Neuvièmes Journées Francophones de Programmation par Contraintes

JFPC 2013

Aix-en-Provence

12 - 14 juin 2013

Actes des
Neuvièmes Journées Francophones
de Programmation par Contraintes

12-14 juin 2013
IUT d'Aix-en-Provence, Aix-Marseille Université

Président des Journées
Djamal Habet, LSIS, Aix-Marseille Université

Présidente du Comité de Programme
Charlotte Truchet, LINA, Université de Nantes

Organisation
Laboratoire des Sciences de l'Information et des Systèmes (LSIS)
Laboratoire d'Informatique Fondamentale de Marseille (LIF)

Les JFPC (Journées Francophones de Programmation par Contraintes) sont le principal congrès de la communauté francophone travaillant sur les problèmes de satisfaction de contraintes (CSP), le problème de la satisfiabilité d'une formule logique propositionnelle (SAT) et/ou la programmation logique avec contraintes (CLP). Issues de la fusion en 2004 des conférences JFPLC [1], nées en 1992, et des JNPC [2], nées en 1994, les JFPC sont patronnées par l'AFPC [3]. Cette neuvième édition des JFPC fait suite aux manifestations antérieures qui se sont tenues à Toulouse (2012), Lyon (2011), Caen (2010), Orléans (2009), Nantes (2008), Rocquencourt (2007), Nîmes (2006) et Lens (2005). Depuis maintenant trois ans, les JFPC sont colocalisées avec les Journées d'Intelligence Artificielle Fondamentale.

1. Journées Francophones de la Programmation Logique avec Contraintes
2. Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets
3. Association Française pour la Programmation par Contraintes, http://afpc.greyc.fr/web/

Présidente du Comité de Programme

Charlotte Truchet, LINA, Université de Nantes, France

Comité de Programme

Patrice Boizumault, GREYC, Université de Caen Basse-Normandie, France
Hadrien Cambazard, G-SCOP, Université de Grenoble, France
Gilles Chabert, LINA, Université de Nantes, France
Rémi Coletta, LIRMM, Université de Montpellier, France
Nadia Creignou, LIF, Université d'Aix-Marseille, France
Thi-Bich-Hanh Dao, LIFO, Université d'Orléans, France
Gilles Dequen, MIS, Université de Picardie Jules Verne, France
Khalil Djelloul, LIFO, Université d'Orléans, France
Hélène Fargier, IRIT, Université Paul Sabatier, France
Frédéric Gardi, Bouygues e-lab, France
Carmen Gervet, German University in Cairo, Egypte
Laurent Granvilliers, LINA, Université de Nantes, France
Djamal Habet, LSIS, Université d'Aix-Marseille, France
Emmanuel Hebrard, LAAS, CNRS, France
George Katsirelos, INRA, Toulouse, France
Jean-Marie Lagniez, FMV, Johannes Kepler University, Austria
Frédéric Lardeux, LERIA, Université d'Angers, France
Nadjib Lazaar, LIRMM, Université de Montpellier, France
Olivier Lhomme, IBM France
Xavier Lorca, LINA, Ecole des Mines de Nantes, France
Florent Madelaine, LIMOS, Université d'Auvergne, France
Arnaud Malapert, I3S, Université Nice Sophia Antipolis, France
Jean-Noël Monette, Uppsala Universitet, Uppsala, Suède
Eric Monfroy, LINA, Université de Nantes, France
Samba-Ndojh Ndiaye, LIRIS, Université Claude Bernard Lyon 1, France
Nicolas Prcovic, LSIS, Université d'Aix-Marseille, France
Benoît Rottembourg, Eurodecision, France
Louis-Martin Rousseau, Ecole Polytechnique de Montréal, Montréal, Canada
Olivier Roussel, CRIL, Université d'Artois, France
Lakhdar Saïs, CRIL, Université d'Artois, France
Pierre Schaus, Université Catholique de Louvain, Belgique
Pierre Siegel, LIF, Université d'Aix-Marseille, France
Laurent Simon, LRI, Université Paris Sud, France
Sylvain Soliman, INRIA Paris-Rocquencourt, France
Igor Stéphan, LERIA, Université d'Angers, France
Elise Vareilles, Ecole des Mines d'Albi-Carmaux, Université de Toulouse, France
Julien Vion, LAMIH, Université de Valenciennes et du Hainaut-Cambrésis, France
Philippe Vismara, LIRMM, SupAgro, France
Bruno Zanuttini, GREYC, Université de Caen Basse-Normandie, France

Relecteurs additionnels

Noureddine Aribi, Vincent Barichard, Loïc Blet, Clément Carbonnel, Laure Devendeville, Line Jakubiec-Jamet, Florian Legendre, Nicolas Paris, Nicolas Schmidt, Johan Thapper, Gilles Trombettoni

Président du Comité d'Organisation

Djamal Habet, LSIS, Université d'Aix-Marseille, France

Comité d'organisation

André Abramé, LSIS, Marseille, France
Nadia Creignou, LIF, Université d'Aix-Marseille, France
Achref El Mouelhi, LSIS, Marseille, France
Stéphane Grandcolas, LSIS, Université d'Aix-Marseille, France
Philippe Jégou, LSIS, Université d'Aix-Marseille, France
Farid Nouioua, LSIS, Université d'Aix-Marseille, France
Nicola Olivetti, LSIS, Université d'Aix-Marseille, France
Richard Ostrowski, LSIS, Université d'Aix-Marseille, France
Cyril Pain-Barre, LSIS, Université d'Aix-Marseille, France
Odile Papini, LSIS, Université d'Aix-Marseille, France
Lionel Paris, LSIS, Université d'Aix-Marseille, France
Nicolas Prcovic, LSIS, Université d'Aix-Marseille, France
Vincent Risch, LSIS, Université d'Aix-Marseille, France
Pierre Siegel, LIF, Université d'Aix-Marseille, France
Cyril Terrioux, LSIS, Université d'Aix-Marseille, France
Donia Toumi, LSIS, Marseille, France
Eric Würbel, LSIS, Université du Sud Toulon - Var, France
Safa Yahi, LSIS, Université d'Aix-Marseille, France

Préface

Les JFPC (Journées Francophones de Programmation par Contraintes) réunissent chaque année la communauté francophone des domaines de la Programmation par Contraintes (PPC), de la Satisfiabilité de formules booléennes (SAT) et de la Programmation Logique avec Contraintes (PLC) . Pour la troisième année consécutive, les JFPC sont colocalisées avec les JIAF (Journées d'Intelligence Artificielle Fondamentale) : nous partagerons certains exposés scientifiques et le programme social, notamment grâce au travail de Sébastien Konieczny, président du Comité de Programme des JIAF. Nul doute que ce sera l'occasion d'échanges fructueux au croisement de la PPC et de l'une de ses disciplines parentes, historiquement.

Cette année, les JFPC ont reçu 48 soumissions d'articles courts et longs, certains originaux, certains traduits d'articles de grandes conférences internationales du domaine. A l'issue du travail du comité de programme, 40 articles ont été retenus pour une présentation à la conférence. Une grande partie du programme est au cœur du domaine : prouveurs SAT, solveurs de contraintes, préférences, consistances, etc. Mais les thèmes abordés sont aussi ouverts à d'autres disciplines, proches ou moins proches : programmation linéaire, programmation mathématique, logique, complexité, parallélisation, vérification, etc. Enfin, les applications pratiques des contraintes sont bien représentées, tant pour des problèmes d'informatique fondamentale que pour des applications très pratiques en œnologie, urbanisme ou réalité virtuelle. Ainsi, le programme reflète à la fois la diversité des origines de la programmation par contraintes en Intelligence Artificielle, Programmation Logique ou Recherche Opérationnelle, et la richesse de ses récents développements, fondamentaux et applicatifs.

Ce programme sera complété par trois exposés invités. Le premier, proposé par les JIAF, concerne l'ensemble de la communauté académique et devrait susciter de nombreuses réflexions : Denis Bouyssou (CNRS-Lamsade, Université de Paris Dauphine) présentera une étude scientifique critique des indicateurs bibliométriques. Les deux autres ont été choisis par le comité de programme des JFPC. Stefan Heinz (Zuse Institute Berlin, Allemagne) présentera SCIP, outil intégrant la PPC, SAT et le *Mixed Integer Programming*, un développement prometteur en programmation mathématique discrète. Frédéric Saubion (LERIA, Université d'Angers) proposera de lutter contre la difficile prise en main des méthodes de résolution grâce à des algorithmes capables de se configurer ou reconfigurer de façon autonome.

Les JFPC ont toujours fait une large place aux jeunes chercheurs, grâce à une politique active de l'AFPC en ce sens. Cette année, parmi les soumissions, 18 avaient un doctorant comme premier auteur. Par ailleurs, de nombreux jeunes docteurs ont participé, et très activement, au comité de programme. Alors que les carrières scientifiques sont de plus en plus contraintes, notamment au début, je crois que l'intégration des jeunes chercheurs à la communauté scientifique est fondamentale : cette année encore, les JFPC reposent sur un travail important de leur part, et les accueilleront chaleureusement.

Le comité de programme a réuni 40 scientifiques, académiques et industriels, de 7 pays différents, représentant au total 28 laboratoires et entreprises. Je remercie très sincèrement tous ses membres pour la qualité de leur travail. Les discussions ont été nombreuses, parfois passionnées, toujours constructives, avec une qualité d'échanges remarquable. Je remercie tout particulièrement Laurent Simon, qui a accepté de superviser les articles pour lesquels j'étais en conflit d'intérêt.

Chaque année, les JFPC sont placées sous le signe de la convivialité. Djamal Habet, Vincent Risch et leur comité d'organisation n'ont pas ménagé leur peine pour respecter cette tradition, organisant la conférence dans les meilleures conditions : un immense merci, et bravo à eux ! Merci enfin à nos sponsors : AFPC, GdR I3, INRIA, Aix-Marseille Université, FRIIAM, LSIS, LIF, Conseil Général des Bouches du Rhône, Communauté du Pays d'Aix, Cosytec, Index Education, Prologia, ainsi qu'à l'IUT d'Aix-en-Provence qui accueillera la conférence.

Je vous souhaite une très belle conférence.

Charlotte Truchet
Présidente du Comité de Programme des JFPC 2013

- Exposé invité - From CP to CIP: Building Constraint-Based Hybrid Models and Solvers

Stefan Heinz

Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
heinz@zib.de

Constraint integer programming (CIP) was introduced by Achterberg [2, 3, 4] in 2004. This paradigm builds a foundation for different problem classes, namely, mixed integer programming (MIP), constraint programming (CP), and satisfiability testing (SAT). The basic concept is to combine solving techniques from the different fields and providing a solid framework for hybrid approaches. This hybrid allows for more structural approaches than linear constraints in MIP and clauses in SAT combined with powerful low level sophisticated solving techniques from MIP and SAT such as the linear programming relaxation and conflict analysis, respectively.

In this talk we give an introduction to CIP and introduce the SCIP Optimization Suite [1] which is a framework to solve CIPs. We visualize the flexible design of the solver and show its advantages as well as its limitations. In the main part we discuss the realization of cumulative scheduling within the CIP methodology, focusing mainly on the presolving phase.

Presolving takes place *before* the tree search starts and tries to reduce the size of the model by, for example, removing irrelevant information such as redundant constraints or already-assigned variables ; by decomposing or reformulating constraints (e.g., tightening the bounds of the variables or strengthening coefficients of the constraints) ; and by extracting structural information such as cliques (sets of binary variables that must sum up to one) that can be used by branching heuristics or cutting plane generation. Presolving collects global structure information and transforms the given problem instance into an equivalent instance w.r.t. the optimal value, that is potentially easier to solve. For state-of-the-art mixed integer programming solvers, presolving is one of the most important components. We show how the global cumulative constraint can contribute and make use of the available global structures [5].

Finally, we present computational results, comparing SCIP a constraint based system within the MIP context and against a state-of-the-art scheduling solver.

Références

[1] SCIP – Solving Constraint Integer Programs. http://scip.zib.de.

[2] Tobias Achterberg. Scip - a framework to integrate constraint and mixed integer programming. ZIB-Report 14-19, Zuse Institute Berlin, 2004.

[3] Tobias Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.

[4] Tobias Achterberg. SCIP : Solving constraint integer programs. *Mathematical Programming Computation*, 1(1) :1–41, 2009.

[5] Stefan Heinz, Jens Schulz, and J. Christopher Beck. Using dual presolving reductions to reformulate cumulative constraints. *Constraints*, 18(2) :166–201, 2012.

- Exposé invité - Vers des algorithmes plus autonomes

Frédéric Saubion
LUNAM Université, Université d'Angers
LERIA
Angers, France
saubion@info.univ-angers.fr

Les progrès réalisés au cours des dernières décennies en matière de résolution de problèmes d'optimisation combinatoire et de satisfaction de contraintes ont été considérables. Les algorithmes et solveurs actuels permettent ainsi de traiter des applications réelles de grande taille en contrepartie, toutefois, d'un fonctionnement plus complexe à appréhender.

En effet, ces méthodes incluent souvent de nombreux paramètres et heuristiques spécifiques, et deviennent ainsi hors de portée des utilisateurs cibles - typiquement des ingénieurs R&D - que ce soit pour les implémenter ou pour les utiliser de manière aisée et satisfaisante.

D'un point de vue plus général, sélectionner la meilleure modélisation puis le meilleur algorithme pour résoudre un problème donné, reste un défi important, à l'intersection de plusieurs domaines tels que l'intelligence artificielle, la recherche opérationnelle ou encore l'apprentissage automatique.

Pourtant, l'aide à la conception d'algorithmes et au réglage automatisé de leurs paramètres a fait l'objet d'une attention particulière depuis plusieurs années en optimisation et en programmation par contraintes. étant donné un algorithme dont les paramètres ont été clairement identifiés - qu'ils affectent directement la structure de cet algorithme ou en modifient le comportement - différentes approches peuvent être envisagées et, en particulier :

- le réglage hors ligne des paramètres à partir d'essais,
- l'utilisation de plusieurs algorithmes ou configurations d'un même algorithme au sein d'un portefeuille couplé à des outils de prédiction de performance,
- le contrôle en ligne, de manière dynamique, des paramètres à partir de l'analyse de la situation courante du calcul,
- la découverte de nouvelles heuristiques par des mécanismes d'apprentissage automatique.

Dans cet exposé, nous présentons la problématique générale de la conception d'algorithmes plus autonomes, en lien avec les différentes taxonomies de la littérature. Nous illustrons notre propos sur le problème plus spécifique de la sélection d'opérateurs dans le cadre d'algorithmes de type méta-heuristiques. Enfin, nous dressons quelques perspectives dans cette voie de recherche en plein essor.

Références

[1] Youssef Hamadi, Éric Monfroy, and Frédéric Saubion. *Autonomous Search*. Springer, 2012.

Liste des articles

Règles d'inférence et recherche locale pour Max-SAT et Max-SAT valué
André Abramé, Djamal Habet 1
Détection des fonctions de rang linéaires à terme
Anthony Alezan, Roberto Bagnara, Fred Mesnard, Etienne Payet 11
Compilation de CSPs : carte de complexité des MDDs non-déterministes
Jérôme Amilhastre, Hélène Fargier, Alexandre Niveau, Cédric Pralet 21
Extraction de régions intérieures pour améliorer le majorant en optimisation globale sous contraintes
Ignacio Araya, Gilles Trombettoni, Bertrand Neveu, Gilles Chabert 31
Approches d'élicitation basées sur le coëfficient de Spearman pour l'optimisation multicritère par ordre lexicographique
Noureddine Aribi, Yahia Lebbah 41
Un nouveau cadre diviser pour régner pour SAT distribué
Gilles Audemard, Benoît Hoessen, Said Jabbour, Cédric Piette 51
Du glucose en goutte à goutte pour les coeurs inconsistants
Gilles Audemard, Jean Marie Lagniez, Laurent Simon 59
Cohérences Locales Paramétrées
Amine Balafrej, Christian Bessière, Remi Coletta, El Houssine Bouyakhf 67
La recherche locale pour la pré-programmation d'environnements urbains durables
Bruno Belin, Marc Christie, Charlotte Truchet 75
Compilation de requêtes à préférences en problèmes de contraintes qualitatives
Jean-François Condotta, Souhila Kaci 85
Un modèle général pour la classification non supervisée sous contraintes d'utilisateur
Thi-Bich-Hanh Dao, Khanh-Chuong Duong, Christel Vrain 95
Extension des cohérences WCSP aux tuples
Djamel E Dehani, Christophe Lecoutre, Olivier Roussel 105
Solveurs CP(FD) vérifiés formellement
Catherine Dubois, Arnaud Gotlieb 115
Une approche basée sur les contraintes pour résoudre le problème d'arbre recouvrant de coût minimum avec contraintes de degré
Jean-Guillaume Fages, Xavier Lorca, Louis-Martin Rousseau 119
Compacité pratique des diagrammes de décision valués : normalisation, heuristiques et expérimentations
Hélène Fargier, Nicolas Schmidt, Pierre Marquis 123
Raffinement de la décomposition arborescente par fusion de clusters pour guider DGVNS
Mathieu Fontaine, Samir Loudni, Patrice Boizumault 133
STR et compression de contraintes tables
Nebras Gharbi, Fred Hemery, Christophe Lecoutre, Olivier Roussel 143
Stratégies d'élimination des clauses apprises dans les solveurs SAT modernes
Long Guo, Said Jabbour, Lakhdar Saïs 147
Approche unifiée pour la Décomposition et la Résolution de PCSP : étude expérimentale sur FAP
Zineb Habbas, Sébastien Martin, Lamia Sadeg, Daniel Singer 157
La programmation par contraintes pour l'assistance à l'agencement d'environnements virtuels
Marouene Kefi, Vincent Barichard, Paul Richard 167
cptest4choco : test et mise-au-point des programmes à contraintes
Nadjib Lazaar, Arnaud Gotlieb, Yahia Lebbah 177
Réutiliser ou adapter les prouveurs SAT pour l'optimisation booléenne
Daniel Le Berre, Emmanuel Lonca 185
Résolution du problème WCSP par extraction de noyaux insatisfiables minimaux
Christophe Lecoutre, Nicolas Paris, Olivier Roussel, Sébastien Tabary 195

DGVNS guidée par les séparateurs
Samir Loudni, Mathieu Fontaine, Patrice Boizumault 205
Un Algorithme Optimal de Filtrage pour Contraintes Table
Jean-Baptiste Mairy, Pascal Van Hentenryck, Yves Deville 215
Exploiter les séparateurs : une approche hybride combinant recherche locale et arborescente
Jean-Philippe Métivier, Mathieu Fontaine, Samir Loudni 223
Des Propagateurs Indépendants des Solveurs
Jean-Noël Monette, Pierre Flener, Justin Pearson 227
Sur une classe polynomiale hybride pour les CSP d'arité quelconque
El Mouelhi Achref, Philippe Jegou, Cyril Terrioux 237
Cohérence d'arc virtuelle dynamique
Hiep Nguyen, Thomas Schiex, Christian Bessière 249
Un solveur de contraintes basé sur les domaines abstraits
Marie Pelleau, Antoine Miné, Charlotte Truchet, Frédéric Benhamou 259
Raffiner l'analyse par interprétation abstraite des programmes à l'aide des techniques de programmation par contraintes
Olivier Ponsini, Claude Michel, Michel Rueher 269
Prototyper des moteurs de propagation avec un DSL
Charles Prud'Homme, Xavier Lorca, Rémi Douence, Narendra Jussien 279
Une adaptation simple et efficace du modèle MapReduce pour la programmation par contraintes
Jean-Charles Régin, Mohamed Rezgui, Arnaud Malapert 289
Linear Programming, the Complexity of Finite-valued CSPs
Johan Thapper, Stanislav Zivny 299
Une étude empirique du module d'analyse de conflits dans les solveurs CDCL
Donia Toumi, Djamal Habet 303
Sélection adaptative d'opérateurs en recherche locale et Progressive Party Problem
Nadarajen Veerapen, Youssef Hamadi, Frédéric Saubion 313
Consistance d'arc par MDD-Réduction
Julien Vion 323
Assemblage de vins sous contraintes
Philippe Vismara, Remi Coletta, Gilles Trombettoni 333
Éliminations des solutions symétriques pour l'isomorphisme de sous-graphe
Philippe Vismara 343
Maintien asynchrone de la consistance d'arc dans la recherche distribuée synchrone
Mohamed Wahbi, Redouane Ezzahir, Christian Bessière, El Houssine Bouyakhf 347

Règles d'inférence et recherche locale pour Max-SAT et Max-SAT valué

André Abramé et Djamal Habet

Université d'Aix Marseille , LSIS UMR 7296
Av. Escadrille Normandie Niemen
13397 Marseille Cedex 20 (France)
{andre.abrame,djamal.habet}@lsis.org

Résumé

Ces dernières années, d'importantes avancées ont été accomplies dans la résolution du problème Max-SAT. En particulier, l'utilisation de nouvelles règles d'inférence dans les solveurs complets de type "séparation et évaluation" (Branch and Bound) a permis d'améliorer sensiblement leur efficacité. Cependant, peu d'études ont été réalisées sur les méthodes incomplètes pour Max-SAT bien que celles-ci aient prouvé leur intérêt pour SAT par le biais de méthodes de recherche locale. Dans ce papier, nous montrons que l'inclusion de règles d'inférence dans un algorithme de recherche locale pour SAT améliore ses performances lorsqu'il est appliqué à Max-SAT. Les résultats obtenus confirment l'efficacité de notre approche, en particulier sur les instances max-2-sat.

Abstract

In the last years, many advances have been accomplished in the exact solving of the Max-SAT problem, especially by the definition of new inference rules and a better estimation of lower bounds in branch and bound based methods. However, and oppositely to the SAT problem, fewer works exist on approximate methods for Max-SAT, mainly local search ones which have shown their efficiency for SAT. In this paper, we illustrate that including inference rules in a classical local search solver for SAT improves its performances when solving the Max-SAT problem. The obtained results confirm the efficiency of our approach, especially on max-2-sat instances.

1 Introduction

Le problème Max-SAT consiste à trouver une affectation de valeurs booléennes aux variables d'une formule propositionnelle en forme normale conjonctive (CNF) qui maximise le nombre de clauses satisfaites. Ce problème NP-difficile [23] est la version optimisation du problème SAT. Ces dernières années, l'intérêt porté à l'étude de Max-SAT a crû en raison de sa capacité à exprimer et résoudre des problèmes de la vie réelle, dans des domaines variés comme le routage, la planification, la bio-informatique et aussi des problèmes académiques (Max-Cut, Max-Clique, etc).

Les approches pour résoudre Max-SAT peuvent être divisées en deux catégories principales : les méthodes complètes et les méthodes incomplètes. Les méthodes complètes donnent, en un temps fini mais potentiellement long, la solution optimale. Elles sont généralement basées sur des algorithmes de type "séparation et évaluation" (*Branch'n'Bound*, BnB). Il en existe de nombreuses implémentations [2, 12, 19] qui se distinguent principalement par leurs heuristiques de choix des variables, leurs méthodes d'estimation de la borne inférieure et les règles d'inférence qu'elles utilisent. Les méthodes incomplètes quant à elles sont capables de trouver une solution de bonne qualité en un temps raisonnable, mais pas de prouver l'optimalité. Ces méthodes incluent la programmation semi-définie, l'optimisation pseudo-booléenne, etc [21, 28, 8]. Les algorithmes de recherche locale (RL), efficaces sur le problème SAT, appartiennent également à cette seconde catégorie. Tous les algorithmes de RL pour SAT peuvent être appliqués à Max-SAT moyennant quelques modifications mineures. Cependant, des adaptations spécifiques de RL ont été réalisées pour le problème Max-SAT. La première de ces adaptations est due à Hansen et Jaumard [9]. On peut également citer Zang et al. [16] qui utilisent les backbones pour améliorer un algorithme de RL classique, Smyth et al. [26] qui ont adapté un algorithme de recherche tabou à Max-SAT ou encore Lardeux et al. [17] qui ont com-

biné des algorithmes de recherche complet et incomplet en utilisant un mécanisme à trois valeurs. Plus récemment, Kroc et al. [16] ont utilisé un solveur SAT de type DPLL pour guider un solveur RL s'exécutant en parallèle. A notre connaissance, à part la contribution de Heras et Baneres citée ci-dessous, il n'y a pas de travaux plus récents sur l'adaptation d'algorithme RL à Max-SAT. Enfin, une présentation générale du problème Max-SAT est faite dans [5] et deux états de l'art sur l'adaptation des algorithmes de RL à Max-SAT existent, un complet mais ancien [27] et un plus récent [14].

Les règles d'inférence ont montré leur potentiel dans les algorithmes BnB, par exemple pour simplifier la formule ou améliorer l'estimation de la borne inférieure. Cependant, ces règles ont rarement été exploitées dans un algorithme RL pour faciliter la résolution d'instances Max-SAT. A notre connaissance, à part Heras et Baneres [10] qui ont utilisé ces règles en pré-traitement avant d'appliquer un algorithme RL classique, il n'y a pas de travaux sur le sujet. Dans ce papier, nous utilisons une règle d'inférence combinant la propagation unitaire simulée et la Max-résolution pour détecter et mémoriser une partie des sous-ensembles inconsistants des instances. Une des difficultés à surmonter est le maintien d'un graphe d'implication cohérent pour rendre possible la construction d'arbres de réfutation dans le contexte de la RL (ce point à déjà été étudié dans le contexte de SAT, par exemple dans [20, 13, 3, 4, 7]). De plus, il faut trouver un juste milieu entre le temps passé à appliquer la règle d'inférence et le temps passé à résoudre l'instance. Sinon, l'algorithme peut s'avérer inefficace en pratique. Enfin, ce travail améliore et étend celui présenté dans [1].

Ce papier est organisé comme suit. Dans la section 2, nous rappelons les définitions et notations utilisées dans ce papier. La règle d'inférence et l'algorithme de RL que nous avons utilisés dans notre implémentation sont décrits respectivement dans les sections 3 et 4. Nous détaillons dans la section 5 notre implémentation et dans la section 6 les expérimentations que nous avons menées et les résultats que nous avons obtenus. Enfin, nous concluons dans la section 7.

2 Notations

Soit $X = \{x_1, \ldots, x_n\}$ un ensemble de variables propositionnelles. Un *littéral* l est une variable x_i ou sa négation $\neg x_i$. Une *clause* est une disjonction finie de littéraux et une *formule* Φ en forme normale conjonctive (CNF) est une conjonction de clauses. Une clause est dite valuée si on lui associe un poids w et une conjonction de clauses valuées est une formule valuée. De manière alternative, les clauses peuvent être représentées comme des ensembles de littéraux et les formules comme des multi-ensembles de clauses. De même, les clauses valuées peuvent être représentées comme des ensembles pondérés de littéraux et les formules valuées comme des multi-ensembles de clauses valuées. On notera $|c|$ le nombre de littéraux d'une clause c et $|\Phi|$ le nombre de clauses d'une formule Φ.

Une *interprétation* I est un ensemble fini de littéraux qui ne contient pas à la fois un littéral l et son opposé $\neg l$. Si $|I| = n$ alors I est dite *complète* et *partielle* sinon. Une variable x_i telle que ni x_i ni $\neg x_i$ n'appartiennent à I est non-affectée. Pour une interprétation unitaire $I = \{l\}$ donnée, on notera $\Phi|_I$ la formule obtenue en appliquant I à Φ. Formellement, $\Phi|_I = \{c \mid c \in \Phi, \{l, \neg l\} \cap c = \emptyset\} \cup \{c/\{\neg l\} \mid c \in \Phi, \neg l \in c\}$. Cette notation peut être étendue à n'importe quelle interprétation $I = \{l_1, l_2, \ldots, l_k\}$ comme suit : $\Phi|_I = (\ldots((\Phi|_{l_1})|_{l_2})\ldots|_{l_k})$.

Un littéral l est *satisfait* par une interprétation I ssi $l \in I$ et il est *falsifié* ssi $\neg l \in I$. Une clause est *satisfaite* ssi au moins un de ses littéraux est satisfait et elle est *falsifiée* (ou *conflictuelle*) ssi tous ses littéraux sont falsifiés. Par convention la clause vide, notée $\square$, est conflictuelle.

Le problème Max-SAT consiste à trouver une interprétation qui maximise (minimise) le nombre de clauses satisfaites (falsifiées). La version valuée de Max-SAT consiste, quant à elle, à trouver une interprétation qui maximise (minimise) la somme des poids des clauses satisfaites (falsifiées). Nous considérerons dans la suite de ce papier toutes les formules comme étant valuées, les non-valuées étant un cas particulier où toutes les clauses sont de poids 1.

3 Règles d'inférence : propagation unitaire simulée et Max-résolution

Ces dernières années, les recherches sur le problème Max-SAT ont majoritairement porté sur sa résolution par des méthodes exactes de type BnB, et en particulier sur une de leurs composantes : les règles d'inférence qui transforment une formule Φ en une formule équivalente Φ'. On notera cette transformation $\frac{\Phi}{\Phi'}$.

Dans le contexte du problème SAT, les règles d'inférence doivent seulement préserver la satisfiabilité de la formule. Dans le cadre du problème Max-SAT, elles doivent également préserver le nombre de clauses falsifiées par chaque interprétation. Ces règles d'inférence peuvent avoir différents rôles, comme étendre l'interprétation courante, améliorer l'estimation de la borne inférieure ou servir de système de mémorisation. Dans cette section, nous rappelons la règle d'inférence pour Max-SAT définie dans [12] qui utilise l'adaptation pour

Max-SAT de deux règles d'inférence bien connues dans le contexte de SAT : la propagation unitaire et la résolution.

Une clause qui ne contient qu'un seul littéral est appelée une *clause unitaire*. En présence d'une telle clause, toutes les clauses contenant le littéral et toutes les occurrences de sa négation peuvent être supprimées de la formule. L'application itérative de cette règle jusqu'à ce qu'il n'y ait plus aucune clause unitaire est appelée *propagation unitaire*. On notera $\Phi*$ la formule obtenue en appliquant la propagation unitaire sur une formule Φ. Cette règle d'inférence préserve la satisfiabilité de la formule mais pas le nombre de clauses falsifiées par chaque interprétation et n'est donc pas utilisable dans le contexte de Max-SAT. Cependant, certains solveurs Max-SAT complets récents utilisent la propagation unitaire pour détecter les conflits et améliorer l'estimation de la borne inférieure, mais sans appliquer ses conséquences à la formule. On parlera alors de *propagation unitaire simulée* (SUP).

Une autre règle d'inférence très utilisée dans le contexte de SAT est la *résolution* [24]. Si deux clauses $\{x, A\}$ et $\{\neg x, B\}$ sont présentes dans la formule (avec A et B des disjonctions de littéraux), alors la clause $\{A, B\}$, dite *résolvante*, peut être ajoutée à la formule sans affecter sa satisfiabilité. Comme précédemment, cette règle ne préserve pas le nombre de clauses falsifiées par chaque interprétation et elle n'est donc pas utilisable dans le contexte de Max-SAT. Une variante, la *Max-résolution* a été proposée dans [17] (et étudiée plus en détail dans [11, 6]). Elle peut être définie comme suit :

$$\frac{\{C_{o_1}, C_{o_2}\} \cup \Phi'}{\{C_r, C_{c_1}, C_{c_2}, C_{c_3}, C_{c_4}\} \cup \Phi'}$$

avec $C_{o_1} = (\{x, A\}, w_1)$ et $C_{o_2} = (\{\neg x, B\}, w_2)$ les clauses originales, $C_r = (\{A, B\}, m)$ la clause résolvante, $C_{c_1} = (\{x, A\}, w_1 - m)$, $C_{c_2} = (\{\neg x, B\}, w_2 - m)$, $C_{c_3} = (\{x, A, \neg B\}, m)$ et $C_{c_4} = (\{\neg x, \neg A, B\}, m)$ les clauses de compensation, $m = min\{w_1, w_2\}$ et Φ' un multi-ensemble de clauses valuées. Deux des clauses ajoutées ne sont pas des disjonctions de littéraux et doivent donc être transformées pour préserver la forme CNF de la formule. Cette transformation produit $(|A| * (|B| - 1) + (|A| - 1) * |B|)$ clauses de tailles variant de $(min\{|A|, |B|\} + 2)$ à $(|A| + |B| + 1)$.

Dans [12], ces règles d'inférence sont utilisées pour améliorer l'estimation de la borne inférieure faite par le solveur BnB MiniMaxSat. Le principe est d'utiliser SUP et la Max-résolution pour faire des déductions sur la formule comme le fait le mécanisme d'apprentissage de clauses des solveurs SAT de type *Conflict Driven Clause Learning* (CDCL) : quand un conflit est détecté, une succession d'étapes de résolution est appliquée sur les clauses ayant mené au conflit et une nouvelle clause est ajoutée à la formule. De la même manière, MiniMaxSat applique SUP après chaque décision. Quand un conflit est détecté, l'ensemble des clauses qui y ont mené forme un sous-ensemble de clauses non-satisfiables de la formule. De ce sous-ensemble, la clause vide $\square$ peut être dérivée par plusieurs étapes de Max-résolution faites dans un ordre donné. Ces étapes de résolution constituent l'*arbre de réfutation* et peuvent être facilement déduites du graphe d'implication (ou d'une simple file des propagations). Contrairement au mécanisme d'apprentissage des solveurs SAT CDCL, l'ensemble des clauses ayant mené au conflit sont supprimées de la formule et des clauses de compensation sont ajoutées pour préserver le nombre de clauses satisfaites et falsifiées par chaque interprétation. Pour limiter le nombre de clauses ajoutées à la formule, MiniMaxSat n'applique cette méthode que quand toutes les clauses du sous-

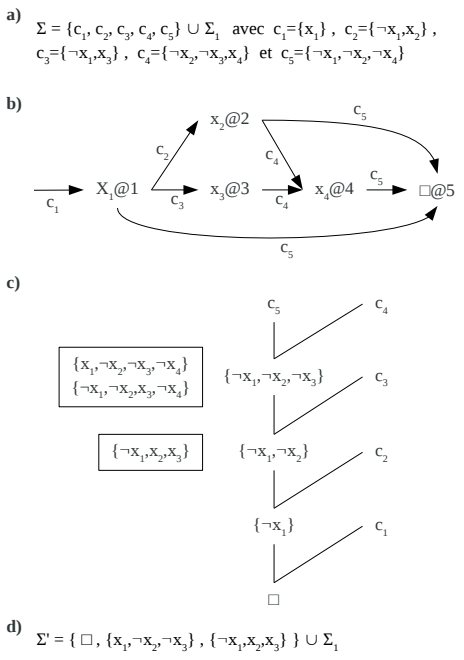


FIGURE 1 – Exemple d'application de la règle d'inférence basée sur SUP et la Max-résolution sur une formule non-valuée. (a) La formule. (b) Le graphe d'implication qui montre les étapes de propagations unitaires (chaque noeud correspond à une affectation, avec leur ordre donné par @X, et chaque arc est étiqueté par la clause qui a conduit à l'affectation). (c) L'arbre de réfutation et les étapes de Max-résolution qui sont appliquées à la formule (dans les cadres, les clauses de compensations qui doivent être ajoutées à la formule). (d) La formule transformée.

ensemble non-satisfiable contiennent moins de cinq littéraux. La fig. 1 donne un exemple d'application de cette règle d'inférence. Enfin, il est intéressant de noter que MiniMaxSat applique cette règle à chaque noeud de l'arbre de recherche, mais il ne conserve les transformations que dans le sous-arbre. Les transformations sont défaites lorsque l'algorithme effectue un retour arrière.

4 Algorithme de recherche locale : AdaptNovelty++

Cette section donne un bref aperçu des algorithmes de recherche locale pour SAT, et en particulier de Novelty++ [18] que nous utilisons dans notre approche.

L'algorithme 1 donne le schéma général d'un algorithme de recherche locale pour SAT basé sur WalkSat (introduit pour la première fois dans [25]). L'algorithme commence par choisir aléatoirement une interprétation complète I (fonction `interprétation_aléatoire`, ligne 4). Puis, itérativement, il choisit aléatoirement une clause falsifiée c et une variable $x_i \in c$ qu'il flippe (lignes 7-9) de manière à améliorer le nombre de clauses satisfaites jusqu'à ce qu'une solution (une interprétation satisfaisant toutes les clauses) soit trouvée ou qu'un nombre maximum donné de flippes (*FreqRelances*) soit atteint. Ce processus est répété pour un nombre donné de relances (*MaxRelances*). Ce système de relances peut être vu comme un mécanisme de diversification dans le sens où il réduit l'impact de l'interprétation aléatoire initiale sur l'efficacité de la recherche.

Les variantes de WalkSat diffèrent principalement par leur heuristique de choix des variables à flipper [22]. Ces heuristiques se basent sur deux compteurs : `break` et `make` qui calculent, pour une variable x_i, le nombre de clauses qui seront respectivement falsifiées et satisfaites si x_i est flippée. La différence entre `break`(x_i) et `make`(x_i) est notée `score`(x_i). La version originale de WalkSat, une fois la clause falsifiée c sélectionnée, choisit la variable à flipper aléatoirement parmi les variables de c qui ont une valeur de break nulle s'il en existe. Sinon, elle choisit aléatoirement une variable de c avec une probabilité p et avec une probabilité $1-p$ elle choisit la variable avec la plus petite valeur de break.

Dans Novelty [22], les variables apparaissant dans c sont triées en fonction de leurs scores. L'algorithme considère les deux variables de plus hauts scores. Si la variable de plus haut score n'est pas celle qui, des variables de c, a été flippée le plus récemment alors elle est choisie. Sinon, avec une probabilité p, l'algorithme choisit celle de second plus haut score et avec une probabilité $1-p$ celle de plus haut score.

Algorithm 1: WalkSat

Data: Φ, *MaxRelances*, *FreqRelances*.
Result: Si une interprétation I satisfaisant Φ a été trouvé alors I, sinon $\emptyset$.

```
1  begin
2      for i ← 1 to MaxRelances do
3          I ← interprétation_aléatoire();
4          for j ← 1 to FreqRelances do
5              if est_satisfaite(Φ|_I) then
6                  return I;
7              c ← choix_clause_falsifiée(Φ|_I);
8              x ← choix_variable_à_flipper(c);
9              I ← flippe(I, x);
10     return ∅
```

Dans [18], Novelty est étendue à Novelty++ comme suit : avec une probabilité dp (*diversification probability*) l'algorithme choisit la variable qui a été flippée le moins récemment (diversification) et avec une probabilité $1-dp$ il fait comme Novelty.

Les valeurs optimales de p et dp dépendent de l'instance traitée et sont difficiles à déterminer. Hoos propose dans [15] un mécanisme d'ajustement dynamique de ces valeurs en fonction de l'évolution de la recherche. Si le nombre de clauses falsifiées n'a pas été réduit pendant un certain nombre de flippes, alors les valeurs de p et dp sont augmentées pour favoriser la diversification. A l'inverse, ces valeurs sont réduites lorsque le nombre de clauses falsifiées diminue. La combinaison de Novelty++ et du mécanisme d'adaptation des probabilités est appelée AdaptNovelty++.

Nous avons ajouté à AdaptNovelty++ une heuristique de sélection des clauses falsifiées pour les formules valuées. Plutôt que de choisir aléatoirement parmi l'ensemble des clauses falsifiées de la formule, AdaptNovelty++ choisit parmi les clauses falsifiées de plus fort poids. Pour éviter de sélectionner toujours les mêmes clauses, nous avons ajouté un mécanisme de tabou. Lorsqu'une clause est choisie elle devient tabou, c'est-à-dire qu'elle ne peut plus être sélectionnée. La durée du tabou est fixée empiriquement à 0,9 fois le nombre de clauses falsifiées de la formule.

5 IRAnovelty++ : recherche locale et règles d'inférence pour Max-SAT

Notre objectif est d'intégrer des règles d'inférence dans un algorithme de recherche locale pour améliorer ses performances lorsqu'il est appliqué au problème Max-SAT. Nous avons donc ajouté à AdaptNovelty++ les éléments nécessaires à l'application de la règle d'inférence décrite dans la section 3, formant ainsi un nou-

veau solveur Max-SAT : IRAnovelty++.

La première difficulté à surmonter est l'intégration de la simulation de propagation unitaire (qui est nécessaire pour la construction d'arbres de réfutation, et donc pour appliquer la Max-résolution) dans AdaptNovelty++. Comme les algorithmes de recherche locale travaillent sur des interprétations complètes tandis que SUP travaille sur une interprétation partielle, notre solveur maintient simultanément deux affectations : une complète pour la recherche locale (I_{RL}) et une partielle pour la détection des conflits par SUP (I_{SUP}). Cette seconde interprétation est remplie par les flippes réalisés par la partie recherche locale de l'algorithme. Le maintient du graphe d'implication, qui mémorise les étapes de propagation unitaire, est plus complexe que dans le contexte de SAT pour deux raisons : (1) les affectations de I_{SUP} ne sont pas faites et défaites chronologiquement et (2) l'application de la règle d'inférence supprime des clauses de la formule, clauses qui peuvent être des sources de propagation. Pour permettre la construction et le maintien d'un graphe d'implication cohérent dans ce contexte, IRAnovelty++ mémorise toutes les sources de propagation des variables. Enfin, une affectation l peut être supprimée de I_{SUP} dans deux situations : si $\neg l$ est ajouté à I_{SUP} ou si elle provoque un conflit direct avec des affectations plus récentes (i.e. si il existe une clause $\{\neg l, \neg l_1, \dots \neg l_k\}$ telle que $l_1, \dots, l_k$ appartiennent à I_{SUP} et sont tous plus récents que l.

La seconde étape pour pouvoir appliquer la règle d'inférence est la construction des arbres de réfutation lorsque des conflits sont découverts par SUP (i.e. quand il existe une clause c telle que pour tout littéral $l \in c$, $\neg l \in I_{SUP}$). Ce point ne nécessite pas d'être détaillé, puisqu'il est en tout point similaire et largement documenté dans le contexte des solveurs SAT complets.

Enfin, la Max-résolution est appliquée selon les arbres de réfutation, comme décrit dans la section 3. Ainsi, chaque sous-ensemble inconsistant détecté (les clauses d'un arbre de réfutation forment un sous-ensemble inconsistant) est remplacé par une clause vide conjuguée à des clauses de compensation. Les modifications apportées à la formule sont conservées durant toute la résolution, même après qu'une affectation qui les a provoquées a été défaite. Utilisée ainsi, la Max-résolution permet de retenir, par les clauses vides ajoutées, les conflits détectés tout au long de la recherche.

Nous avons défini deux limites pour contrôler les transformations appliquées à la formule et limiter sa taille après transformation : *LimSUP* et *LimMR*.

- *LimSUP* définit la taille maximum que peut atteindre la formule transformée, exprimée en ratio de l'originale. Lorsque cette limite est atteinte, SUP n'est plus appliquée (et par conséquent la Max-résolution non plus) et seule la partie recherche locale de l'algorithme reste active. Avec des valeurs < 1, la formule n'est jamais modifiée et l'algorithme se comporte comme AdaptNovelty++.
- *LimMR* définit la taille maximum autorisée des résolvants lors de l'application de la Max-résolution, exprimée en ratio de la taille de la plus petite clause de l'arbre de réfutation. Lorsque cette limite est dépassée, les modifications ne sont pas appliquées et le conflit est simplement ignoré.

Comme AdaptNovelty++, IRAnovelty++ relance la recherche locale avec une nouvelle interprétation complète choisie aléatoirement après $\mathit{FreqRelances} * |\Phi|$ flippes. Il inclut également le mécanisme d'adaptation des probabilités de diversification et l'heuristique de choix des clauses falsifiées basée sur le poids des clauses, tous deux présentés dans la section 4.

IRAnovelty++ est détaillé dans l'algorithme 2 où les notations et fonctions suivantes sont utilisées : b et I_b sont respectivement le nombre minimum de clauses falsifiées (la meilleure solution) trouvé jusqu'à présent et l'interprétation correspondante. La fonction `poids(`c`)` renvoie le poids de la clause c passée en argument. La fonction `affecte` (I, l) ajoute un littéral l à l'interprétation I et résout les éventuelles contradictions comme décrit au début de cette section, `arbre_réfutation` (Φ, c) renvoie l'arbre de réfutation de la clause c dans la formule Φ. `max_résolution` (Φ, R) renvoie la formule obtenue en appliquant la Max-résolution sur Φ selon l'arbre de réfutation R. Enfin, `res(`R`)` et `src(`R`)` renvoient respectivement la clause résolvante et l'ensemble des clauses sources d'un arbre de réfutation R.

Pour *MaxRelances* étapes, IRAnovelty++ agit comme suit. Il commence par initialiser aléatoirement l'interprétation complète I_{RL} (ligne 5). Puis, pour $\mathit{FreqRelances} * |\Phi|$ étapes, si I_{RL} ne satisfait pas la formule (lignes 7-8) l'algorithme choisit et flippe une variable (lignes 9-11) et ajoute l'affectation correspondante à I_{SUP} (ligne 12). Si $|\Phi'| < \mathit{LimSUP} * |\Phi|$ l'algorithme applique la simulation de propagation unitaire sur $\Phi'|_{I_{SUP}}$ (ligne 16) et pour chaque clause vide trouvée il génère l'arbre de réfutation (ligne 17) et applique la Max-résolution (lignes 18-19). b et I_b sont mis à jour (lignes 20-22) si le nombre de clauses falsifiées par l'interprétation courante est inférieur au meilleur connu jusqu'à présent.

6 Résultats expérimentaux

Comme nous l'avons décrit précédemment, IRAnovelty++ contient plusieurs paramètres qui peuvent in-

Algorithm 2: IRAnovelty++

Data: Φ, *MaxRelances*, *FreqRelances*, *LimMR* and *LimSUP*.
Result: (b, I_b) : b est le nombre minimum de clauses falsifiées trouvé durant la recherche et I_b l'interprétation correspondante.

```
1  begin
2      I_b ← ∅ ; b ← Σ_{c∈Φ} poids(c);
3      Φ' ← Φ ; I_SUP ← ∅;
4      for j ← 1 to MaxRelances do
5          I_RL ← interprétation_aléatoire();
6          for k ← 1 to FreqRelances do
7              if est_satisfaite(Φ|_{I_RL}) then
8                  return (0, I_RL);
9              c ← choix_clause_falsifiée(Φ|_{I_RL});
10             x ← choix_variable_à_flipper(c);
11             I_RL ← flippe(I_RL, x);
12             I_SUP ← affecte(I_SUP, l);
13             // l est le littéral correspondant à x_i
14             // modulo sa valeur dans I_RL
15             if |Φ| < LimSUP*|Φ'| then
16                 for clause c vide ∈ Φ'|_{I_SUP} * do
17                     R ← arbre_réfutation(I_SUP, c);
18                     if |res(R)| ≤
                       LimMR*min{|c|, c ∈src(R)} then
19                         Φ' ← max_résolution(Φ', R);
20             if Σ_{c∈Φ'} poids(c) < b then
21                 b ← Σ_{c∈Φ'} poids(c);
22                 I_b ← I_RL;
23     return (b, I_b)
```

fluencer ses performances. Dans cette section, nous étudions le comportement du solveur en fonction des valeurs de ses paramètres puis nous donnons une vue d'ensemble des résultats obtenus avec les paramètres optimisés.

Les tests ont été réalisés sur des machines sous système d'exploitation GNU/Linux équipées de deux processeurs Intel Xeon cadencés à 2.4 GHz et de 24 Gb de mémoire vive. Chaque processeur contient 4 coeurs et chaque coeur est dédié au traitement d'une instance. Le benchmark que nous avons utilisé est tiré de ceux des Max-SAT Compétitions de 2011 et 2012. Nous avons choisi 694 instances, 150 instances non-valuées aléatoires Max-2-SAT, 150 non-valuées aléatoires Max-3-SAT, 167 non-valuées crafted (instances issues de problèmes académiques) Max-2-SAT, 80 valuées aléatoires Max-2-SAT, 80 valuées aléatoires Max-3-SAT et 65 valuées crafted Max-2-SAT. Le temps d'exécution est fixé à 900 secondes.

Les valeurs de l'optimum de la plupart des instances ont été obtenues avec le solveur complet maxsatz [19]. Pour les autres, nous avons retenu le meilleur résultat trouvé lors de nos expérimentations.

Le critère de comparaison le plus naturel est le pourcentage d'instances pour lesquelles l'optimum a été trouvé. Quand deux solveurs (ou deux versions d'un même solveur) ont des résultats très proches sur ce premier critère, nous utilisons un critère additionnel : le temps médian mis par le solveur pour atteindre l'optimum la première fois. Lorsque c'est pertinent, nous montrerons l'évolution en focntion des valeurs des paramètres du pourcentage médian de clauses vides générées à partir des sous-ensembles inconsistants détectés par SUP.

6.1 Optimisation d'AdaptNovelty++

Nous avons tout d'abord étudié les performances d'AdaptNovelty++ en fonction de la fréquence de relance (exprimée comme un facteur de la taille de la formule originale). Le pourcentage d'optimums trouvés (fig. 2, graphique *(a)*) est légèrement plus élevé avec de faibles valeurs de fréquence de relance (entre une et dix fois le nombre de clauses de la formule originale) et il décroît doucement lorsque cette valeur augmente. Le temps médian pour trouver la première fois l'optimum (fig. 2, graphique *(b)*) confirme cette observation.

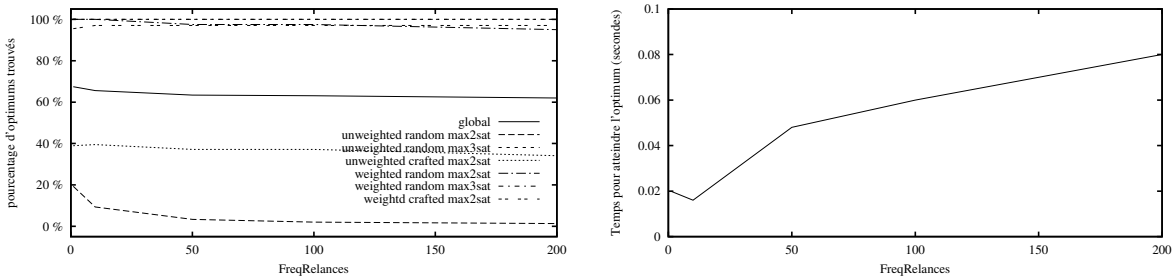
Nous n'avons pas cherché à optimiser les paramètres relatifs au mécanisme d'adaptation des probabilités de diversification, ni pour AdaptNovelty++ ni pour IRAnovelty++.

6.2 Optimisation d'IRAnovelty++

Nous avons étudié individuellement chaque paramètre d'IRAnovelty++ : celui relatif au mécanisme de diversification, *FreqRelances*, et ceux qui contrôlent l'application de la règle d'inférence sur la formule, *LimSUP* and *LimMR*.

FreqRelances (fig. 3, graphiques *(a)* et *(b)*) Comme pour AdaptNovelty++, le meilleur pourcentage d'optimums trouvés est obtenu avec une fréquence de relance faible (entre 5 et 25 fois la taille de la formule originale). Le temps médian pour trouver la première fois l'optimum confirme cette observation et permet de l'affiner, en situant la meilleur valeur autour de dix. Enfin, il est intéressant de noter que la fréquence de relance n'a pas d'impact sur la taille de la formule après transformation ni sur le nombre de clauses vides trouvées par l'algorithme.

LimSUP (fig. 3, graphiques *(c)*, *(d)* et *(e)*) Ce paramètre définit la taille maximale possible de la formule transformée, exprimée en un ratio de la taille de la formule originale. Quand la formule atteint ou dépasse cette taille, la simulation de propagation unitaire n'est plus appliquée et la formule reste inchangée. Avec des valeurs inférieures à 1, la règle d'inférence



(a) Pourcentage d'optimums trouvés

(b) Temps médian pour atteindre l'optimum

FIGURE 2 – Impact de la fréquence de relance sur les performances d'AdaptNovelty++.

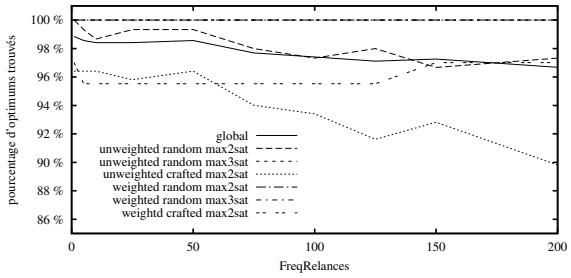
n'est jamais appliquée et IRAnovelty++ agit comme AdaptNovelty++. En augmentant sa valeur, le pourcentage d'optimum trouvé augmente jusqu'à devenir stable autour de 98,5% avec des valeurs supérieures à 4. Le meilleur temps médian pour trouver l'optimum est obtenu avec la valeur 8. Des valeurs plus grandes donnent des performances légèrement moins bonnes. Nous voyons deux raisons possibles expliquant ce comportement : (1) trop de temps est passé à transformer la formule et/ou (2) la taille plus importante de la formule transformée ralentit la recherche locale. On peut noter qu'aucune clause vide n'est trouvée sur les instances max-3-sat (valuées et non-valuées).

LimMR (fig. 3, graphiques *(f)*, *(g)* et *(h)*) Comme le précédent, ce paramètre contrôle l'application de la règle d'inférence sur la formule. Il limite la production de clauses résolvantes de tailles supérieures à un certain ratio de la plus petite des clauses en entrée. Pour un conflit donné, si la taille du résolvant est supérieure à *LimMR* fois celle de la plus petite clause de l'arbre de réfutation, alors la règle d'inférence n'est pas appliquée et le conflit est simplement ignoré. Comme nous l'avons expliqué précédemment, cette limite a deux objectifs : limiter la taille de la formule transformée et favoriser la production de clauses vides. Avec la valeur 1 la taille de la formule n'augmente quasiment pas et l'algorithme détecte 60% des clauses vides mais il ne trouve l'optimum que sur moins de 70% des instances. Cela peut s'expliquer par le fait que *LimSUP* n'est jamais atteinte, donc SUP est appliquée durant toute la résolution et ralentit la partie recherche locale de l'algorithme. Avec des valeurs supérieures à 1, IRAnovelty++ trouve l'optimum sur plus de 95% des instances et détecte environ 40% des clauses vides. Les meilleurs temps pour atteindre la première fois l'optimum sont obtenus avec des valeurs de *LimMR* comprises entre 1,5 et 2. Là encore, aucune clause vide n'est générée sur les instances Max-3-SAT et ce même avec de petites valeurs de *LimMR*.

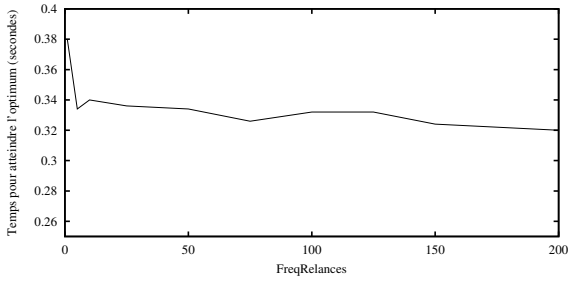
6.3 Analyse des résultats

La Table 1 compare les performances d'AdaptNovelty++ et d'IRAnovelty++ avec leurs paramètres optimisés (pour AdaptNovelty++ $FreqRelances = 1$ et pour IRAnovelty++ $FreqRelances = 10$, $LimSUP = 5$ and $LimMR = 1.5$). Pour chaque solveur, la colonne $\%O$ donne le pourcentage d'instances pour lesquelles l'optimum a été trouvé et ftO donne le temps médian pour atteindre l'optimum la première fois. Nous ne comparons pas notre solveur à l'état de l'art des solveurs incomplets pour Max-SAT. Notre objectif ici est de montrer que les règles d'inférences permettent d'améliorer un algorithme de recherche locale, pas de développer un solveur concurrentiel. D'autant que les résultats obtenus dépendent nécessairement des performances de l'algorithme RL original, qui dans notre cas n'est pas spécifiquement conçu pour Max-SAT. Dans de futurs travaux, nous chercherons à étendre les résultats obtenus ici à d'autres algorithmes incomplets spécifiquement conçus pour Max-SAT et nous conduirons alors une étude expérimentale plus poussée.

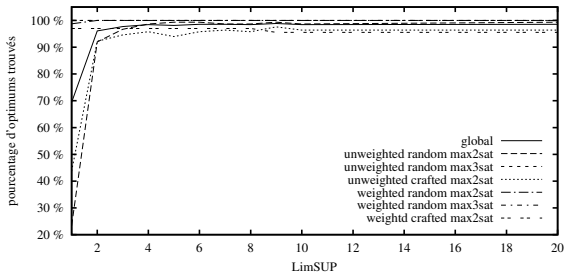
On voit sur la Table 1 qu'IRAnovelty++ avec ses paramètres optimisés trouve l'optimum sur plus de 99% des instances du benchmark, tandis qu'AdaptNovelty++ ne trouve l'optimum que sur moins de 70% d'entre elles. Les résultats sont particulièrement flagrant sur les instances sur lesquelles AdaptNovelty++ est peu performante, les non valuées Max-2-SAT (aléatoires et crafted). Sur les autres classes d'instances, AdaptNovelty++ et IRAnovelty++ présentent des taux d'optimum trouvés similaires, proches des 100%. On peut noter qu'à l'exception des deux classes d'instances citées précédemment, AdaptNovelty++ a un temps médian pour atteindre l'optimum plus faible que celui d'IRAnovelty++. Cela peut s'expliquer par deux faits : (1) l'application de la règle d'inférence est coûteuse en temps, particulièrement au



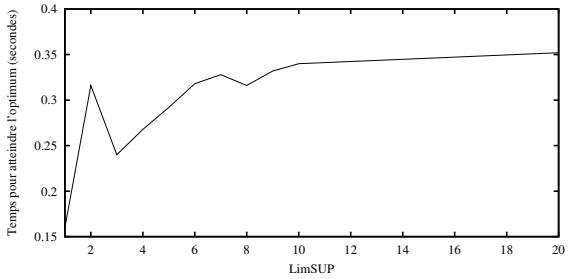
(a) *FreqRelances* - Pourcentage d'optimums trouvés



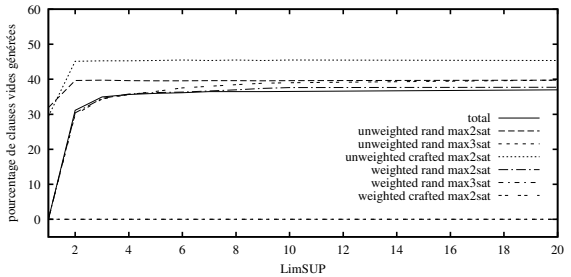
(b) *FreqRelances* - Temps médian pour atteindre l'optimum



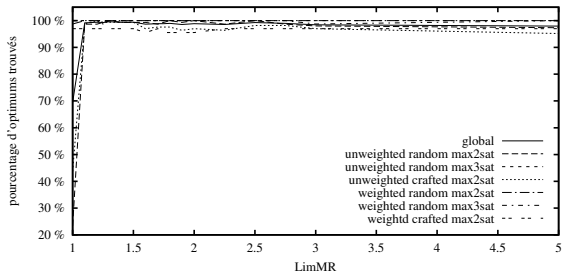
(c) *LimSUP* - Pourcentage d'optimums trouvés



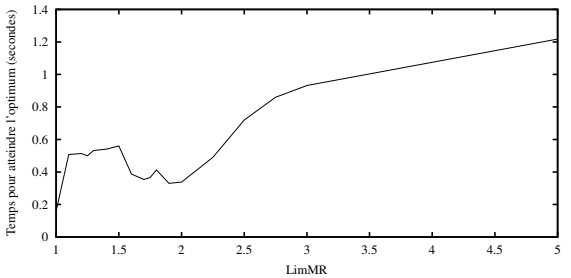
(d) *LimSUP* - Temps médian pour atteindre l'optimum



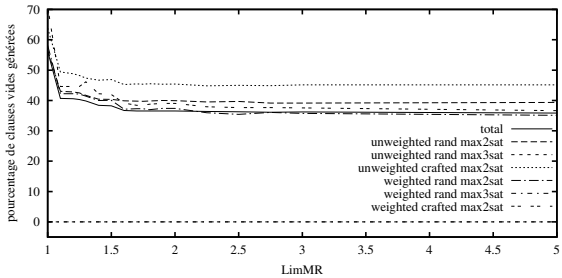
(e) *LimSUP* - Pourcentage médian de clauses vides générées



(f) *LimMR* - Pourcentage d'optimums trouvés



(g) *LimMR* - Temps médian pour atteindre l'optimum



(h) *LimMR* - Pourcentage médian de clauses vides générées

FIGURE 3 – Impact de la valeur des paramètres d'IRAnovelty++ sur ses performances.

	AdaptNovelty++		IRAnovelty++	
Instances	**%O**	**ftO**	**%O**	**ftO**
aléatoires non-valuées Max-2-SAT	19.33	129.41	100	0.66
aléatoires non-valuées Max-3-SAT	100	< 0.01	100	0.09
crafted non-valuées Max-2-SAT	38.92	0.11	99.4	0.08
aléatoires valuées Max-2-SAT	100	0.07	100	0.31
aléatoires valuées Max-3-SAT	100	0.04	100	0.1
crafted valuées Max-2-SAT	95.52	0.01	97.01	0.47
Total	67.44	0.02	99.57	0.16

TABLE 1 – Comparaison des performances d'AdaptNovelty++ et d'IRAnovelty++.

début de la recherche (sur la plupart des instances, *LimSUP* est atteinte dans les toutes premières secondes de la résolution) et donc peut retarder la découverte de l'optimum et (2) la taille plus grande de la formule transformée ralentit la recherche locale et donc la découverte de l'optimum.

Même si les résultats présentés ne le font pas apparaître clairement, nous pensons que notre algorithme n'améliore pas les performances sur les instances Max-3-SAT. Nous supposons que la génération de clauses vides permet de simplifier la résolution des instances. Force est de constater que sur les instances Max-3-SAT, aucune n'est trouvée. Cela peut s'expliquer par la manière dont la règle d'inférence est appliquée. À part dans certains cas particuliers (quand plusieurs clauses de l'arbre de réfutation contiennent des littéraux communs), les clauses de compensations et les résolvants produits par la règle d'inférence sont de tailles égales (avec des clauses originales de tailles 2) ou supérieures (avec des clauses originales de tailles > 2) à celles des clauses originales. Sur les instances Max-2-SAT, le cas particulier présenté ci-dessus est suffisamment fréquent pour permettre la production de clauses unitaires et vides. Ce n'est visiblement pas le cas sur les instances Max-3-SAT et en conséquence IRAnovelty++ passe du temps à transformer la formule, augmentant ainsi sa taille, sans pour autant la simplifier.

Enfin, il est intéressant de noter que sur certaines instances IRAnovelty++ trouve toutes les clauses vides possibles. Autrement dit, la formule transformée est de la forme $\Phi = \{(\square, o)\} \cup \Phi'$, avec Φ' une formule satisfiable et o la somme des poids des clauses vides. Dans cette situation, o est l'optimum de l'instance [6]. IRAnovelty++ peut donc parfois prouver l'optimalité comme le font les solveurs Max-SAT complet. Cependant, ce cas est rare (environ 1% des instances) et ne concerne que des instances avec de petites valeurs d'optimums (c'est-à-dire dont le nombre minimum de clauses falsifiées est faible).

7 Conclusion

Nous avons présenté dans cet article un nouvel algorithme, IRAnovelty++, qui intègre une règle d'inférence dans un solveur de recherche locale classique pour résoudre le problème Max-SAT. Les tests que nous avons effectués donnent des informations utiles sur le comportement de notre algorithme et prouvent sa robustesse sur un grand nombre d'instances. Les résultats montrent que les performances sont nettement améliorées par rapport à celles de l'algorithme original, ce qui confirme l'intérêt de notre approche.

Dans le futur, nous étendrons notre algorithme au problème partial Max-SAT et nous étudierons plus en détail sont comportement sur les instances Max-3-SAT, avec en perspective l'objectif de permettre la génération de clauses vides sur ces instances, par exemple en guidant l'application de la règle d'inférence par des heuristiques dédiées. Nous essaierons d'étendre les résultats obtenues ici en appliquant la règle d'inférence à d'autres algorithmes de recherche locale, et en particulier à ceux conçus pour le problème Max-SAT.

Enfin, les travaux récents sur le problème Max-SAT ont majoritairement porté sur sa résolution par des méthodes complètes, tandis que moins d'intérêt a été porté sur les méthodes incomplètes. Notre travail à aussi pour but de participer au développement de ces approches.

Références

[1] A. ABRAME et D. HABET : Inference rules in local search for max-sat. *In 24th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2012)*, pages 207 – 214, Athens, Greece, 07 november 2012.

[2] Teresa ALSINET, Felip MANYÀ et Jordi PLANES : A max-sat solver with lazy data structures. *In* Christian LEMAÎTRE, Carlos REYES et Jesùs GONZÁLEZ, éditeurs : *Advances in Artificial Intelligence - IBERAMIA'04*, volume 3315 de *LNCS*, pages 334–342. Springer Berlin / Heidelberg, 2004.

[3] Gilles AUDEMARD, Jean-Marie LAGNIEZ, Bertrand MAZURE et Lakhdar SAIS : Learning in local search. *In 21st IEEE International Conference on Tools with Artificial Intelligence - ICTAI 2009*, pages 417–424. IEEE Computer Society, 2009.

[4] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure et Lakhdar Saïs : Boosting local search thanks to cdcl. *In* Christian Fermüller et Andrei Voronkov, éditeurs : *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 6397 de *LNCS*, pages 474–488. Springer Berlin / Heidelberg, 2010.

[5] A. Biere, M. Heule, H. van Maaren et T. Walsh, éditeurs. *Handbook of Satisfiability*, volume 185 de *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[6] María Luisa Bonet, Jordi Levy et Felip Manyà : Resolution for max-sat. *Artificial Intelligence*, 171(8-9):606–618, 2007.

[7] Oliver Gableske et Marijn Heule : Eagleup : Solving random 3-sat using sls with unit propagation. *In* Karem Sakallah et Laurent Simon, éditeurs : *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 de *LNCS*, pages 367–368. Springer Berlin / Heidelberg, 2011.

[8] Carla Gomes, Willem-Jan van Hoeve et Lucian Leahu : The power of semidefinite programming relaxations for max-sat. *In* J. Beck et Barbara Smith, éditeurs : *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CPAIOR'06*, volume 3990 de *LNCS*, pages 104–118. Springer Berlin / Heidelberg, 2006.

[9] Pierre Hansen et Brigitte Jaumard : Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.

[10] Federico Heras et David Bañeres : The impact of max-sat resolution-based preprocessors on local search solvers. *JSAT*, 7(2-3):89–126, 2010.

[11] Federico Heras et 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.

[12] Federico Heras, Javier Larrosa et Albert Oliveras : Minimaxsat : An efficient weighted max-sat solver. *Journal of Artificial Intelligence Research - JAIR*, 31:1–32, 2008.

[13] Edward A. Hirsch et Arist Kojevnikov : Unitwalk : A new sat solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43:91–111, 2005.

[14] H. Hoos et T. Stutzle : *Stochastic Local Search : Foundations and Applications*. Morgan Kaufmann Publishers Inc., 2004.

[15] H. H. Hoos : An adaptive noise mechanism for walksat. *In Proceedings of the 18th National Conference on Artificial Intelligence - AAAI'02*, pages 655–660, 2002.

[16] Lukas Kroc, Ashish Sabharwal, Carla P. Gomes et Bart Selman : Integrating systematic and local search paradigms : A new strategy for maxsat. *In* Craig Boutilier, éditeur : *Proceedings of the 21st International Joint Conference on Artificial Intelligence - IJCAI 2009*, pages 544–551, 2009.

[17] Javier Larrosa et Federico Heras : Resolution in max-sat and its relation to local consistency in weighted csps. *In* Leslie Pack Kaelbling et Alessandro Saffiotti, éditeurs : *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence - IJCAI'05*, pages 193–198. Morgan Kaufmann Publishers Inc., 2005.

[18] Chu Li et Wen Huang : Diversification and determinism in local search for satisfiability. *In* Fahiem Bacchus et Toby Walsh, éditeurs : *Theory and Applications of Satisfiability Testing - SAT'05*, volume 3569 de *LNCS*, pages 158–172. Springer Berlin / Heidelberg, 2005.

[19] Chu Min Li, Felip Manyà, Nouredine Mohamedou et Jordi Planes : Exploiting cycle structures in max-sat. *In* Oliver Kullmann, éditeur : *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 de *LNCS*, pages 467–480. Springer Berlin / Heidelberg, 2009.

[20] Xiao Li, Matthias Stallmann et Franc Brglez : A local search sat solver using an effective switching strategy and an efficient unit propagation. *In* Enrico Giunchiglia et Armando Tacchella, éditeurs : *Theory and Applications of Satisfiability Testing*, volume 2919 de *LNCS*, pages 276–277. Springer Berlin / Heidelberg, 2004.

[21] Mark Liffiton et Karem Sakallah : On finding all minimally unsatisfiable subformulas. *In* Fahiem Bacchus et Toby Walsh, éditeurs : *Theory and Applications of Satisfiability Testing - SAT 2005*, volume 3569 de *LNCS*, pages 34–42. Springer Berlin / Heidelberg, 2005.

[22] David A. McAllester, Bart Selman et Henry A. Kautz : Evidence for invariants in local search. *In* Benjamin Kuipers et Bonnie L. Webber, éditeurs : *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference - AAAI'97/IAAI'97*, pages 321–326. AAAI Press / The MIT Press, 1997.

[23] Christos H. Papadimitriou : *Computational complexity*. Addison-Wesley, 1994.

[24] J. A. Robinson : A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[25] Bart Selman, Henry A. Kautz et Bram Cohen : Noise strategies for improving local search. *In* Barbara Hayes-Roth et Richard E. Korf, éditeurs : *Proceedings of the 12th National Conference on Artificial Intelligence - AAAI'94*, volume 1, pages 337–343. AAAI Press / The MIT Press, 1994.

[26] Kevin Smyth, Holger Hoos et Thomas Stützle : Iterated robust tabu search for max-sat. *In* Yang Xiang et Brahim Chaib-draa, éditeurs : *Advances in Artificial Intelligence*, volume 2671 de *LNCS*, pages 995–995. Springer Berlin / Heidelberg, 2003.

[27] T. Stützle, H. Hoos et A. Roli : A review of the literature on local search algorithms for max-sat. Rapport technique AIDA-01-02, Intellectics Group, Darmstadt University of Technology, Germany, 2001.

[28] Hans van Maaren et Linda van Norden : Sums of squares, satisfiability and maximum satisfiability. *In* Fahiem Bacchus et Toby Walsh, éditeurs : *Theory and Applications of Satisfiability Testing - SAT'05*, volume 3569 de *LNCS*, pages 294–308. Springer Berlin / Heidelberg, 2005.

Détection des fonctions de rang linéaires à terme

Anthony Alezan[1] **Roberto Bagnara**[2] **Fred Mesnard**[1] **Étienne Payet**[1]

[1] LIM/ERIMIA, université de la Réunion, France

[2] BUGSENG & Dipartimento di Matematica e Informatica, Università di Parma, Italie

{anthony.alezan, fred, epayet}@univ-reunion.fr bagnara@cs.unipr.it

Résumé

La terminaison des programmes est un sujet actif de recherche en informatique. Ces dernières années ont vu l'apparition d'analyseurs de terminaison performants pour des langages comme C ou Java où l'emploi des techniques et outils de la programmation par contraintes est omniprésent. Dans cet article, nous rappelons un algorithme particulier basé sur l'emploi du lemme de Farkas pour le calcul de fonctions de rang linéaires garantissant la terminaison d'une certaine classe de boucles. Puis nous présentons une extension de cette méthode pour la découverte de fonctions linéaires qui deviennent des fonctions de rang linéaire à terme, c.-à-d. après un certain nombre de passages dans la boucle. Nous montrons la correction et la complétude d'un algorithme polynomial pour ce problème.

Abstract

Program termination is a hot research topic in program analysis. The last few years have witnessed the development of termination analyzers for programming languages such as C and Java with remarkable precision and performance. These systems are largely based on techniques and tools coming from the field of constraint programming. In this paper, we first recall an algorithm based on Farkas' Lemma for discovering linear ranking functions proving termination of a certain class of loops. Then we propose an extension of this method for showing the existence of *eventual linear ranking functions*, i.e., linear functions that become ranking functions after a finite unrolling of the loop. We show correctness and completeness of this polynomial algorithm.

1 Introduction et préliminaires

La terminaison des programmes est un sujet actif de recherche en informatique. Ces dernières années ont vu l'apparition d'analyseurs de terminaison performants pour des langages comme C [10] et Java [1, 13, 16] où l'emploi des techniques et outils de la programmation par contraintes est omniprésent.

Au delà des particularités des langages de programmation visés et après plusieurs abstractions (voir par exemple [16]), l'analyse de terminaison se ramène à la preuve de terminaison de boucles, dont les boucles dites *SLC* sont peut-être la catégorie la plus simple. Précisons ce type de boucles.

Une boucle SLC (pour *single-path linear constraint* [3]) sur n variables x_1, ..., x_n est une boucle de la forme :

$$\textbf{tant que } (B\ \mathbf{x} \leq \mathbf{b}) \quad \textbf{faire } A \begin{pmatrix} \mathbf{x} \\ \mathbf{x}' \end{pmatrix} \leq \mathbf{c}$$

où $\mathbf{x} = (x_1, \ldots, x_n)^T$ et $\mathbf{x}' = (x'_1, \ldots, x'_n)^T$ sont des vecteurs colonnes de variables, B est une matrice de $\mathbb{Z}^{p \times n}$, $\mathbf{b} \in \mathbb{Z}^p$, $A \in \mathbb{Z}^{q \times 2n}$ et $\mathbf{c} \in \mathbb{Z}^q$.

Lorsque les variables sont à valeurs dans $\mathbb{Z}$ (resp. $\mathbb{Q}$), on parle de boucle *entière* (resp. *rationnelle*). Opérationnellement, de telles boucles modélisent le calcul suivant : on part d'un point $\mathbf{x}$ quelconque ; si le test $B\ \mathbf{x} \leq \mathbf{b}$ est faux, on sort de la boucle ; si le test est vrai, on choisit un nouveau point $\mathbf{x}'$ satisfaisant le corps de la boucle et on réitère en remplaçant les valeurs de $\mathbf{x}$ par celles de $\mathbf{x}'$.

Une boucle SLC peut être notée, et c'est ce que nous ferons par la suite, sous la forme d'une clause de programmation logique avec contraintes :

$$p(\mathbf{x}) \leftarrow B\mathbf{x} \leq \mathbf{b},\ A \begin{pmatrix} \mathbf{x} \\ \mathbf{x}' \end{pmatrix} \leq \mathbf{c},\ p(\mathbf{x}').$$

La terminaison des boucles en général peut toujours être établie en exhibant une *fonction de rang* ρ, c.-à-d. une fonction de $\mathbb{Z}^n$ ou $\mathbb{Q}^n$ suivant le cas vers un ensemble bien-fondé telle que à chaque passage dans la boucle la valeur $\rho(\mathbf{x})$ décroît strictement. Comme l'ensemble d'arrivée de ρ est bien-fondé, le calcul termine.

À notre connaissance, la décidabilité de la terminaison universelle des boucles SLC (c.-à-d. quelque soit le point de départ du calcul et quelque soit le choix des points suivants) est une question ouverte dans le cas général. Certains cas particuliers ont été montrés décidables [6, 9, 17]. Diverses généralisations des boucles SLC sont indécidables [5].

Une approche possible pour l'analyse de la terminaison des boucles SLC consiste à restreindre la classe des fonctions de rang et le domaine des variables. Dans la section qui suit, nous présentons les résultats connus pour les fonctions de rang linéaires dans le cas $\mathbb{Q}$. Puis nous étudions les fonctions de rang linéaires à terme : ce sont les fonctions linéaires qui, après un certain nombre de passages dans la boucle, deviennent des fonctions de rang. Nous proposons un algorithme polynomial correct et complet pour cette classe de fonctions de rang, qui constitue la contribution de cet article à l'état de l'art. Enfin, nous concluons.

2 Les fonctions de rang linéaires

Précisons tout d'abord ce que nous entendons par fonction de rang linéaire.

Définition 2.1 *Soit C la boucle SLC rationnelle, exprimée sous forme clausale, $p(\mathbf{x}) \leftarrow c(\mathbf{x}, \mathbf{x}'),\ p(\mathbf{x}')$, où p est une relation n-aire. Une* fonction de rang linéaire *ρ pour C est une application linéaire de $\mathbb{Q}^n$ vers $\mathbb{Q}$ vérifiant :*

$$\forall \mathbf{x}, \mathbf{x}' : c(\mathbf{x}, \mathbf{x}') \implies \begin{cases} \rho(\mathbf{x}) \geq 1 + \rho(\mathbf{x}'), \\ \rho(\mathbf{x}) \geq 0. \end{cases}$$

Autrement dit, ρ reste toujours positive et décroît d'au moins une unité[1] à chaque passage dans la boucle. Remarquons que si $c(\mathbf{x}, \mathbf{x}')$ est insatisfiable, la boucle termine immédiatement et toute fonction linéaire est une fonction de rang. Dans ce qui suit, nous supposerons $c(\mathbf{x}, \mathbf{x}')$ satisfiable.

Nous pouvons généraliser en considérant les fonctions de rang affines, mais cette classe de fonctions de rang peut toujours se ramener au cas linéaire. En effet, une fonction de rang affine pour la clause $p(\mathbf{x}) \leftarrow c(\mathbf{x}, \mathbf{x}'),\ p(\mathbf{x}')$ est une fonction de rang linéaire pour $p(\mathbf{x}, y) \leftarrow c(\mathbf{x}, \mathbf{x}'), y = 1, y' = 1,\ p(\mathbf{x}', y')$ où y et y' sont des nouvelles variables distinctes de $\mathbf{x}$ et $\mathbf{x}'$. En conséquence, si on dispose d'une méthode correcte et complète pour les fonctions de rang linéaires, il en est de même pour les fonctions de rang affines. Nous nous focalisons donc sur les fonctions de rang linéaires pour les boucles SLC rationnelles et, après avoir précisé une formulation du lemme de Farkas, nous considérons la question de la vérification et de la détection.

[1] Tout rationnel strictement positif fixé une fois pour toutes convient.

2.1 Lemme de Farkas

Dans les rationnels, une inéquation linéaire I est une conséquence logique d'une conjonction S finie satisfiable d'inéquations linéaires quand I est une combinaison linéaire positive des inéquations de S. Formellement, posons S :

$$\begin{cases} a_{1,1}x_1 & + \cdots & + a_{1,n}x_n & + b_1 & \geq 0 \\ \cdots & + \cdots & + \cdots & + \cdots & \geq 0 \\ a_{m,1}x_1 & + \cdots & + a_{m,n}x_n & + b_m & \geq 0. \end{cases}$$

Nous supposons que S admet au moins une solution. Le lemme de Farkas établit l'équivalence entre :

$$\forall x_1, \ldots, x_n : S \implies (c_1x_1 + \cdots + c_nx_n + d \geq 0)$$

et

$$\exists \lambda_1 \geq 0, \ldots, \lambda_m \geq 0\ .\ \begin{cases} c_1 = \sum_{i=1}^{m} \lambda_i a_{i,1} \\ \quad \vdots \\ c_n = \sum_{i=1}^{m} \lambda_i a_{i,n} \\ d \geq \sum_{i=1}^{m} \lambda_i b_i. \end{cases}$$

2.2 Vérification

Si on se donne une boucle SLC et une fonction ρ linéaire, on peut aisément vérifier si ρ est une fonction de rang en s'assurant du caractère insatisfiable des formules $c(\mathbf{x}, \mathbf{x}'), \rho(\mathbf{x}) < 1 + \rho(\mathbf{x}')$ et $c(\mathbf{x}, \mathbf{x}'), \rho(\mathbf{x}) < 0$. Cette vérification peut être déléguée à un solveur complet comme CLP($\mathbb{Q}$) [12]. En théorie, ce problème est polynomial.

Exemple 2.2 *Pour la boucle C :*

$$p(x, y) \leftarrow x \geq 0, y' \leq y - 1, x' \leq x + y, y \leq -1, p(x', y')$$

la fonction $\rho(x, y) = x$ est une fonction de rang linéaire, comme le montre la session Prolog ci-dessous.

```
?- use_module(library(clpq)).
% library(clpq) compiled
true.
?- {X >= 0, Y1 =< Y - 1, X1 =< X + Y, Y =< -1,
    X < 1 + X1}.
false.
?- {X >= 0, Y1 =< Y - 1, X1 =< X + Y, Y =< -1,
    X < 0}.
false.
```

2.3 Détection

Cette fois, on se donne une boucle SLC et on se demande s'il existe une fonction de rang linéaire ρ. Ce problème a été largement étudié [2, 14, 15] : il est décidable et de complexité polynomiale.

Sur l'exemple 2.2, nous explicitons un des algorithmes connus [14]. En posant $\rho(x, y) = ax + by$, exprimons formellement la question : existe-t-il une fonction de rang linéaire pour la boucle C ?

$$\exists a, b \,.\, \forall x, y, x', y' : \left.\begin{array}{r} x \geq 0 \\ y' \leq y - 1 \\ x' \leq x + y \\ y \leq -1 \end{array}\right\} \implies \left\{\begin{array}{l} ax + by \geq 1 + ax' + by' \\ ax + by \geq 0. \end{array}\right. \quad (1)$$

Notons que cette définition du problème est *a priori* exécutable par élimination des quantificateurs sur un système de calcul formel comme Reduce [11] :

```
1: load_package redlog;
2: rlset r;
3: F:=ex({a,b},all({x,y,x1,y1},
  (x>=0 and y1<=y-1 and x1<=x+y and y<= -1)
  impl
  (a*x+b*y>=1+a*x1+b*y1 and a*x+b*y>=0)));
4: rlqe F;
```

La commande 1 charge le module d'élimination des quantificateurs. La commande 2 définit $\mathbb{R}$ comme domaine de discours. La commande 3 initialise la formule F. La commande 4 lance l'élimination des quantificateurs de la formule F et retourne une formule équivalente, ici `true`. La formule F est donc vraie : il existe au moins une fonction de rang linéaire. Déterminons les coefficients de la fonction ρ :

```
5: G:=all({x,y,x1,y1},
  (x>=0 and y1<=y-1 and x1<=x+y and y<= -1)
  impl
  (a*x+b*y>=1+a*x1+b*y1 and a*x+b*y>=0));
6: rlqe G;
```

On obtient :

$$\begin{array}{l} a^2 - ab \geq 0 \wedge a - b \neq 0 \wedge a > 0 \wedge b = 0 \\ \qquad \wedge (a^2 b - ab^2 \leq 0 \vee a^2 - ab = 0 \\ \qquad\qquad \vee a^2 - 2ab - a + b^2 + b \geq 0) \\ \qquad \wedge (a^2 - ab = 0 \vee a^2 - 2ab - a + b^2 b \geq 0). \end{array}$$

Donc tous les a et b vérifiant la formule ci-dessus conviennent. On vérifie que $a = 1, b = 0$ est bien une solution. Néanmoins, *a fortiori*, la complexité des algorithmes mis en jeu ne permettra pas systématiquement l'obtention d'une réponse avec des ressources raisonnables.

En revanche, en considérant a et b comme des *paramètres* du problème, nous pouvons appliquer le lemme de Farkas. Pour la condition de décroissance de la fonction de rang :

$$\forall x, y, x', y' : \left.\begin{array}{r} x \geq 0 \\ y' \leq y - 1 \\ x' \leq x + y \\ y \leq -1 \end{array}\right\} \implies ax + by \geq 1 + ax' + by'$$

en faisant apparaître les coefficients λ_i sur la colonne de gauche de façon à faciliter son emploi, le lemme de Farkas s'écrit :

$$\begin{array}{ll} \lambda_1 : & 1x + 0y + 0x' + 0y' + 0 \geq 0 \\ \lambda_2 : & 1x + 1y - 1x' + 0y' + 0 \geq 0 \\ \lambda_3 : & 0x + 1y + 0x' - 1y' - 1 \geq 0 \\ \lambda_4 : & 0x - 1y + 0x' + 0y' - 1 \geq 0 \\ \implies & \\ & ax + by - ax' - by' - 1 \geq 0. \end{array}$$

La condition de décroissance est donc équivalente à l'existence de quatre rationnels positifs ou nuls λ_1, …, λ_4 tels que :

$$\left\{\begin{array}{r} a = \lambda_1 + \lambda_2 \\ b = \lambda_2 + \lambda_3 - \lambda_4 \\ -a = -\lambda_2 \\ -b = -\lambda_3 \\ -1 \geq -\lambda_3 - \lambda_4. \end{array}\right. \quad (2)$$

Pour la condition de positivité de la fonction de rang :

$$\forall x, y, x', y' : \left.\begin{array}{r} x \geq 0 \\ y' \leq y - 1 \\ x' \leq x + y \\ y \leq -1 \end{array}\right\} \implies ax + by \geq 0$$

le lemme de Farkas s'écrit :

$$\begin{array}{ll} \lambda'_1 : & 1x + 0y + 0x' + 0y' + 0 \geq 0 \\ \lambda'_2 : & 1x + 1y - 1x' + 0y' + 0 \geq 0 \\ \lambda'_3 : & 0x + 1y + 0x' - 1y' - 1 \geq 0 \\ \lambda'_4 : & 0x - 1y + 0x' + 0y' - 1 \geq 0 \\ \implies & \\ & ax + by + 0x' + 0y' + 0 \geq 0. \end{array}$$

La condition de positivité est cette fois équivalente à l'existence de quatre autres rationnels positifs ou nuls

$\lambda'_1, \ldots, \lambda'_4$ tels que :

$$\begin{cases} a = \lambda'_1 + \lambda'_2 \\ b = \lambda'_2 + \lambda'_3 - \lambda'_4 \\ 0 = -\lambda'_2 \\ 0 = -\lambda'_3 \\ 0 \geq -\lambda'_3 - \lambda'_4. \end{cases} \quad (3)$$

En résumé, via le lemme de Farkas, la formule (1) est équivalente à la conjonction des formules (2) et (3) :

$$\exists a, b \,.\, \exists \lambda_1 \geq 0, \ldots, \lambda_4 \geq 0, \lambda'_1 \geq 0, \ldots, \lambda'_4 \geq 0 \,.\, \begin{cases} a = \lambda_1 + \lambda_2 \\ b = \lambda_2 + \lambda_3 - \lambda_4 \\ -a = -\lambda_2 \\ -b = -\lambda_3 \\ -1 \geq -\lambda_3 - \lambda_4 \\ a = \lambda'_1 + \lambda'_2 \\ b = \lambda'_2 + \lambda'_3 - \lambda'_4 \\ 0 = -\lambda'_2 \\ 0 = -\lambda'_3 \\ 0 \geq -\lambda'_3 - \lambda'_4. \end{cases}$$

En théorie, le problème de l'existence d'une fonction de rang linéaire est polynomial, ainsi que le calcul d'une fonction témoin, *un certificat* de terminaison, puisque le calcul d'une solution quelconque donne des valeurs à a et b qui répondent au problème. Si on s'intéresse à l'ensemble des fonctions de rang linéaires, c.-à-d. aux conditions sur a et b, il suffit d'éliminer les λ_i et λ'_i du système précédent par exemple en utilisant l'algorithme de Fourier-Motzkin. En voici une illustration. En compilant ce programme :

```
fm(A, B) :-
   {L1 >= 0, L2 >= 0, L3 >= 0, L4 >= 0,
    LP1 >= 0, LP2 >= 0, LP3 >= 0, LP4 >= 0,
    A = L1 + L2,
    B = L2 + L3 - L4,
    A = L2,
    B = L3,
    -1 >= -L3 - L4,
    A = LP1 + LP2,
    B = LP2 + LP3 - LP4,
    0 = LP2,
    0 = LP3,
    0 >= -LP3 - LP4}.
```

puis en interrogeant Prolog, on obtient la condition :

```
| ?- fm(A, B).
B = 0, {A >= 1}.
| ?-
```

qu'on peut montrer équivalente à la condition générée par Reduce, une fois simplifiée.

3 Les fonctions de rang linéaires à terme

Dans la section précédente, nous avons exposé une méthode qui permet de décider de l'existence d'une fonction de rang linéaire pour une boucle SLC rationnelle. Il est bien sûr possible que la méthode échoue.

Exemple 3.1 *La boucle :*

$$p(x, y) \leftarrow x \geq 0, y' \leq y - 1, x' \leq x + y, p(x', y')$$

n'admet pas de fonction de rang linéaire.

Peut-on en conclure que cette boucle ne termine pas ? Non, car il existe peut-être une fonction de rang non-linéaire. Dans cette section, nous allons étendre la méthode précédente pour la détection de fonctions de rang linéaires à terme, c.-à-d. des fonctions linéaires qui deviennent des fonctions de rang *après un nombre fini de passages dans le corps de la boucle*. Pour modéliser ce point, nous allons supposer que la boucle SLC considérée est donnée avec une fonction linéaire $f(x, y)$ dont la valeur croît strictement à chaque passage dans le corps de la boucle.

Exemple 3.2 *La fonction $f(x, y) = -y$ croît strictement pour la boucle de l'exemple 3.1 puisque y', la nouvelle valeur de y, est nécessairement inférieure d'au moins une unité à celle de y.*

Définition 3.3 *Soient C la boucle SLC rationnelle, sous forme clausale : $p(\mathbf{x}) \leftarrow c(\mathbf{x}, \mathbf{x}')$, $p(\mathbf{x}')$, où p est une relation n-aire, et $f(\mathbf{x})$ une fonction linéaire vérifiant $\forall \mathbf{x}, \mathbf{x}' : c(\mathbf{x}, \mathbf{x}') \implies f(\mathbf{x}') \geq 1 + f(\mathbf{x})$. Une* fonction de rang linéaire à terme *ρ pour (C, f) est une application linéaire de $\mathbb{Q}^n$ vers $\mathbb{Q}$ vérifiant :*

$$\exists k \,.\, \forall \mathbf{x}, \mathbf{x}' : \left\{ \begin{array}{l} c(\mathbf{x}, \mathbf{x}') \\ f(\mathbf{x}) \geq k \end{array} \right\} \implies \begin{cases} \rho(\mathbf{x}) \geq 1 + \rho(\mathbf{x}') \\ \rho(\mathbf{x}) \geq 0. \end{cases}$$

Relativement à la définition 2.1, le seuil k est quantifié existentiellement et on impose $f(\mathbf{x}) \geq k$ dans la partie gauche de l'implication.

Notons que si nous déterminons un tel rationnel k, alors tout $k' \geq k$ vérifie également la définition 3.3. D'autre part, comme par hypothèse la valeur de f croît d'une unité à chaque passage dans la boucle, si f est bornée supérieurement pour la contrainte c, la boucle termine. Sinon, après un nombre fini de passages dans la boucle, la valeur de f dépasse le seuil k, ρ devient alors une fonction de rang linéaire au sens de la section 2 et la boucle termine. Nous nous intéressons à présent au problème de la détection puis de la vérification des fonctions de rang linéaires à terme.

3.1 Détection

Considérons la boucle de l'exemple 3.1 associée à la fonction croissante de l'exemple 3.2. En posant $\rho(x,y) = ax + by$, ρ est une fonction de rang linéaire à terme quand :

$$\exists a,b,k \,.\, \forall x,y,x',y' : \quad \left.\begin{array}{l} x \geq 0 \\ y' \leq y - 1 \\ x' \leq x + y \\ -y \geq k \end{array}\right\} \implies \begin{cases} ax + by \geq 1 + ax' + by' \\ ax + by \geq 0. \end{cases}$$

Cette définition du problème, que nous noterons $\exists a,b,k \,.\, \phi(a,b,k)$, est aussi exécutable par élimination des quantificateurs, donc le problème est décidable. En considérant a, b et k comme des paramètres, appliquons le lemme de Farkas :

$$\begin{array}{lrcl} \lambda_1 : & 1x + 0y + 0x' + 0y' + 0 & \geq & 0 \\ \lambda_2 : & 1x + 1y - 1x' + 0y' + 0 & \geq & 0 \\ \lambda_3 : & 0x + 1y + 0x' - 1y' - 1 & \geq & 0 \\ \lambda_4 : & 0x - 1y + 0x' + 0y' - k & \geq & 0 \\ \implies & & & \\ & ax + by - ax' - by' - 1 & \geq & 0 \\ & ax + by & \geq & 0. \end{array}$$

On en déduit que $\phi(a,b,k)$ est équivalente à la conjonction $DEC(a,b,k)$:

$$\exists \lambda_1 \geq 0, \ldots, \lambda_4 \geq 0 \,.\, \begin{cases} a = \lambda_1 + \lambda_2 \\ b = \lambda_2 + \lambda_3 - \lambda_4 \\ -a = -\lambda_2 \\ -b = -\lambda_3 \\ -1 \geq -\lambda_3 - k\lambda_4 \end{cases}$$

assurant la décroissance de la fonction de rang et $POS(a,b,k)$:

$$\exists \lambda'_1 \geq 0, \ldots, \lambda'_4 \geq 0 \,.\, \begin{cases} a = \lambda'_1 + \lambda'_2 \\ b = \lambda'_2 + \lambda'_3 - \lambda'_4 \\ 0 = -\lambda'_2 \\ 0 = -\lambda'_3 \\ 0 \geq -\lambda'_3 - k\lambda'_4 \end{cases}$$

assurant la positivité de la fonction de rang.

Examinons $DEC(a,b,k)$. Nous nous apercevons que le produit $k\lambda_4$ introduit une non-linéarité que nous pouvons éliminer en remarquant que, comme $\lambda_4 \geq 0$, soit $\lambda_4 = 0$ (et donc $k\lambda_4 = 0$) soit $\lambda_4 > 0$. Pour ce dernier cas nous introduisons la variable $P = k\lambda_4$. On a alors la propriété :

Lemme 3.4 *La formule $\exists k \, DEC(a,b,k)$ est équivalente à la disjonction $DEC_1(a,b) \vee DEC_2(a,b)$.*

Dans notre cas, $DEC_1(a,b)$ équivaut à :

$$\exists \lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0 \,.\, \begin{cases} a = \lambda_1 + \lambda_2 \\ b = \lambda_2 + \lambda_3 \\ -a = -\lambda_2 \\ -b = -\lambda_3 \\ -1 \geq -\lambda_3 \end{cases}$$

et $DEC_2(a,b)$ à :

$$\exists \lambda_1 \geq 0, \lambda_2 \geq 0, \lambda_3 \geq 0, \lambda_4 > 0, P \,.\, \begin{cases} a = \lambda_1 + \lambda_2 \\ b = \lambda_2 + \lambda_3 - \lambda_4 \\ -a = -\lambda_2 \\ -b = -\lambda_3 \\ -1 \geq -\lambda_3 - P. \end{cases}$$

Preuve ($\Rightarrow$) Soient k un rationnel et quatre rationnels positifs ou nuls λ_i pour $1 \leq i \leq 4$ vérifiant la définition de $DEC(a,b,k)$. Si $\lambda_4 = 0$, alors $DEC(a,b,k)$ se simplifie en $DEC_1(a,b)$ qui est vraie. Si $\lambda_4 > 0$, en posant $P = k\lambda_4$, nous constatons que $DEC_2(a,b)$ est vraie.

($\Leftarrow$) Supposons $DEC_1(a,b)$ vraie. Alors en prenant $\lambda_4 = 0$ et k rationnel quelconque, 0 par exemple, il existe k tel que $DEC(a,b,k)$ est vraie. Supposons $DEC_2(a,b)$ vraie. Posons $k = P/\lambda_4$ (toujours possible puisque $\lambda_4 > 0$) : nous constatons qu'il existe k tel que $DEC(a,b,k)$ est vraie.

Concernant la condition de positivité, nous avons également :

Lemme 3.5 *La formule $\exists k \,.\, POS(a,b,k)$ est équivalente à la disjonction $POS_1(a,b) \vee POS_2(a,b)$.*

Dans notre cas, $POS_1(a,b)$ équivaut à :

$$\exists \lambda'_1 \geq 0, \lambda'_2 \geq 0, \lambda'_3 \geq 0 \,.\, \begin{cases} a = \lambda'_1 + \lambda'_2 \\ b = \lambda'_2 + \lambda'_3 \\ 0 = -\lambda'_2 \\ 0 = -\lambda'_3 \\ 0 \geq -\lambda'_3 \end{cases}$$

et $POS_2(a,b)$ à :

$$\exists \lambda'_1 \geq 0, \lambda'_2 \geq 0, \lambda'_3 \geq 0, \lambda'_4 > 0, P' \,.\, \begin{cases} a = \lambda'_1 + \lambda'_2 \\ b = \lambda'_2 + \lambda'_3 - \lambda'_4 \\ 0 = -\lambda'_2 \\ 0 = -\lambda'_3 \\ 0 \geq -\lambda'_3 - P'. \end{cases}$$

Nous pouvons à présent combiner les deux résultats précédents :

Proposition 3.6 *La formule* $\exists k \,.\, \phi(a,b,k)$ *équivaut à* $[DEC_1(a,b) \vee DEC_2(a,b)] \wedge [POS_1(a,b) \vee POS_2(a,b)]$.

Preuve D'après les deux lemmes précédents, il reste à justifier l'équivalence entre les formules $\exists k \,.\, \phi(a,b,k)$ et $\exists k \,.\, DEC(a,b,k) \wedge \exists k \,.\, POS(a,b,k)$.

($\Rightarrow$) Soit k_0 un rationnel vérifiant $\phi(a,b,k_0)$. Comme $\phi(a,b,k) \iff DEC(a,b,k) \wedge POS(a,b,k)$, on a $DEC(a,b,k_0)$ et $POS(a,b,k_0)$.

($\Leftarrow$) Supposons l'existence de k_d tel que $DEC(a,b,k_d)$ et k_p tel que $POS(a,b,k_p)$. Alors $k_0 = \max(k_d, k_p)$ valide $DEC(a,b,k_0) \wedge POS(a,b,k_0)$ et montre $\exists k \,.\, \phi(a,b,k)$.

Pour revenir à notre problème initial, l'existence d'une fonction de rang linéaire à terme se résume donc à la satisfiabilité d'au moins un des quatre systèmes linéaires :

$$DEC_1(a,b) \wedge POS_1(a,b)$$
$$DEC_1(a,b) \wedge POS_2(a,b)$$
$$DEC_2(a,b) \wedge POS_1(a,b)$$
$$DEC_2(a,b) \wedge POS_2(a,b)$$

ce que nous pouvons décider en temps polynomial. Pour notre exemple courant, il s'agit de $DEC_2(a,b) \wedge POS_1(a,b)$, ce que confirme la requête Prolog suivante :

```
?- dec2pos1.
true.
?-
```

après compilation du programme :

```
dec2pos1 :-
   {L1 >= 0, L2 >= 0, L3 >= 0, L4 > 0,
       A = L1 + L2,
       B = L2 + L3 - L4,
       A = L2,
       B = L3,
       1 =< L3 + P,
    LP1 >= 0, LP2 >= 0, LP3 >= 0,
       A = LP1 + LP2,
       B = LP2 + LP3,
       0 = LP2,
       0 = LP3,
       0 =< LP3}.
```

Nous sommes maintenant en mesure de définir un algorithme de détection d'existence d'une fonction de rang linéaire à terme (cf. l'algorithme 1).

Théorème 3.7 *Pour toute boucle SLC* C *et toute fonction linéaire strictement croissante associée à* C*, l'algorithme 1 décide l'existence d'une fonction de rang linéaire à terme en temps polynomial.*

Algorithme 1 Existence d'une fonction de rang linéaire à terme

Entrée : Une boucle SLC $p(\mathbf{x}) \leftarrow c(\mathbf{x},\mathbf{x}'), p(\mathbf{x}')$ et $f(\mathbf{x})$ une fonction linéaire vérifiant la condition $\forall \mathbf{x},\mathbf{x}' : c(\mathbf{x},\mathbf{x}') \implies f(\mathbf{x}') \geq 1 + f(\mathbf{x})$.
Sortie : Décide l'existence d'une fonction de rang à terme $\rho(\mathbf{x}) = \mathbf{ax} = \sum a_i x_i$ à partir du seuil k.

1: $DEC(\mathbf{a},k) \leftarrow$ Farkas pour la décroissance de ρ
2: $DEC_1(\mathbf{a}), DEC_2(\mathbf{a}) \leftarrow$ linéarisat. de $DEC(\mathbf{a},k)$
3: $POS(\mathbf{a},k) \leftarrow$ Farkas pour la positivité de ρ
4: $POS_1(\mathbf{a}), POS_2(\mathbf{a}) \leftarrow$ linéarisation de $POS(\mathbf{a},k)$
5: **si** $\bigvee_{1\leq i,j\leq 2} DEC_i(\mathbf{a}) \wedge POS_j(\mathbf{a})$ est satisfiable **alors**
6: **retourner vrai**
7: **sinon**
8: **retourner faux**
9: **fin si**

Le calcul d'un certificat de terminaison, c.-à-d. d'une fonction de rang linéaire à terme ρ et du seuil k associé peut s'effectuer lors de l'exécution de l'algorithme comme suit :

- si $DEC_1(\mathbf{a}) \wedge POS_1(\mathbf{a})$ est satisfiable, on calcule une solution $\mathbf{a}$ de la conjonction et tout rationnel k convient ;
- si $DEC_1(\mathbf{a}) \wedge POS_2(\mathbf{a})$ est satisfiable, on calcule une solution $\mathbf{a}$, λ', P' de la conjonction, on prend $k = P'/\lambda'_n$;
- si $DEC_2(\mathbf{a}) \wedge POS_1(\mathbf{a})$ est satisfiable, on calcule une solution $\mathbf{a}$, λ, P de la conjonction, on prend $k = P/\lambda_n$;
- si $DEC_2(\mathbf{a}) \wedge POS_2(\mathbf{a})$ est satisfiable, on calcule une solution $\mathbf{a}$, λ, P, λ', P' de la conjonction, on prend $k = \max(P/\lambda_n, P'/\lambda'_n)$.

Exemple 3.8 *Poursuivons l'exemple 3.1. Voici la solution la plus générale de* $DEC_2(a,b) \wedge POS_1(a,b)$

```
?-  {L1 >= 0, L2>= 0, L3 >= 0, L4 > 0,
        A = L1 + L2,
        B = L2 + L3 - L4,
        A = L2,
        B = L3,
        1 =< L3 + P,
     LP1 >= 0, LP2 >= 0, LP3 >= 0,
        A = LP1 + LP2,
        B = LP2 + LP3,
        0 = LP2,
        0 = LP3,
        0 =< LP3}.
B = 0, L1 = 0, L3 = 0, LP2 = 0, LP3 = 0,
{LP1 = L4, L2 = L4, A = L4, L4 > 0, P >= 1}.
?-
```

Une solution particulière est $b = 0 = \lambda_1 = \lambda_3 = \lambda'_2 =$

λ'_3, $a = 1 = \lambda'_1 = \lambda_2 = \lambda_4$, $P = 1$. *Nous en déduisons que* $\rho(x, y) = x$ *est une fonction de rang linéaire à terme à partir du seuil* $k = P/\lambda_4 = 1$.

3.2 Vérification

Étant données une boucle SLC C, une fonction strictement croissante associée f et une fonction linéaire ρ, il s'agit à présent de déterminer si ρ est bien une fonction de rang à terme. La vérification peut s'effectuer en appliquant l'algorithme 1, avec les coefficients $\mathbf{a}$ instanciés comme précisé par ρ. On peut au besoin déterminer le seuil k comme indiqué précédemment.

3.3 D'autres exemples

Les exemples qui suivent complémentent ceux de la partie 3.1. Nous cherchons des fonctions de rang linéaires à terme pour ces boucles SLC. Nous nous restreignons aux fonctions strictement croissantes de la forme $f(x_1, \ldots, x_n) = x_i$ et $f(x_1, \ldots, x_n) = -x_i$, aisément détectables via une technique similaire à celle exposée section 2.2.

Exemple 3.9 *Étudions la boucle suivante :*

$$\begin{aligned} p(x, y, z) \leftarrow\ & y' = 10,\ y - z \geq 1, \\ & x = z,\ z - z' \leq -1,\ x' = z', \\ & p(x', y', z') \end{aligned}$$

qui n'admet pas de fonction de rang linéaire. La fonction $f(x, y, z) = z$ *croît strictement pour cette boucle puisque* z'*, la nouvelle valeur de* z*, est nécessairement supérieure d'une unité à celle de* z*.*

En posant $\rho(x, y, z) = ax+by+cz$, ρ est une fonction de rang linéaire à terme quand :

$$\exists a, b, c, k\ .\ \forall x, y, z, x', y', z' : \left.\begin{array}{r} y' \geq 10 \\ y' \leq 10 \\ y - z \geq 1 \\ x \geq z \\ x \leq z \\ z - z' \leq -1 \\ x' \geq z' \\ x' \leq z' \\ z \geq k \end{array}\right\} \implies \begin{cases} ax + by + cz \\ \quad \geq 1 + ax' + by' + cz' \\ ax + by + cz \geq 0. \end{cases}$$

En considérant a, b, c et k comme des paramètres et en appliquant le lemme de Farkas ainsi que les lemmes 3.4 et 3.5, nous obtenons :

– $DEC_1(a, b, c)$:

$$\exists \lambda_1 \geq 0, \ldots, \lambda_8 \geq 0\ . \begin{cases} a = \lambda_4 - \lambda_5 \\ b = \lambda_3 \\ c = -\lambda_3 - \lambda_4 + \lambda_5 - \lambda_6 \\ -a = \lambda_7 - \lambda_8 \\ -b = \lambda_1 - \lambda_2 \\ -c = \lambda_6 - \lambda_7 + \lambda_8 \\ -1 \geq -10\lambda_1 + 10\lambda_2 - \lambda_3 - \lambda_6 \end{cases}$$

– $DEC_2(a, b, c)$:

$$\exists \lambda_1 \geq 0, \ldots, \lambda_8 \geq 0, \lambda_9 > 0, P\ . \begin{cases} a = \lambda_4 - \lambda_5 \\ b = \lambda_3 \\ c = -\lambda_3 - \lambda_4 + \lambda_5 - \lambda_6 + \lambda_9 \\ -a = \lambda_7 - \lambda_8 \\ -b = \lambda_1 - \lambda_2 \\ -c = \lambda_6 - \lambda_7 + \lambda_8 \\ -1 \geq -10\lambda_1 + 10\lambda_2 - \lambda_3 - \lambda_6 - P \end{cases}$$

– $POS_1(a, b, c)$:

$$\exists \lambda'_1 \geq 0, \ldots, \lambda'_8 \geq 0\ . \begin{cases} a = \lambda'_4 - \lambda'_5 \\ b = \lambda'_3 \\ c = -\lambda'_3 - \lambda'_4 + \lambda'_5 - \lambda'_6 \\ 0 = \lambda'_7 - \lambda'_8 \\ 0 = \lambda'_1 - \lambda'_2 \\ 0 = \lambda'_6 - \lambda'_7 + \lambda'_8 \\ 0 \geq -10\lambda'_1 + 10\lambda'_2 - \lambda'_3 - \lambda'_6 \end{cases}$$

– $POS_2(a, b, c)$:

$$\exists \lambda'_1 \geq 0, \ldots, \lambda'_8 \geq 0, \lambda'_9 > 0, P'\ . \begin{cases} a = \lambda'_4 - \lambda'_5 \\ b = \lambda'_3 \\ c = -\lambda'_3 - \lambda'_4 + \lambda'_5 - \lambda'_6 + \lambda'_9 \\ 0 = \lambda'_7 - \lambda'_8 \\ 0 = \lambda'_1 - \lambda'_2 \\ 0 = \lambda'_6 - \lambda'_7 + \lambda'_8 \\ 0 \geq -10\lambda'_1 + 10\lambda'_2 - \lambda'_3 - \lambda'_6 - P'. \end{cases}$$

La formule $DEC_2(a, b, c) \wedge POS_2(a, b, c)$ admet une solution, par exemple : $\lambda_6 = \lambda'_5 = \lambda'_6 = 0$, $a = b = 1 = \lambda_1 = \lambda_3 = \lambda_5 = \lambda_7 = \lambda_9 = \lambda'_1 = \lambda'_2 = \lambda'_3 = \lambda'_4 = \lambda'_7 = \lambda'_8 = \lambda'_9$, $\lambda_2 = \lambda_4 = \lambda_8 = 2$, $c = -1$, et $P = P' = 11$.

Nous en déduisons que $\rho(x,y,z) = x + y - z$ est bien une fonction de rang linéaire à terme à partir du seuil $k = P/\lambda_9 = P'/\lambda'_9 = 11$.

Exemple 3.10 *Étudions la boucle de l'exemple* 2.2 *et montrons que cette boucle admet également une fonction de rang linéaire à terme. La fonction $f(x,y) = -y$ croît strictement pour cette boucle car y', la nouvelle valeur de y, est nécessairement inférieure d'une unité à celle de y.*

En posant $\rho(x,y) = ax + by$, ρ est une fonction de rang linéaire à terme quand :

$$\exists a,b,k\ .\ \forall x,y,x',y' : \left.\begin{array}{r} x \geq 0 \\ y' \leq y - 1 \\ x' \leq x + y \\ y \leq -1 \\ -y \geq k \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} ax + by \geq 1 + ax' + by' \\ ax + by \geq 0. \end{array}\right.$$

En considérant a ,b et k comme des paramètres et en appliquant le lemme de Farkas ainsi que les lemmes 3.4 et 3.5, nous obtenons :

– $DEC_1(a,b)$:

$$\exists \lambda_1 \geq 0, \ldots, \lambda_4 \geq 0\ .\ \left\{\begin{array}{r} a = \lambda_1 + \lambda_2 \\ b = \lambda_2 + \lambda_3 - \lambda_4 \\ -a = -\lambda_2 \\ -b = -\lambda_3 \\ -1 \geq -\lambda_3 - \lambda_4 \end{array}\right.$$

– $DEC_2(a,b)$:

$$\exists \lambda_1 \geq 0, \ldots, \lambda_4 \geq 0, \lambda_5 > 0, P\ .\ \left\{\begin{array}{r} a = \lambda_1 + \lambda_2 \\ b = \lambda_2 + \lambda_3 - \lambda_4 - \lambda_5 \\ -a = -\lambda_2 \\ -b = -\lambda_3 \\ -1 \geq -\lambda_3 - \lambda_4 - P \end{array}\right.$$

– $POS_1(a,b)$:

$$\exists \lambda'_1 \geq 0, \ldots, \lambda'_4 \geq 0\ .\ \left\{\begin{array}{l} a = \lambda'_1 + \lambda'_2 \\ b = \lambda'_2 + \lambda'_3 - \lambda'_4 \\ 0 = -\lambda'_2 \\ 0 = -\lambda'_3 \\ 0 \geq -\lambda'_3 - \lambda'_4 \end{array}\right.$$

– $POS_2(a,b)$:

$$\exists \lambda'_1 \geq 0, \ldots, \lambda'_4 \geq 0, \lambda'_5 > 0, P'\ .\ \left\{\begin{array}{l} a = \lambda'_1 + \lambda'_2 \\ b = \lambda'_2 + \lambda'_3 - \lambda'_4 - \lambda'_5 \\ 0 = -\lambda'_2 \\ 0 = -\lambda'_3 \\ 0 \geq -\lambda'_3 - \lambda'_4 - P'. \end{array}\right.$$

La formule $DEC_1(a,b) \wedge POS_1(a,b)$ admet une solution, par exemple : $b = 0 = \lambda_1 = \lambda_3 = \lambda'_2 = \lambda'_3 = \lambda'_4$, $a = 1 = \lambda_2 = \lambda_4 = \lambda'_1$. Il n'y pas de condition sur k, donc $k = 0$ convient. Nous en déduisons que $\rho(x,y) = x$ est une fonction de rang linéaire à terme à partir du seuil $k = 0$.

Exemple 3.11 *Étudions la boucle suivante :*

$$p(x) \leftarrow x = -1,\ x' = -2,\ p(x').$$

La fonction $f(x) = -x$ croît strictement pour cette boucle.

En posant $\rho(x) = ax$, ρ est une fonction de rang linéaire à terme quand :

$$\exists a,k\ .\ \forall x,x' : \left.\begin{array}{r} x \geq -1 \\ x \leq -1 \\ x' \geq -2 \\ x' \leq -2 \\ -x \geq k \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} ax \geq 1 + ax' \\ ax \geq 0. \end{array}\right.$$

En considérant a et k comme des paramètres et en appliquant le lemme de Farkas ainsi que les lemmes 3.4 et 3.5, nous obtenons :

– $DEC_1(a)$:

$$\exists \lambda_1 \geq 0, \ldots, \lambda_4 \geq 0\ .\ \left\{\begin{array}{r} a = \lambda_1 - \lambda_2 \\ -a = \lambda_3 - \lambda_4 \\ -1 \geq \lambda_1 - \lambda_2 + 2\lambda_3 - 2\lambda_4 \end{array}\right.$$

– $DEC_2(a)$:

$$\exists \lambda_1 \geq 0, \ldots, \lambda_4 \geq 0, \lambda_5 > 0, P\ .\ \left\{\begin{array}{r} a = \lambda_1 - \lambda_2 - \lambda_5 \\ -a = \lambda_3 - \lambda_4 \\ -1 \geq \lambda_1 - \lambda_2 + 2\lambda_3 - 2\lambda_4 - P \end{array}\right.$$

– $POS_1(a)$:

$$\exists \lambda'_1 \geq 0, \ldots, \lambda'_4 \geq 0\ .\ \left\{\begin{array}{l} a = \lambda'_1 - \lambda'_2 \\ 0 = \lambda'_3 - \lambda'_4 \\ 0 \geq \lambda'_1 - \lambda'_2 + 2\lambda'_3 - 2\lambda'_4 \end{array}\right.$$

– $POS_2(a)$:

$$\exists \lambda'_1 \geq 0, \ldots, \lambda'_4 \geq 0, \lambda'_5 > 0, P' \ . \begin{cases} a = \lambda'_1 - \lambda'_2 - \lambda'_5 \\ 0 = \lambda'_3 - \lambda'_4 \\ 0 \geq \lambda'_1 - \lambda'_2 + 2\lambda'_3 - 2\lambda'_4 - P'. \end{cases}$$

La formule $DEC_1(a) \wedge POS_2(a)$ admet une solution, par exemple : $a = 1 = \lambda_1 = \lambda_4 = \lambda'_5$, $\lambda_2 = 0 = \lambda_3 = \lambda'_2 = \lambda'_3 = \lambda'_4$, $\lambda'_1 = 2 = P'$. Nous en déduisons que $\rho(x) = x$ est une fonction de rang linéaire à terme à partir du seuil $k = P'/\lambda'_5 = 2$.

4 Conclusion

Dans cet article, pour les boucles SLC rationnelles, nous avons proposé une définition des fonctions de rang linéaires à terme assortie d'un algorithme polynomial correct et complet pour la détection de telles fonctions de rang.

Cet algorithme est implanté dans l'analyseur de terminaison BinTerm [16]. Il admet une formulation élégante en programmation logique avec contraintes. Il nous reste à approfondir le choix de la "bonne" fonction croissante. Pour le moment, nous nous restreignons aux fonctions candidates de la forme $f(x_1, \ldots, x_n) = x_i$ ou $-x_i$.

Le concept d'*eventual termination*, similaire à la notion de fonction de rang à terme, apparaît dans [7, 8]. La classe de boucles qui y est abordée est plus vaste, mais comme les auteurs s'appuient sur des techniques de différences finies, les approches sont incomplètes.

Concernant les fonctions de rang linéaires des boucles SLC entières, la *vérification*, c.-à-d. déterminer dans $\mathbb{Z}$ la satisfiabilité des formules $c(\mathbf{x}, \mathbf{x}') \wedge \rho(\mathbf{x}) < 1 + \rho(\mathbf{x}')$ et $c(\mathbf{x}, \mathbf{x}') \wedge \rho(\mathbf{x}) < 0$, est à présent un problème de programmation linéaire en nombres entiers, de complexité NP. Pour la *détection*, comme le lemme de Farkas n'est plus vrai dans $\mathbb{Z}$, la méthode exposée à la section 2 n'est pas directement transposable. La question de l'existence d'une fonction de rang linéaire dans $\mathbb{Z}$ vient d'être résolue très récemment par [4]. Le problème appartient à la classe coNP et les auteurs présentent un algorithme de complexité exponentielle qui génère les fonctions de rang linéaires.

L'étude des fonctions de rang linéaires à terme pour les boucles SLC entières est une extension naturelle à notre travail à envisager.

Références

[1] E. Albert, P. Arenas, S. Genaim, M. Gómez-Zamalloa, G. Puebla, D. V. Ramírez, G. Román, and D. Zanardini. Termination and cost analysis with COSTA and its user interfaces. *Electronic Notes in Theoretical Computer Science*, 258(1) :109–121, 2009.

[2] R. Bagnara, F. Mesnard, A. Pescetti, and E. Zaffanella. A new look at the automatic synthesis of linear ranking functions. *Information and Computation*, 215 :47–67, 2012.

[3] A. M. Ben-Amram and S. Genaim. On the linear ranking problem for integer linear-constraint loops. Technical Report arXiv :1208.4041v2 [cs.PL], 2013. Available from `http://arxiv.org/`.

[4] A. M. Ben-Amram and S. Genaim. On the linear ranking problem for integer linear-constraint loops. In R. Giacobazzi and R. Cousot, editors, *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*, pages 51–62, Rome, Italy, 2013. Association for Computing Machinery.

[5] A. M. Ben-Amram, S. Genaim, and A. N. Masud. On the termination of integer loops. *ACM Transactions on Programming Languages and Systems*, 34(4) :16 :1–16 :24, 2012.

[6] M. Bozga, R. Iosif, and F. Konecný. Deciding conditional termination. In C. Flanagan and B. König, editors, *Tools and Algorithms for the Construction and Analysis of Systems : Proceedings of the 18th International Conference (TACAS 2012)*, volume 7214 of *Lecture Notes in Computer Science*, pages 252–266, Tallinn, Estonia, 2012. Springer.

[7] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *Automata, Languages and Programming : Proceedings of the 32nd International Colloquium (ICALP 2005)*, volume 3580 of *Lecture Notes in Computer Science*, pages 1349–1361, Lisbon, Portugal, 2005. Springer.

[8] A. R. Bradley, Z. Manna, and H. B. Sipma. Termination of polynomial programs. In R. Cousot, editor, *Verification, Model Checking and Abstract Interpretation : Proceedings of the 6th International Conference (VMCAI 2005)*, volume 3385 of *Lecture Notes in Computer Science*, pages 113–129, Paris, France, 2005. Springer-Verlag, Berlin.

[9] M. Braverman. Termination of integer linear programs. In T. Ball and R. B. Jones, editors, *Computer Aided Verification : Proceedings of the 18th International Conference (CAV 2006)*, vo-

lume 4144 of *Lecture Notes in Computer Science*, pages 372–385, Seattle, WA, USA, 2006. Springer.

[10] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In M. I. Schwartzbach and T. Ball, editors, *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 415–426, Ottawa, Ontario, Canada, 2006. Association for Computing Machinery.

[11] A. C. Hearn. REDUCE : the first forty years. In A. Dolzmann, A. Seidl, and T. Sturm, editors, *Algorithmic Algebra and Logic : Proceedings of the A3L 2005 Conference in Honor of the 60th Birthday of Volker Weispfenning*, pages 19–24, Passau, Germany, 2005.

[12] C. Holzbaur. OFAI clp(Q,R) manual, edition 1.3.3. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.

[13] C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java bytecode by term rewriting. In Christopher Lynch, editor, *RTA*, volume 6 of *LIPIcs*, pages 259–276. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[14] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In B. Steffen and G. Levi, editors, *Verification, Model Checking and Abstract Interpretation : Proceedings of the 5th International Conference (VMCAI 2004)*, volume 2937 of *Lecture Notes in Computer Science*, pages 239–251, Venice, Italy, 2004. Springer.

[15] K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes (extended abstract). In D. J. Rosenkrantz, editor, *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 216–226, Denver, CO, USA, 1991. Association for Computing Machinery.

[16] F. Spoto, F. Mesnard, and É. Payet. A termination analyzer for Java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

[17] A. Tiwari. Termination of linear programs. In R. Alur and D. Peled, editors, *Computer Aided Verification : Proceedings of the 16th International Conference (CAV 2004)*, volume 3114 of *Lecture Notes in Computer Science*, pages 70–82, Boston, MA, USA, 2004. Springer.

Compilation de CSPs : carte de complexité des MDDs non-déterministes *

Jérôme Amilhastre[1] **Hélène Fargier**[2] **Alexandre Niveau**[3] **Cédric Pralet**[4]

[1] Cameleon Software, Le Galilée, 185 rue Galilée, F-31670 Labège
[2] IRIT, Université Paul Sabatier, F-31062 Toulouse Cedex 9
[3] CRIL, Université d'Artois, F-62307 Lens Cedex
[4] ONERA—The French Aerospace Lab, F-31055, Toulouse

jamilhastre@cameleon-software.com fargier@irit.fr niveau@cril.fr cpralet@onera.fr

Résumé

Les CSPs fournissent un cadre puissant pour la représentation de problèmes très divers. La difficulté est que la plupart des requêtes associées aux CSPs sont NP-difficiles, mais doivent dans certains contextes être traitées « en ligne ». C'est pour cette raison que les diagrammes de décision multivalués (MDDs) ont été proposés pour la compilation de CSPs.

Cet article dresse une carte de compilation des MDDs, dans l'esprit de la carte de la famille des NNFs de Darwiche et Marquis, en analysant les MDDs selon leur compacité et les requêtes et transformations qu'ils supportent en temps polynomial. Les MDDs déterministes et ordonnés généralisant les diagrammes de décision binaire ordonnés à des variables non-booléennes, le fait que leurs propriétés soient similaires n'est pas surprenant. Cependant, notre étude met en avant l'intérêt des MDDs ordonnés *non-déterministes* : restreint aux variables booléennes, ce fragment est strictement plus compact que ceux des OBDDs et des DNFs, et admet des performances proches de celles des DNNFs. La comparaison aux MDDs classiques montre que relâcher la contrainte du déterminisme améliore la compacité et permet à plus de transformations d'être supportées en temps polynomial. Des expériences sur des problèmes aléatoires confirment le gain en compacité.

Abstract

CSPs offer a powerful framework for representing a great variety of problems. The difficulty is that most of the requests associated with CSPs are NP-hard. As these requests must be addressed online, Multivalued Decision Diagrams (MDDs) have been proposed as a way to compile CSPs.

In the present paper, we draw a compilation map of MDDs, in the spirit of the NNF compilation map, analyzing MDDs according to their succinctness and to their polytime transformations and queries. Deterministic ordered MDDs are a generalization of ordered binary decision diagrams to non-Boolean domains : unsurprisingly, they have similar capabilities. More interestingly, our study puts forward the interest of *non-deterministic* ordered MDDs : when restricted to Boolean domains, this fragment captures OBDD and DNF as proper subsets and has performances close to those of DNNF. The comparison to classical, deterministic MDDs shows that relaxing the determinism requirement leads to an increase in succinctness and allows more transformations to be satisfied in polytime (typically, the disjunctive ones). Experiments on random problems confirm the gain in succinctness.

1 Introduction

Le cadre très puissant des problèmes de satisfaction de contraintes (CSPs) permet de représenter des problèmes très variés. Un CSP peut faire l'objet de requêtes de différentes sortes, telles que la classique extraction d'une solution, mais aussi l'exigence de cohérence des domaines, l'ajout dynamique de nouvelles contraintes, le comptage de solutions, et même des combinaisons de ces requêtes. Ainsi, la résolution interactive d'un problème de configuration de produit se ramène à l'ajout successif de contraintes unaires tout en maintenant la cohérence forte.

La plupart de ces requêtes sont NP-difficiles ; elles doivent pourtant être parfois effectuées en ligne. Pour

*Cet article est la traduction de « Compiling CSPs: A Complexity Map of (Non-Deterministic) Multivalued Decision Diagrams », paru dans les actes d'ICTAI 2012 [2]. Les travaux ont été partiellement financés par l'ANR dans le cadre du projet BR4CP (ANR-11-BS02-008).

résoudre cette contradiction, une possibilité consiste à représenter l'ensemble des solutions du CSP par un diagramme de décision multivalué (MDD) [14, 10, 5], c'est-à-dire un graphe dont chaque nœud est étiqueté par une variable et chaque arc représente l'instanciation d'une variable. Dans de tels diagrammes, chaque chemin depuis la racine jusqu'au puits représente une solution du CSP. Sur un MDD, diverses opérations — dont celles précédemment citées — peuvent être effectuées en temps polynomial en la taille du diagramme ; cette taille peut théoriquement être exponentiellement plus grande que celle du CSP d'origine, mais reste raisonnable en pratique dans beaucoup d'applications. En effet, les MDDs tirent parti de leur structure de graphe et de l'interchangeabilité (conditionnelle) des valeurs du CSP pour gagner de l'espace en fusionnant les sous-problèmes identiques. Les diagrammes de décision ont été utilisés dans des contextes variés, comme la configuration de produit [4], les systèmes de recommandation [6], ou, dans leur forme booléenne originelle, pour la planification [8, 9] et le diagnostic [13]. À notre connaissance, ces applications considèrent toujours des MDDs *déterministes*, c'est-à-dire que les étiquettes des arcs sortant d'un même nœud sont mutuellement exclusives. Cela implique que chaque solution est représentée exactement une fois dans le graphe ; cependant, cette contrainte est superflue pour beaucoup d'opérations. Il semble donc intéressant de considérer des structures *non-déterministes*.

Pour évaluer les avantages et les inconvénients du déterminisme par rapport au non-déterminisme dans les MDDs, nous proposons d'en dresser une carte de compilation, dans l'esprit de la carte des NNFs [7]. Une carte de compilation permet d'identifier les langages les plus concis qui supportent en temps polynomial les opérations nécessaires à une application donnée. Pour construire une telle carte, nous conduisons une analyse générale de la complexité des MDDs sur un ensemble d'opérations, utilisées dans des problèmes de raisonnement ou de décision.

La section suivante présente le langage `MDD` et ses sous-langages, comme ceux des MDDs déterministes ou ordonnés. Dans la section 3, nous étudions le cas des domaines booléens, pour situer `MDD` dans la carte de `NNF` : nous montrons qu'au-delà de `dOMDD`, qui correspond plus ou moins à `OBDD`, la famille de `MDD` inclut le langage des diagrammes de décision non-déterministes (`OMDD`) qui généralise à la fois `OBDD` et `DNF`. La section 4 est dédiée à la carte de `MDD`, avec une analyse de la concision des différents sous-langages de la famille de `MDD`, ainsi que l'analyse de la complexité de plusieurs requêtes et transformations. La section 5 présente ensuite quelques résultats expérimentaux sur la compacité relative des MDDs déterministes et non-déterministes. Enfin, les preuves les plus significatives sont rassemblées en annexe.

2 Diagrammes de décision multivalués

Un problème de satisfaction de contraintes $P = \langle X, C\rangle$ consiste en un ensemble fini de variables $X = \{x_1, \ldots, x_n\}$ possédant un domaine fini de valeurs, et un ensemble fini de contraintes C qui spécifient les combinaisons de valeurs autorisées pour certains sous-ensembles de variables. Pour $y \in X$, $\mathrm{Dom}(y)$ désigne le domaine de y. Pour $Y = \{y_1, \ldots, y_k\} \subseteq X$, $\mathrm{Dom}(Y)$ représente $\mathrm{Dom}(y_1) \times \cdots \times \mathrm{Dom}(y_k)$, et $\vec{y}$ désigne une Y-*instanciation* des variables de Y, c'est-à-dire $\vec{y} \in \mathrm{Dom}(Y)$. Quand $Z \cap Y = \varnothing$, $\vec{z} \,.\, \vec{y}$ est la *concaténation* de $\vec{z}$ et $\vec{y}$. Enfin, $\vec{y}|_{y_i}$ désigne la valeur assignée à y_i dans $\vec{y}$.

L'ensemble de solutions $\mathrm{Sol}(P)$ d'un CSP $P = \langle X, C\rangle$ est l'ensemble des éléments de $\mathrm{Dom}(X)$ qui satisfont toutes les contraintes dans C. On peut représenter $\mathrm{Sol}(P)$ par un diagramme de décision multivalué sur X [14, 5].

Définition 1 (MDDs)**.** Un *diagramme de décision multivalué* (MDD) sur un ensemble X de variables à domaines finis, est un graphe acyclique orienté $\varphi = \langle \mathcal{N}, \mathcal{E}\rangle$ où $\mathcal{N}$ est un ensemble de nœuds contenant au plus une racine (notée $\mathrm{Root}(\varphi)$) et au plus une feuille (appelée *puits* et notée $\mathrm{Sink}(\varphi)$).

Excepté le puits, chaque nœud $N \in \mathcal{N}$ est étiqueté par une variable $\mathrm{Var}(N)$ de X, et chaque arc $E \in \mathcal{E}$ sortant de N est étiqueté par une valeur appartenant au domaine de $\mathrm{Var}(N)$, notée [1] $\mathrm{Lbl}(E)$.

La taille de φ, notée $\|\varphi\|$, est égale au nombre de ses nœuds et arcs, plus les cardinalités des domaines des variables de X.

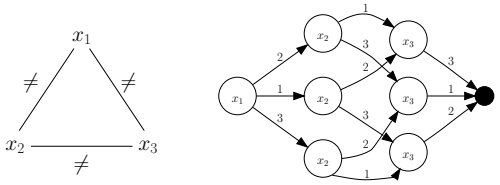


FIGURE 1 – Le problème « alldiff » de coloriage sur une clique (3 variables, 3 couleurs) et un dOMDD représentant son ensemble de solutions ($x_1 < x_2 < x_3$).

Chaque solution de P est représentée par un chemin depuis la racine du MDD jusqu'à son puits (voir par

1. On note également $\mathrm{Out}(N)$ (resp. $\mathrm{In}(N)$) l'ensemble des arcs sortants (resp. entrants) de N. Un arc $E \in \mathcal{E}$ est souvent désigné par le triplet $\langle N, N', a\rangle$ constitué par son nœud d'origine N, noté $\mathrm{Src}(E)$, son nœud de destination N', noté $\mathrm{Dest}(E)$, et sa valeur associée $a = \mathrm{Lbl}(E)$.

exemple la figure 1). Pour vérifier si une instanciation $\vec{x} = \langle a_1, \ldots, a_n \rangle \in \mathrm{Dom}(X)$ appartient à $\mathrm{Sol}(P)$, il suffit de parcourir le MDD de la racine au puits, et pour chaque nœud N étiqueté par la variable x_i, de suivre l'arc étiqueté a_i. Si un tel chemin existe, alors $\vec{x}$ appartient à $\mathrm{Sol}(P)$. Dans le cas contraire — si le puits ne peut être atteint — alors $\vec{x}$ n'est pas solution de P.

Au-delà du cadre CSP, un MDD peut représenter n'importe quel sous-ensemble de $\mathrm{Dom}(X)$ — ou, de manière équivalente, n'importe quelle fonction booléenne sur $\mathrm{Dom}(X)$.

Définition 2 (Sémantique des MDDs)**.** Un MDD φ sur X représente une fonction $\mathrm{I}(\varphi)$ de $\mathrm{Dom}(X)$ vers $\{\top, \bot\}$, appelée son *interprétation* et définie comme suit : pour toute X-instanciation $\vec{x}$, $\mathrm{I}(\varphi)(\vec{x}) = \top$ si et seulement s'il existe un chemin p de la racine au puits de φ tel que pour tout arc $E = \langle N, N', a \rangle$ sur p, $\vec{x}|_{\mathrm{Var}(N)} = a$.

On dit que $\vec{x}$ est un modèle de φ si $\mathrm{I}(\varphi)(\vec{x}) = \top$; $\mathrm{Mod}(\varphi)$ désigne l'ensemble des modèles de φ. Un MDD φ est dit *équivalent* à un autre MDD ψ (ce que l'on note $\varphi \equiv \psi$) si et seulement si $\mathrm{Mod}(\varphi) = \mathrm{Mod}(\psi)$.

Les travaux existants relatifs à la compilation de CSPs en diagrammes de décision supposent que ceux-ci sont déterministes et ordonnés [14, 5].

Définition 3 (MDD déterministe)**.** Un nœud N d'un MDD est *déterministe* si et seulement si les valeurs étiquetant les arcs sortant de N sont distinctes deux à deux. Un *MDD déterministe* (dMDD) est un MDD ne contenant que des nœuds déterministes.

Définition 4 (MDD ordonné)**.** Soit $<$ un ordre total sur X. Un MDD est dit *ordonné suivant* $<$ si et seulement si tout couple de nœuds $\langle N, M \rangle$ tel que N est un ancêtre de M vérifie $\mathrm{Var}(N) < \mathrm{Var}(M)$.

Dans les sections suivantes, nous étudions l'influence du déterminisme sur (i) la concision relative des formes compilées, et (ii) les performances de la forme compilée pour différentes opérations. Pour ce faire, six langages sont considérés.

Définition 5 (La famille de MDD)**.** On définit les langages suivants :

- MDD est le langage[2] de tous les diagrammes de décision multivalués sur X ;
- dMDD est le langage de tous les MDDs déterministes ;
- $\mathtt{OMDD}_<$ est le langage de tous les MDDs ordonnés suivant $<$ (ces MDDs pouvant être non-déterministes) ;
- OMDD est l'union de tous les langages $\mathtt{OMDD}_<$;
- $\mathtt{dOMDD}_<$ est le langage de tous les MDDs déterministes et ordonnés suivant $<$;
- dOMDD est l'union de tous les langages $\mathtt{dOMDD}_<$.

Il est clair que $\mathtt{dOMDD}_< \subseteq \mathtt{dOMDD} \subseteq \mathtt{dMDD} \subseteq \mathtt{MDD}$, que $\mathtt{dOMDD}_< \subseteq \mathtt{OMDD}_< \subseteq \mathtt{OMDD} \subseteq \mathtt{MDD}$, et que $\mathtt{dOMDD} \subseteq \mathtt{OMDD}$.

De la même manière que pour les diagrammes de décision binaires, nous considérons que les MDDs sont *réduits*, c'est-à-dire que (i) les nœuds isomorphes (étiquetés par la même variable et pointant vers les mêmes fils pour les mêmes valeurs) ont été fusionnés, (ii) les nœuds redondants (ayant un unique fils et un arc sortant par valeur dans le domaine de la variable) ont été éliminés, et (iii) les nœuds sans successeur (sauf le puits) ont été retirés. Supposer les MDDs réduits est « indolore », car la réduction peut être effectuée en temps polynomial en la taille du graphe.

Proposition 6 (Réduction)**.** *Soit* L *un langage parmi* $\{\mathtt{dOMDD}_<, \mathtt{OMDD}_<, \mathtt{dOMDD}, \mathtt{OMDD}, \mathtt{dMDD}, \mathtt{MDD}\}$*. Il existe un algorithme polynomial transformant tout* φ *de* L *en un graphe réduit équivalent* φ' *de* L *vérifiant* $\|\varphi'\| \leq \|\varphi\|$*.*

3 Les diagrammes de décision dans la carte de NNF

Pour souligner la relation entre les MDDs et la carte de compilation de NNF [7], considérons le cas booléen. Pour tout sous-langage L de MDD, on note $\mathtt{L}^{\mathfrak{B}}$ le sous-langage de L obtenu en le restreignant aux variables booléennes. Les langages $\mathtt{dMDD}^{\mathfrak{B}}$, $\mathtt{dOMDD}^{\mathfrak{B}}$ et $\mathtt{dOMDD}^{\mathfrak{B}}_<$ correspondent respectivement à BDD, OBDD et $\mathtt{OBDD}_<$, moyennant une transformation linéaire.

Définition 7. Un sous-langage $\mathtt{L}_2$ de MDD est *polynomialement traduisible* (resp. *linéairement traduisible*) en un autre sous-langage $\mathtt{L}_1$ de MDD, ce que l'on note $\mathtt{L}_1 \leq_{\mathcal{P}} \mathtt{L}_2$ (resp. $\mathtt{L}_1 \leq_{\mathcal{L}} \mathtt{L}_2$), si et seulement s'il existe un algorithme $A_{\mathtt{L}_2 \mapsto \mathtt{L}_1}$ associant en temps polynomial (resp. linéaire) à tout élément de $\mathtt{L}_2$ un élément de $\mathtt{L}_1$ équivalent.

Quand les traductions $A_{\mathtt{L}_1 \mapsto \mathtt{L}_2}$ et $A_{\mathtt{L}_2 \mapsto \mathtt{L}_1}$ sont stables (c'est-à-dire quand $A_{\mathtt{L}_1 \mapsto \mathtt{L}_2} = A^{-1}_{\mathtt{L}_2 \mapsto \mathtt{L}_1}$) et linéaires, on dit que $\mathtt{L}_1$ et $\mathtt{L}_2$ sont linéairement équivalents, et on note $\mathtt{L}_1 \equiv_{\mathcal{L}} \mathtt{L}_2$.

Définition 8 (Concision)**.** Un sous-langage $\mathtt{L}_1$ de MDD est *au moins aussi concis* qu'un autre sous-langage $\mathtt{L}_2$ de MDD (ce que l'on note $\mathtt{L}_1 \leq_s \mathtt{L}_2$) si et seulement s'il existe un polynôme $P(\cdot)$ tel que pour tout élément

2. Un *langage* est un ensemble de graphes, muni d'une fonction d'interprétation et d'une fonction de taille. Il faudrait écrire $\mathtt{MDD}_X$, mais nous omettons l'indice car il n'y a jamais d'ambiguïté sur X. Un élément du langage MDD est un MDD (noter la différence de police).

φ de $\mathtt{L}_2$, il existe un élément équivalent ψ de $\mathtt{L}_1$ qui vérifie $\|\psi\| \leq P(\|\varphi\|)$.

La relation $\leq_s$ est un préordre (partiel). On note $\sim_s$ sa partie symétrique, et $<_s$ sa partie asymétrique. Il est clair que $\mathtt{L}_1 \leq_{\mathcal{L}} \mathtt{L}_2 \Rightarrow \mathtt{L}_1 \leq_{\mathcal{P}} \mathtt{L}_2 \Rightarrow \mathtt{L}_1 \leq_s \mathtt{L}_2$.

Les langages BDD et $\mathtt{dMDD}^{\mathfrak{B}}$ sont linéairement équivalents : pour transformer un BDD en un MDD booléen déterministe, il suffit de retirer la feuille $\bot$ et d'appliquer la réduction. Pour transformer un MDD booléen déterministe en BDD, il suffit de lui adjoindre une feuille $\bot$, et d'ajouter à tout nœud n'ayant qu'un seul arc sortant E un deuxième arc sortant, pointant vers la feuille $\bot$ et étiqueté 0 si $\mathrm{Lbl}(E) = 1$, et 1 sinon.

Proposition 9. $\mathtt{dMDD}^{\mathfrak{B}} \equiv_{\mathcal{L}} \mathtt{BDD}$, $\mathtt{dOMDD}^{\mathfrak{B}} \equiv_{\mathcal{L}} \mathtt{OBDD}$ *et* $\mathtt{dOMDD}^{\mathfrak{B}}_{<} \equiv_{\mathcal{L}} \mathtt{OBDD}_{<}$.

Les diagrammes de décision multivalués capturent également des fragments hors de la famille de BDD. Ainsi, les DNFs peuvent être transformés en OMDDs en temps linéaire, bien que certaines DNFs ne puissent pas être représentés par des OMDDs de taille polynomiale. Cela implique que OMDD est strictement plus concis que DNF — on peut le voir comme un « sur-ensemble » strict de ce fragment.

Proposition 10. $\mathtt{OMDD}^{\mathfrak{B}}_{<} <_s \mathtt{DNF}$.

Enfin, $\mathtt{dOMDD}^{\mathfrak{B}} \subseteq \mathtt{d\text{-}DNNF}$ et $\mathtt{OMDD}^{\mathfrak{B}} \subseteq \mathtt{DNNF}$; de plus, $\mathtt{d\text{-}DNNF} <_s \mathtt{dOMDD}^{\mathfrak{B}}$ (puisque $\mathtt{dOMDD}^{\mathfrak{B}} \equiv_{\mathcal{L}} \mathtt{OBDD}$). La figure 2 résume nos résultats : $\mathtt{OMDD}^{\mathfrak{B}}$ apparaît comme un nouveau fragment de la carte de concision de NNF, situé en-dessous de DNNF et au-dessus de DNF et OBDD. La question de la concision relative de $\mathtt{OMDD}^{\mathfrak{B}}$ et DNNF, et plus généralement l'extension des diagrammes multivalués à des structures de type DNNF, est laissée à des travaux ultérieurs.

4 Carte de compilation de MDD

4.1 Concision

Les résultats de notre analyse de concision sont présentés dans le tableau 1 (voir aussi la figure 2 pour le cas booléen).

Proposition 11. *Les résultats du tableau 1 sont démontrés.*

Certains de ces résultats ne sont guère surprenants, et viennent directement du fait que dOMDD et $\mathtt{dOMDD}_{<}$ deviennent OBDD et $\mathtt{OBDD}_{<}$ quand les domaines sont booléens : $\mathtt{dOMDD}_{<} \not\leq_s \mathtt{dOMDD}$ est une conséquence directe de $\mathtt{OBDD}_{<} \not\leq_s \mathtt{OBDD}$. Certains autres résultats sont dérivés de la carte de NNF de manière moins immédiate. Par exemple, $\mathtt{dOMDD} \not\leq_s \mathtt{OMDD}$ (et donc $\mathtt{OMDD} <_s \mathtt{dOMDD}$) vient du fait que $\mathtt{OMDD}^{\mathfrak{B}} \leq_{\mathcal{P}} \mathtt{DNF}$ et $\mathtt{OBDD} \sim_s \mathtt{dOMDD}^{\mathfrak{B}}$: si $\mathtt{dOMDD} \leq_s \mathtt{OMDD}$ était vrai, on pourrait en déduire que $\mathtt{OBDD} \leq_s \mathtt{DNF}$, ce qui est faux [7].

L	MDD	dMDD	OMDD	dOMDD	$\mathtt{OMDD}_{<}$	$\mathtt{dOMDD}_{<}$
MDD	$\leq_s$	$\leq_s$	$\leq_s$	$\leq_s$	$\leq_s$	$\leq_s$
dMDD	?	$\leq_s$	?	$\leq_s$	?	$\leq_s$
OMDD	$\not\leq_s$	$\not\leq_s$	$\leq_s$	$\leq_s$	$\leq_s$	$\leq_s$
dOMDD	$\not\leq_s$	$\not\leq_s$	$\not\leq_s$	$\leq_s$	$\not\leq_s$	$\leq_s$
$\mathtt{OMDD}_{<}$	$\not\leq_s$	$\not\leq_s$	$\not\leq_s$	$\not\leq_s$	$\leq_s$	$\leq_s$
$\mathtt{dOMDD}_{<}$	$\not\leq_s$	$\not\leq_s$	$\not\leq_s$	$\not\leq_s$	$\not\leq_s$	$\leq_s$

TABLE 1 – Résultats sur la concision.

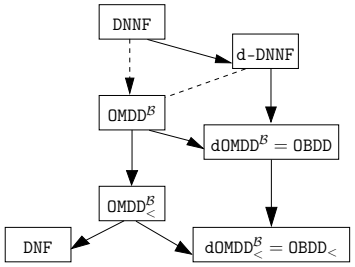


FIGURE 2 – $\mathtt{OMDD}^{\mathfrak{B}}$ et ses sous-langages dans la carte de concision de DNNF. Un arc $\mathtt{L}_1 \to \mathtt{L}_2$ indique que $\mathtt{L}_1$ est strictement plus concis que $\mathtt{L}_2$. Les arcs en pointillés indiquent des résultats incomplets.

D'autres résultats peuvent être obtenus indépendamment. Par exemple, pour prouver que $\mathtt{OMDD} \not\leq_s \mathtt{MDD}$, on peut passer par le problème de la n-coloration d'une clique de n nœuds. On peut montrer que l'ensemble de solutions S de ce problème peut être représenté par un MDD de taille polynomiale en n ; cependant, tout dOMDD ou OMDD représentant S est de taille exponentielle en n. La figure 1 donne une idée de l'explosion en espace sur une petite instance.

On montre que $\mathtt{dOMDD}_{<} \not\leq_s \mathtt{OMDD}_{<}$ en considérant un autre CSP, celui du problème de la n-coloration d'un graphe en étoile avec n nœuds (voir la figure 3 pour un exemple où $n = 3$). Soit x_1 la variable centrale, et soit l'ordre de variables $x_n < \cdots < x_2 < x_1$: le $\mathrm{dOMDD}_{<}$ représentant ce problème contient au moins 2^n nœuds et $n \cdot 2^{n-1}$ arcs, tandis qu'on peut montrer que ce CSP admet une représentation en $\mathrm{OMDD}_{<}$ (suivant le même ordre $<$) dont la taille est polynomiale en n.

4.2 Requêtes et transformations

Comme l'ont souligné Darwiche et Marquis [7], évaluer l'adéquation d'un langage-cible de compilation à une application donnée revient à comparer sa concision à l'ensemble d'opérations qu'il supporte en temps

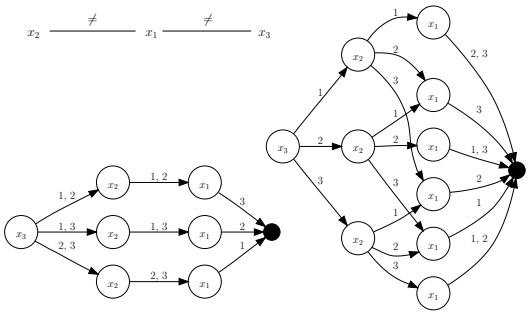


FIGURE 3 – Le problème de la coloration sur un graphe en étoile (3 variables, 3 couleurs), ainsi qu'un OMDD et un dOMDD représentant son ensemble de solutions (pour l'ordre $x_3 < x_2 < x_1$).

polynomial. Les opérations identifiées par Darwiche et Marquis sont orientées vers des applications de gestion de connaissances, mais la plupart d'entre elles ont également du sens dans certaines applications visées par le cadre CSP. Nous enrichissons cet ensemble de quelques nouvelles opérations, nécessaires pour des applications plus orientées décision.

- Les opérations les plus courantes sont celles consistant à tester la cohérence du CSP (on la note **CO**), à extraire une ou toutes les solutions (on note ces requêtes **MX** et **ME**), et à compter (**CT**) le nombre de solutions.
- La requête d'« extraction de contexte » (**CX**) vise à fournir à l'utilisateur toutes les valeurs possibles d'une variable donnée.
- L'opération de « conditionnement » (**CD**) assigne des valeurs à certaines variables ; plus généralement, la « restriction à un terme » (**TR**) restreint les valeurs possibles de certaines variables à un sous-ensemble de leur domaine. Les opérations **TR**, **CD** et **CX** sont souvent utilisées en séquence dans la résolution interactive de CSPs, au cours de laquelle l'utilisateur examine itérativement les valeurs possibles de certaines variables puis restreint leur domaine selon ses préférences [4].
- L'« implication clausale » (**CE**) provient de problèmes de raisonnement : elle consiste à déterminer si toutes les solutions d'un problème satisfont une disjonction de conditions élémentaires (contraintes unaires). Les problèmes de raisonnement en IA nécessitent également d'autres opérations, comme **VA** (la formule est-elle valide ?) et $\neg$**C** (construire la négation d'une formule).
- L'équivalence (**EQ**) et l'implication (**IM**) viennent du *model-checking*. Elles peuvent être utiles pour des problèmes de modélisation en CSP : **IM** revient à vérifier si un ensemble de contraintes est une relaxation sémantique d'un autre (c'est-à-dire, si les instanciations satisfaisant le second satisfont aussi le premier) ; **EQ** est le test d'équivalence.
- Les applications de modélisation interactive peuvent aussi nécessiter la manipulation de contraintes compilées, par exemple leur conjonction ($\wedge$**C**), leur disjonction ($\vee$**C**) ou leur négation ($\neg$**C**), de manière à construire des contraintes composées à partir de quelques opérateurs. Le processus de modélisation peut encore s'appuyer sur d'autres opérations pour vérifier la contrainte résultante, par exemple en la projetant (via **CX**) sur une de ses variables.
- L'opération d'« oubli » (**FO**) permet d'éliminer des variables (intermédiaires) du problème — elle revient à une projection existentielle. Son opération duale, l'« enforcement » (**EN**), effectue l'élimination universelle de variables. Ces deux opérations sont notamment utilisées dans la planification basée sur la compilation, comme le paradigme de *planning as model-checking* [8].

Toutes ces opérations sont souvent appliquées en séquence. Le conditionnement (**CD**) suivi d'une extraction de modèle (**MX**) peut s'avérer utile pour la décision dans l'incertain : si on dispose de la forme compilée d'une politique de décision π, associant des décisions à chaque état possible, alors **CD** permet de conditionner π suivant l'état courant, et **MX** de récupérer une décision appropriée pour cet état. Les applications de diagnostic en ligne constituent un autre exemple : une fois que le modèle (compilé) du système a été conditionné par les observations avec **CD**, **CX** peut fournir les différents modes d'échec de chaque composant. Des hypothèses d'échec plus complexes peuvent être testées via l'implication clausale (**CE**).

Plus généralement, les opérations susmentionnées peuvent être divisées en deux catégories, celle des transformations et celle des requêtes. Les transformations prennent en entrée un problème (compilé) et en retournent un autre (par exemple, conditionner un CSP par une instanciation produit un CSP avec moins de variables). Les requêtes ne modifient pas le problème, mais répondent simplement à une question (**CO**, **CT** et **CE** font ainsi partie des requêtes).

Formalisons à présent ces opérations ; on les définit comme des propriétés qu'un langage de compilation peut ou non satisfaire. Les requêtes et transformations qui sont des généralisations directes du cas booléen [7] ne sont pas définies explicitement.

Définition 12 (Requêtes)**.** Soit `L` un sous-langage de `MDD`.

L	CO	VA	MC	CE	IM	EQ	SE	MX	CX	CT	ME
MDD	○	○	√	○	○	○	○	○	○	○	○
dMDD	○	○	√	○	○	○	○	○	○	○	○
OMDD	√	○	√	√	○	○	○	√	√	○	√
dOMDD	√	√	√	√	√	√	○	√	√	√	√
$\text{OMDD}_<$	√	○	√	√	○	○	○	√	√	○	√
$\text{dOMDD}_<$	√	√	√	√	√	√	√	√	√	√	√
DNF	√	○	√	√	○	○	○	√	√	○	√
DNNF	√	○	√	√	○	○	○	√	√	○	√
d-DNNF	√	√	√	√	√	?	○	√	√	√	√
OBDD	√	√	√	√	√	√	○	√	√	√	√
$\text{OBDD}_<$	√	√	√	√	√	√	√	√	√	√	√

L	CD	TR	FO	SFO	EN	SEN	$\vee$C	$\vee$BC	$\wedge$C	$\wedge$BC	$\neg$C
MDD	√	○	○	√	○	√	√	√	√	√	?
dMDD	√	○	○	√	○	√	√	√	√	√	√
OMDD	√	√	√	√	○	○	[○]	[○]	○	○	○
dOMDD	√	•	•	•	•	•	•	○	•	○	√
$\text{OMDD}_<$	√	√	√	√	○	○	√	√	○	√	○
$\text{dOMDD}_<$	√	•	•	•	•	•	•	√	•	√	√
DNF	√	√	√	√	○	√	√	√	•	√	•
DNNF	√	√	√	√	○	○	√	√	○	○	○
d-DNNF	√	√	○	○	○	○	○	○	○	○	?
OBDD	√	•	•	√	•	√	•	○	•	○	√
$\text{OBDD}_<$	√	•	•	√	•	√	•	√	•	√	√

TABLE 2 – Résultats sur les requêtes et les transformations ; √ signifie « satisfait », • signifie « ne satisfait pas », et ○ signifie « ne satisfait pas, sauf si $\mathsf{P} = \mathsf{NP}$ ». Les crochets [·] désignent une conjecture. La plupart des résultats pour DNF, DNNF, d-DNNF, OBDD et $\text{OBDD}_<$ viennent de la carte de NNF et sont donnés à titre de comparaison.

– L satisfait **CE** (resp. **IM**) si et seulement s'il existe un polynôme $P(\cdot,\cdot)$ et un algorithme associant à tout MDD φ dans L, tout ensemble de variables $\{y_1,\ldots,y_k\} \subseteq \text{Var}(\varphi)$ et toute suite $(A_1,\ldots,A_k)$ d'ensembles finis d'entiers, 1 si $\text{I}(\varphi) \models [y_1 \in A_1] \vee \cdots \vee [y_k \in A_k]$ (resp. $[y_1 \in A_1] \wedge \cdots \wedge [y_k \in A_k] \models \text{I}(\varphi)$), et 0 sinon, en un temps borné par $P(\|\varphi\|, n_A)$, où $n_A = \max_{1\leq i\leq k}|A_i|$.

– L satisfait **MC** si et seulement s'il existe un algorithme polynomial associant à tout MDD φ dans L et toute $\text{Var}(\varphi)$-instanciation $\vec{x}$, 1 si $\vec{x}$ est un modèle de φ, et 0 sinon.

– L satisfait **MX** si et seulement s'il existe un algorithme polynomial associant à tout MDD φ dans L un modèle de φ s'il en existe un, et s'arrêtant sans rien retourner dans le cas contraire.

– L satisfait **CX** si et seulement s'il existe un algorithme polynomial associant à tout φ dans L et tout $y \in \text{Var}(\varphi)$, l'ensemble de toutes les valeurs prises par y dans au moins un modèle de φ.

Avant de définir les transformations, présentons les opérations sémantiques sur lesquelles elles sont basées.

Définition 13. Soient I et J les fonctions d'interprétation, sur Var(I) et Var(J), de deux MDDs.

– La projection existentielle de I sur $Y \subseteq \text{Var(I)}$ est la fonction $\text{I}^{\downarrow Y}$ sur les variables de Y définie comme suit : $\text{I}^{\downarrow Y}(\vec{y}) = \top$ si et seulement s'il existe une Z-instanciation $\vec{z}$ (avec $Z = \text{Var(I)} \setminus Y$) telle que $\text{I}(\vec{z}\,.\,\vec{y}) = \top$. L'opération d'« oubli » est son dual : $\text{Forget}(\text{I}, Y) = \text{I}^{\downarrow \text{Var(I)}\setminus Y}$.

– La projection universelle de I sur $Y \subseteq \text{Var(I)}$ est la fonction $\text{I}^{\Downarrow Y}$ sur les variables de Y définie comme suit : $\text{I}^{\Downarrow Y}(\vec{y}) = \top$ si et seulement si pour toute Z-instanciation $\vec{z}$ (avec $Z = \text{Var(I)} \setminus Y$), $\text{I}(\vec{z}\,.\,\vec{y}) = \top$. L'opération d'« enforcement » est sa duale : $\text{Ensure}(\text{I}, Y) = \text{I}^{\Downarrow \text{Var(I)}\setminus Y}$.

– La restriction de I à J, notée $\text{I}|_\text{J}$, est définie par $\text{I}|_\text{J} = \text{Forget}(\text{I} \wedge \text{J}, \text{Var(J)})$.

– Pour une instanciation $\vec{y}$ d'un ensemble de variables $Y \subseteq \text{Var(I)}$, le conditionnement de I par $\vec{y}$ est la fonction $\text{I}|_{\vec{y}}$ sur les variables de $Z = \text{Var(I)} \setminus Y$ définie par $\text{I}|_{\vec{y}}(\vec{z}) = \text{I}(\vec{y}\,.\,\vec{z})$.

Définition 14 (Transformations)**.** Soit L un sous-langage de MDD.

– L satisfait **FO** (resp. **EN**) si et seulement s'il existe un algorithme polynomial associant à tout MDD φ dans L et tout $Y \subseteq \text{Var}(\varphi)$, un MDD φ' dans L vérifiant $\text{I}(\varphi') = \text{Forget}(\text{I}(\varphi), Y)$ (resp. $\text{I}(\varphi') = \text{Ensure}(\text{I}(\varphi), Y)$).

– L satisfait **SFO** (resp. **SEN**) si et seulement s'il satisfait **FO** (resp. **EN**) pour une seule variable (c'est-à-dire quand $|Y| = 1$).

– L satisfait **TR** si et seulement s'il existe un polynôme $P(\cdot,\cdot)$ et un algorithme associant à tout MDD φ dans L, tout ensemble de variables $\{y_1,\ldots,y_k\} \subseteq \text{Var}(\varphi)$ et toute suite $(A_1,\ldots,A_k)$ d'ensembles finis d'entiers, un MDD φ' dans L vérifiant $\text{I}(\varphi') = \text{I}(\varphi)|_{[y_1\in A_1]\wedge\cdots\wedge[y_k\in A_k]}$ en temps borné par $P(\|\varphi\|, n_A)$, où $n_A = \max_{1\leq i\leq k}|A_i|$.

– L satisfait **CD** si et seulement s'il existe un algorithme polynomial associant à tout MDD φ dans L et toute instanciation $\vec{y}$ de $Y \subseteq \text{Var}(\varphi)$, un MDD φ' dans L vérifiant $\text{I}(\varphi') = \text{I}(\varphi)|_{\vec{y}}$.

Les résultats de notre analyse de complexité de ces opérations sont présentés dans le tableau 2.

Proposition 15. *Les résultats du tableau 2 sont démontrés.*

En ce qui concerne dMDD, dOMDD et $\texttt{dOMDD}_<$, les résultats sont généralement connus ou viennent directement de travaux antérieurs [11, 10, 7]. Les dOMDDs ont presque les mêmes capacités que les OBDDs, ce qui n'est pas surprenant vu leur forte proximité. Cependant, il est notable que l'oubli et l'enforcement simples (**SFO**, **SEN**), bien que satisfaits par OBDD, ne le sont pas par dOMDD ; cela est dû au fait que les tailles des domaines ne sont pas bornées.

La proposition 15 met en avant l'attractivité des OMDDs *non-déterministes* au niveau des transformations : $\texttt{OMDD}_<$ satisfait les mêmes transformations que DNF à part **SEN**, et plus de transformations que $\texttt{dOMDD}_<$ à part la négation, tout en étant strictement plus concis que ces deux langages.

En ce qui concerne les performances sur les requêtes, OMDD et $\texttt{OMDD}_<$ sont moins intéressants que leur version déterministe ; malgré tout, on ne perd « que » **CT**, **VA**, **EQ**, **IM** et **SE** en relâchant le déterminisme (à noter que les mêmes requêtes sont perdues quand on passe de DNNF à d-DNNF). Ces langages gardent donc tout leur intérêt pour de nombreuses applications ne s'appuyant pas sur ces requêtes, comme la planification, où l'on a besoin de tester la cohérence, d'oublier des variables et d'extraire des modèles, ou la configuration interactive, qui nécessite essentiellement le conditionnement et l'extraction de contexte.

5 Résultats expérimentaux

Le tableau 3 rapporte quelques résultats [1] obtenus sur des CSPs générés aléatoirement contenant 15 variables de domaines de taille 6. Pour chaque ligne, 50 problèmes aléatoires ont été générés ; pour chaque problème, l'ordre des variables utilisé dans les formes compilées a été construit grâce à l'heuristique MCSInv [12], qui choisit itérativement la variable connectée au plus grand nombre de variables restantes dans le graphe de contraintes. Le compilateur de dOMDDs utilisé suit l'approche *bottom-up* [14] : chaque contrainte est compilée en un dOMDD, et les dOMDDs élémentaires ainsi obtenus sont alors combinés par conjonction. La compilation continue par l'application d'opérations de compactage, plus puissantes que la fusion des nœuds isomorphes, mais ne préservant pas le déterminisme [3] ; on obtient finalement un OMDD.

Il apparaît que l'intérêt des structures non-déterministes ne se limite pas à quelques problèmes bien spécifiques comme ceux utilisés dans les preuves : même sur des CSPs aléatoires, autoriser des opérations de compactage non-déterministes peut parfois diviser par 2 ou plus le nombre de nœuds dans le graphe.

%T	%C	#SOL	#N OMDD	#N dOMDD
70	10	290 888 073	80	81
	20	136 056 826	1338	1558
	30	5 006 576	5662	8132
	40	95 131	3315	5005
	50	2367	737	897
80	10	391 615 179	79	80
	20	1 581 648 506	2572	2932
	30	189 551 100	12 223	16 370
	40	11 557 737	20 501	35 486
	50	1 035 884	13 815	25 240
	60	70 185	5776	9253
	70	4662	1719	2246
	80	229	54	401

TABLE 3 – Résultats pour des CSPs binaires générés aléatoirement (15 variables, domaines de taille 6). %T désigne le pourcentage de tuples satisfaisant chaque contrainte, %C la densité du graphe de contraintes, #SOL le nombre de solutions du CSP, et #N le nombre de nœuds dans la forme compilée.

6 Conclusion

Autant les résultats théoriques de complexité que les expérimentations montrent que relâcher la contrainte du déterminisme dans les diagrammes de décision ordonnés peut améliorer la concision. Relâcher le déterminisme permet également à plus de transformations d'être traitables en temps polynomial : typiquement, toutes les transformations satisfaites par DNF le sont aussi par $\texttt{OMDD}_<$ (sauf **SEN**, qui dépend de la taille du domaine). Cela inclut l'oubli et la disjonction, qui ne sont pas satisfaites par les langages déterministes. Le prix à payer quand on s'affranchit du déterminisme est la perte de la transformation de négation et des requêtes de comptage, validité, équivalence et implication (il est à noter que les mêmes opérations sont perdues quand on relâche le déterminisme sur les DNNFs). Par conséquent, $\texttt{OMDD}_<$ est particulièrement intéressant pour les applications s'appuyant sur des transformations ($\neg$**C** exceptée) et des requêtes basiques de cohérence (**CO**, **MX**, **ME**, **CX**), comme c'est le cas en planification et en configuration interactive. D'un point de vue théorique, il est établi que, restreint à des domaines booléens, $\texttt{OMDD}_<$ est un nouveau fragment dans la carte de NNF, situé sous DNNF et au-dessus de DNF et OBDD. De plus, $\texttt{OMDD}_<$ satisfait plus de requêtes et de transformations que DNNF.

Une prochaine étape est l'introduction de nœuds « et » décomposables dans le cadre MDD. Cela permettrait de capturer les graphes AND/OR, et de définir de nouveaux fragments, tels les graphes AND/OR non-déterministes.

Références

[1] Jérôme Amilhastre. *Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes*. PhD thesis, Université Montpellier II, 1999.

[2] Jérôme Amilhastre, Hélène Fargier, Alexandre Niveau, and Cédric Pralet. Compiling CSPs : A complexity map of (non-deterministic) multivalued decision diagrams. In *ICTAI*, pages 1–8, 2012.

[3] Jérôme Amilhastre, Philippe Janssen, and Marie-Catherine Vilarem. FA minimisation heuristics for a class of finite languages. In *International Workshop on Implementing Automata (WIA)*, pages 1–12, 1999.

[4] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs — application to configuration. *Artif. Intell. J.*, 135(1–2) :199–234, 2002.

[5] Henrik Reif Andersen, Tarik Hadzic, John N. Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In *CP*, pages 118–132, 2007.

[6] Hadrien Cambazard, Tarik Hadzic, and Barry O'Sullivan. Knowledge compilation for itemset mining. In *ECAI*, pages 1109–1110, 2010.

[7] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *JAIR*, 17 :229–264, 2002.

[8] Fausto Giunchiglia and Paolo Traverso. Planning as model checking. In *ECP*, pages 1–20, 1999.

[9] Jesse Hoey, Robert St-Aubin, Alan J. Hu, and Craig Boutilier. SPUDD : Stochastic planning using decision diagrams. In *UAI*, pages 279–288, 1999.

[10] Timothy Kam, Tiziano Villa, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Multivalued Decision Diagrams : Theory and Applications. *Multiple-Valued Logic*, 4(1–2) :9–62, 1998.

[11] Arvind Srinivasan, Timothy Kam, Sharad Malik, and Robert K. Brayton. Algorithms for discrete function manipulation. In *ICCAD*, pages 92–95, November 1990.

[12] Robert Endre Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. on Computing*, 13(3) :566–579, 1984.

[13] Pietro Torasso and Gianluca Torta. Model-based diagnosis through OBDD compilation : A complexity analysis. In *Reasoning, Action and Interaction in AI Theories and Systems*, pages 287–305, 2006.

[14] Nageshwara Rao Vempaty. Solving constraint satisfaction problems using finite state automata. In *AAAI*, pages 453–458, 1992.

A Preuves

Cette annexe rassemble les preuves les plus significatives. Les autres sont soit directes, soit héritées du cadre des BDDs ; elles se trouvent dans l'article original en anglais [2], disponible en ligne à l'adresse `ftp://ftp.irit.fr/IRIT/ADRIA/PapersFargier/ICTAI12-AFNP-long.pdf`.

Comme c'est souvent le cas dans les cartes de compilation, de nombreuses preuves tirent parti du fait que tous les langages présentés permettent de représenter termes et clauses booléennes en temps et en espace linéaires. La preuve est similaire au cas booléen [7].

Lemme 16. *Étant donné un ordre $<$ sur les variables, tout terme et toute clause de la logique propositionelle peut être mis sous la forme d'un dOMDD$^{\mathfrak{B}}_{<}$ en temps linéaire.*

A.1 Preuves de concision (proposition 11)

• $\texttt{OMDD}_< \not\leq_s \texttt{dOMDD}$. On considère le CSP Π portant sur $2n$ variables booléennes partitionnées en deux ensembles $Y = \{y_1, \ldots, y_n\}$ et $Z = \{z_1, \ldots, z_n\}$, ayant pour ensemble de contraintes $C = \{y_i = z_i, 1 \leq i \leq n\}$. Soient les deux ordres $y_1 <^a z_1 <^a y_2 <^a z_2 <^a \cdots <^a y_n <^a z_n$ et $y_1 <^b y_2 <^b \cdots <^b y_n <^b z_1 <^b \cdots <^b z_n$.

Le CSP Π a une représentation en $\texttt{dOMDD}_{<^a}$ de taille polynomiale en n, que l'on note φ_a. Montrons que la taille de tout $\text{OMDD}_{<^b}$ représentant $\text{Sol}(\Pi)$ est exponentielle en n.

Soit φ_b un $\text{OMDD}_{<^b}$ représentant $\text{Sol}(\Pi)$. On considère un arc $E = \langle N, N', v\rangle$ de φ_b correspondant à l'instanciation d'une variable de Y (c'est-à-dire un arc dans la première moitié du graphe). Soit y_i la variable étiquetant N, et soit p le chemin $\langle N, N', v\rangle \rightsquigarrow \text{Sink}(\varphi)$. Sur ce chemin, la valeur de z_i doit être v ; par conséquent, φ_b ne peut pas contenir d'arc $\langle N'', N', u\rangle$ avec $\text{Var}(N'') = y_i$ et $u \neq v$. Pour cette raison, deux chemins de la racine à N' sont soit disjoints, soit associés à la même instanciation de $\{y_1, \ldots, y_i\}$. Chacune des instanciations possibles des variables de Y est représentée par (au moins) un chemin menant à un nœud z_1 distinct. Les instanciations des variables de Y n'étant pas contraintes, φ_b contient donc au moins 2^n nœuds au niveau z_1.

Au final, φ_a est un $\text{dOMDD}_{<^a}$ n'ayant aucune représentation en $\text{OMDD}_{<^b}$ de taille polynomiale.

• $\texttt{OMDD} \not\leq_s \texttt{dMDD}$. On considère le problème de la n-coloration d'une clique de n nœuds (ou, de manière équivalente, la contrainte « alldiff » sur n variables — voir la figure 1). Ce problème n'a aucune représentation de taille polynomiale en OMDD ; en effet, de manière similaire à la preuve précédente, on peut montrer qu'au niveau i, il doit y avoir autant de nœuds que de possibilités de choisir i valeurs dans $\{1, \ldots, n\}$ (s'il y en a moins, on peut construire un chemin sur lequel la même valeur est assignée à deux variables différentes, une fois avant le niveau i, et une fois après). Le nombre total de nœuds est donc supérieur à $\sum_{i=1}^{n} \binom{n}{i} = 2^n$.

Cependant, ce problème peut facilement être exprimé comme un dMDD de taille polynomiale, puisque chaque contrainte $x_i \neq x_j$ a une représentation en dMDD comportant $n(n-1)$ arcs, et dMDD satisfait $\wedge\mathbf{C}$ (voir la proposition 15).

• $\texttt{dOMDD} \not\leq_s \texttt{OMDD}_<$. La proposition 10 établit que $\texttt{OMDD}_<^{\mathfrak{B}} \leq_s \texttt{DNF}$. S'il était vrai que $\texttt{dOMDD} \leq_s \texttt{OMDD}_<$, et donc que $\texttt{dOMDD}^{\mathfrak{B}} \leq_s \texttt{OMDD}_<^{\mathfrak{B}}$, alors puisque $\texttt{OBDD} \equiv_{\mathcal{L}} \texttt{dOMDD}^{\mathfrak{B}}$, on aurait $\texttt{OBDD} \leq_s \texttt{DNF}$ — ce qui est faux [7].

• Tous les autres résultats sont obtenus par transitivité et inclusion entre langages.

A.2 Preuves des requêtes (proposition 15)

• **MDD et ses sous-langages satisfont MC.** Tous ces langages satisfaisant **CD** (voir la section A.3), on peut vérifier en temps polynomial si $\vec{x}$ est un modèle de φ en conditionnant le MDD par $\vec{x}$; en effet, on obtient soit le MDD vide (l'instanciation n'est alors pas un modèle) soit le MDD puits (alors $\vec{x}$ est un modèle). Par conséquent, tous ces langages satisfont **MC**.

• **Requêtes sur dMDD et MDD.** BDD est un sous-langage de dMDD et MDD, donc dMDD et MDD ne satisfont pas les autres requêtes considérées, sauf si P = NP.

• **OMDD et $\texttt{OMDD}_<$ ne satisfont pas VA, IM, EQ, SE, CT.** Le lemme 16 établit que tout terme peut être traduit dans $\texttt{OMDD}_<^{\mathfrak{B}}$ en temps linéaire, et $\texttt{OMDD}_<^{\mathfrak{B}}$ satisfait $\vee\mathbf{C}$ (proposition 15) ; ainsi $\texttt{OMDD}_<^{\mathfrak{B}} \leq_{\mathcal{L}} \texttt{DNF}$, donc $\texttt{OMDD}^{\mathfrak{B}} \leq_{\mathcal{L}} \texttt{DNF}$ (par inclusion), d'où le résultat, puisque DNF ne satisfait pas **VA**, **IM**, **EQ**, **SE** ou **CT** sauf si P = NP.

• **OMDD et $\texttt{OMDD}_<$ satisfont CO.** En effet, le seul OMDD réduit incohérent est l'OMDD vide.

• **OMDD et $\texttt{OMDD}_<$ satisfont CE.** Vérifier si φ implique $[x_1 \in A_1] \vee \cdots \vee [x_k \in A_k]$ revient à vérifier si $\varphi \wedge [x_1 \notin A_1] \wedge \cdots \wedge [x_k \notin A_k]$ est incohérent. Comme (i) $[x \notin A] \equiv [x \in \mathrm{Dom}(x) \setminus A]$, (ii) un « terme » peut être représenté en temps polynomial en un OMDD suivant le même ordre de variables que φ, et (iii) $\texttt{OMDD}_<$ satisfait $\wedge\mathbf{BC}$ (voir plus loin), OMDD satisfait **CE**.

Algorithme 1 Conditionnement d'un MDD φ par une instanciation $\vec{y}$ de $Y \subseteq \mathrm{Var}(\varphi)$.

```
Numéroter les nœuds de φ suivant < (si pertinent)
pour tout E = ⟨N, N', v⟩ tel que Var(N) ∈ Y faire
  si v ≠ y⃗|_{Var(N)} alors
    Retirer E de φ : Out(N) := Out(N) \ {E}
  sinon
    Étiqueter E avec ⊤
pour tout x_i ∈ Y, i allant de 1 à n faire
  pour tout N tel que Var(N) = x_i faire
    si N est racine alors
      Créer une nouvelle racine N' qui est la disjonction de tous les fils de N (si φ est déterministe, il n'y en a qu'un)
    sinon
      I := In(N), O := Out(N), E_new := ∅
      pour tout E_I = ⟨N_I, N, a⟩ ∈ I faire
        pour tout E_O = ⟨N, N_O, ⊤⟩ ∈ O faire
          Ajouter un arc E = ⟨N_I, N_O, a⟩ à E_new
      Retirer N, I et O du graphe
      Ajouter E_new au graphe
```

• **OMDD et $\texttt{OMDD}_<$ satisfont ME, MX, CX.** Cela vient du fait que OMDD satisfait **CD** et **CO**. Énumérer les modèles peut être fait en explorant l'arbre entier des instanciations, comme dans le cas booléen [7] (vérifier la cohérence après chaque branchement évite d'avoir à revenir en arrière — la taille de l'arbre de recherche est donc polynomiale en la taille de la sortie). L'extraction de modèle est un cas particulier de **ME**, et l'extraction de contexte peut se faire en vérifiant, pour chaque variable x_i et chaque valeur v de son domaine, si $\varphi|_{[x_i=v]}$ est cohérent.

A.3 Preuves des transformations (proposition 15)

Plusieurs preuves s'appuient sur le fait qu'un MDD peut être conditionné en temps polynomial, grâce à l'algorithme 1, qui effectue un conditionnement syntaxique de φ par une instanciation $\vec{y}$ de $Y \subseteq X$. On note $\varphi|_{\vec{y}}$ le MDD construit par cet algorithme.

• **MDD et ses sous-langages satisfont CD.** On peut montrer que l'algorithme 1 :

- préserve le déterminisme et l'ordre des variables ;
- est en temps polynomial ;
- est tel que pour tout MDD φ, $\mathrm{I}(\varphi|_{\vec{y}}) = \mathrm{I}(\varphi)|_{\vec{y}}$.

Seule la preuve de complexité est détaillée ici ; on montre que l'algorithme 1 est en temps polynomial. En effet, la première boucle visite chaque arc au plus une fois. La seconde boucle, quant à elle, visite chaque nœud N une fois, retire $|\mathrm{In}(N)| + |\mathrm{Out}(N)|$ arcs et en

Algorithme 2 Étant donné un dMDD φ, construit un dMDD Compl(φ).

1: Soit ψ le MDD puits
2: **pour tout** nœud N de φ, ordonné du puits à la racine, en excluant le puits **faire**
3: Créer un nœud N' de même variable x que N
4: $U := \{v \in \mathrm{Dom}(x) : \forall E \in \mathrm{Out}(N), \mathrm{Lbl}(E) \neq v\}$
5: **pour tout** $v \in U$ **faire**
6: Ajouter à N' un arc sortant $\langle N', \mathrm{Sink}(\psi), v\rangle$
7: **pour tout** arc $\langle N, D, v\rangle$ sortant de N **faire**
8: **si** D correspond à un nœud D' de ψ **alors**
9: Ajouter à N' un arc sortant $\langle N', D', v\rangle$
10: **si** N' a au moins un arc sortant **alors**
11: Ajouter N' à ψ
12: **renvoyer** ψ

crée au plus $|\mathrm{In}(N)| \times |\mathrm{Out}(N)|$, étiquetés seulement avec des valeurs mentionnées dans φ : le nombre d'arcs sortants est donc borné par $|\mathrm{Edges}(\varphi)| \times |\mathrm{Nodes}(\varphi)|$ (au pire cas, il y a un arc pour chaque valeur et chaque nœud). La complexité de l'opération est donc polynomiale ; et puisqu'aucun nœud n'est créé, c'est aussi le cas pour la procédure toute entière.

• **dMDD, dOMDD et dOMDD$_<$ satisfont ¬C.** On peut montrer (on omet ici la preuve du dernier point) que l'algorithme 2 :

– préserve le déterminisme et l'ordre des variables ;
– est polynomial en $\|\varphi\|$;
– est tel que pour tout dMDD φ, Compl(φ) $\equiv \neg\varphi$.

Ordre : Cela vient du fait que (i) aucun « nouveau » nœud n'est ajouté, et (ii) les seuls arcs ajoutés pointent vers le puits.

Déterminisme : Cela vient du fait qu'aucun « nouveau » nœud n'est ajouté, et que les seuls arcs ajoutés sont disjoints avec les autres arcs qui sortent de leur nœud d'origine.

Complexité : Chaque nœud N est rencontré une fois, et pour chacun, toutes les valeurs de $\mathrm{Dom}(\mathrm{Var}(N))$ sont considérées ; on ajoute au plus un arc par valeur. La complexité temporelle et spatiale est donc en $\mathrm{O}(d \times |\mathrm{Nodes}(\varphi)|)$, où $d = \max_{x \in X} |\mathrm{Dom}(x)|$, et est donc polynomiale en $\|\varphi\|$, puisque d et $|\mathrm{Nodes}(\varphi)|$ sont tous deux bornés par $\|\varphi\|$.

• **OMDD$_<$ et OMDD ne satisfont pas ¬C.** La négation d'une CNF est une DNF, que l'on peut traduire en un OMDD suivant n'importe quel ordre en temps linéaire (proposition 10). Ainsi, si OMDD$_<$ ou OMDD satisfaisait **¬C**, on pourrait transformer toute CNF en un OMDD équivalent en temps polynomial ; comme OMDD$_<$ et OMDD satisfont **CO**, on aurait alors un algorithme polynomial décidant si une CNF est cohérente, ce qui est impossible sauf si P = NP.

• **OMDD$_<$ satisfait ∨C.** Pour disjoindre k OMDDs $\varphi_1, \ldots, \varphi_k$ de même ordre $<$, il faut s'assurer que les racines sont étiquetées par la même variable (sinon, il suffit d'en ajouter une nouvelle, avec un arc par valeur dans le domaine, pointant vers l'ancienne racine), puis fusionner toutes les racines en une.

• **Oubli et enforcement.** Excepté les trois preuves suivantes, les résultats peuvent être adaptés du cas booléen, en utilisant par exemple une décomposition de Shannon pour **SEN** et **SFO**, et les relations entre transformations pour **CO** ou **VA**.

• **OMDD$_<$ et OMDD satisfont FO, SFO.** L'algorithme 1 peut être modifié pour effectuer un oubli syntaxique des variables de $Y \subseteq X$ dans φ au lieu d'un conditionnement : à la place des instructions des lignes 3 à 6, il suffit d'étiqueter E par $\top$. L'algorithme modifié remplace alors, pour chaque N tel que $\mathrm{Var}(N) \in Y$, tout chemin $\langle N_1, N, u\rangle \rightsquigarrow \langle N, N_2, \top\rangle$ par un raccourci $\langle N_1, N_2, u\rangle$. La preuve est similaire à celle de **CD**. Il est cependant à noter que cet algorithme ne préserve pas le déterminisme.

• **dOMDD et dOMDD$_<$ ne satisfont pas SFO, FO.** Supposons que dOMDD (resp. dOMDD$_<$) satisfasse **SFO**. On pourrait alors obtenir en temps polynomial un dOMDD (resp. un dOMDD$_<$) représentant la disjonction d'un nombre arbitraire de dOMDDs$_<$, en les joignant grâce à une nouvelle racine étiquetée par une nouvelle variable, puis en oubliant cette variable. Comme tout terme a une représentation polynomiale dans dOMDD$_<$ (lemme 16), on pourrait construire un dOMDD (resp. dOMDD$_<$) de taille polynomiale équivalent à n'importe quelle DNF. Cependant, c'est impossible, puisque OBDD $\not\leq_s$ DNF (resp. OBDD$_<$ $\not\leq_s$ DNF).

• **OMDD et OMDD$_<$ ne satisfont pas SEN.** La même technique permet de montrer que si OMDD ou OMDD$_<$ satisfaisait **SEN**, on pourrait traduire toute CNF vers un de ces langages en temps polynomial, et donc vérifier sa cohérence en temps polynomial, ce qui est impossible sauf si P = NP.

• **MDD, dMDD, dOMDD, et dOMDD$_<$ ne satisfont pas TR.** La satisfaction de **TR** implique celle de **FO** (l'oubli est une restriction à $[y_1 \in \mathrm{Dom}(y_1)] \wedge \cdots \wedge [y_k \in \mathrm{Dom}(y_k)]$), donc MDD et dMDD ne satisfont pas **TR** sauf si P = NP, et dOMDD et dOMDD$_<$ ne satisfont pas **TR**.

• **OMDD et OMDD$_<$ satisfont TR.** Tout « terme » peut être traduit en temps polynomial vers un OMDD de n'importe quel ordre, et OMDD$_<$ satisfait ∧**BC** et **FO** ; il satisfait donc **TR**. Tout OMDD étant un élément de OMDD$_<$ pour un certain $<$, OMDD satisfait également **TR**.

Extraction de régions intérieures pour améliorer le majorant en optimisation globale sous contraintes

Ignacio Araya[1] **Gilles Trombettoni**[2] **Bertrand Neveu**[3] **Gilles Chabert**[4]

[1] UTFSM, Universitad Federico Santa Maria, Valparaiso, Chile
[2] LIRMM, Université Montpellier 2, France
[3] Imagine LIGM Université Paris–Est, France
[4]LINA, Ecole de Mines de Nantes, France

iaraya@inf.utfsm.cl,Gilles.Trombettoni@lirmm.fr,Bertrand.Neveu@enpc.fr,Gilles.Chabert@emn.fr

Résumé

En optimisation globale continue sous contraintes, la recherche d'un meilleur point réalisable utilise généralement des méthodes d'optimisation locale en chaque nœud de l'arbre de recherche développé par une méthode de type *Branch & Bound*. Nous proposons une approche alternative quand les contraintes sont des inégalités et que l'espace réalisable a un volume non nul. Tout d'abord, on extrait une région intérieure, c.-à-d. un polyèdre convexe ou une boîte entièrement réalisable. Ensuite, on sélectionne un point dans la région intérieure extraite et on met à jour le majorant de la fonction objectif si ce point l'améliore.

Nous décrivons dans cet article deux algorithmes originaux d'extraction de régions intérieures implantés dans `IbexOpt`. Ils s'appliquent à des contraintes continues non convexes contenant des opérateurs comme +, •, /, $power$, $sqrt$, exp, log, sin. Cette approche produit de très bons résultats sur des systèmes de taille moyenne du banc d'essai `COCONUT`.

1 Introduction

En optimisation globale sous contraintes[1], la principale méthode exacte utilisée est basée sur l'algorithme du *Branch and Bound* (B&B). La recherche d'un majorant de l'objectif consiste à trouver un point réalisable qui améliore le meilleur coût trouvé jusqu'à présent. La plupart des optimiseurs globaux ont recours à la minimisation locale utilisant une relaxation lagrangienne. La relaxation considérée peut être complexe et rendre l'optimisation locale coûteuse en temps de calcul. Cet article décrit une approche alternative qui évite l'évaluation répétée de cette relaxation. L'approche cherche à construire un espace réalisable de volume non nul formé par les contraintes d'inégalité.

Plus précisément, le problème d'optimisation globale que nous traitons est défini par :

$$\min_{x\in[x]} f(x) \text{ soumis à } g(x) \leq 0,$$

où $f : \mathbb{R}^n \rightarrow \mathbb{R}$ est la fonction objectif et $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$ la fonction représentant les contraintes. $x = (x_1, ..., x_i, ...x_n)$ est un vecteur de variables ayant pour domaine une boîte $[x] = [x_1] \times ... \times [x_i] \times ... \times [x_n]$. Un point x est **réalisable** s'il satisfait les contraintes.

L'idée principale de cet article est l'exploitation de *régions intérieures*, c.-à-d. des sous-ensembles de l'espace de recherche dont tous les points sont réalisables.

Deux communautés étudiant les méthodes à intervalles ont proposé plusieurs heuristiques pour extraire des boîtes intérieures à partir d'un domaine donné (boîte extérieure). La communauté d'analyse numérique par intervalles a étudié les systèmes linéaires ayant pour coefficients des intervalles [20, 21]. La communauté de programmation par contraintes a proposé plusieurs heuristiques générales pour des systèmes d'inégalités non convexes [8, 3, 9].

Cet article décrit deux algorithmes originaux d'extraction de région intérieure. Ces heuristiques sont appliquées pour la première fois en optimisation globale sous contraintes.

A chaque nœud de notre B&B appelé `IbexOpt` [23], un majorant est calculé en utilisant deux pro-

[1]Cet article considère le problème de minimisation, sans perte de généralité.

cédures d'extraction de région intérieure appelées `InnerPolytope` et `InHC4`. L'algorithme `InnerPolytope`, décrit dans la partie 3, construit un hyperplan pour chaque contrainte d'inégalité en utilisant une formule de Taylor sur intervalles avec pour point d'expansion un coin de la boîte courante. Si l'on réussit à construire un polytope intérieur non vide, le point minimisant la linéarisation de l'objectif est utilisé pour mettre à jour le majorant.

`InHC4` est décrit dans la partie 4. Il essaie tout d'abord d'extraire une boîte intérieure contrainte par contrainte. En cas d'échec, il tire aléatoirement un point dans la boîte courante et teste s'il est réalisable. En cas de succès, une analyse de monotonie de f remplace les intervalles des variables monotones par leur borne adéquate dans la boîte intérieure trouvée par `InHC4`. Aucun test des contraintes n'est alors nécessaire car tous les points de cette boîte sont réalisables. Le point réalisable finalement obtenu est utilisé pour mettre à jour le majorant.

Contrairement aux approches existantes, les algorithmes de mise à jour du majorant que nous proposons séparent la partie réalisable (traitée en premier par l'extraction de région intérieure) et le calcul de l'objectif (traité ensuite dans la région intérieure).

La partie 5 met en évidence le fait que, comme les autres algorithmes d'extraction de région intérieure, nos algorithmes sont des heuristiques. En d'autres termes, il peuvent parfois ne pas trouver de région intérieure, même si une telle région existe. Cependant, nous soulignons qu'ils sont en général peu coûteux.

Les expérimentations de la partie 6 montrent que la recherche de majorant utilisant les deux procédures `InnerPolytope` et `InHC4` apporte d'importants gains de performance à `IbexOpt`.

2 Prérequis et problème traité

Les intervalles offrent des calculs rigoureux sur des ordinateurs en gérant des arrondis extérieurs sur des bornes flottantes.

Un **intervalle** $[x_i] = [\underline{x_i}, \overline{x_i}]$ définit l'ensemble des réels x_i t.q. $\underline{x_i} \leq x_i \leq \overline{x_i}$, où $\underline{x_i}$ et $\overline{x_i}$ sont des nombres flottants. L'ensemble de tous les intervalles est noté $\mathbb{IR}$. La taille ou **largeur** de $[x_i]$ est $w([x_i]) = \overline{x_i} - \underline{x_i}$. Une **boîte** $[x]$ est le produit cartésien d'intervalles $[x_1] \times ... \times [x_i] \times ... \times [x_n]$. Sa largeur est définie par $\max_i w([x_i])$. $m([x])$ désigne le milieu de $[x]$. L'**enveloppe** (ou *hull*) d'un sous-ensemble S de $\mathbb{R}^n$ est la plus petite boîte n-dimensionnelle contenant S.

L'*arithmétique d'intervalles* [16] a été définie pour étendre à $\mathbb{IR}$ les fonctions élémentaires sur $\mathbb{R}$. Par exemple, la somme sur les intervalles est définie par $[x_1] + [x_2] = [\underline{x_1} + \underline{x_2}, \overline{x_1} + \overline{x_2}]$. Quand une fonction f est une composition de fonctions élémentaires, on doit définir une *extension* de f aux intervalles pour assurer un calcul de l'image conservatif.

Définition 1 (Extension d'une fonction à $\mathbb{IR}$; fonction d'inclusion)
Considérons une fonction $f : \mathbb{R}^n \to \mathbb{R}$.
$[f] : \mathbb{IR}^n \to \mathbb{IR}$ est une **extension** *de f aux intervalles ssi :*

$$\forall [x] \in \mathbb{IR}^n \quad [f]([x]) \supseteq \{f(x),\ x \in [x]\}$$
$$\forall x \ \in \mathbb{R}^n \quad f(x) = [f]([x,x])$$

L'**extension naturelle** $[f]_N$ d'une fonction sur les réels f fait la correspondance de f aux intervalles en utilisant l'arithmétique d'intervalles. Les linéarisations *intérieures* proposées dans cet article sont liées à l'**extension de Taylor sur intervalles** [16], définie comme suit :

$$[f]_T([x]) = f(\tilde{x}) + \sum_i [a_i] \,.\, ([x_i] - \tilde{x_i})$$

où $\tilde{x}$ désigne un point quelconque de $[x]$, comme $m([x])$; $[a_i]$ désigne $\left[\frac{\partial f}{\partial x_i}\right]_N([x])$.

Exemple Etudions $f(x_1, x_2) = 3{x_1}^2 + {x_2}^2 + x_1 x_2$ dans la boîte $[x] = [-1, 3] \times [-1, 5]$. L'évaluation naturelle donne : $[f]_N([x_1], [x_2]) = 3[-1, 3]^2 + [-1, 5]^2 + [-1, 3][-1, 5] = [0, 27] + [0, 25] + [-5, 15] = [-5, 67]$. Les dérivées partielles sont : $\frac{\partial f}{\partial x_1}(x_1, x_2) = 6x_1 + x_2$, $[\frac{\partial f}{\partial x_1}]_N([-1, 3], [-1, 5]) = [-7, 23]$, $\frac{\partial f}{\partial x_2}(x_1, x_2) = x_1 + 2x_2$, $[\frac{\partial f}{\partial x_2}]_N([x_1], [x_2]) = [-3, 13]$. L'évaluation de Taylor sur intervalles avec $\tilde{x} = m([x]) = (1, 2)$ donne : $[f]_T([x_1], [x_2]) = 9 + [-7, 23][-2, 2] + [-3, 13][-3, 3] = [-76, 94]$.

Problème traité

Un problème d'optimisation globale continue sous contraintes est défini de la manière suivante :

Définition 2 (Optimisation globale sous contraintes)
Soit $x = (x_1, ..., x_i, ...x_n)$ un vecteur de variables de domaine $[x]$, une fonction $f : \mathbb{R}^n \to \mathbb{R}$, des fonctions $g : \mathbb{R}^n \to \mathbb{R}^m$ et $h : \mathbb{R}^n \to \mathbb{R}^p$.
Etant donné le système $S = (f, g, h, [x])$, le problème d'optimisation globale sous contraintes consiste à trouver :

$$\min_{x \in [x]} f(x) \ \textit{ soumis à } \ g(x) \leq 0 \wedge h(x) = 0.$$

f est appelée **fonction objectif***; g et h sont des* **contraintes** *d'inégalité et d'égalité respectivement. x est dit* **réalisable** *s'il satisfait les contraintes.*

Notre optimiseur à intervalles extrait des *boîtes intérieures* et des *polytopes intérieurs* à partir de boîtes (extérieures) classiques.

Définition 3 *Soit $(f, g, [x])$ un système comprenant seulement des contraintes d'inégalité. Une* **région intérieure** *r^{in} est un sous-ensemble de $[x]$ entièrement réalisable, c.-à-d., $r^{in} \subset [x]$ et tous les points $x \in r^{in}$ vérifient $g(x) \leq 0$.*

Seuls quelques résolveurs traitent rigoureusement l'optimisation globale telle que définie ci-dessus. Des optimiseurs globaux utilisant un B&B à intervalles, comme GlobSol [10] et Icos [12], renvoient une petite boîte garantie de contenir un vecteur réel x ϵ-minimisant : $f(x)$ t.q. $g(x) \leq 0 \wedge h(x) = 0$.[2]

Inspirés par le résolveur IBBA [18], nous proposons de relâcher les égalités $h_j(x) = 0$ par des équations "épaisses" $h_j(x) \in [-\epsilon_{eq}, +\epsilon_{eq}]$, soit 2 inégalités : $-\epsilon_{eq} \leq h_j(x) \leq +\epsilon_{eq}$. Ainsi, notre optimiseur global `IbexOpt` [23] calcule un vecteur flottant x ϵ-minimisant :

$$f(x)\ s.t.\ g(x) \leq 0 \wedge (-\epsilon_{eq} \leq h(x) \leq +\epsilon_{eq}).$$

Soulignons que `IBBA` et `IbexOpt` peuvent seulement garantir l'optimum global du système relâché, quoique ϵ_{eq} puisse souvent être choisi arbitrairement petit. Notons aussi que la plupart des optimiseurs globaux déterministes (généralement basés sur un B&B), comme Baron [22], ne peuvent garantir la solution, ni du problème d'optimisation globale (Définition 2), ni du problème relâché. Notons enfin qu'en pratique la plupart des équations sont initialement "épaisses" et ne nécessitent donc pas d'être encore plus relaxées avec un ϵ_{eq}. Les équations modélisant les systèmes physiques ont en effet souvent au moins un coefficient pouvant être représenté par une constante intervalle. Ce paramètre correspond à une incertitude bornée, comme une imprécision sur une mesure, ou une constante irrationnelle, comme π (approximée par un intervalle comme $[3.14159265, 3.14159266]$).

Par souci de simplicité, puisque notre optimiseur accepte des inégalités et des égalités relaxées qui sont aussi des inégalités, nous considérons par la suite le problème d'optimisation globale suivant :

$$\min_{x \in [x]} f(x)\ \ s.c.\ \ g(x) \leq 0,$$

c'est-à-dire que les fonctions $h(x) - \epsilon_{eq}$ (≤ 0) et $-h(x) - \epsilon_{eq}$ (≤ 0) appartiennent au vecteur de fonctions $g : \mathbb{R}^n \to \mathbb{R}^m$.

De nombreux opérateurs mathématiques convexes et non-convexes peuvent être pris en compte, comme $+, \cdot, /, x^n, exp, log, sqrt, sinus$, etc.

[2] ϵ-minimiser $f(x)$ signifie minimiser $f(x)$ avec une précision ϵ, i.e., trouver x tel que pour tout y, on a $f(y) \geq f(x) - \epsilon$.

3 Algorithme du polytope intérieur

L'idée principale est de construire un demi-espace pour chaque inégalité $g_j(x) \leq 0$, tel que *tous* les points de ce demi-espace satisfassent la contrainte. Pour cela, nous approximons une fonction non linéaire par une formule de Taylor au premier ordre sur intervalles. La formule usuelle, définie dans la partie 2, peut sélectionner n'importe quel point d'expansion $\tilde{x}$ dans la boîte pour réaliser la linéarisation. Au lieu du point milieu, nous choisissons un *coin* de la boîte, ici $\underline{x}$. Considérons une fonction $g_j : \mathbb{R}^n \to \mathbb{R}$ et un domaine $[x]$. Pour toute variable $x_i \in x$, soit $[a_i^j] = \left[\frac{\partial g_j}{\partial x_i}\right]_N([x])$. L'idée est de borner $g_j(x)$ par au-dessus avec une fonction affine $g_j^l(x)$ obtenue par une formule de Taylor sur un coin. Pour tout vecteur réel $x \in [x]$, nous avons :

$$g_j(x) \leq g_j^l(x) = g_j(\underline{x}) + \sum_i \overline{a_i^j} \bullet (x_i - \underline{x_i}). \qquad (1)$$

Si nous considérons une inégalité $g_j(x) \leq 0$, la relation (1) nous permet de construire un hyperplan $g_j^l(x) = 0$ bornant l'espace réalisable par au-dessus : $g_j(x) \leq g_j^l(x) \leq 0$. La fonction linéaire $g_j^l(x)$ peut donc être utilisée pour définir une région intérieure de $[x]$.

Proposition 1 *La linéarisation sur intervalles (1) est correcte et rigoureuse, c.-à-d. qu'elle est robuste aux erreurs dues aux calculs sur les nombres à virgule flottante.*

La rigueur est assurée par Taylor sur intervalles [17]. La correction de la relation (1) tient au fait que chaque "variable" $(x_i - \underline{x_i})$ est positive puisque son domaine est $[0, d_i]$, avec $d_i = w([x_i]) = \overline{x_i} - \underline{x_i}$. Donc, le maximum de chaque terme $[a_i^j].(x_i - \underline{x_i})$ pour tout point $(x_i - \underline{x_i}) \in [0, d_i]$ est obtenu avec $\overline{a_i^j}$.

Exemple. Considérons la contrainte

$$g_1(x_1, x_2) = x_1^3 + cos(x_1) - sin(x_2) - 0.15 \leq 0$$

dans une boîte $[x_1] \times [x_2] = [-0.32, 0.52] \times [0.90, 1.06]$. Le gradient de g_1 est :

- $\frac{\partial g_1}{\partial x_1} = 3.[-0.32, 0.52]^2 - sin([-0.32, 0.52]) = 3[0, 0.2704] - [-0.314, 0.497] = [-0.497, 1.1252]$
- $\frac{\partial g_1}{\partial x_2} = -cos([0.90, 1.06]) = [-0.6216, -0.4888]$

La fonction g_1 est illustrée par la figure 1. L'espace réalisable apparaît en gris.

Appliquer (1) à g_1 donne : $g_1(-0.32, 0.90) + 1.1252\,(x1 + 0.32) - 0.4889\,(x2 - 0.90) \leq 0$. Le demi-espace intérieur correspondant est en gris foncé sur la partie gauche de la figure. La partie droite montre un espace calculé par une procédure de Taylor sur intervalles standard avec pour point d'expansion le milieu de la boîte :

$g_1(0.10, 0.98) + [-0.497, 1.1252](x1 - 0.10) + [-0.6216, -0.4889](x2 - 0.98)$ Cette formule donne quatre hyperplans sur les quatre quadrants (séparés par des pointillés sur la figure) :

1. $0.0155067948 - 0.497(x1 - 0.1) - 0.6216(x_2 - 0.98) \leq 0$ ($x_1 < m[x_1]$ et $x_2 < m[x2]$)
2. $0.0155067948 - 0.497(x1 - 0.1) - 0.4889(x_2 - 0.98) \leq 0$ ($x_1 < m[x_1]$ et $x_2 > m[x2]$)
3. $0.0155067948 + 1.1252(x1 - 0.1) - 0.4889(x_2 - 0.98) \leq 0$ ($x_1 > m[x_1]$ et $x_2 > m[x2]$)
4. $0.0155067948 + 1.1252(x1 - 0.0) - 0.6216(x2 - 0.98) \leq 0$ ($x_1 > m[x_1]$ et $x_2 < m[x2]$)

On remarque que seulement les hyperplans 2 et 3 apparaissent sur la figure ; les autres ne coupent pas la boîte.

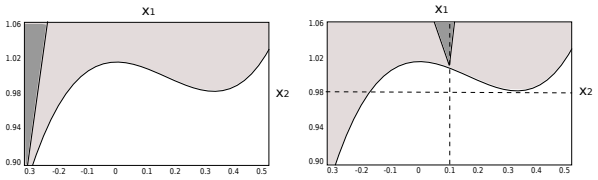


FIG. 1 – A gauche : demi-espace intérieur créé par une formule de Taylor par intervalles sur un coin $(\underline{x_1}, \underline{x_2})$. A droite : deux demi-espaces intérieurs créés à partir du point milieu dans deux des quatre quadrants.

Un programme linéaire pour améliorer le majorant

En appliquant cette idée à la fonction objectif $f(x)$ et aux inégalités $g_j(x) \leq 0$, on peut construire le programme linéaire LP^{ub} :

$$\begin{array}{ll} LP^{ub} = \min & f(\underline{x}) + \sum_i \overline{a_i} \bullet (x_i - \underline{x_i}) \\ \text{soumis à :} & \forall j \;\; g_j(\underline{x}) + \sum_i \overline{a_i^j} \bullet (x_i - \underline{x_i}) \leq 0 \\ & \forall i \;\; \underline{x_i} \leq x_i \wedge x_i \leq \overline{x_i} \end{array}$$

Un algorithme du simplexe résout LP^{ub} et renvoie l'infaisabilité ou une solution optimale x^l (cf. l'algorithme 1). L'infaisibilité ne prouve rien car le système linéarisé est plus contraint que l'original, qui pourrait avoir des solutions. Si l'algorithme du simplexe calcule une solution x^l optimale de l'approximation intérieure, alors x^l est aussi une solution du système original, mais peut-être pas l'optimale. Le processus précédent est correct sur les nombres réels, mais n'est pas nécessairement correct sur un ordinateur à cause des erreurs d'arrondi. En effet, nous utilisons un algorithme du simplexe standard sur les flottants et il est possible que le meilleur point retourné par le résolveur de programmation linéaire soit légèrement

Algorithm 1 `InnerPolytopeUB` (**in :** S, $[x]^{out}$; **in-out :** ub, x_{ub})

$LP^{ub} \leftarrow$ `InnerLinearization` (S, $[x]^{out}$)
$x^l \leftarrow$ `Simplex`(LP^{ub})
if $x^l \neq \bot$ **and** `FeasibilityCheck` (x^l, S) **then**
  $cost \leftarrow \overline{[f]_N([x^l, x^l])}$
  if $cost < ub$ **then** $ub \leftarrow cost$; $x_{ub} \leftarrow x^l$ **end if**
end if

en dehors du polytope intérieur. C'est pourquoi nous rendons le processus rigoureux en testant le caractère réalisable de x avec une évaluation par intervalles (voir la ligne 3 de l'algorithme 1).

Le pseudo-code montre finalement que l'on évalue la fonction objectif (l'originale, pas la linéarisée) au point x^l ce qui peut baisser le majorant. Dans ce cas, on met à jour le point x_{ub} et son coût, le nouveau majorant ub.

Travaux connexes, discussion

La formule de Taylor sur intervalles a souvent été utilisée pour produire une approximation linéaire de l'espace réalisable ou de la fonction objectif. Cependant, quand le point d'expansion est choisi strictement à l'intérieur du domaine, le système obtenu n'est pas convexe. Il est formé d'une intersection de sous-espaces non convexes [10]. Contracter de manière optimale une boîte contenant cette relaxation non convexe a été prouvé comme étant NP-difficile [11]. Cela explique pourquoi la communauté d'analyse par intervalles a beaucoup étudié ce problème pendant des décennies [19, 5].

Plusieurs chercheurs ont proposé de choisir comme point d'expansion de la formule de Taylor un coin de la boîte courante, au lieu du point milieu [2, 13, 14, 17]. Le principal inconvénient est de construire une relaxation de plus grand volume, l'avantage principal étant que cette approximation est formée d'un unique quadrant et est un polytope convexe.

La formulation duale que nous avons définie dans (1) n'a jamais été utilisée pour extraire un polytope intérieur et améliorer le majorant en optimisation globale sous contraintes.

4 L'algorithme InHC4

La boucle principale d'`InHC4` suit le schéma général proposé dans [3, 8]. Elle consiste à traiter chaque contrainte à tour de rôle et à intersecter à la volée les différentes boîtes produites.[3] Plus précisément :

[3] Ce schéma diffère cependant d'une propagation du type `HC4`, où une même contrainte peut être traitée plusieurs fois.

– Le traitement d'une contrainte $g_j(x) \leq 0$ prend une boîte en entrée et construit à l'intérieur de celle-ci une sous-boîte qui est intérieure pour cette seule contrainte.
– La boîte en entrée pour la $j^{\text{ème}}$ contrainte est la boîte intérieure produite par le traitement de la contrainte précédente $g_{j-1}(x) \leq 0$. Par récurrence, la boîte produite est donc intérieure pour les j premières contraintes.
– Le traitement de la première contrainte $g_1(x) \leq 0$ est effectué avec la boîte extérieure courante du B&B.

Ainsi, la boîte (éventuellement vide) produite par le traitement de la dernière contrainte est intérieure pour toutes les contraintes.

Le traitement d'une contrainte dans `InHC4` est en revanche radicalement différent des procédés de réfutation de [8, 3]. Notre procédure `InHC4-Revise` tente d'extraire une boîte intérieure pour chaque opérateur de la contrainte. De façon similaire à la procédure interne `HC4-Revise` (en abrégé, `HC4R`) de l'algorithme de propagation `HC4` [4, 15], notre procédure `InHC4-Revise` (en abrégé, `InHC4R`) est basée sur une représentation arborescente (et plus généralement, sous forme de DAG) de l'expression de la contrainte, comme illustré à la figure 2.

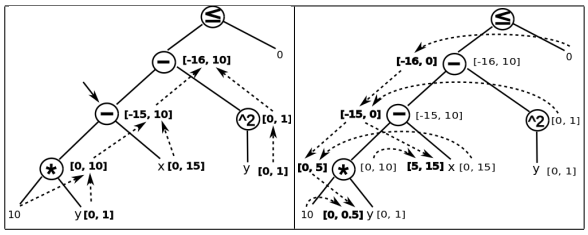


FIG. 2 – Représentation arborescente de la contrainte $10y - x - y^2 \leq 0$. **Gauche :** Phase montante (évaluation). **Droite :** Phase descendante (projection intérieure).

Notons $[x]$ la boîte initiale et $g_j(x) \leq 0$ la contrainte. Chaque nœud de l'arbre est associé à un intervalle, les intervalles associés aux feuilles étant les composantes de $[x]$. Les deux phases suivantes sont ensuite effectuées.

Phase montante d'évaluation (cf. fig. 2–gauche). L'arbre est parcouru des feuilles à la racine et les intervalles associés à chaque opérateur sont calculés avec l'arithmétique d'intervalles. Par exemple, l'intervalle du nœud marqué par une flèche est initialisé à $[0, 10] - [0, 15] = [-15, 10]$. A chaque nœud, l'intervalle est le résultat d'une évaluation naturelle sur intervalles de la sous-expression qu'il représente. Cette phase est la même que dans l'algorithme de propagation `HC4` [4, 15].

Phase descendante de projection (cf. fig. 2–droite). Des racines aux feuilles, les intervalles de chaque nœud sont contractés en utilisant des opérateurs spécifiques de *projection intérieure*.

Pour chaque nœud $(z = x_1 \; op \; x_2)$ correspondant à un opérateur binaire op et associé à un intervalle $[z]$, le rectangle formé par les intervalles des nœuds-fils x_1 et x_2 est contracté en une boîte $[x_1]^{in} \times [x_2]^{in}$ telle que :

$$\forall (x_1, x_2) \in [x_1]^{in} \times [x_2]^{in} : \;\; x_1 \; op \; x_2 \in [z]. \qquad (2)$$

Si op est un opérateur unaire (c.-à-d., $z = op(x)$), son unique nœud-fils est contracté en un intervalle $[x]^{in}$, tel que :

$$\forall x \in [x]^{in} : \;\; op(x) \in [z]. \qquad (3)$$

Si la projection intérieure produit une boîte vide (aucune boîte satisfaisant (2) ou (3) n'a pu être construite), alors la phase descendante est interrompue. Cela signifie qu'`InHC4R` n'a pu parvenir à construire une boîte intérieure pour $g_j(x) \leq 0$ donc, a fortiori pour les n inégalités. Dans ce cas, la boucle principale d'`InHC4` est interrompue.

Considérons par exemple la phase descendante appliquée à l'opérateur de multiplication de la figure 2–droite et ses deux nœuds-fils. Les intervalles contractés apparaissent en gras sur la gauche de chaque nœud (par ex., $[z]$ a été contracté à $[0, 5]$). Avant la contraction, les nœuds-fils ont pour intervalles associés $[10, 10]$ et $[0, 1]$. Ils sont alors réduits à $[10, 10]$ et $[0, 0.5]$ respectivement. Cette contraction est conforme à (2), c.-à-d., $\forall y \in [0, 0.5]$, $10 \cdot y \in [0, 5]$. Le paragraphe suivant détaille comment une telle opération élémentaire est effectuée par `InHC4R`.

4.1 Opérateurs élémentaires de projection intérieure

Nous allons décrire maintenant la façon dont les projections intérieures peuvent être implantées pour chaque opérateur élémentaire. Cette étude étend et simplifie l'étude au cas par cas faite dans le §3 de [6]. Nous proposons ici une projection générique basée uniquement sur des propriétés de monotonie et sur laquelle s'appuieront les projections spécifiques aux opérateurs.[4] Le premier cas est trivial mais il sert de point de départ pour comprendre les autres cas.

Cas 1 : opérateurs unaires monotones

Le premier cas est celui des opérateurs unaires monotones, comme log et exp. Plus précisément, il rassemble les opérateurs $z = op(x)$ continus et monotones pour x variant dans $[x]$.

[4] Par ailleurs, Chabert & Beldiceanu traitent en fait le problème dual consistant à trouver des boîtes *interdites* (c.-à-d., contenant uniquement des points qui violent au moins une contrainte). De plus, leur approche nécessite un point initial autour duquel les boîtes sont construites suivant le principe d'*inflation*.

Dans ce cas, une projection intérieure maximale (c.-à-d, non extensible, aux erreurs d'arrondi près) est évidente à calculer. Il suffit d'appliquer l'opérateur inverse de op (cf. fig. 3), comme le fait la projection extérieure dans HC4R. Cependant, pour prendre en compte rigoureusement les erreurs d'arrondi, le mode d'arrondi doit être *intérieur* (et non *extérieur*, comme dans HC4R).[5]

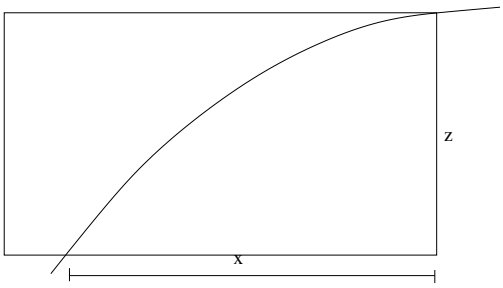


FIG. 3 – Cas 1 : Projection intérieure d'un opérateur monotone. Le segment horizontal en bas de figure est le résultat de la projection sur x.

Cas 2 : opérateurs unaires non-monotones

Le deuxième cas est celui des opérateurs $z = op(x)$ non-monotones pour $x \in [x]$, tels que x^2 ou $\sin$.

Ces opérateurs sont en fait tous monotones par morceaux. On peut donc calculer une projection intérieure pour chaque branche monotone et retourner aléatoirement l'un des intervalles formés. Bien sûr, lorsque des intervalles se touchent, on considère plutôt leur union.

Notons que pour les opérateurs unaires non-monotones, HC4R calcule l'enveloppe des intervalles obtenus pour chaque branche. Mais dans le cas d'une projection intérieure, un seul intervalle est conservé car les espaces entre les intervalles représentent des points qui ne sont pas intérieurs.

Cas 3 : opérateurs binaires monotones

Pour les opérateurs binaires (ou n-aire) monotones vis-à-vis de chacune de leurs variables, une procédure générique, appelée MonoMaxInnerBox, calcule aléatoirement une boîte intérieure maximale (si une telle boîte existe), suivant le principe représenté à la figure 5.

[5]Pour qu'un calcul soit conservatif en arithmétique flottante, c.-à-d., pour que l'image d'un intervalle contienne toutes les images réelles possibles, la borne inférieure (resp. supérieure) de l'image doit être arrondie vers $-\infty$ (resp. $+\infty$). Ces deux opérations consituent un arrondi *extérieur*.

Avec l'arrondi *intérieur*, les arrondis sont inversés : vers $+\infty$ et $-\infty$ respectivement. La propriété vérifiée par l'arrondi extérieur (resp. intérieur) pour un opérateur op est (1) et (2) respectivement :

1. $\forall x \in [x] \;\; \exists z \in op([x]) \;\; z = op(x)$
2. $\forall z \in op([x]) \;\; \exists x \in [x] \;\; z = op(x)$

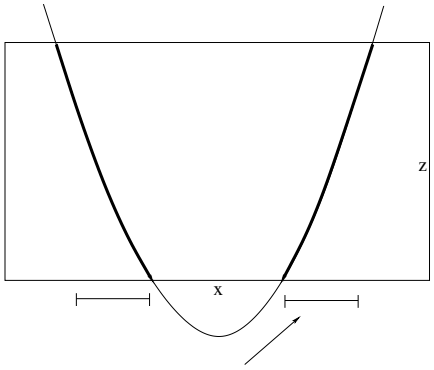


FIG. 4 – Cas 2 : projection intérieure d'un opérateur non-monotone : sqr. Deux intervalles (les segments horizontaux) sont obtenus en projetant sur x (c.-à-d., en utilisant l'inverse de sqr), et l'un d'entre eux est choisi aléatoirement.

Un opérateur binaire monotone revient à considérer deux inégalités $\underline{z} \leq (x_1 \; op \; x_2) \leq \overline{z}$. Chaque inégalité est traitée en séquence, la boîte intérieure calculée avec la première servant d'entrée pour la seconde. Comme décrit sur la fig. 5, il existe en général une infinité de boîtes maximales si bien que l'une d'entre elles est sélectionnée aléatoirement.

Cette procédure est utilisée, bien entendu, pour la projection intérieure de l'addition et de la soustraction. Elle est également utilisée pour les différents cas (monotones) des opérateurs non-monotones : la multiplication et la division, décrits ci-dessous.

Cas 4 : opérateurs binaires non-monotones

La fig. 6 illustre les deux cas à considérer pour la multiplication $x_1 \cdot x_2 \in [z]$, suivant que 0 appartienne ou non à $[z]$. Pour le cas $x_1 \cdot x_2 \in [z] \ni 0$, notons qu'une procédure directe (un peu plus complexe) pourrait être appliquée sans avoir recours à MonoMaxInnerBox.

Différentes implantations de la division sont possibles. Une approche consiste à réécrire $x_1/x_2 = x_1 \cdot \frac{1}{x_2} \in [z]$. IbexOpt propose aussi une procédure directe. Des tests (non exhaustifs) montrent que les deux versions présentent des performances similaires en pratique.

4.2 Amélioration du majorant avec InHC4

L'algorithme 2 montre comment InHC4 est utilisé par IbexOpt pour améliorer la borne supérieure.

Si une boîte intérieure $[x]^{in}$ est trouvée par InHC4, on applique une analyse de monotonie (MonotonicityAnalysis) sur la fonction objectif par rapport à chaque variable x_i. Si la dérivée partielle

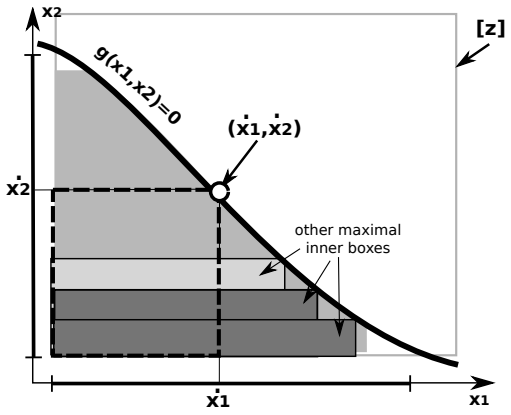


FIG. 5 – Cas 3, procédure `MonoMaxInnerBox` : La boîte en pointillés correspond à une boîte intérieure maximale de $[z]$ pour la contrainte monotone $g(x_1, x_2) \leq 0$. Un point $\dot{x}_1$ est tiré aléatoirement dans le segment horizontal des valeurs admissibles. Une seule valeur $\dot{x}_2$ peut alors rendre la boîte intérieure maximale.

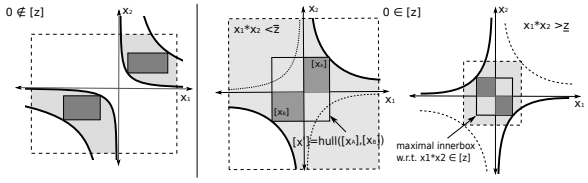


FIG. 6 – Projection intérieure pour la multiplication. **Gauche :** Deux boîtes maximales sont calculées indépendamment par `MonoMaxInnerBox` dans les deux régions intérieures disjointes (quadrants) définies par l'opérateur $x_1.x_2 \in [z] \geq 0$. **Milieu et droite :** Boîte maximale calculée pour $x_1.x_2 \in [z] \ni 0$ ($\overline{z} \geq -\underline{z}$) via deux appels à `MonoMaxInnerBox` (boîtes en gris).

$[a_i] = \left[\frac{\partial f}{\partial x_i}\right]_N([x]^{in}) \geq 0$ alors f est croissante par rapport à $x_i \in [x]$ si bien que $[x_i]$ est remplacé par l'intervalle dégénéré $[\underline{x_i}, \underline{x_i}]$ puisqu'on minimise $f(x)$ sur $[x]^{in}$. Si $[a_i] \leq 0$, f est décroissante et $[x_i]$ est remplacé par $[\overline{x_i}, \overline{x_i}]$.

Enfin, nous tirons aléatoirement un point x dans la boîte résultante [6], et remplaçons x_{ub} par x si x satisfait les contraintes et améliore le coût ub. Deux cas peuvent en fait se produire. Si une boîte intérieure a été extraite par `InHC4`, un point est sélectionné à l'intérieur de $[x]^{in}$. La satisfiabilité de x n'a plus besoin d'être testée puisque $[x]^{in}$ ne contient que des points admissibles. Si aucune boîte intérieure n'est disponible, un point aléatoire est tiré dans la boîte extérieure $[x]^{out}$ et les contraintes doivent cette fois être testées.

[6] Un échantillon constitué de plusieurs points (au lieu d'un seul) s'avère être contre-productif en pratique.

Algorithm 2 `Inhc4UB` (**in :** S, $[x]^{out}$; **in-out :** ub, x_{ub})

```
[x]^in ← InHC4 (S, [x]^out)   /* Inner box extraction */
if [x]^in ≠ ∅ then
   [x]^in ← MonotonicityAnalysis (f, [x]^in)
   x ← RandomProbing([x]^in) // or gradient descent
else
   x ← RandomProbing([x]^out)
end if
cost ← overline([f]_N([x, x]))   /* Cost evaluation */
if cost < ub  and  ([x]^in ≠ ∅ or overline([g]_N([x, x])) ≤ 0)
then
   ub ← cost ; x_ub ← x
end if
```

Remplacer ce simple échantillonage par une recherche locale simple (de type descente de gradient) n'a pas amélioré notre stratégie. Une comparaison avec des algorithmes plus sophistiqués (quasi-Newton, etc) serait cependant nécessaire pour émettre un avis définitif sur l'intérêt de la recherche locale dans ce contexte.

5 Propriétés

On peut déduire quelques propriétés à partir des descriptions des deux sections précédentes. La première est négative.

Observation 1 *Les deux algorithmes* `InnerPolytope` *et* `InHC4` *sont corrects mais incomplets : ce sont des heuristiques qui peuvent parfois ne rien trouver alors qu'il existe une région intérieure.*

L'algorithme `InnerPolytope` est incomplet (même en négligeant les erreurs d'arrondis) car le polytope intérieur (intersecté avec la boîte) possède un volume plus petit que celui de l'espace réalisable non-convexe et peut ainsi parfois être vide.

L'algorithme `InHC4` est aussi incomplet (en négligeant les erreurs d'arrondis) car tous les cas de figure, sauf le premier, ignorent une partie de l'espace réalisable. Puisque en chaque nœud du parcours de l'arbre de haut en bas une partie de l'espace réalisable n'est pas extraite, le processus peut passer à côté d'une boîte intérieure non vide obtenue par composition des opérateurs de base.

Dans le cas 2, on sélectionne seulement un intervalle parmi ceux de l'union d'intervalles possibles. Dans le cas 3, une seule boîte intérieure maximale est calculée (en faisant un choix aléatoire sur $[x_1]$) parmi un choix infini de boîtes possibles. De plus, une boîte (seule) ne peut pas contenir ou définir une région intérieure. Le dernier cas rassemble les deux défauts (des cas 2 et 3). Considérons par exemple le cas de

la multiplication $x_1.x_2 \in [z]$ où z est positif (cf. fig. 6–gauche).

En revanche, la proposition suivante souligne un aspect intéressant de InHC4.

Proposition 2 *Soit g_j une fonction contenant au plus une occurrence de chaque variable.*

Si InHC4-Revise *réussit à extraire une boîte intérieure $[x]^{in}$ vis à vis de la contrainte $g_j(x) \leq 0$ dans le domaine $[x]$, alors $[x]^{in}$ est maximal (en négligeant les erreurs d'arrondi dus à l'arithmétique flottante), c'est-à-dire qu'il n'existe pas de boîte $[x]^{in'} \supset [x]^{in}$ qui soit une boîte intérieure de $[x]$ vis à vis de g_j.*

Démonstration (esquisse)

Nous pouvons démontrer que chaque opérateur unaire et binaire calcule une boîte intérieure maximale, en négligeant la perte entraînée par les arrondis intérieurs. La preuve est directe pour les cas 1 et 2. Dans le cas 3, on peut l'obtenir par construction à partir de la procédure MonoMaxInnerBox. La seule difficulté réside dans le cas $x_1 \,.\, x_2 \in [z]$ quand $0 \in [z]$.[7]

Dans le cas où une contrainte contient seulement une occurrence de chaque variable, InHC4R calcule ainsi une boîte intérieure maximale, quand une telle boîte existe. La condition de simple occurrence rejoint celle du HC4R standard [4]. Elle assure que l'expression forme une structure d'arbre et non un graphe acyclique orienté (une variable apparaissant plusieurs fois dans l'expression a plusieurs parents dans le graphe) qui ne préserve pas la propriété par récurrence. Autrement dit, la condition assure que, pendant le parcours de haut en bas de l'expression, la composition des boîtes intérieures produit des intervalles intérieurs maximaux sur toutes les dimensions. □

Observons cependant que la propriété de maximalité ne tient pas pour un *système* d'inégalités traité par InHC4.

Une deuxième proposition donne la complexité en temps en pire cas de nos algorithmes d'extraction de régions intérieures.

Proposition 3 *Soit $(f, g, [x])$ un système de n variables et m inégalités. Soit k le nombre maximum d'opérateurs unaires et binaires dans une fonction g_j. Soit t le temps maximum requis pour évaluer un opérateur mathématique primitif.*

La complexité en temps au pire cas de la procédure d'extraction InnerPolytope *est $O(m(k.t+n))$. La complexité en temps au pire de la procédure d'extraction* InHC4 *est $O(m.k.t)$.*

[7] L'implantation directe est plus facile à vérifier car un coin de la boîte cherche à maximiser la boîte dans une dimension tandis que l'autre coin maximise la boîte dans l'autre dimension...

Démonstration

Pour InnerPolytope, un hyperplan (ou une forme linéaire de la fonction objectif) se calcule en temps $O(k.t+n)$. En effet, calculer le gradient d'une fonction g_j s'obtient en temps $O(k.t+n)$ en utilisant la différenciation automatique ; évaluer $g_j(\underline{x})$ requiert un temps en $O(k.t)$; la somme des termes $\overline{a_i^j}.(x_i - \underline{x_i})$ est en $O(n)$ pour toutes les variables. Finalement, générer le programme linéaire LP^{ub} demande $m+1$ appels à la procédure précédente (pour les m contraintes plus la fonction objectif). Notons qu'il faut compter en plus la complexité d'un appel à un résolveur de PL dans celle de la procédure InnerPolytopeUB.

La procédure InHC4 traite au plus m contraintes une par une. La procédure InHC4R lance au plus k évaluations sur intervalles en $O(t)$ chacune pendant la phase d'évaluation (de bas en haut), et k fois un nombre constant (entre 1 et 4) d'appels à la procédure MonoMaxInnerBox pendant la phase de projection (de haut en bas). Pour le traitement d'un opérateur binaire, la complexité de MonoMaxInnerBox est en $O(t)$ car il effectue deux itérations (sur les deux variables), chacune étant dominée par une évaluation sur intervalles de l'opérateur primitif. □

Soulignons que contrairement à InnerPolytope, la complexité en pire cas de InHC4 n'est pas atteinte quand InHC4R échoue dans le traitement d'une contrainte, puisque la boucle sur toutes les contraintes est alors interrompue. C'est pourquoi moins InHC4 a de chance d'extraire une boîte intérieure (parce que la boîte initiale est grande comparée à l'espace réalisable), moins de contraintes sont traitées par InHC4 et la complexité en temps reste modérée en pratique.

6 Expérimentations

Nous avons implanté ces deux algorithmes d'extraction de régions intérieures dans IbexOpt [23], qui étend la bibliothèque en C++ Ibex (Interval Based EXplorer) [7] de résolution sur intervalles à l'optimisation globale.

A chaque nœud du B&B, IbexOpt est appelé avec nos meilleurs opérateurs de réduction de l'espace de recherche et d'amélioration du minorant de l'objctif :

- L'opérateur ACID(Mohc) est une version adaptative de CID [24] utilisant Mohc [1] comme contracteur de base. Mohc exploite la monotonie des contraintes pour mieux contracter la boîte courante.
- L'opérateur X-Newton utilise la forme duale de (1) pour contracter l'espace de recherche et améliorer le minorant [2].

La plupart des problèmes sont résolus en utilisant comme heuristique de bissection une variante

de la fonction *Smear* de Kearfott décrite dans [23]. Quelques problèmes, indiqués avec `rr` dans le tableau 1 ont utilisé l'heuristique du tour de rôle.

Pour trouver un majorant, `IbexOpt` appelle les procédures `InnerPolytopeUB` et `InHC4UB` à chaque nœud du B&B.

Nous avons testé ces procédures sur un ensemble d'instances provenant de la série 1 du banc d'essai d'optimisation globale Coconut. Les équations $h_k(x) = 0$ sont remplacées par les inégalités $-\epsilon_{eq} \leq h_k(x) \leq \epsilon_{eq}$, avec $\epsilon_{eq} = $ `1.e-8`. Nous avons sélectionné les 59 systèmes résolus avec un temps de calcul allant de 1 seconde à 1 heure par `IbexOpt` ou `IbexOpt0` sur un ordinateur standard avec un processeur Pentium à 3 GHz. `IbexOpt0` désigne une version précédente de notre optimiseur global où la recherche du majorant était effectuée en tirant un point au hasard dans la boîte (extérieure) courante à chaque nœud du B&B et en testant les contraintes sur ce point (comme l'optimiseur global IBBA [18] qui choisit le milieu de la boîte). Le tableau 1 donne le nom et le nombre de variables de chaque système testé.

nom	n	nom	n	nom	n
alkyl (rr)	14	ex6_1_4	6	ex9_2_6	16
bearing	14	ex6_2_6	3	ex14_1_2	6
ex2_1_3	13	ex6_2_8	3	ex14_1_6	9
ex2_1_5	10	ex6_2_9	4	ex14_1_7	10
ex2_1_6	10	ex6_2_10	6	ex14_2_1	5
ex2_1_7	20	ex6_2_11	3	ex14_2_3	6
ex2_1_8	24	ex6_2_12	4	ex14_2_4	6
ex2_1_9	10	ex6_2_14	4	ex14_2_6	5
ex2_1_10	20	ex7_2_1	7	ex14_2_7	6
ex3_1_1	8	ex7_2_3	9	haverly	12
ex3_1_3	6	ex7_2_7 (rr)	4	hhfair	28
ex5_2_2_c1	9	ex7_2_8 (rr)	8	himmel11	9
ex5_2_2_c2	9	ex7_2_9 (rr)	10	himmel16	18
ex5_2_2_c3	9	ex7_3_4	12	house	8
ex5_2_4	7	ex7_3_5	13	hydro	30
ex5_3_2	22	ex8_1_8	6	immun (rr)	21
ex5_4_2	8	ex8_5_1	6	launch	38
ex5_4_3	16	ex8_5_2	6	meanvar	7
ex6_1_1	8	ex8_5_3	5	process	10
ex6_1_3	12	ex8_5_6	6		

TAB. 1 – Ensemble des 59 systèmes sélectionnés dans la 1ère série du banc d'essais Coconut.

La figure 7 et le tableau 2 montrent une comparaison des performances entre `IbexOpt` et `IbexOpt0`. IN représente `IbexOpt` et RAND `IbexOpt0`. `IP` représente une variante de l'optimiseur appelant uniquement `InnerPolytopeUB` à chaque nœud et INHC4 une variante n'appelant que `InHC4UB` à chaque nœud.

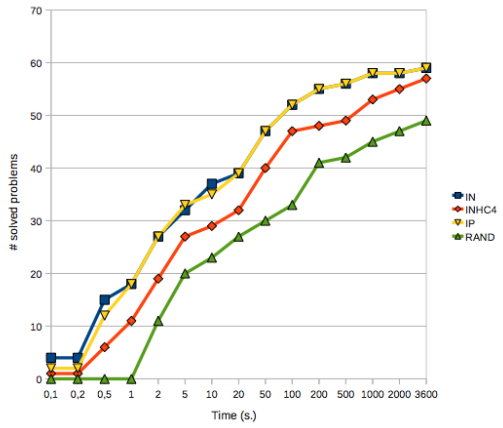


FIG. 7 – Profil de performance. Un point sur une courbe indique le nombre de systèmes résolus dans le temps CPU en abscisse par l'optimiseur correspondant.

Gain	IN	InHC4	IP
< 0.69	0	0	0
0.69 – 0.9	1	4	2
0.9 – 1	2	2	2
1 – 1.1	0	11	0
1.1 – 2	14	13	13
2 – 10	19	15	22
10 – 100	11	3	8
> 100	3	2	3
Explosion mémoire avec `IbexOpt0`	9	7	9
Résolus dans le temps imparti	59	57	59

TAB. 2 – Gains obtenus par les différents optimiseurs X (IN, InHC4, IP) par rapport à `IbexOpt0`. Le gain est défini par $\frac{time(IbexOpt0)}{time(X)}$. Chaque ligne (intervalle de gains), indique le nombre de problèmes obtenant ce gain pour les différents optimiseurs. L'avant dernière ligne indique le nombre de systèmes résolus par X mais qui explosent en mémoire avec `IbexOpt0`.

Résultats

Ces expérimentations montrent les apports significatifs de notre recherche de majorant, par rapport au test des contraintes sur un point de la boîte extérieure courante. Une perte de performance de 31% n'a été observée que sur une seule instance alors qu'un gain d'un facteur au moins 2 a été obtenu sur 33 systèmes. De plus, `IbexOpt0` n'a pas pu résoudre 10 des 59 systèmes : il a atteint la limite de temps pour 1 système et a causé une explosion mémoire pour 9 systèmes.

InnerPolytope apparaît plus utile que `InHC4`, mais l'ajout des 2 algorithmes rend le B&B plus robuste et donne de meilleures performances que l'utilisation de `IP` et `InHC4` seuls.

Etude qualitative

Plusieurs analyses qualitatives ont été menées pour mieux comprendre les tendances générales de nos algorithmes.

Nous avons d'abord cherché à déterminer quelle procédure d'extraction de région intérieure était la plus utile pour chaque système. Pour cela, nous avons compté le nombre de fois où `InHC4UB` et `InnerPolytopeUB` amélioraient le majorant. Aucune conclusion générale ne s'est dégagée, le résultat dépendant de chaque instance. Nous avons ensuite mesuré la taille moyenne des boîtes extérieures pour lesquelles les algorithmes arrivaient à extraire une région intérieure. Il est apparu qu'`InHC4UB` améliorait souvent le majorant dans des boîtes plus grandes qu'`InnerPolytopeUB`. Ceci confirme le fait qu'une forme de Taylor donne une meilleure approximation d'une fonction non convexe dans des petites boîtes. Nous avons aussi observé une plus grande variabilité dans le temps de résolution quand `IbexOpt` est appelé avec seulement `InHC4UB` plutôt qu'avec `InnerPolytopeUB` seul. Ceci est sans doute dû aux choix aléatoires faits dans les cas 2, 3 et 4 de `MonoMaxInnerBox`.

En conclusion, la recherche d'un majorant dans des régions intérieures semble prometteuse. La prochaine étape consistera à étudier si cette approche peut être étendue à de plus gros systèmes et, dans le cas contraire, comment l'améliorer.

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 *CPAIOR*, pages 1–16. LNCS 7298, 2012.

[3] F. Benhamou and F. Goualard. Universally Quantified Interval Constraints. In *Proc. CP, 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*, pages 230–244, 1999.

[5] C. Bliek. *Computer Methods for Design Automation*. PhD thesis, MIT, 1992.

[6] G. Chabert and N. Beldiceanu. Sweeping with Continuous Domains. In *Proc. CP, LNCS 6308*, pages 137–151, 2010.

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

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

[9] A. Goldsztejn. *Définition et Applications des Extensions des Fonctions Réelles aux Intervalles Généralisés : Nouvelle Formulation de la Théorie des Intervalles Modaux et Nouveaux Résultats*. PhD thesis, University of Nice Sophia Antipolis, 2005.

[10] R. B. Kearfott. *Rigorous Global Search : Continuous Problems*. Kluwer Academic Publishers, 1996.

[11] V. Kreinovich, A.V. Lakeyev, J. Rohn, and P.T. Kahl. *Computational Complexity and Feasibility of Data Processing and Interval Computations*. Kluwer, 1997.

[12] Y. Lebbah, C. Michel, M. Rueher, D. Daney, and J.P. Merlet. Efficient and safe global constraints for handling numerical constraint systems. *SIAM Journal on Numerical Analysis*, 42(5) :2076–2097, 2005.

[13] Y. Lin and M. Stadtherr. LP Strategy for the Interval-Newton Method in Deterministic Global Optimization. *Industrial & engineering chemistry research*, 43 :3741–3749, 2004.

[14] D. McAllester, P. Van Hentenryck, and D. Kapur. Three Cuts for Accelerated Interval Propagation. Technical Report AI Memo 1542, MIT, 1995.

[15] 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.

[16] R.E. Moore. *Interval Analysis*. Prentice-Hall, 1966.

[17] A. Neumaier. *Interval Methods for Systems of Equations*. Cambridge Univ. Press, 1990.

[18] J. Ninin, F. Messine, and P. Hansen. A Reliable Affine Relaxation Method for Global Optimization. Technical Report RT-APO-10-05, IRIT, 2010.

[19] W. Oettli. On the Solution Set of a Linear System with Inaccurate Coefficients. *SIAM J. Numerical Analysis*, 2(1) :115–118, 1965.

[20] J. Rohn. Inner Solutions of Linear Interval Systems. In *Proc. Interval Mathematics 1985, LNCS 212*, pages 157–158, 1986.

[21] S.P. Shary. Solving the Linear Interval Tolerance Problem. *Mathematics and Computers in Simulation*, 39 :53–85, 1995.

[22] M. Tawarmalani and N. V. Sahinidis. A Polyhedral Branch-and-Cut Approach to Global Optimization. *Mathematical Programming*, 103(2) :225–249, 2005.

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

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

Approches d'élicitation basées sur le coefficient de Spearman pour l'optimisation multicritère par ordre lexicographique

Noureddine Aribi[1,2] * **Yahia Lebbah**[1,2]

[1] Laboratoire LITIO, Université d'Oran, BP 1524, El-M'Naouer, 31000 Oran, Algérie
[2] Laboratoire I3S/CNRS, Université de Nice - Sophia Antipolis, Nice, France
{aribi.noureddine,ylebbah}@gmail.com

Résumé

Dans cet article, nous étudions l'exploitation du coefficient de corrélation rho de Spearman pour résoudre le problème d'élicitation des paramètres de l'opérateur multicritère d'ordre lexicographique (LO). Une première exploitation du coefficient statistique rho de Spearman consiste à l'utiliser comme une méthode heuristique. Une deuxième exploitation propose l'intégration de ce coefficient statistique dans deux méthodes exactes basées respectivement sur la programmation par contraintes et sur la programmation en nombres entiers. Comme toutes les méthodes d'optimisation multicritères, la méthode LO a un paramètre qui doit être fixé avec soin, soit pour déterminer la solution optimale (meilleur compromis) ou pour classer l'ensemble des solutions faisables (alternatives). C'est pourquoi nous proposons des méthodes d'élicitation afin d'assister le décideur (DM) à fixer ces paramètres. Ces méthodes nécessitent des connaissances préalables, que le DM peut proposer facilement. Nous présentons une étude expérimentale montrant l'efficacité de nos approches.

Abstract

In this paper, we study the use of Spearman's rho correlation coefficient, first in a heuristic method, and then in combination with two exact methods based on constraint programming and mixed integer programming, to tackle the parameter elicitation problem in the lexicographic ordering (LO). Like all multicriteria optimization methods, the LO method have a parameter that should be fixed carefully. Indeed, the criteria usually conflict with each other, and thus, finding an appropriate parameter settings for a specific multicriteria method is challenging. This is why we propose elicitation methods in order to assist the Decision Maker (DM) in fixing the parameters. These methods require some prior knowledge, that the DM can give easily. We present some numerical experiments, showing the effectiveness of our approaches.

*Ce travail est soutenu par le programme de recherche TASSILI 11MDU839 (France, Algérie), et par le projet de recherche PNR 70/TIC/2011 (Algérie).

1 Introduction

De nombreuses méthodes existent pour résoudre des problèmes d'optimisation multicritère, et il n'est pas aisé de choisir une méthode suffisamment adaptée à un problème donné. Après le choix d'une méthode multicritère, différents paramètres (e.g., poids, fonctions d'utilité, ...) doivent être déterminés, soit pour trouver la solution optimale (meilleur compromis) ou pour classer l'ensemble des solutions faisables (alternatives). Justement, vue cette difficulté pour fixer les paramètres, les méthodes d'élicitation sont utilisées pour aider le décideur dans cette tâche de fixation des paramètres.

Dans cet article, nous nous intéressons à la méthode d'ordre lexicographique [4], qui est utilisée quand une relation stricte de dominance entre les critères peut être établie. Le paramètre de cette méthode est un ordre total sur les critères. Nous faisons l'hypothèse de la disponibilité des informations préalables sur les préférences du décideur, à savoir un vecteur de valeurs agrégées (cf. [1]) où chaque solution est affecté d'une valeur quantifiant la préférence du décideur. Nous nous concentrons sur la façon d'utiliser ces préférences, plutôt que sur la façon de les obtenir. Nous supposons que nous disposons d'un sous-ensemble d'alternatives avec leurs valeurs agrégées.

Par exemple, considérons une situation dans laquelle

un vendeur souhaite expliquer/justifier à ses clients le prix de ses différents produits, à savoir des appareils photo numérique. Un appareil photo numérique est caractérisé par plusieurs caractéristiques : le nombre de méga-pixels, le poids, la plage de zoom maximal, etc. Le vendeur voudrait fournir à ses clients une explication simple et cohérente des prix des appareils. Les clients de ce vendeur ne comprennent pas les compromis subtils entre les critères. En effet, ils ne peuvent seulement comprendre qu'un critère (e.g., le zoom) est plus important qu'un autre (e.g., le poids). Il s'agit donc de trouver une permutation entre les critères qui justifient au mieux le prix des produits.

Dans le travail présent, nous tirons profit d'une mesure statistique simple et rapidement calculable, à savoir, le coefficient de corrélation de Spearman, afin de développer une méthode heuristique (approximative), et deux approches exactes basées sur la programmation par contraintes (PPC) et la programmation linéaire en nombres entiers (PLNE). Ces méthodes sont ensuite utilisées pour éliciter automatiquement les paramètres appropriés de l'optimisation multicritère via l'ordre lexicographique.

Le reste de ce papier est organisé comme suit. La section 2 fournit quelques préliminaires. La section 3 donne une formulation du problème d'élicitation des paramètres. La section 4 illustre les méthodes heuristiques et exactes sur un exemple de motivation. La section 5 décrit en détail la méthode heuristique proposée. La Section 6 présente à la fois nos approches exactes, ainsi qu'une approche combinée. Des résultats empiriques sont présentés et discutés respectivement dans les sections 7 et 8. La section 9 conclut le papier.

2 Préliminaires

Nous présentons des définitions préliminaires nécessaires pour développer nos approches d'élicitation.

Definition 1 (Ordre lexicographique) *[9] Les critères sont classés par le décideur en fonction de leur importance valorisée. La solution sélectionnée est celle qui a la meilleure valeur pour le critère le plus important. Les solutions indifférentes par rapport au premier critère seront discriminées à l'aide du deuxième critère le plus important, et ainsi de suite.*

Formellement, soient $x, y \in \mathbb{R}^n$,

$$x <_{lex} y \equiv (x_1 < y_1) \vee ((x_1 = y_1) \wedge \langle x_2, ..., x_n\rangle <_{lex} \langle y_2, ..., y_n\rangle) \quad (1)$$

Dans le contexte du présent papier, nous avons besoin de mesurer l'écart entre un ordre donné par l'opérateur lexicographique et un ordre idéal dérivé à partir des valeurs agrégées. Il y a différentes façons pour mesurer cet écart. Nous proposons de faire appel à la distance tau de Kendall qui est très utilisée pour comparer le degré de désordre d'un vecteur par rapport à un vecteur référence.

Definition 2 (Distance tau de Kendall) *La distance tau de Kendall [7] est une métrique qui compte le nombre de désaccords par paires entre deux listes (τ_1, τ_2).*

$$\begin{aligned} K(\tau_1, \tau_2) = |\{(i,j) : i \in \{1, ..., n-1\}, j \in \{i+1, ..., n\}, \\ (\tau_1(i) < \tau_1(j) \wedge \tau_2(i) > \tau_2(j)) \\ \vee(\tau_1(i) > \tau_1(j) \wedge \tau_2(i) < \tau_2(j))\}| \end{aligned} \quad (2)$$

Quand les deux listes sont identiques, alors $K(\tau_1, \tau_2) = 0$. Si τ_1 est strictement croissante, et τ_2 contient les mêmes éléments de τ_1 dans l'ordre inverse, alors $K(\tau_1, \tau_2) = n(n-1)/2$, où $n = |\tau|$. La complexité en temps du calcul de la métrique tau de Kendall est en $\mathcal{O}(n^2)$ [15].

Exemple 1 : La distance tau de Kendall entre $[0, 3, 1, 6, 2, 5, 4]$ et $[1, 0, 3, 6, 4, 2, 5]$ est égale à six, i.e., $|\{(0,1), (1,2), (1,4), (2,5), (4,5), (5,6)\}|$, et toutes les autres paires sont dans le même ordre [13].

Definition 3 (Coefficient rho de Spearman) *Le coefficient de corrélation rho de Spearman (r_s) [3] est une mesure non paramétrique de corrélation entre deux variables X et Y. Il évalue dans quelle mesure une fonction monotone pourrait décrire la relation entre deux variables classées. Ce coefficient est calculé avec la formule :*

$$r_s = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2 \sum_{i=1}^{n}(y_i - \overline{y})^2}} \quad (3)$$

où $x_i = rank(X_i)$ et $y_i = rank(Y_i)$ sont les rangs de la $i^{ème}$ valeur dans l'échantillon. Le coefficient de Spearman varie entre -1 et $+1$, et la valeur absolue de r_s désigne le taux de la monotonie.

Le rang des valeurs identiques est donné par la moyenne de leurs positions par rapport à l'ordre croissant de ces valeurs. Par exemple, si deux valeurs sont égales au $2^{\text{ième}}$ plus grand rang, ils auront un rang de 2,5 (i.e., la moyenne de 2 et 3). Pour un échantillon de taille n, le coefficient rho de Spearman a une complexité en temps en $\mathcal{O}(n \log n)$, puisque la complexité de cette méthode est dominée par la complexité de l'algorithme de tri.

Exemple 2 : Le coefficient rho de Spearman entre $[1, 2, 3, 4, 5, 6, 7]$ et $[1, 2, 3, 5, 4, 7, 6]$ est égal à 0.929, synonyme d'une forte similarité, tandis que le coefficient de Spearman entre $[1, 2, 3, 4, 5, 6, 7]$ et $[1, \mathbf{7}, 3, 4, 5, 6, \mathbf{2}]$ est égal à 0.107, synonyme d'une faible similarité.

3 Formulation du problème

Le problème d'élicitation des paramètres de la méthode d'ordre lexicographique peut être posé avec les éléments suivants :

- Un ensemble de critères $X = \{X_1, ..., X_n\}$ où $n \geq 2$;
- Un ensemble d'alternatives $A = \{A_1, ..., A_m\}$, où A_{ij} désigne la valeur de la $i^{\text{ème}}$ alternative sur le $j^{\text{ème}}$ critère. Ces alternatives sont triées suivant leurs valeurs agrégées Y, i.e., $Y(A_i) \leq Y(A_j), i = 1, ..., m-1, j = i+1, ..., m$, où $Y(A_i)$ désigne la valeur agrégée issue de la $i^{\text{ème}}$ alternative. Ces valeurs agrégées peuvent être recueillies expérimentalement, ou bien aussi à partir des réponses d'un questionnaire sur les préférences (cf. [1]).

Le problème consiste à trouver un ordre total entre les critères θ, pour lequel l'ordre lexicographique induit sur l'ensemble des alternatives A_θ, devient aussi proche que possible de l'ordre "idéal" donné par A. Nous utilisons la distance suivante pour mesurer le désordre entre A_θ et A.

Definition 4 (Mesure du désordre) *Soit* $\theta = [X_{\theta(1)}, ..., X_{\theta(n)}]$ *une permutation entre les critères* $[X_1, ..., X_n]$. *Suivant* θ, *nous obtenons un ordre lexicographique* $A_\theta = (A_{\sigma(1)}, ..., A_{\sigma(m)})$, *où* $[\sigma(1), ..., \sigma(m)]$ *est une permutation de* $[1, ..., m]$. *La mesure du désordre associée à* A_θ *est donnée par :*

$$D(A_\theta) = K([\sigma(1), ..., \sigma(m)], [1, ..., m]) \tag{4}$$

Definition 5 (Permutation optimale) *Une permutation* θ^* *est dite optimale si et seulement si pour toute permutation* θ,

$$D(A_\theta) \geq D(A_{\theta^*}) \tag{5}$$

Notez que pour un problème multicritère ayant n critères, nous avons $n!$ permutations possibles. Ainsi, l'espace des permutations possibles a une taille exponentielle suivant le nombre de critères. Par conséquent, le défi consiste à trouver le meilleur ordre, sans énumérer toutes les possibilités.

TABLE 1 – Un échantillon de 18 appareils pour le problème du vendeur

#	X_1	X_2	X_3	Y
1	440	2	5	389
2	510	2	2	416
3	470	4	2	421
4	400	4	4	425
5	340	6	12	434
6	390	8	1	449
7	250	8	5	461
8	250	12	6	465
9	360	14	8	468
10	220	14	7	473
11	320	18	7	478
12	280	20	9	484
13	310	20	10	485
14	300	22	9	488
15	280	24	3	527
16	200	26	12	529
17	270	28	14	532
18	210	30	13	566
r_s	$\underline{-0.763}$	$\underline{0.997}$	$\underline{0.689}$	

4 Exemple de motivation

Revenons à l'exemple introductif dans lequel nous disposons du vecteur des valeurs agrégées Y, correspondant aux *prix* des produits. Le vendeur voudrait expliquer/justifier ces prix auprès de ses clients. Un échantillon de 18 appareils sont présentés dans le tableau (1), où X_1 indique le *poids*, X_2 est la *résolution* du capteur d'image, et X_3 est le *zoom* de l'appareil.

Dans cet exemple, nous cherchons à trouver une permutation entre les critères qui minimise le critère d'optimalité donnée par l'équation (4). Selon la méthode heuristique, nous mesurons le degré de monotonie en utilisant la méthode basée sur le coefficient de corrélation de Spearman (r_s), entre chaque critère et le vecteur des valeurs agrégées. Nous rappelons ici qu'une valeur r_s proche de 1, désigne une monotonie forte et donc un désordre faible. Pour notre cas d'étude, nous avons obtenu $[\underline{-0.763}, \underline{0.997}, \underline{0.689}]$ comme vecteur de coefficients, pour X_1, X_2 et X_3 respectivement. Ici, le soulignement indique que le résultat est inférieur au seuil d'importance statistique [1] (α, le plus souvent égal à 5%, cf. [3]). En conséquence, nous concluons qu'une monotonie forte existe entre Y et chaque critère X_i. Ensuite, une permutation est obtenue en triant les critères suivant les valeurs décroissantes de ce vecteur.

1. Le seuil d'importance indique la probabilité qu'un résultat est dû au hasard.

Nous obtenons ainsi $[X_2, X_3, X_1]$ qui désigne une permutation robuste entre les critères (θ_{rs}). X_2 est le critère le plus important, et ainsi de suite en terminant par X_1. Notez que la valeur -0.763 du premier critère signifie que X_1 est pratiquement dans l'ordre inverse par rapport à Y. Si nous prenons $-X_1$, alors le premier critère va surclasser X_3. Mais puisque notre approche d'élicitation n'élicite pas le signe des critères, nous retenons $[X_2, X_3, X_1]$ comme le meilleur ordre. De toute façon, l'élicitation du signe du critère pourrait être facilement intégrée dans notre approche, mais pour plus de clarté, notre travail a porté sur la difficulté réelle du problème d'élicitation : faire surgir la meilleure permutation entre les critères.

Par ailleurs, les approches exactes que nous proposons (i.e., PPC et PLNE) permettent de trouver la permutation optimale par construction, et pour notre exemple, les deux ont donné la même permutation (optimale), c'est-à-dire $\theta_{rs} = \theta_{ppc} = \theta_{plne}$.

5 Méthode heuristique basée sur le coefficient rho de Spearman

Nous voulons exploiter le coefficient de corrélation de Spearman afin de déterminer d'une façon heuristique une permutation robuste entre les critères. Le schéma algorithmique de notre méthode heuristique est introduit dans l'Algorithme 1. Cet algorithme applique une heuristique de recherche basée sur le coefficient rho de Spearman. Il est décrit comme suit. Nous commençons par convertir la variable liée aux valeurs agrégées à des rangs (*inst.* 1/fonction RANK). Ensuite, l'algorithme itère sur tous les critères dans X (*boucle* $2-5$). Pour chaque critère X_i, nous calculons son vecteur de rang x_i (*inst.* 3). Ensuite, le coefficient rho de Spearman est calculé à partir des deux vecteurs de rangs x_i et y (*inst.* 4/fonction COMPUTERHO basée sur la Formule 3), afin d'évaluer le taux de désordre issu du critère x_i par rapport à l'ordre idéal donné par Y. Le coefficient de corrélation obtenu est ajouté à la liste des coefficients de corrélation (*inst.* 5). L'Algorithme 1 tri la liste des coefficients de corrélation $rhoList$ par ordre décroissant, puis renvoi une permutation robuste entre les critères dans θ (*inst.* $6-7$).

Dans la section 7 de l'évaluation expérimentale, nous verrons que l'Algorithme 1 est très efficace, car il parvient généralement à proposer un ordre entre les critères qui est souvent similaire à celui donné par les méthodes exactes. Néanmoins, vue la nature heuristique de cette méthode basée sur le coefficient statistique de Spearman, elle peut trouver un ordre non-optimal. En effet, dans la section expérimentale 7, nous avons repéré certains cas où cette démarche heuristique n'a pas trouvé l'ordre optimal donné par les méthodes exactes.

Algorithm 1: STATELICIT cherche une permutation robuste entre les critères

Input: un ensemble fini de critères $X = \{X_1, ..., X_n\}$;
vecteur des valeurs agrégées Y donnant l'ordre idéal entre les solutions ;
Output: une permutation robuste θ

1 $y \leftarrow$RANK(Y)
2 **for** $i \leftarrow 1$ ***to*** n **do**
3 $\quad x_i \leftarrow$RANK(X_i)
4 $\quad rho \leftarrow$COMPUTERHO(x_i, y)
5 $\quad rhoList \leftarrow rhoList \cup \{rho\}$
6 $\theta \leftarrow$ SORT$(rhoList)$
7 **return** θ

Ces cas sont des contre-exemples, et donc des éléments de sa preuve de non-optimalité.

Du moment que cette approche est basée sur une mesure statistique, elle s'appuie sur l'*hypothèse nulle* (H_0) [8]. Cette hypothèse stipule qu'il y a une forte probabilité qu'il n'y ait pas d'association entre les deux variables dans *la population sous-jacente*, représentée par l'échantillon. Habituellement, une valeur statistique (p-value) est calculée avec un *Test d'importance* [16], pour voir s'il convient d'accepter ou de rejeter H_0, selon un seuil d'importance, $\alpha = 5\%$. Ainsi, une p-value suffisamment faible (i.e., p-value $< \alpha$) est considérée comme suffisante pour rejeter H_0, et donc, le résultat est dit "statistiquement significatif". [2]

L'Algorithme 1 implémente une heuristique simple et efficace, que nous avons trouvée en utilisant le coefficient de Spearman. En effet, après avoir capturé le premier critère ayant la plus grande valeur de Spearman, nous avons pu choisir le second critère en ne considérant que les alternatives non-ordonnées ; Malheureusement, dans ce cas, le coefficient de corrélation de Spearman sera moins efficace, car il sera appliqué sur des échantillons de petites tailles. Plus précisément, le fait de considérer toutes les alternatives permet d'avoir un coefficient de Spearman beaucoup plus précis.

6 Méthodes basées sur la PPC et sur la PLNE

6.1 Modèle de programmation par contraintes

Les environnements de programmation par contraintes (e.g., CPO, Gecode, Choco, ...) visent à faciliter la formulation et la résolution des problèmes

2. La p-value ne mesure pas la force des corrélations, mais elle pourrait être utile pour décider si réellement une corrélation existe.

[12]. L'utilisation des contraintes globales est assez souvent à l'origine du succès de la PPC, telles que `All-different` [11] et `Element` [5] exploitées dans notre modèle d'optimisation PPC. Ce modèle est décrit dans l'Algorithme 2, où :

- D_{θ^*} (*inst.* 1) spécifie le domaine des variables θ^*.
- Les variables dans θ^* doivent être toutes différentes (*inst.* 2).
- Les préférences sont extraites à partir du vecteur des valeurs agrégées, et elle sont traitées en utilisant les *contraintes réifiées*[3] (*boucle* $4-12$). Ainsi, pour chaque relation de préférence $Y(A_i) < Y(A_j)$ nous utilisons la contrainte réifiée $(A_i <_{lex} A_j) \Longleftrightarrow b$, où $b \in \{0,1\}$. Par ailleurs, aucun traitement n'est associé aux cas des égalités (i.e., $Y(A_i) = Y(A_j)$), puisque chaque alternative parmi (A_i et A_j) peut surclasser une autre, et n'affecte pas le degré du désordre.
- L'objectif est modélisé par une *fonction coût* (*inst.* 13), dont le nombre de préférences satisfaites est maximisé.
- Les variables Pr_1 et Pr_2 permettent d'ignorer les cas d'indifférence et les préférences redondantes.

Algorithm 2: LEXMAXCSP cherche la permutation optimale entre les critères

Input: n critères ; m alternatives : $\{A_l, ..., A_m\}$
Output: Permutation optimale θ^*

1 $D_{\theta^*} \leftarrow \{1, ..., n\}$
2 ALL-DIFFERENT(θ^*)
3 $Pr_1 \leftarrow min(Y) - 1$; $k \leftarrow 1$
4 **for** $i \leftarrow 1$ ***to*** $m-1$ **do**
5 **if** $Y(A_i) \neq Pr_1$ **then**
6 $Pr_2 \leftarrow Y(A_i)$
7 **for** $j \leftarrow i+1$ ***to*** m **do**
8 **if** $Y(A_j) \neq Pr_2$ **then**
9 $pref_k \leftarrow$ LEXLE(A_i, A_j, θ^*)
10 $k \leftarrow k+1$
11 $Pr_2 \leftarrow Y(A_j)$
12 $Pr_1 \leftarrow Y(A_i)$
13 MAXIMIZE$(\sum_{k=1}^{|pref|} pref_k)$
14 **return** θ^*

L'Algorithme 3 est proposé comme une version itérative de la formule récursive (1). Etant donnés deux vecteurs X et Y, cet algorithme cherche une permutation entre les critères, qui permet à X d'être inférieur à Y au sens lexicographique.

L'Algorithme 3 contient des contraintes de la forme x_z *op* y_z, où x, y sont deux vecteurs d'entiers, z est

3. Les contraintes réifiées reflètent la validité d'une contrainte dans un domaine fini 0/1.

Algorithm 3: LEXLE cherche une permutation θ, de sorte que $(x <_{lex} y)$

Input: x, y : deux vecteurs de n valeurs entières
Output: $(x <_{lex} y)$

1 $lex_{n-1} \leftarrow (x_{\theta_{n-1}} < y_{\theta_{n-1}}) \vee ((x_{\theta_{n-1}} = y_{\theta_{n-1}}) \wedge (x_{\theta_n} < y_{\theta_n}))$
2 **for** $k \leftarrow n-2$ ***downto*** 1 **do**
3 $lex_k \leftarrow (x_{\theta_k} < y_{\theta_k}) \vee ((x_{\theta_k} = y_{\theta_k}) \wedge (lex_{k-1}))$
4 **return** lex_1

une variable de décision dont le domaine est $D_z = \{1, ..., n\}$ et $op \in \{<, =\}$. Ce type de contraintes est considéré efficacement avec la contrainte globale `Element`.

A savoir,

$$(x_z \ op \ y_z) \equiv \begin{cases} \texttt{Element}(z, [x_1, ..., x_n], t_1) \\ \texttt{Element}(z, [y_1, ..., y_n], t_2) \\ t_1 \ op \ t_2 \\ \text{integer}(t_i) \text{ for } i = \{1, 2\} \end{cases} \quad (6)$$

todo La contrainte $\texttt{Element}(z, [x_1, ..., x_n], t_1)$ est satisfaite si t_1 est égal à la $z^{\text{ième}}$ valeur du vecteur $[x_1, ..., x_n]$. Pour des raisons de facilité d'écriture, nous avons préféré préserver la notation x_z *op* y_z. Le nombre de contraintes élémentaires du modèle PPC est en $\mathcal{O}(n \times m^2)$ dans le pire des cas (i.e., $m(m-1)/2$ contraintes de préférences et $2n$ contraintes élémentaires pour chaque contrainte LEXLE). Par construction, l'approche PPC calcule l'ordre optimal entre les critères. En effet, le modèle PPC trouve la meilleure permutation des critères pour satisfaire le plus $m(m-1)/2$ relations de préférences possibles. De façon similaire, le critère tau de Kendall s'intéresse aux $m(m-1)/2$ relations possibles.

6.2 Modèle d'élicitation PLNE

Dans cette section, nous introduisons une formulation du problème d'élicitation basée sur la programmation linéaire en nombres entiers (PLNE). Notre développement du modèle PLNE est justifié par les performances des solveurs PLNE dans le traitement des problèmes d'optimisation combinatoire. Une reformulation directe du modèle PPC vers un modèle PLNE en vue d'être résolu par un solveur PLNE, n'est pas nécessairement une bonne idée, car la linéarisation de la contrainte globale `Element` produira des modèles linéaires volumineux, qui nuiront aux performances du solveur PLNE. Pour plus de détails, voir les travaux de Refalo [10] et Hooker [6].

L'idée consiste à reformuler l'opérateur d'ordre lexicographique comme une fonction de *somme pondérée*, à condition que le vecteur de poids choisi interdise

la compensation entre les critères. En fait, pour réagir comme un opérateur lexicographique, nous avons choisi des poids assez grands pour neutraliser tout comportement compensatoire entre les critères. Cependant, même si cette reformulation semble être efficace, elle peut engendrer une instabilité numérique provoquée par les grandes valeurs flottantes, et donc une dégradation des performances et de la justesse du solveur PLNE.

Linéarisation de la contrainte `All-different` La reformulation linéaire proposée pour cette contrainte est illustrée dans l'Algorithme 4 ; dans lequel on associe à chaque variable de décision un vecteur de variables binaires, de sorte qu'au plus une variable soit fixée à 1. Les contraintes dans les deux boucles $2-3$ et $4-5$, permettent de s'assurer que les variables θ vont avoir des valeurs distinctes par rapport à la liste de toutes les valeurs possibles (a).

Algorithm 4: allDiffMip - reformulation PLNE de All-different

Input: b : matrice de $n \times n$ variables binaires ;
a : vecteur de n entiers ;
θ : vecteur de n variables entières.
Output: AllDiff(θ)

1 $D_\theta \leftarrow \{1, .., n\}$
2 **for** $i \leftarrow 1$ ***to*** n **do**
3 $\quad \sum_{j=1}^{n} b_{ij} \leq 1$
4 **for** $j \leftarrow 1$ ***to*** n **do**
5 $\quad \sum_{i=1}^{n} b_{ij} \leq 1$
6 **for** $i \leftarrow 1$ ***to*** n **do**
7 $\quad \theta_i = \sum_{j=1}^{n} b_{ij} \cdot a_j$
8 **return** θ

Notre modèle d'optimisation PLNE est décrit dans l'Algorithme 5, où :

- *inst.* 1 détermine la valeur de chaque coefficient c_i.
- D_{θ^*} et D_γ (*inst.* 2 & 3) définissent respectivement le domaine des variables θ^* et γ.
- Le domaine des variables w est défini en utilisant la boucle $4-6$.
- Les variables dans w doivent être toutes différentes (*inst.* 7).
- Les préférences sont extraites à partir du vecteur des valeurs agrégées, et elles sont modélisées avec les contraintes réifiées de la somme pondérée (*boucle* $9-17$).
- L'objectif est de modéliser par une *fonction coût* (*inst.* 18), dont le nombre de préférences satisfaites est maximisé.

Algorithm 5: mipElicit - modèle d'élicitation PLNE basé sur une fonction somme pondérée

Input: n critères ; m alternatives : $\{A_l, ..., A_m\}$
Output: Permutation optimale (θ^*)

1 $c_i \leftarrow 10^i, i \in \{1, .., n\}$
2 $D_{\theta^*} \leftarrow \{1, ..., n\}$
3 $D_{\gamma_{ij}} \leftarrow \{0, 1\},\ i, j \in \{1, .., n\}$
4 **for** $i \leftarrow 1$ ***to*** n **do**
5 $\quad w_i \leftarrow \sum_{j=1}^{n} c_j \cdot \gamma_{ij}$
6 $\quad \sum_{j=1}^{n} \gamma_{ij} = 1$
7 allDiffMip(w)
8 $Pr_1 \leftarrow min(Y) - 1$; $k \leftarrow 1$
9 **for** $i \leftarrow 1$ ***to*** $m - 1$ **do**
10 $\quad$ **if** $Y(A_i) \neq Pr_1$ **then**
11 $\quad\quad Pr_2 \leftarrow Y(A_i)$
12 $\quad\quad$ **for** $j \leftarrow i + 1$ ***to*** m **do**
13 $\quad\quad\quad$ **if** $Y(A_j) \neq Pr_2$ **then**
14 $\quad\quad\quad\quad pref_k \leftarrow \sum_{k=1}^{n} w_k \cdot A_{ik} < \sum_{k=1}^{n} w_k \cdot A_{jk}$
15 $\quad\quad\quad\quad k \leftarrow k + 1$
16 $\quad\quad\quad Pr_2 \leftarrow Y(A_j)$
17 $\quad Pr_1 \leftarrow Y(A_i)$
18 maximize($\sum_{k=1}^{|pref|} pref_k$)
19 **for** $i \leftarrow 1$ ***to*** n **do**
20 $\quad \theta_i^* = \sum_{j=1}^{n} \gamma_{ij} \cdot j$
21 **return** θ^*

– La permutation optimale entre les critères est récupérée à l'aide de la boucle $19 - 20$.
– Les variables Pr_1 et Pr_2 permettent d'ignorer les cas d'indifférence et les préférences redondantes.

Le nombre de contraintes élémentaires du modèle PLNE est en $\mathcal{O}(m^2)$ dans le pire des cas (i.e., $m(m-1)/2$ contraintes de préférences. Par construction, cette reformulation linéaire permet de trouver la permutation optimale entre les critères.

6.3 Utilisation de l'heuristique dans les deux méthodes exactes

L'obtention d'une bonne solution réalisable très tôt peut améliorer considérablement les performances de résolution des deux méthodes exactes, car elle fournit une borne inférieure (vs. supérieure) efficace pour la fonction objectif. Comme il a déjà été mentionné, l'utilisation du coefficient de Spearman dans la méthode heuristique conduit à une solution robuste. Sur la base de cette idée, nous proposons d'injecter la solution de la méthode heuristique dans les méthodes exactes, et d'examiner dans quelle mesure cela peut affecter les performances de résolution des solveurs PPC/PLNE.

Les modélisations PPC et PLNE visent à maximiser la satisfaction des relations de préférence données par l'utilisateur. La solution calculée avec la méthode heuristique basée sur Spearman est une solution réalisable pour les deux modèles PPC et PLNE. En conséquence, cette solution est nécessairement une borne inférieure, c'est-à-dire que la valeur de la fonction objectif au niveau de la solution heuristique est toujours inférieure à sa valeur à l'optimum.

7 Evaluation expérimentale

Nous avons évalué expérimentalement nos approches d'élicitation sur un ensemble d'instances réalistes [14, 2]. Ces expérimentations ont été effectuées en utilisant les outils suivants : (i) IBM ILOG Concert C++ et CP Optimizer (CPO)/CPLEX (version 12.5) [4]. Nos modèles d'optimisation PPC et PLNE ont été résolus en utilisant les solveurs CPO et CPLEX respectivement ; (ii) IBM SPSS Statistics (version 20.0) [5]. L'approche basée sur Spearman a été testée à l'aide de l'outil SPSS.

Les expérimentations ont été réalisées sur une machine x86_64 dotée d'un processeur Intel Xeon (X5460) 4xCœurs @ 3.16 Ghz et de 8 Go de mémoire libre. Nous avons utilisé la mesure de désordre (voir l'Equation 4) pour évaluer la robustesse de la méthode heuristique.

4. http://www-142.ibm.com/software/products/fr/fr/ibmilogcplecpopti/

5. http://www-01.ibm.com/software/fr/analytics/spss/products/statistics/

Les tableaux (2 & 3) résument nos résultats empiriques menés sur une série de 11 instances avec 100 alternatives pour chaque instance. Dans le tableau (2), les colonnes `PPC` et `PPC + SP` désignent le temps de résolution de la méthode PPC (en secondes), avec un temps max de 3600 *sec* pour chacune des instances ; tandis que les colonnes `Cr`, `Vars`, `Cons` et `Mem` indiquent respectivement le nombre de critères, variables de décision, contraintes et l'espace mémoire utilisé (en `Mo`).

Nous notons que la première méthode basée sur le coefficient de Spearman est à la fois heuristique et instantanée, avec un temps d'exécution quasi-nul. Par conséquent, ses résultats ne sont pas comparables à ceux des méthodes exactes. Nous avons préféré mentionner le degré d'optimalité au lieu du temps d'exécution. Dans la colonne D_{SP}/D_{PPC} du tableau (2), le côté droit du symbole "/" (slash) désigne le degré du désordre, calculé à partir des résultats de la méthode PPC, tandis que le côté gauche désigne le degré du désordre calculé à partir des résultats de la méthode heuristique.

Par ailleurs, les résultats expérimentaux de notre approche PLNE sont présentés dans le tableau (3). Pour chaque problème, ce tableau indique le nombre de lignes, colonnes, variables binaires, et indicateurs, respectivement dans les colonnes `rows`, `cols`, `bin` et `ind`. Le temps de résolution (en secondes) est également illustré dans les colonnes `PLNE` et `PLNE + SP` avec un temps max d'une heure sur tous les problèmes.

Nous constatons à partir des résultats empiriques que le modèle PPC est nettement plus efficace que le modèle PLNE sur toutes les instances. En fait, l'approche PLNE ne passe pas à l'échelle sur des grandes instances, en particulier pour les problèmes d'optimisation ayant plus de 9 critères. Par exemple, le modèle PPC associé à l'instance #6, contient peu de variables (9), tandis que le modèle PLNE équivalent contient 4 861 variables. Cependant, sur les instances #9 à #11 (voir le tableau 3) le solveur CPLEX a signalé un état de non-faisabilité, car les coefficients ont des valeurs trop grandes, de l'ordre de 10^{18}, qui vont au delà de la précision machine.

Combinaison de la méthode heuristique avec les méthodes exactes Il est intéressant de noter que la méthode heuristique capture les solutions optimales dans presque tous les cas (voir le tableau 2, colonne D_{SP}/D_{PPC}). Il ressort également de ce tableau que cette méthode gloutonne accélère considérablement les performances des deux méthodes exactes (voir colonnes `PLNE+SP` et `PPC+SP`), notamment en raison de sa rapidité. Cette méthode combinée fournit donc une borne inférieure de bonne qualité, et concentre l'effort du solveur dans la preuve d'optimalité.

TABLE 2 – Résultats empiriques de l'approche heuristique et de l'approche PPC. Les colonnes `Cr`, `Vars`, `Cons` et `Mem` désignent respectivement, le nombre de critères, de variables de décision, de contraintes et l'espace mémoire utilisé (en `Mo`). La colonne D_{SP}/D_{PPC}, illustre la mesure du désordre calculée à partir des résultats de la méthode heuristique, contre la mesure optimale du désordre obtenue avec l'approche PPC. Les colonnes `PLNE` et `PLNE+SP` montrent le temps de résolution (en `sec`) des approches PPC et PPC + SP. Le symbole # désigne le numéro d'instance.

#	Cr	D_{SP}/D_{PPC}	Vars	Cons	Mem	PPC	PPC + SP
1	3	**.40**/.40	3	44,551	71.6	2.36	**1.09**
2	5	**.44**/.44	5	84,151	145.1	13.67	**5.94**
3	6	.51/.48	6	103,951	173.0	85.28	**35.22**
4	8	**.42**/.42	8	143,551	239.5	62.44	**11.47**
5	8	**.44**/.44	8	143,551	240.1	156.93	**51.77**
6	9	.60/.47	9	163,351	276.6	TO	**427.8**
7	11	**.44**/.44	11	202,951	342.6	257.71	**14.94**
8	11	.53/.42	11	202,951	342.9	505.90	**65.33**
9	18	**.37**/.37	18	341,551	2,700.0	TO	**735.16**
10	19	**.16**/.16	19	361,351	686.7	23.19	**17.22**
11	44	.43/.41	44	856,351	3,900.0	TO	**2,052.06**

TO: timeout (3,600 sec)

8 Discussion

Nous avons étudié l'utilisation du coefficient de corrélation rho de Spearman, d'abord dans une méthode heuristique, puis en combinaison avec deux méthodes exactes, pour faire face au problème d'élicitation du paramètre de l'opérateur d'ordre lexicographique. De toute évidence, cela n'a pas de sens de comparer les deux approches exactes avec l'approche heuristique. Il en résulte que la méthode heuristique devient un excellent choix, si le décideur est à la recherche de solutions robustes et rapidement calculables. Sinon, le choix le plus évident est d'opter pour une approche exacte.

Approche PPC vs PLNE Une comparaison plus approfondie entre les deux approches exactes comprend les considérations (non exhaustives) suivantes :

- **Complexité des calculs**
  Le temps de résolution des modèles PLNE augmente rapidement par rapport au nombre de critères. Par contre, le nombre de critères n'affecte pas considérablement les performances de l'approche PPC, plus particulièrement la version améliorée (`PPC +SP`).
- **Traitement parallèle**
  Lors de l'utilisation du mode de résolution parallèle (multi-Threads) de CPLEX, nous avons réussi à améliorer le temps de résolution des instances étudiées. Néanmoins, sur ces instances, CPO n'a pas bénéficié de ce mode. Par exemple, pour l'instance #9, avec un seul Thread, et seulement 50 alternatives, CPO a terminé la résolution après 331.59 *sec*, tandis qu'avec 4 Threads, le temps de calcul a augmenté considérablement jusqu'à 2 094.63 *sec*.

Paramètres personnalisés pour les performances de calcul Les résultats des deux solveurs CPO et CPLEX (voir les tableaux 2 & 3) ont été obtenus selon un paramétrage avancé. Suivant ces paramètres, nous avons réussi à augmenter les performances de résolution d'un facteur allant [6] jusqu'à 19.

- **Paramètres CPLEX**
  CPLEX a un certain nombre de caractéristiques sophistiquées qui améliorent considérablement les performances de résolution. Durant nos expérimentations, nous avons exploité les paramètres suivants :

  1. **Ordres de priorité** Ces ordres permettent d'attribuer une plus grande priorité aux variables de décision qui devraient être instanciées plutôt.
  2. **Cutoff** L'activation de ce paramètre démarre la résolution avec des bornes inférieures (vs. supérieures) de bonne qualité, et de ce fait, peut considérablement accélérer la résolution. Dans notre cas, ces valeurs sont obtenues à partir de la méthode heuristique.
  3. **Cuts** En général (et en particulier dans nos expérimentations), la désactivation des

6. En raison du manque d'espace, nous ne pouvons pas présenter tous nos résultats antérieurs.

TABLE 3 – Résultats empiriques de l'approche PLNE. Les colonnes `Cr`, `rows`, `cols`, `bin` et `ind` désignent respectivement le nombre de critères, de lignes, de colonnes, de variables binaires et des indicateurs du modèle PLNE, tandis que les colonnes `Mem`, `PLNE` et `PLNE+SP` montrent l'utilisation de la mémoire (en `Mo`), et le temps de résolution (en `sec`) de l'approche PLNE et sa version PLNE + SP. Le symbole # désigne le numéro de l'instance.

#	Cr	rows	cols	bin	ind	Mem	PLNE	PLNE + SP
1	3	2,414	904	192	3,795	.01	4.16	**2.59**
2	5	5,219	6,429	3,694	7,288	.42	78.37	**16.39**
3	6	4,609	9,242	4,649	8,170	3.98	1,596.53	**174.37**
4	8	4,625	9,306	4,713	9,170	166.9	1,439.18	**645.91**
5	8	4,683	9,422	4,771	9,286	494.85	TO	**1,786.10**
6	9	4,861	9,865	5,048	9,612	2,273.15	TO	**2,165.40**
7-8	11	—	—	—	—	—	TO	TO
9-11	≥18	—	—	—	—	—	infeasible	infeasible

TO: timeout (3,600 sec)

coupes inutiles ajoutées par CPLEX permet d'accélérer les performances du solveur PLNE sur la plupart des problèmes.

4. **Probing** Il permet de contrôler l'appel d'une procédure de fixation des variables binaires. Cette procédure est activée après l'étape de pré-résolution, mais avant la séparation/évaluation (*Branch and Bound*). Ici, nous avons choisi de faire un appel intensif à cette procédure, améliorant sensiblement nos résultats.
5. **Sélection des variables** Sur certaines instances difficiles, l'usage d'une séparation forte (*Strong Branching*) a été très utile.

– **CP Optimizer**
Il y a plusieurs paramètres qui peuvent améliorer considérablement les performances de CP-Optimizer. Dans ce qui suit, nous examinons, sans être exhaustif, quelques paramètres importants que nous avons exploités dans nos expérimentations.

1. **Phases de recherche avec sélecteurs** Ce paramètrage indique explicitement à l'algorithme de recherche les variables clés, et quelle variable doit être instanciée. Sur nos expérimentations, nous avons décidé de sélectionner des variables de décision ayant la plus petite taille de domaine.
2. **Niveaux d'inférence** Ce paramètre est utilisé pour réaliser plus de réduction sur les domaines, en changeant le niveau d'inférence de certaines contraintes spécialisées, par exemple, `All-different`, dans notre cas.
3. **Paramètres de recherche**

– **Paramètre de type de recherche** Ce paramètre contrôle le type de recherche appliquée à un problème. Le type de recherche choisi est basé sur la stratégie de relance (`Restart`), conjointement avec deux autres paramètres, à savoir, la limite de l'échec (`Fail limit`) et le facteur de relance (`Restart grow factor`). La justification de notre choix est que la solution de départ (fournie par la méthode heuristique), ne peut être utilisée que par une recherche de type `Restart` ou `multi-point`.
– **Tolérance d'optimalité relative** Ce paramètre doit être manipulé avec précaution, surtout si nous voulons maintenir l'optimalité des solutions. Par exemple, si nous voulons trouver une solution d'environ 1% de l'optimum, nous mettons la tolérance d'optimalité relative à 0.01. Toutefois, dans ce cas, le solveur ne prouve pas nécessairement l'optimalité, ni même l'absence d'une solution.

9 Conclusion et perspectives

Nous avons mis en évidence la pertinence du coefficient de corrélation de Spearman exploité dans les deux méthodes heuristique et exacte, pour résoudre le problème d'élicitation des paramètres de la méthode d'optimisation multicritère d'ordre lexicographique. Nous avons montré que l'utilisation de la méthode heuristique en combinaison avec une méthode exacte est efficace, car elle bénéficie de la simplicité, la rapidité et l'efficacité de la borne inférieure fournie par la méthode heuristique. La méthode exacte (PPC en particulier) assure l'optimalité, et offre une forme

concise et expressive pour la formulation du problème.

Nous avons également présenté des résultats expérimentaux évaluant l'efficacité des approches proposées, portant sur des problèmes multicritère significatifs. Ces expérimentations ont été réalisées avec un paramétrage avancé, afin de diriger et accélérer les solveurs PPC et PLNE. Toutefois, la décision finale du choix entre une méthode heuristique ou exacte, s'appuie sur les exigences du décideur. La méthode heuristique est un excellent choix, si le décideur est à la recherche de solutions robustes et rapidement calculables. Sinon, le choix le plus évident est d'employer l'approche PPC qui prouve l'optimalité.

Parmi les perspectives de ce travail, nous envisageons d'améliorer le modèle PLNE, notamment en s'inspirant des autres approches de linéarisation.

Remerciements

Nous remercions Olivier Lhomme pour ses conseils constructifs et ses critiques dans la réalisation de ce travail, ainsi que ses commentaires enrichissants dans la rédaction de l'article.

Références

[1] G. Beliakov. How to build aggregation operators from data. *International Journal of Intelligent Systems*, 18 :903–923, 2003.

[2] C.H. Brase and C.P. Brase. *Understandable Statistics : Concepts and Methods*. Cengage Learning, 2011.

[3] G.W. Corder and D.I. Foreman. *Nonparametric Statistics for Non-Statisticians : A Step-By-Step Approach*. Wiley, 2009.

[4] J. Figueira, S. Greco, and M. Ehrgott. *Multiple Criteria Decision Analysis : State of the Art Surveys*. Springer Verlag, Boston, Dordrecht, London, 2005.

[5] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. Logic programming. MIT Press, 1989.

[6] J. Hooker. *Integrated Methods for Optimization*. International series in operations research & management science. Springer US, 2012.

[7] M. G. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30(1/2) :81–93, June 1938.

[8] J.S. Maritz. *Distribution-Free Statistical Methods*. Monographs on Statistics and Applied Probability. Chapman & Hall, 1995.

[9] R.T. Marler and J.S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26 :369–395, 2004. 10.1007/s00158-003-0368-6.

[10] Philippe Refalo. Linear formulation of constraint programming models and hybrid solvers. In Rina Dechter, editor, *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 2000.

[11] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In Barbara Hayes-Roth and Richard E. Korf, editors, *AAAI*, pages 362–367. AAAI Press / The MIT Press, 1994.

[12] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*. Number vol. 35 in Foundations of Artificial Intelligence. Elsevier, 2006.

[13] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[14] Barbara G. Tabachnick and Linda S. Fidell. *Using Multivariate Statistics (5th Edition)*. Allyn & Bacon, Inc., Needham Heights, MA, USA, 2006.

[15] Weichao Xu, Chunqi Chang, Yeung Sam Hung, S. K. Kwan, and Peter Chin Wan Fung. Order statistics correlation coefficient as a novel association measurement with applications to biosignal analysis. *IEEE Transactions on Signal Processing*, 55(12) :5552–5563, 2007.

[16] Jerrold H. Zar. Significance testing of the spearman rank correlation coefficient. *Journal of the American Statistical Association*, 67(339) :578–580, 1972.

Un nouveau cadre diviser pour régner pour SAT distribué

Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Cédric Piette *

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

{audemard,hoessen,jabbour,piette}@cril.fr

Résumé

Nous proposons dans cet article une nouvelle technique parallèle pour SAT. Celle-ci, basée sur le paradigme "diviser pour régner", met en place une architecture distribuée de type maître/esclave, où chaque esclave est un solveur CDCL travaillant sur une sous-formule du problème initial, tandis que le maître est chargé de mettre en relation les esclaves, et ne travaille donc pas "activement" à la résolution. Le schéma de répartition de charge adopté est celui du vol de travail, dont la division peut être effectuée "classiquement" par séparation sur une variable, mais autorise également des formes plus sophistiquées de division. Notre implémentation préliminaire permet d'obtenir des résultats très honorables.

Abstract

In this paper, we propose a new distributed approach for SAT. This approach, based on the divide-and-conquer principles, relies on master/slave architecture, where each slave is a CDCL solver working on a subformula of the initial problem, whereas the master does not actively participate to the solving process, since its main task is to coordonate the slaves, and etasblishing communication between them. The load balancing strategy is the work stealing technique, where the load division can be achieved in a classical way by splitting on variables. However, our division scheme allows more sophisticated splitting forms. Our preliminary experiments delivers very satisfying results.

1 Introduction

Le problème SAT, qui consiste à décider de la satisfaisabilité d'une formule booléenne sous forme normale conjonctive, occupe un rôle central en théorie de la complexité, où il représente le problème NP-complet de référence [6], alors que de nombreux problèmes issus d'applications pratiques (vérification formelle, cryptographie, bio-informatique, etc.) s'y ramènent naturellement ou le contiennent.

Il est le sujet, depuis des décennies, de recherches actives concernant sa résolution pratique. De très nombreux leviers (structure de données, heuristique, apprentissage, etc.) ont été mis en évidence pour réduire encore et toujours l'effort calculatoire nécessaire à déterminer la satisfaisabilité d'un problème donné. Ces dernières années, l'évolution matérielle des unités de calcul a principalement consisté en une multiplication de cœur de calculs. Ainsi, depuis quelques années, on voit émerger un intérêt croissant pour la résolution parallèle de SAT.

Dans ce papier, nous présentons un cadre (appelé `Dolius`) permettant de résoudre SAT de manière distribuée, selon le principe du diviser pour régner. Ce travail préliminaire représente la première étape d'un projet à plus large échelle. Dans cette approche, le maître est chargé de mettre en relation les esclaves et ne travaille donc pas activement. Lorsqu'un esclave a terminé son travail, il peut répondre SAT et mettre un terme à la recherche globale, soit répondre UNSAT et dans ce cas demander du travail au maître. Le maître le met alors en relation avec un autre esclave, celui-ci va alors diviser son travail en deux permettant au premier esclave d'effectuer une nouvelle recherche. Il existe différentes possibilités pour diviser la recherche. La séparation sur une variable est la solution classique. Une autre est la séparation du travail en utilisant deux ensembles de clauses mutuellement inconsistants [12]. Si tous les esclaves ont répondu UNSAT alors le pro-

*Supporté par le CNRS et OSEO, avec le projet ISI "Pajero".

blème initial est lui même UNSAT. Notre approche distribuée utilise les protocoles réseau et peut donc mettre en relation un nombre quelconque d'esclaves.

Le reste de cet article est structuré de la manière suivante : la section 2 introduit les définitions et notations nécessaires à la compréhension, la section 3 passe en revue les différentes approches existantes pour paralléliser la résolution du problème SAT. La section 4 présente notre nouvelle approche. Enfin, cet article se termine par une première évaluation expérimentale de `Dolius` + `PeneLoPe`, notre solveur SAT distribué.

2 Définitions et notations

Le lecteur est supposé familier avec les notions liées au problème de satisfaisabilité d'une formule propositionnelle (variable x, littéral positif x ou négatif $\neg x$, clause, clause unitaire, interprétation et CNF). Chaque esclave de notre approche utilise un solveur de type CDCL (*Conflict-Driven, Clause Learning*) dont voici une brève explication du fonctionnement. Une branche d'un solveur CDCL est une séquence de décisions, suivies de propagations des littéraux unitaires, répétées jusqu'à ce qu'un conflit survienne. Chaque variable de décision choisie au moyen d'une heuristique, généralement basée sur l'activité, est affectée à un niveau de décision donné, les littéraux propagés en conséquence ayant le même niveau de décision. Chaque fois qu'un conflit survient, un *nogood* est calculé avec une méthode donnée, généralement celle nommée FUIP (*First Unique Implication Point*) [14, 20]. Le *nogood* est ajouté à la base des clauses apprises et un saut arrière est effectué. En outre, des redémarrages sont effectués périodiquement. Le lecteur désirant plus de détails sur les solveurs de type CDCL peut se référer à [7].

Dans cet article, la distinction est faite entre une application parallèle et une application distribuée. La première utilise un système de mémoire centrale pour la communication inter processus, tandis que la seconde communique au travers d'interface(s) réseau. L'avantage d'un système à mémoire centrale est la rapidité des communications. Les applications distribuées permettent en revanche de faciliter l'utilisation d'un plus grand nombre d'unités de calcul par des machines distinctes.

Dans le cadre d'applications multi-processus, un processus est dit *affamé* si celui-ci est disponible mais n'a pas de travail à effectuer.

3 État de l'art

Il existe deux approches générales pour paralléliser la résolution SAT. D'un côté, les algorithmes de type *portofolio* consistent à placer en concurrence plusieurs instances d'un solveur, chacune d'entre elles étant paramétrées différemment. L'idée de cette approche est d'utiliser des stratégies orthogonales afin de raccourcir en pratique le temps global de résolution, toutes les instances du solveur étant stoppées dès l'obtention de la solution par la stratégie la plus adaptée au problème traité.

Toutefois, certains portofolios de solveurs ne sont pas uniquement basés sur la concurrence. Certains mettent également en place des techniques de partage d'information. En pratique, ce partage est basé sur l'échange de clauses apprises au cours de la recherche (*nogood*). À titre d'exemple, chaque instance du solveur `plingeling` [4] transmet les clauses unitaires produites au cours de la recherche, afin de faciliter les autres recherches. D'autre mécanismes plus sophistiqués ont été récemment proposés, tels que ceux introduits dans le portofolio `ManySAT` [9]. Toutefois, le partage d'informations au sein d'un portofolio est délicat. D'un côté, certaines informations obtenues par une instance de solveur peuvent être cruciales à la recherche d'une de ces concurrentes, et accélérer considérablement l'exploration de l'espace de recherche. Il est donc souhaitable de transmettre ces informations. D'un autre côté, la surabondance d'informations, et le coût de leur transmission d'une instance aux autres peuvent véritablement pénaliser l'efficacité du portofolio. Ce problème est accru par le fait (i) qu'un nogood est appris après chaque conflit rencontré par chacune des instances (ii) il est extrêmement difficile de prédire l'utilité d'un *nogood*. Pour pallier ces difficultés, `PeneLoPe` [1] se base sur différents concepts heuristiques tels que le *Progress Saving Measure* (`PSM`) [2] et le *Literal Block Distance* (`LBD`) [3] qui ont pour objectif de ne partager que les clauses jugées les plus pertinentes pour les recherches en cours.

L'autre approche majeure pour paralléliser SAT se fonde sur l'idée de diviser pour régner. Cette classe d'algorithmes a pour principe de réduire récursivement un problème en un ou plusieurs sous-problèmes du même type, plus simple à résoudre. Appliqué à SAT, cette technique vise à diviser la formule à résoudre Σ en plusieurs sous-formules $\Sigma_1, \Sigma_2, ...$ qui sont ensuite résolues indépendamment par différentes unités de calcul. Il s'agit de l'un des aspects importants de ce modèle, qui réside dans le partage de la charge de travail entre les processeurs.

Le projet `SAT@HOME` [16], est un exemple de parallélisation de type diviser pour régner. Celui-ci, initié en 2010, se base sur la plate-forme de calcul distribué `BOINC` [17] pour effectuer une résolution massivement parallèle, de type diviser pour régner. Ces travaux héritent de l'architecture du célèbre projet `SETI@HOME`,

visant à distribuer l'observation et l'analyse des rayons issus de l'espace, dans le but de détecter de la vie intelligente non terrestre.

Il existe également d'autres implémentations de type diviser pour régner pour résoudre SAT. Par exemple, Feldman *et al.* [8] proposent une telle architecture, qui permet en outre le partage des clauses apprises (*nogood*) entre les différents processus. Cependant, ce partage de clauses se fait au sein d'une zone mémoire commune à différents fils d'exécution, ce qui rend cette implémentation impossible à exécuter sur des machines distinctes.

On compte aussi parmi les approches distribuées pour résoudre SAT le portail `GridSat` [5]. Lancé en 2000, il a pour but de fournir une interface publique simple à utiliser pour exécuter une résolution distribuée à large échelle (c'est à dire sur un grand nombre de calculateurs). Cet ambitieuse plate-forme est malheureusement ancienne et n'exploite pas les dernières avancées algorithmiques pour la résolution de SAT. Elle est en effet basée sur `Chaff`, le solveur connu pour avoir introduit les *watched littéraux*. Ce solveur n'est toutefois plus maintenu et n'est plus représentatif de l'état de l'art pour la résolution de SAT.

Récemment, Schulz et Blochinger [18] ont proposé une approche originale pour résoudre SAT de manière distribuée. En effet, contrairement à la plupart des autres techniques distribuées, où un élément centralisateur (souvent appelé *maître*) est nécessaire, leur contribution consiste à proposer un système *pair-à-pair* (P2P), où tous les calculateurs jouent le même rôle, sans avoir recours à une quelconque centralisation. Le système P2P résultant se nomme `SatCiety` [18].

Enfin, une étude sur le partitionnement statique de formules CNF a été proposée [11]. Ce partitionnement, dans lequel la formule CNF est représentée sous forme d'arbre, est donc effectué avant le début de la recherche. Ce type de partitionnement statique n'est pas connu pour être la technique la plus efficace pour l'obtention de formules équilibrées (i.e. de difficulté équivalente), mais les auteurs défendent cette façon de faire en arguant qu'une fois découpée et générée, une sous-formule (de type CNF) peut être fournie à n'importe quel solveur sans aucune modification de son code, le considérant ainsi comme une boîte noire.

Dans la section suivante, notre nouvelle plate-forme distribuée, baptisée `Dolius`, est présentée.

4 Détails techniques de `Dolius`

Dans cette section, nous détaillons les différents mécanismes et structures de données implémentés dans `Dolius`.

Le but de cette implémentation est de fournir une plateforme capable de résoudre une instance du problème SAT en distribué. Pour cela, cette plateforme est destinée à diviser le travail par l'approche diviser pour régner. De plus, il est également possible de transmettre une partie des connaissances accumulées dans l'espoir de limiter au maximum le travail redondant.

4.1 Structure générale / Initialisation

La structure générale de `Dolius` est de type maître/esclave. Les esclaves sont des solveurs de type CDCL ; dans la pratique nous utilisons `PeneLoPe` [1], ce qui nous permet d'obtenir un solveur distribué dont les esclaves peuvent être des solveurs séquentiels "classiques", ou des portofolios de solveurs, permettant d'exploiter au mieux les architectures multi-cœurs [1] ; tandis que le maître est un processus dont la tâche est de mettre en relation les esclaves, quand l'un d'eux a terminé sa tâche (voir plus bas).

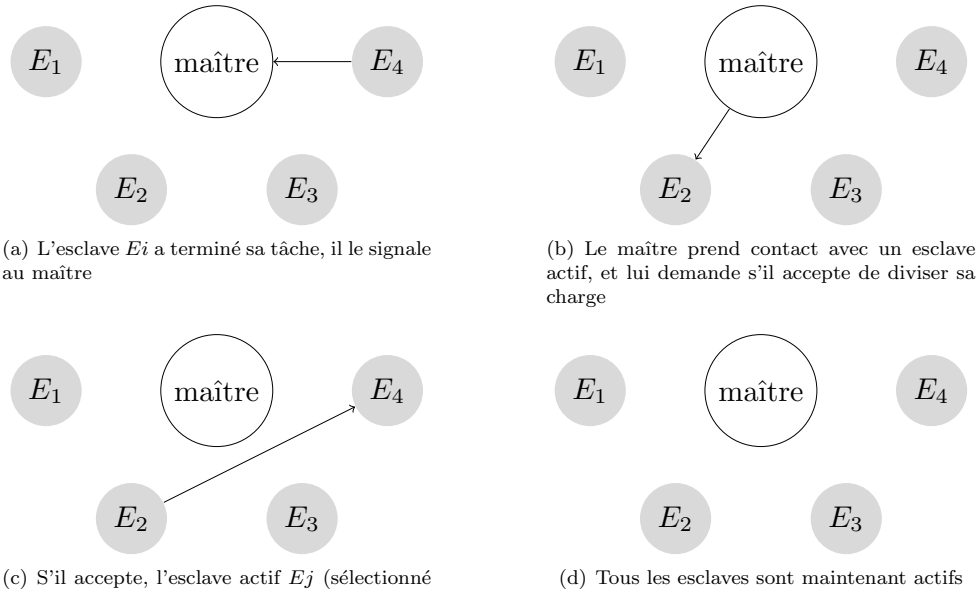
L'initiation de `Dolius` consiste simplement à démarrer le maître. Celui-ci est alors en attente d'un ou plusieurs esclaves, qui entameront alors la résolution du problème. Il est à noter que l'implémentation `Dolius` autorise l'ajout de ressources (esclaves) en cours de résolution. Ainsi, il est possible à n'importe quel moment de la recherche d'augmenter le nombre d'esclaves, ceux-ci n'ayant qu'à contacter le maître. Au travers de notre stratégie, une formule est prouvée satisfiable (SAT) si l'un des esclaves trouve un modèle. En effet, chaque esclave travaille avec la formule initiale simplifiée par la division du travail (voir section 4.3). Une formule est prouvée insatisfaisable (UNSAT) si tous les esclaves ont prouvé que la sous-formule dont ils s'occupent est UNSAT.

4.2 Rééquilibrage de charge

Il semble impossible en pratique de partitionner l'espace de recherche de manière à la fois statique et optimale au début de l'algorithme. Idéalement, les esclaves doivent avoir une charge de travail équivalente, mais il est en effet très difficile de prédire à l'avance la difficulté d'une (sous-)formule.

Un mécanisme de rééquilibrage de charge de travail doit donc être mis en place, afin d'être opportuniste vis-à-vis des ressources disponibles. En effet, certains esclaves termine en pratique leur tâche bien avant certains autres, dans la large majorité des cas. Nous avons ainsi implanté au sein de notre plate-forme un mécanisme qui permet de répartir les tâches sur les différentes unités de calcul, ceci à l'aide d'un processus de

1. Nous travaillons sur une API qui permettra de distribuer `Dolius` afin que chacun puisse y greffer son propre solveur SAT comme esclave



(a) L'esclave Ei a terminé sa tâche, il le signale au maître

(b) Le maître prend contact avec un esclave actif, et lui demande s'il accepte de diviser sa charge

(c) S'il accepte, l'esclave actif Ej (sélectionné par le maître) envoie une partie de sa charge à l'esclave actif Ei

(d) Tous les esclaves sont maintenant actifs

FIGURE 1 – Illustration de l'équilibrage de charge via le vol de travail

plus en plus répandu : le *vol de travail*. L'avantage de cette technique est qu'un esclave cherchant une solution ne doit pas se soucier d'autre esclaves qui seraient affamés. Ceux-ci viendront d'eux même demander du travail. Les différentes étapes de ce processus sont illustrées en Figure 1. Lorsqu'un esclave (noté Ei) termine la tâche qui lui est confiée, il contacte le maître (M) pour signaler qu'il n'a pas plus de travail (étape 1(a)). Le maître demande à l'un des esclaves encore actifs (noté Ej) s'il accepte de diviser sa tâche (étape 1(b)). Si Ej accepte, il contacte alors Ei pour se délester d'une partie de sa charge de travail (étape 1(c)). Les esclaves entrent donc directement en communication l'un avec l'autre sans passer par le maître, qui n'est contacté que pour fournir à un esclave nouvellement inactif les coordonnées d'un autre esclave acceptant de diviser sa tâche.

Dans notre plate-forme, un esclave ne peut refuser de diviser son travail que lorsqu'un des trois cas suivants se présentent :

1. l'esclave vient de trouver un modèle à la formule, la recherche d'une solution devient donc inutile
2. il n'œuvre sur sa tâche que depuis peu de temps (en pratique $< x$ secondes, où x peut être spécifié par l'utilisateur. Par défaut, $x = 2,5$)
3. il est déjà en cours de division de sa charge avec un autre esclave inactif

En outre, au sein de `Dolius`, le maître conserve l'ensemble des esclaves actifs dans une file. Lorsqu'un esclave demande du travail, il sélectionne le premier esclave actif de sa file afin de le contacter. Ce choix a été fait pour éviter, en cas de demandes de travail simultanées par plusieurs esclaves inactifs, qu'un même esclave soit contacté pour plusieurs demandes de division de travail, ce qui serait très inefficace.

Dans la section suivante, nous détaillons les stratégies de division du travail offertes par `Dolius`.

4.3 Division du travail

Lorsqu'il en reçoit la requête, la manière dont un esclave divise sa charge de travail pour en céder une partie est un facteur important de l'efficacité d'un algorithme de type diviser pour régner. La division sera considérée comme parfaite si les temps de résolution des sous-formules sont égaux entre eux, et inférieur au temps nécessaire pour résoudre la formule complète φ. Si la division est mal effectuée, deux problèmes de nature différente peuvent survenir. Premièrement, il est possible que les sous-formules φ_i aient un même temps de résolution que la formule complète. De ce fait, le temps nécessaire pour résoudre l'instance augmentera en fonction du nombre de ressources : le temps nécessaire pour résoudre l'instance auquel il faut ajouter le temps nécessaire pour la division du travail. Le deuxième problème lié à une mauvaise division se produit si le temps de résolution de φ_i est négligeable ou bien moindre que celui pour φ_j. Cela implique pour le système d'être capable de mettre en place une po-

litique de ré-équilibrage de travail. Si cette politique souffre de mauvaise division, des cas pathologiques peuvent apparaitre, comme le cas dit *ping-pong* [13].

Exemple 1 *Soit ϕ une formule CNF. La formule $\Sigma = ((a \vee b) \wedge \phi) \wedge ((a \vee \neg b) \wedge \phi)$ est également une CNF. Les divisions sur les variables a et b posent toutes les deux problème.*

(a) si la division du travail de Σ est réalisée sur la variable a, l'une des deux charges de calculs est alors très faible, car il est très facile de prouver que $\Sigma \vDash a$. Ainsi, par simple propagation unitaire, il est possible de montrer que $\Sigma \wedge (\neg a) \vDash \bot$. L'esclave ayant reçu cette partie du problème va donc la prouver incohérente sans la moindre exploration (la propagation unitaire étant suffisante), et redemander du travail très rapidement. Ceci est problèmatique, car la division du travail a un coût, notamment en transfert réseau. Le phénomène ping-pong est le fait de multiplier de tels mauvais choix pour la division du travail, menant à des sous-problèmes d'une trivialité telle que certains esclaves passent plus de temps à demander du travail qu'à véritablement participer à la résolution de la formule.

(b) si la division est réalisée sur b, alors chacun des deux esclaves travaille sur la même formule : $a \wedge \phi$. En outre, si ϕ est UNSAT, c'est seulement lorsque le moins efficace des deux (i.e. celui répondant en dernier) transmet sa solution que l'incohérence de Σ peut être établie.

Ainsi, dans une telle situation, il est largement préférable de diviser selon l'une des variables de ϕ plutôt que sur a ou b.

Dans l'idéal, la tâche doit être divisée en deux sous-tâches différentes de difficulté similaire, afin de répartir au mieux la charge de travail, tout en s'assurant que les sous-tâches soient de difficulté moindre que la tâche avant division. Malheureusement, comme précisé plus haut, il semble difficile sinon impossible de déterminer à l'avance la difficulté d'une formule. Les algorithmes diviser pour régner basent donc leur division sur des concepts heuristiques.

Les algorithmes de ce type utilisent généralement la notion de *guiding path* pour la division du travail. La plupart du temps le *guiding path* est réduit à la simple *séparation* d'une variable, et affecte (pour lui même) l'une des variables de la formule tandis qu'il transmet à l'esclave inactif l'opposé de cette même variable [15, 12].

Notre plate-forme permet ce genre de division, mais se veut plus générique. En effet, `Dolius` permet de diviser sa charge de travail selon une formule booléenne ϕ quelconque, l'un des esclaves considérant cette formule ϕ vraie, tandis que l'autre fait l'hypothèse que sa négation $\neg\phi$ est vraie. La division du travail a donc pour effet d'obtenir un arbre de résolution dont la racine est la formule initiale Σ et les feuilles les formules simplifiées par les *guiding paths* ϕ_i successifs. Un exemple d'un tel arbre est montré dans la Figure 2

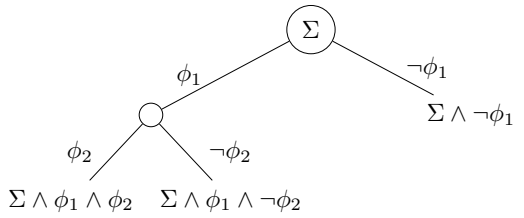


FIGURE 2 – Une exemple de division de travail

Formellement, l'ensemble des *guiding paths* doit vérifier les conditions suivantes :

Propriété 1 *Soit Σ la formule propositionnelle à résoudre et $\phi_1, \phi_2, \ldots \phi_n$ les guidings paths successifs utilisés pour diviser Σ. La condition suivante doit alors être vérifiée : $\Sigma \wedge \bigwedge_{i=1}^{n} \phi_i$ est SAT.*

Triviale dans le cas d'une division simple, cette propriété ne l'est plus dans le cas de divisions en utilisant des formules plus complexes. Nous étudions en ce moment des *guiding paths* dont la structure même nous assurent qu'ils sont mutuellement consistants. Clairement, qu'il s'agisse d'une simple variable ou d'une formule booléenne, le choix du *guiding path* est un élément déterminant pour l'efficacité de la procédure en général. Dans la première version de notre plate forme, nous avons fait le choix de conserver une division simple. Ainsi, l'esclave recevant une requête de division de travail sélectionne la variable ayant le VSIDS le plus important [15]. Cette valeur heuristique, utilisée au sein des solveurs CDCL comme choix de variable, est connue pour désigner les variables centrales de la formule [19]. Ainsi, diviser sur la variable présentant le score le plus important a pour but de scinder la formule sur une variable jugée pertinente.

En pratique, chaque esclave est une instance de `Penelope` travaillant sur plusieurs cœurs. Une difficulté supplémentaire apparait donc quand au choix de la variable de séparation. Pour l'instant, nous sélectionnons le cœur ayant généré le plus de conflits et sélectionnons sa "*meilleure*" variable. Il est clair que ce choix heuristique doit être amélioré et des expérimentations sont actuellement réalisées afin de déterminer un meilleur critère de sélection. Mais, à terme, nous souhaitons vivement déterminer une stratégie de séparation utilisant des formules plus complexes qu'une simple clause unitaire.

			5 esclaves			10 esclaves			20 esclaves		
Instance	SAT ?	`Penelope`	min	moy	F	min	moy	F	min	moy	F
ACG-15-5p1	O	97	101	127	0	93	118	2	79	112	1
traffic_3_uc_sat	O	43	90	209	0	82	105	0	77	85	0
q_query_3_L70_coli.sat	O	74	81	2485	1	63	88	1	80	84	0
q_query_3_L150_coli.sat	N	150	288	368	0	319	440	0	345	534	0
traffic_pcb_unknown	N	269	239	301	0	194	279	0	176	221	0
SAT_dat.k80	N	770	-	-	5	-	-	5	-	-	5
sortnet-8-ipc5-h19-sat	O	752	1173	1173	4	720	720	4	-	-	5
aes_64_1_keyfind_1	O	-	568	568	4	128	372	1	716	718	3
partial-10-13-s	O	-	576	797	3	185	284	3	376	470	2
AProVE07-01	N	-	-	-	5	1101	1164	2	558	638	1
UTI-20-10p1	O	-	-	-	5	-	-	5	766	766	4
eq.atree.braun.12.unsat	N	-	-	-	5	-	-	5	354	354	4

TABLE 1 – Résumé des résultats pour 12 instances. Pour chaque instance on affiche si elle est satisfiable, le temps de résolution nécessaire à `Penelope`, et le temps minimum (min) moyen sur les runs réussis (moy) et le nombre d'echecs (F) pour 5, 10 et 20 esclaves.

4.4 Transfert d'informations

Il est connu que la gestion des clauses apprises est un composant très important des solveurs CDCL [3] et des approches coopératives comme `ManySAT` ou `Penelope` [10, 1]. Dans le cas de notre approche distribuée ce problème perdure. Comme nous le montrerons dans la partie expérimentale, les performances de `Dolius` s'écroulent si l'esclave en manque de travail ne reçoit pas de clauses apprises en plus du *guiding path*. En effet, l'esclave nouvellement créé doit alors recommencer la recherche depuis le début et ne profite d'aucun effort fait précédemment. Dès lors, il faut déterminer quelles clauses apprises les esclaves partagent. Dans cette première version de `Dolius` + `PeneLoPe`, nous avons décider de partager toutes les clauses ayant un LBD [3] inférieur à 15. Les critères de sélection des nogoods à partager peuvent influer grandement, et des études plus approfondies sont encore nécessaires pour optimiser ce composant de `Dolius`.

Un autre type d'information que nous n'avons malheureusement pas eu le temps de mettre en œuvre concerne l'initialisation de l'heuristique : il semble en effet intéressant d'initialiser la recherche dans le même état que l'esclave qui partage son travail. La structure de `Dolius` permet de telles approches, que nous prévoyons de tester dans de futurs travaux. C'est une des nombreuses perspectives offertes par `Dolius`.

5 Évaluation expérimentale

Les expérimentations conduites dans cet article sont réalisées sur des bi-processeurs `Intel XEON X5550` 4 cœurs à 2.66 GHz avec 8Mo de cache et 32 Go de RAM, sous Linux CentOS 6 (kernel 2.6.32). Chaque esclave utilise 8 *threads*. Le temps limite alloué pour résoudre une instance est de 1200 secondes WC (Wall Clock) (Les temps seront toujours donnés en secondes). Nous considérons ici le temps réel (WC) plutôt que le temps CPU car ce dernier n'intègre que le temps d'activité des unités de calcul, et occulte donc certains aspects, comme les temps d'attente des processus lors des communications réseau. Nous avons choisi un pool de 12 instances de difficulté variable issues de la compétition SAT 2011. Chaque instance a été testée avec 5, 10 et 20 esclaves.

Précisons qu'il est délicat d'évaluer expérimentalement une plate-forme de cette nature. C'est probablement l'une des raisons pour lesquelles depuis des années, aucun solveur de type diviser pour régner n'est soumis aux compétitions SAT. En effet, ce type d'approche n'est pas déterministe par nature, ni en temps d'exécution, ni dans la solution (preuve de réfutation ou modèle) reportée. Plusieurs exécutions de `Dolius` + `PeneLoPe` sur un même ensemble de machines peuvent amener à des résultats disparates, dont la variance peut être plus ou moins grande. Ceci est inhérent à la plupart des processus distribués en informatique. Afin d'obtenir des résultats viables, nous avons donc lancé chaque instance à 5 reprises.

La table 1 donne une vue globale de l'ensemble des résultats obtenus. Nous avons également mis les résultats obtenus par `Penelope` (correspondant à `Dolius` avec un seul esclave).

Jetons tout d'abord un œil aux 5 premières instances de ce tableau. Elles sont (relativement) faciles pour `PeneLoPe` et il semble que l'approche distribuée ne soit pas adaptée à ce type de formules simples. Pour

					Attente		
E	Temps	Mo	req	dernière	somme	moyenne	médiane
5	-	-	56	19.77	66.70	13.34	13.70
5	-	-	46	14.18	50.75	10.15	11.02
5	-	-	46	14.30	51.13	10.23	11.00
5	576	2	47	15.32	63.90	12.78	12.79
5	1019	1	48	14.10	50.23	10.05	10.92
10	-	-	141	27.30	143.47	14.35	14.93
10	-	-	160	32.97	162.61	16.26	16.79
10	-	-	211	31.40	201.61	20.16	20.86
10	383	4	164	42.26	175.99	17.60	18.17
10	185	2	182	24.37	161.81	16.18	16.49
20	-	-	472	35.15	376.62	18.83	19.16
20	-	-	466	37.34	352.25	17.61	18.80
20	637	8	415	30.82	344.25	17.21	17.21
20	376	7	438	49.17	330.61	16.53	16.95
20	398	5	479	21.05	325.88	16.29	17.58

TABLE 2 – Détail des résultats pour l'instance partial-10-13-s. Chaque ligne correspond à un lancement. On y reporte le nombre d'esclaves (E), Le temps nécessaire à la résolution, le nombre de demandes de travail (req), la quantité totale transférée sur le réseau en mégaoctet (Mo), La dernière demande de travail par un esclave, la somme, moyenne, et médiane des temps d'attente de chaque esclave.

de tels problèmes, notre implémentation fournit même de moins bons résultats qu'une approche centralisée, et dans certains cas, l'approche distribuée est d'ailleurs incapable de répondre à chaque exécution. Les 2 instances qui suivent (SAT_dat.k80 et sortnet-8-ipc5-h19-sat) présentent plus de difficultés pour `PeneLoPe`. Ici, `Dolius` obtient des résultats décevants, puisque dans les deux cas, l'approche contenant 20 esclaves ne parvient jamais à résoudre l'instance. Les versions contenant moins d'esclaves réussissent quant à eux une seule fois.

Les dernières instances listées dans la table 1 sont en revanche des instances difficiles, pour lesquels `PeneLoPe` ne réussit pas à déterminer la cohérence en moins de 1200 secondes. On commence à voir ici l'intérêt d'une approche distribuée puisque sur de telles instances, l'utilisation d'un certain nombre d'esclaves (chacun d'eux étant basés sur `PeneLoPe`) permet de trouver une solution là où l'approche portofolio seule montre ses limites. En effet, l'utilisation de ces ressources supplémentaires permet l'obtention d'une réponse lors de certaines exécutions (d'autres terminent en échec), en des temps de calcul très raisonnables pour des problèmes de cette difficulté.

Encore une fois, `Dolius` est un projet de longue haleine, dont nous ne présentons ici qu'un travail préliminaire. Il est clair que `Dolius` peut être plus finement paramétré afin d'accroître son efficacité pratique, toutefois les résultats obtenus par sa première version sont très prometteurs.

La table 2 donne des détails sur l'ensemble des runs de l'instance partial-10-13-s. Malheureusement, lorsque les instances ne sont pas résolues en 1200 secondes (-) nous n'avons pas la possibilité d'afficher les transferts réseaux réalisés. Comparons tout d'abord les lancements avec des nombres d'esclaves différents. Il est clair que le nombre de demandes de travail et la quantité transférée sur le réseau augmente avec le nombre de travailleurs. Il est par contre difficile de mettre en relation le temps d'exécution avec le nombre de travailleurs : nous ne sommes actuellement pas en mesure de proposer une méthode assurant une diminution du temps de calcul en augmentant le nombre de travailleurs.

La table 2 donne par contre des indications très intéressantes sur la division du travail. En effet, pour un nombre de travailleurs donné, le nombre de requêtes demandées et le temps où tous les esclaves travaillent jusqu'à la fin sont relativement proches y compris lorsque le solveur échoue à trouver une solution après 1200 secondes. Dans ce cas, la division de travail échoue quelque peu : Aucun espace de recherche n'est prouvé insatisfiable et l'affectation de certaines variables n'aide pas à la détection d'un modèle ; d'où l'importance du choix du *guiding path*.

6 Conclusion

Dans cet article, nous avons proposé un cadre permettant de résoudre le problème SAT de manière pa-

rallèle. Cette première version est capable de résoudre des instances très difficiles pour les solveurs centralisés. Il est de toute façon clair qu'une approche distribuée n'est utile que pour des instances très difficiles, le surcoût dû aux différents demandes de travail, transfert, etc. s'avérant rédhibitoires sur des instances dont le temps de résolution séquentiel est relativement court.

Clairement, le choix du *guiding path* est un élément important pour un tel algorithme. La séparation sur une seule variable (celle avec le VSIDS le plus élevé) permet d'obtenir des premiers résultats relativement satisfaisants. Il est toutefois clair qu'il est possible d'améliorer l'efficacité de la procédure en divisant avec soin le travail à accomplir. L'utilisation de formules plus générales que de simples clauses unitaires constitue également un point qu'il est important d'étudier.

De manière plus générale, ce travail offre de nombreuses perspectives de travail. Le choix de l'information à transférer entre esclaves, le temps avant qu'un esclave accepte de partager son travail, l'étude des cas pathologiques comme le cas du *ping-pong* sont autant de pistes de recherches que nous allons étudier dans les prochains mois.

Enfin, en mettant en place une API qui permettra à tout un chacun d'utiliser la plateforme `Dolius`, nous espérons contribuer à l'étude des solveurs SAT sur les environnements distribués.

Références

[1] Gilles Audemard, Benoît Hoessen, Said Jabbour, Jean-Marie Lagniez, and Cédric Piette. Revisiting clause exchange in parallel SAT solving. In *Proceedings of SAT'12*, pages 200–213, 2012.

[2] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Saïs. On freezeing and reactivating learnt clauses. In *proceedings of SAT'11*, pages 147–160, 2011.

[3] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of IJCAI'09*, pages 399–404, 2009.

[4] Armin Biere. (p)lingeling. `http://fmv.jku.at/lingeling`.

[5] Wahid Chrabakh and Richard Wolski. The gridsat portal : a grid web-based portal for solving satisfiability problems using the national cyberinfrastructure. *Concurrency and Computation : Practice and Experience*, 19(6) :795–808, 2007.

[6] Stephen Cook. The complexity of theorem proving procedures. In *Proceedings of STOC'71*, 1971.

[7] Adnan Darwiche and Knot Pipatsrisawat. *Complete Algorithms*, chapter 3, pages 99–130. IOS Press, 2009.

[8] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. Parallel multithreaded satisfiability solver : Design and implementation. *Electronic Notes in Theoretical Computer Science*, 128(3) :75–90, 2005.

[9] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Control-based clause sharing in parallel SAT solving. In *Proceedings of IJCAI'09*, pages 499–504, 2009.

[10] Youssef Hamadi, Said Jabbour, and Lakhdar Saïs. Manysat : a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :245–262, 2009.

[11] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning sat instances for distributed solving. In *Proceedings of LPAR'10*, pages 372–386, 2010.

[12] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Grid-based SAT solving with iterative partitioning and clause learning. In *Proceedings of CP*, pages 385–399, 2011.

[13] Bernard Jurkowiak, Chu Min Li, and Gil Utard. Parallelizing satz using dynamic workload balancing. *Electronic Notes in Discrete Mathematics*, 9 :174–189, 2001.

[14] Joao Marques-Silva and Karem. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *Proceedings of ICCAD'96*, pages 220–227, 1996.

[15] Ruben Martins, Vasco Manquinho, and Ines Lynce. Improving search space splitting for parallel SAT solving. In *Proceedings of ICTAI'10*, pages 336–343, 2010.

[16] Mikhail Posypkin, Alexander Semenov, and Oleg Zaikin. `SAT@HOME` web page. `http://sat.isa.ru/pdsat`, 2008.

[17] C.B. Ries. *BOINC : Hochleistungsrechnen mit Berkeley Open Infrastructure for Network Computing*. Springer, 2012.

[18] Sven Schulz and Wolfgang Blochinger. Parallel SAT solving on peer-to-peer desktop grids. *Journal of Grid Computing*, 8(3) :443–471, 2010.

[19] Laurent Simon and George Katsirelos. Eigenvector centrality in industrial SAT instances. In *Proceedings of CP'12*, pages 348–356, 2012.

[20] Lintao Zhang, Connor Madigan, Matthew Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *proceedings of ICCAD'01*, pages 279–285, 2001.

Du glucose en goutte à goutte pour les coeurs inconsistants

Gilles Audemard[1] **Jean-Marie Lagniez** [1] **Laurent Simon**[2]

[1] Univ. Lille-Nord de France – CRIL/CNRS UMR8188 – Lens, F-62307, France
[2] Univ. Paris-Sud – LRI/CNRS UMR 8623 / INRIA Saclay – Orsay, F-91405, France

audemard@cril.fr lagniez@cril.fr simon@lri.fr

Résumé

Les dernières avancées réalisées dans le cadre de la résolution pratique du problème SAT ont rejailli bien au delà de ses frontières. Ainsi, à l'heure actuelle, de nombreux problèmes dont la classe de complexité est supérieure à NP peuvent être traités de manière pratique. Pour la plupart, leur résolution consiste à appeler un solveur SAT sur plusieurs instances analogues. Ce type de résolution, appelé résolution incrémentale de SAT, est en passe de devenir l'état de l'art dans bien des domaines. Il devient nécessaire de concevoir les nouveaux démonstrateurs SAT afin de prendre en considération ce nouveau paradigme d'utilisation. Dans cet article, nous proposons d'améliorer le démonstrateur SAT `Glucose` afin d'en faire un démonstrateur incrémental efficace. Pour cela, nous étendons la notion de qualité de clauses apprises dans le cas intensif de démonstrateur incrémental utilisant des hypothèses. Afin de valider expérimentalement nos contributions, nous avons étudié ces performances sur une utilisation typique des démonstrateurs SAT incrémentaux : la recherche de noyaux (coeurs) minimaux inconsistants. Nous pensons que ces améliorations peuvent directement bénéficier à la plupart des autres applications basées sur les démonstrateurs SAT incrémentaux.

1 Introduction

Décider de la satisfiabilité d'une formule propositionnelle fait partie des questions fondamentales en intelligence artificielle. Ce problème, nommé *problème de SATisfiabilité* (SAT), a grandement été étudié pratiquement et théoriquement depuis les années 70. Étant donné sa place centrale dans la théorie de la complexité (premier problème a avoir été prouvé NP-complet par Cook [8]), il est l'un des problèmes de référence car il permet de capturer la difficulté d'un large panel d'autres problèmes provenant d'applications très diverses. Ainsi, les progrès réalisés dans le contexte de la résolution pratique du problème SAT ont un impact direct sur la résolution en pratique de nombreux autres problèmes pouvant s'y réduire ou pouvant être résolu par plusieurs appels successifs à un oracle NP.

Depuis l'avénement des solveurs SAT modernes [17, 10], initialement introduits par Marques-Silva et Sakallah [23], les démonstrateurs peuvent résoudre des problèmes de plus en plus conséquent (des millions de clauses et de variables). Puisque de nombreux problèmes peuvent se réduire facilement au problème SAT, l'amélioration de ces solveurs a eu un impact direct dans de nombreux autres domaines d'application. Ainsi, à l'heure actuelle, les méthodes les plus efficaces pour la résolution de nombreux problèmes NP-complets consistent à utiliser un démonstrateur SAT. De plus, ce gain de performance laisse envisager la possibilité de traiter pratiquement, à l'aide de solveurs SAT, des problèmes au-delà de NP nécessitant l'appel à plusieurs oracles NP afin de pouvoir être traités [6, 24].

Ainsi, depuis quelques années une nouvelle utilisation des démonstrateurs SAT, appelé "SAT incrémental", a vu le jour afin de répondre à ce nouveau type de besoin. Ce mécanisme, initialement proposé dès les premières versions de `Minisat` [10], est aujourd'hui une technique utilisée dans des applications très diverses : vérification formelle bornée [11], extraction de noyaux inconsistants [18], MAXSAT [12], ou encore en vérification inductive [7]. Dans ce contexte, les démonstrateurs SAT ne sont pas lancés une seule fois sur des formules potentiellement énormes, mais sont potentiellement appelés des milliers de fois sur des instances proches les unes des autres avec des clauses ajoutées et/ou supprimées à chaque appel. Ainsi, la proximité des différents problèmes laisse entrevoir la pos-

sibilité de ré-utiliser un maximum les informations pouvant être collectées entre deux appels successifs au démonstrateur. Malgré tout, il est clair que lorsque des clauses sont supprimées en cours de route, les informations pouvant être dérivées (clauses apprises) à partir de ces dernières ne peuvent plus être ré-utilisées. Pour palier ce problème, il est nécessaire d'ajouter des hypothèses au démonstrateur SAT. Ces hypothèses sont des littéraux assignés à chaque clauses et qui sont affectés au début de la recherche.

Dans cet article, nous prenons un cas d'utilisation typique des démonstrateurs SAT incrémentaux, qui est la recherche de noyaux minimum inconsistants [14, 18, 22], et nous nous focalisons uniquement à perfectionner le moteur SAT afin d'améliorer les performances globales de l'extracteur de MUS. Pour effectuer cela, nous avons choisi d'étudier expérimentalement notre travail sur `Muser` [4], qui est un extracteur MUS open source très efficace et facilement modifiable. Nous nous intéressons plus particulièrement à améliorer les performances de `glucose` [2] lorsque celui-ci est utilisé comme moteur SAT. Ainsi, nous espérons que les améliorations induites pourront directement être utilisées dans d'autres applications typiques de SAT incrémental.

2 SAT et SAT Incrémental

Les démonstrateurs SAT "*modernes*" sont basés sur le paradigme CDCL (*conflict driven clause learning*) [17]. Bien qu'à l'origine, ils ont été proposés comme une extension de l'algorithme DPLL [9] comprenant l'apprentissage de clauses [23, 25], il est maintenant admis qu'ils doivent être considérés comme un mélange d'algorithmes de retour arrière et de moteurs de résolution. Ces démonstrateurs intègrent également de nombreuses techniques telles que : l'heuristiques de choix de variable dynamique [17], le choix de la polarité des variables [21], la stratégie de suppression des clauses apprises [13, 2, 1] et de redémarrages [5, 3] (le lecteur pourra se référer à [16] pour une description détaillée des démonstrateurs CDCL). Leur haut degré de sophistication leur permet aujourd'hui de traiter des problèmes applicatifs conséquents [15].

2.1 SAT incrémental avec sélecteurs

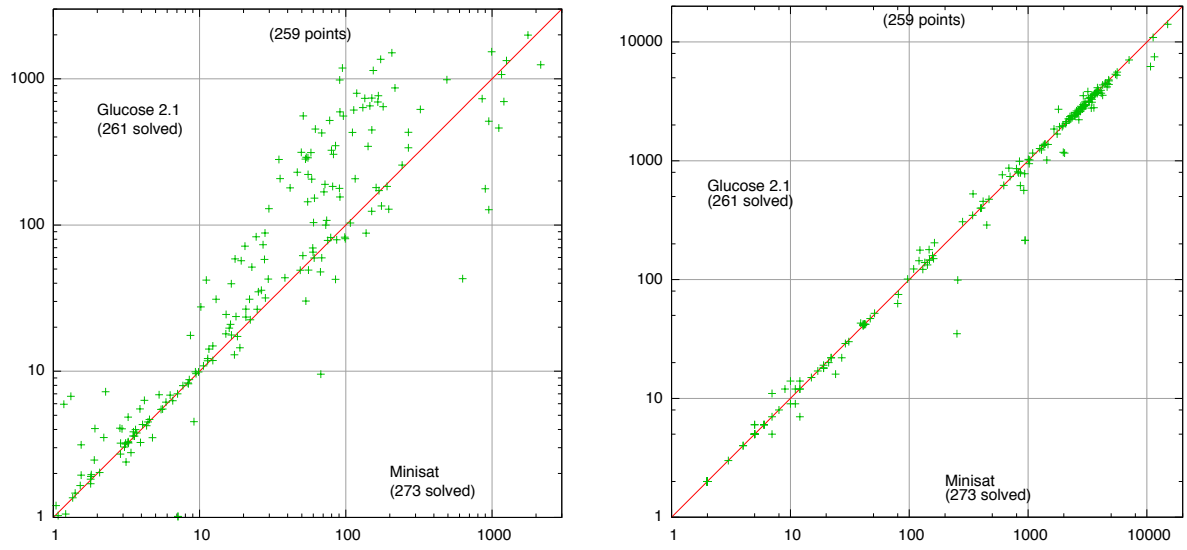
Dans cet article, nous nous concentrons sur l'amélioration des démonstrateurs SAT incrémentaux (voir [19] pour une description détaillée). Dans ce type d'approche, le même démonstrateur est lancé sur un certain nombre d'instances proches les unes des autres et l'état du solveur est mémorisé entre chaque appel. Par exemple, les heuristiques de choix de variables [17], les choix de polarité [21] ou encore les stratégies de redémarrages et de nettoyage de la base de clauses apprises [13, 2, 1] sont maintenus en l'état entre deux appels successifs. Dans le cadre incrémental, certaines clauses peuvent être supprimées entre deux appels au démonstrateur. Il n'est alors pas possible de ré-utiliser directement les clauses apprises générées à l'aide de ces dernières. Pour palier ce problème, il est possible d'ajouter des hypothèses aux démonstrateurs. Celles-ci sont caractérisées par un ensemble $\mathcal{A}$ de littéraux qui sont choisis comme variables de décisions et affectés à vrai avant toute autre variable de décision (une autre possibilité est donnée dans [19]). Ainsi, si durant la recherche une variable issue des hypothèses doit absolument changer de polarité, alors le problème est UNSAT suivant l'ensemble de littéraux $\mathcal{A}$.

Quand une hypothèse est utilisée pour activer/désactiver une clause, celle-ci doit alors être associée à une nouvelle variable (s_i) du problème, qui est appelée *sélecteur* pour cette clause. La variable s_i est alors ajoutée à la clause. Si le littéral associé est faux (resp. vrai) suivant les hypothèses courantes, alors la clause est activée (resp. désactivée). Les sélecteurs n'apparaissant que positivement dans la formule, les clauses apprises obtenues au cours de la recherche gardent donc une empreinte (le sélecteur s_i associé) de toutes les clauses initiales de la formule utilisées pour les produire. Ainsi, en affectant à vrai le sélecteur s_i, on désactive non seulement la clause associée mais également toutes les clauses apprises dérivées à partir de celle-ci. Ceci est illustré dans l'exemple suivant :

Exemple 1 *On considère une formule contenant (entre autres) les clauses $c_1 = a \vee b$ et $c_2 = \neg a \vee b \vee c$. On ajoute donc deux sélecteurs s_1 et s_2 associés aux clauses c_1 et c_2. Les clauses c_1 et c_2 de la formule initiale sont supprimées et remplacées par $(c_1 \vee s_1)$ et $(c_2 \vee s_2)$. Lorsque l'on choisi d'affecter comme hypothèse s_1 et $\neg s_2$ alors la clause c_1 est activée pendant cet appel à SAT et la clause c_2 est désactivée. Si lors des précédents appels au moteur SAT on a appris la clause (les sélecteurs de ces deux clauses étaient alors faux) issue de la résolution entre c_1 et c_2 et égale à $b \vee c \vee s_1 \vee s_2$ alors cette clause apprise ne sera activée que lorsque les hypothèses s_1 et s_2 seront fausses.*

2.2 Utiliser Glucose dans le problème d'extraction de noyau minimum inconsistant

Nous allons nous focaliser sur une application typique utilisant SAT incrémental ainsi que les sélecteurs. Il s'agit du problème de calcul de MUS (*Minimum Unsatisfiable Set*) sur une formule insatisfiable [20]. Ce problème consiste à extraire une sous formule également insatisfiable de la formule initiale, telle que toute sous formule plus petite soit satisfiable. Dans cette approche, un sélecteur est ajouté à chaque clause initiale et de nombreux appels SAT sont effectués en activant/désactivant les clauses initiales afin de détecter les MUS. Cette application est donc typique et difficile pour les démonstrateurs SAT incrémentaux : le nombre d'hypothèses est très important (égal aux



(a) Temps de résolution

(b) Nombre d'appels au moteur SAT

FIGURE 1 – Comparaison de `Glucose` 2.1 et de `Minisat` comme moteur de `Muser`. La figure de gauche est relative au temps CPU (en secondes), la figure de droite est relative au nombre d'appels au moteur SAT

nombre de clauses initiales) et de nombreux appels au solveur SAT sont effectués. Cette application rentre donc bien dans notre objectif principal, à savoir l'amélioration des moteurs incrémentaux. Nous avons basé notre travail sur `Muser` [4], un extracteur de MUS open source et efficace. Rappelons que nous ne travaillons pas du tout du coté des algorithmes de recherche de MUS, mais uniquement du coté du moteur SAT. Nous espérons qu'en améliorant les performances des moteurs SAT (vu comme des boites noires) nous pouvons améliorer les performances de l'extracteur de MUS. Nous avons choisi d'utiliser `Muser` avec ces options par défaut, i.e. l'algorithme hybride est utilisé (essentiellement basé sur l'oubli de clauses) avec l'affinage des ensembles de clauses et la rotation de modèles (voir [4] pour plus de détails). Nous utilisons les 300 instances issues de la catégorie MUS de la compétition SAT 2011. Toutes les expérimentations ont été réalisées sur un cluster Intel XEON X5550 quatre coeurs 2.66 GHz avec 32Go de RAM avec un temps CPU limite de 2400 secondes.

Si, entre chaque appel au démonstrateur SAT, on veut pouvoir ré-utiliser des clauses apprises, il est alors essentiel de déterminer quelles clauses seront utiles pour les futurs appels. Dans ce contexte, nous avons choisi d'utiliser le démonstrateur `Glucose` [2], une variante de `Minisat` [10], basé sur un score (appelé LBD) évaluant la qualité des clauses apprises. Nous avons donc démarré notre travail en comparant `Glucose` contre `Minisat` comme moteur SAT de `Muser`. Les résultats, étonnamment mauvais, sont résumés dans la figure 2 (pour chaque figure de ce type, nous donnons le nombre d'instances résolues par chaque méthode, le nombre d'instances résolues par les deux (259 pour la figure 1(a) ; moins il y a de points dans la région d'une méthode, meilleure est cette dernière comparée à l'autre). Malgré des résultats très mauvais pour `Glucose`, les deux parties de la figure 2 sont intéressantes à observer. Le temps CPU est clairement en faveur de `Minisat`. Mais la figure 1(b), comparant les deux approches du point de vue du nombre d'appel au moteur SAT, montre que la plupart du temps, les deux approches nécessitent approximativement autant d'appels démonstrateur SAT, avec dans certains cas un nombre beaucoup plus faible lorsque `Glucose` est utilisé. Si nous regardons plus en détails, sur les 259 instances résolues par `Minisat` et `Glucose`, 103 nécessitent moins d'appels au démonstrateur si `Glucose` est utilisé, 68 plus d'appels si `Glucose` est utilisé, et dans 88 cas on obtient le même nombre d'appels. Si nous arrivons à réduire le temps nécessaire pour chaque appel au moteur `Glucose`, nous pouvons donc espérer obtenir de très bons résultats.

Dès lors, comment interpréter des résultats aussi décevants. `Glucose` met à jour les scores des clauses durant le processus de propagation unitaire. Avec des clauses apprises pouvant avoir des milliers de littéraux (les sélecteurs apparaissent dans ces clauses), cela peut avoir un surcoût prohibitif. Néanmoins, il est aussi important de noter que `Minisat` a du mal à résoudre des problèmes de recherche de MUS difficiles (observation non reportée), en partie à cause de problèmes de mémoire. Sur de tels problèmes, on

		LBD					Nouveau LBD				
			taille		LBD			taille		LBD	
Instance	#C	temps	moyenne	max	moyenne	max	temps	moyenne	max	moyenne	max
fdmus_b21_96	8541	29	1145	5980	1095	5945	11	972	6391	8	71
longmult6	8853	46	694	3104	672	3013	14	627	2997	11	61
dump_vc950	360419	110	522	36309	498	35873	67	1048	36491	8	307
g7n	15110	190	1098	16338	1049	16268	75	1729	17840	27	160

TABLE 1 – Pour quelques instances significatives, nous reportons le nombre de clauses (#C), et pour chaque définition de LBD (initiale et nouvelle), nous donnons aussi le temps nécessaire pour calculer le MUS, la taille moyenne et max des clauses apprises et la valeur moyenne et max des LBD.

va lancer le moteur SAT un grand nombre de fois avec de nombreuses clauses apprises et de nombreux sélecteurs par clause. La capacité de `Glucose` à maintenir efficacement les *bonnes* clauses apprises peut alors s'avérer cruciale. Nous montrons dans la partie suivante comment adapter les mécanismes de `Glucose` dans le cadre incrémental, avec, au final, des améliorations sensibles par rapport à `Minisat`.

3 Améliorer les moteurs SAT

LBD et sélecteurs

Dans `Minisat`, chaque hypothèse possède son propre niveau de décision (excepté lorsque elle est propagée par une précédente hypothèse). Ainsi, dans de nombreux cas, le score LBD des clauses va être dominé par le nombre de sélecteurs que possède cette clause. Par conséquent, le LBD va très souvent valoir approximativement la taille de la clause. Si on ajoute à cela, le fait que le score LBD est très discriminant (les clauses de LBD égal à $n+1$ sont significativement moins importantes que celles de LBD égal à n), nous devons prendre en compte les sélecteurs dans le calcul du LBD. Nous proposons d'adapter simplement ce score en ne prenant pas en compte les sélecteurs dans son calcul. Ce nouveau score LBD est simplement nommé "Nouveau LBD".

La table 1 montre quelques statistiques relatives aux remarques précédentes. Sur 4 instances représentatives, nous détaillons les score LBD initiaux et les scores des nouveaux LBD. Comme nous pouvons le constater, il est clair que l'utilisation du LBD initial n'est pas du tout adapté dans le cadre de moteurs SAT utilisant les sélecteurs. La valeur des LBD se rapproche significativement de la taille des clauses apprises. Par contre, en utilisant la nouvelle définition des LBD, nous pouvons constater que le temps nécessaire est fortement amélioré. De plus, la valeur des LBD n'est plus du tout reliée à la taille des clauses apprises, mais bien plus petite.

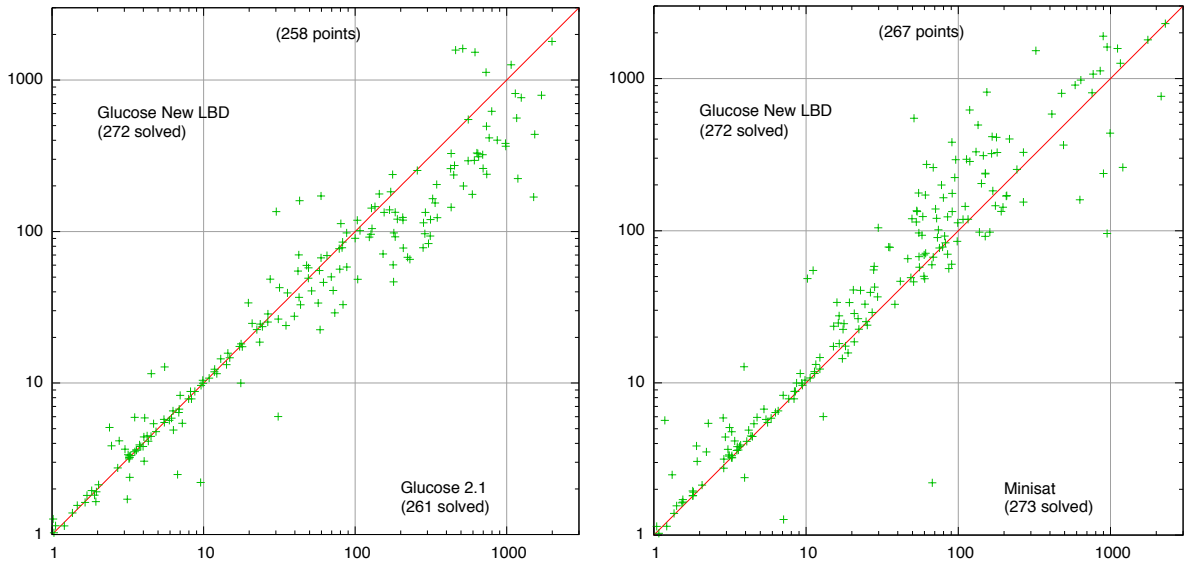
Malgré tout, comme nous pouvons le constater sur la figure 2, même si nous améliorons les performances de `Glucose`, cette nouvelle version obtient des résultats très proches de `Minisat` (par rapport au nombre d'instances résolues), mais plus lente. À ce niveau de nos expérimentations, les résultats sont assez décevants : `Glucose` est supposé être beaucoup plus efficace que `Minisat` dans le cadre non incrémental. Afin, d'en améliorer les performances globales, nous proposons par la suite quelques modifications importantes.

Amélioration des performances

Comme le montre le tableau 1, les clauses apprises peuvent rapidement devenir extrêmement grandes. Par exemple, les clauses apprises ont une taille moyenne de 1729 littéraux pour l'instance `g7n` (avec certaines clauses contenant quelques 10 000 littéraux !). Dès lors, même une opération simple comme le parcours d'une clause peut rapidement avoir un surcoût prohibitif. Ceci est très problématique, puisque de nombreuses opérations fondamentales des solveurs SAT sont justement basées sur le parcours de clauses (propagation unitaire, calcul du LBD, calcul des clauses apprises, suppression des clauses satisfaites...). Nous proposons des améliorations spécialement dédiées à ces opérations.

Parcours efficaces des clauses contenant des sélecteurs

Comme nous venons de le voir, dans de nombreux cas, les clauses apprises vont contenir majoritairement des variables sélecteurs qui n'ont pas à être prises en compte dans le calcul du nouveau LBD. Comme ces littéraux peuvent être utilisés comme témoin (*watchers*) nous ne pouvons malheureusement splitter ces dernières en deux sous ensembles indépendants (sélecteurs/non sélecteurs) : nous devons être à même de visiter ces littéraux particuliers lorsque nécessaire. Il est néanmoins possible d'enregistrer dans chaque clause sa taille et le nombre de sélecteurs qu'elle contient. De plus, lors de la création de chaque clause, nous poussons les sélecteurs à la fin. Nous pouvons espérer qu'un tel arrangement accélérera la mise à jour des



(a) `Glucose` vs. `Glucose` avec le nouveau LBD

(b) `Minisat` vs. `Glucose` avec le nouveau LBD

FIGURE 2 – Comparaison, au sein de `Muser`, des performances de `Glucose` (2.1) en le nouveau LBD. La figure 2(a) compare cette version avec `Glucose` classique. La figure 2(b) compare cette version avec `Minisat`.

LBDs : nous pouvons stopper le parcours de la clause dès lors que tous le littéraux initiaux ont été visités.

Accéleration de la propagation

Si les modifications des structures de données décrites précédemment permettent d'améliorer le calcul de la mise à jour du LBD, nous avons également à faire face à un autre problème : la propagation unitaire nécessite également de parcourir les clauses pour détecter de nouveaux littéraux témoins. Supposons que durant la propagation d'une hypothèse (sélecteur), on cherche un nouveau témoin pour une clause c. Supposons également que le nouveau témoin s_i de la clause c est choisi parmi les autres selecteurs. Ainsi, si s_i est également choisi comme hypothèse (tous les sélecteurs le sont au fur et à mesure), alors la clause c sera à nouveau parcourue. Ceci peut facilement être évité : Lors de la propagation d'une hypothèse, nous parcourons la clause en entier pour trouver un nouveau témoin qui est vrai (la clause est alors satisfaite) ou qui n'est pas un sélecteur. De cette manière, nous pouvons espérons limiter le nombre de clauses à visiter lors de la recherche de nouveaux littéraux témoins. De plus, comme `Glucose` effectue de nombreux restarts, nous limitons également le backtrack au premier niveau de décision qui n'est pas une hypothèse.

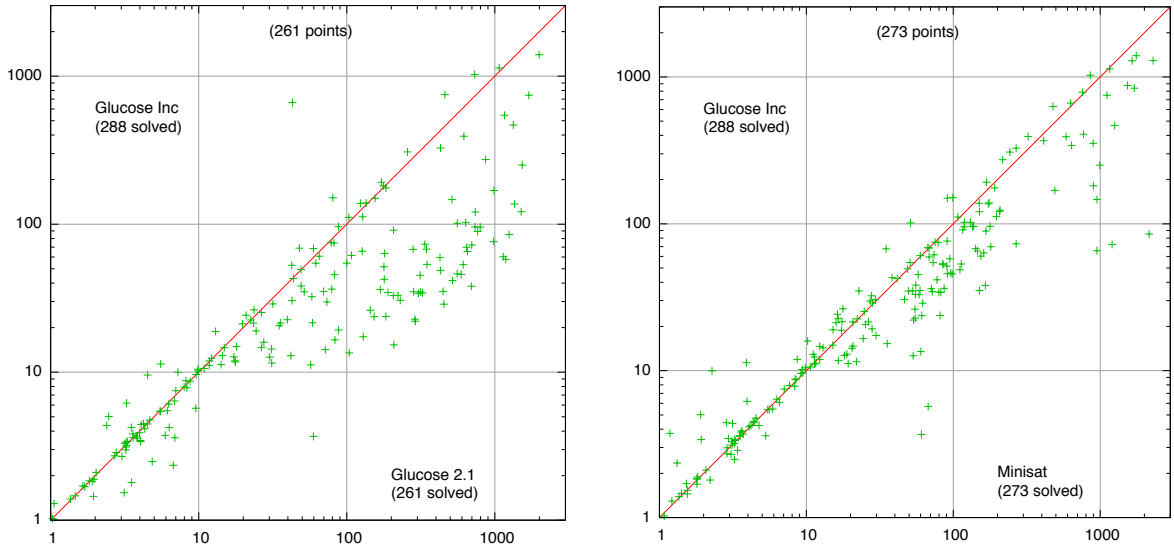
Simplification de la base de clauses

La dernière, et importante, modification que nous avons introduite est reliée à la suppression définitive des clauses satisfaites. C'est très important dans le cas de `Muser` : à chaque fois que l'on détermine qu'une clause est hors d'un MUS, son sélecteur associé est définitivement affectée à vrai (au nivau de décision 0). Ainsi, toutes les clauses apprises où il apparait sont également définitivement satisfaites. Parcourir toutes les clauses apprises pour détecter ces littéraux satisfaits peut s'avérer contre-productif. Dans cette nouvelle version de `Glucose`, nous supprimons uniquement les clauses apprises dont un des deux littéraux témoins est définitivement satisfait. Comme nous avons modifié le processus de propagation unitaire (voir plus haut), nous pouvons espérer que cela soit suffisant pour supprimer la majorité des clauses apprises définitivement satisfaites.

3.1 Comparaison globale

Toutes les modifications proposées on été implantés et nous avons appelé le moteur SAT résultant `GlucoseInc`. La figure 3 compare `GlucoseInc` avec `Minisat`, mais aussi avec `Glucose`. Les résultats sont clairs : cette nouvelle version dépasse les performances des deux autres moteurs SAT. Cela est aussi explicite sur la fameuse courbe cactus de la figure 4.

Comme nous l'avons écrit dans la section 2.2, notre objectif consistait à améliorer les performances des moteurs SAT incrémentaux, et donc, les performances de chaque appel au moteur SAT. La table 2 montre que cet objectif est atteint. Elle montre, pour `Minisat Glucose` et `GlucoseInc`, et pour les 4 mêmes instances que précédemment, le nombre d'appels SAT et le temps moyen né-



(a) running time

(b) number of SAT solver calls

FIGURE 3 – Comparaison, au sein de `Muser`, des performances de `GlucoseInc`. La figure 2(a) compare cette version avec `Glucose` classique. La figure 2(b) compare cette version avec `Minisat`.

cessaire pour chacun d'entre eux. Même dans le cas où plus d'appels à SAT sont réalisés, le temps total pour extraire le MUS est toujours amélioré.

3.2 Autres idées aux résultats négatifs

Habituellement, seuls les « *bons* » résultats sont publiés. Nous avons décidé de passer en revue un certains nombre d'idées prometteuses que nous avons testées, mais qui se sont malheureusement avérées mauvaises au cours des expérimentations. Par exemple, nous avons essayer de détecter les vieilles clauses apprises (créées lors d'un ancien run) et qui n'ont pas été utilisées dans le processus de propagation unitaire depuis un certain temps et avons alors donné une pénalité à leur LBD. Si elles sont à nouveau utilisées, alors le LBD sera remis à jour, sinon ces clauses seront supprimées lors du prochain nettoyage de la base. Nous avons également essayé de favoriser les clauses avec peu de littéraux initiaux en utilisant cette valeur comme second critère de tri. Nous avons finalement essayer de prédire le résultat (SAT/UNSAT) de l'appel courant afin de modifier la stratégie de suppression des clauses apprises. En effet, la plupart des appels SAT sont fait avec très peu de conflits. Nous avons donc essayé d'être plus agressif et de supprimer plus de clauses lorsque de nombreux conflits étaient atteints.

Malheureusement, aucune de ces idées n'a donné de bons résultats. Il faut quand même noter que la nouvelle version de `Muser` obtenue avec `GlucoseInc` est capable de trouver un MUS pour 96% des instances (en 2400 secondes). Il y a des chances (nous avons augmenté le temps CPU à 15000 secondes et n'avons été en mesure de résoudre que 4 instances supplémentaires) que les instances restantes soient très dures et nécessitent alors de nouvelles stratégies quand à la recherche des MUS proprement dite.

4 Conclusion

Une des raisons du succès des démonstrateurs SAT est certainement leur capacité à être utilisés comme des boites noires dans de nombreuses applications. Néanmoins, lorsqu'une nouvelle et très spécifique utilisation est proposée il peut s'avérer nécessaire d'en adapter les démonstrateurs. Ceci est typiquement le cas dans le cadre SAT incrémental. Dans cet article, nous nous somme focalisé sur une des applications SAT incrémentales les plus représentatives, car utilisant de nombreux sélecteurs : la recherche de noyaux minimum inconsistants. Nous avons uniquement travaillé du coté du moteur SAT, c'est-à-dire que nous nous sommes attelé à en améliorer ses performances, afin d'obtenir des meilleures performances au niveau de l'extracteur de noyau. Nous pensons que cette amélioration peut avoir des applications directes dans d'autres domaines utilisant la technologie SAT incrémental.

Ce travail préliminaire offre de nombreuses et intéressantes perspectives de recherche. Par exemple, nous comptons étudier les dépendances entre les sélecteurs afin de les prendre en compte pour améliorer la recherche. Une autre piste de recherche est reliée à l'adaptation des démonstrateurs (par rapport aux heuristiques, redémarrages...) par

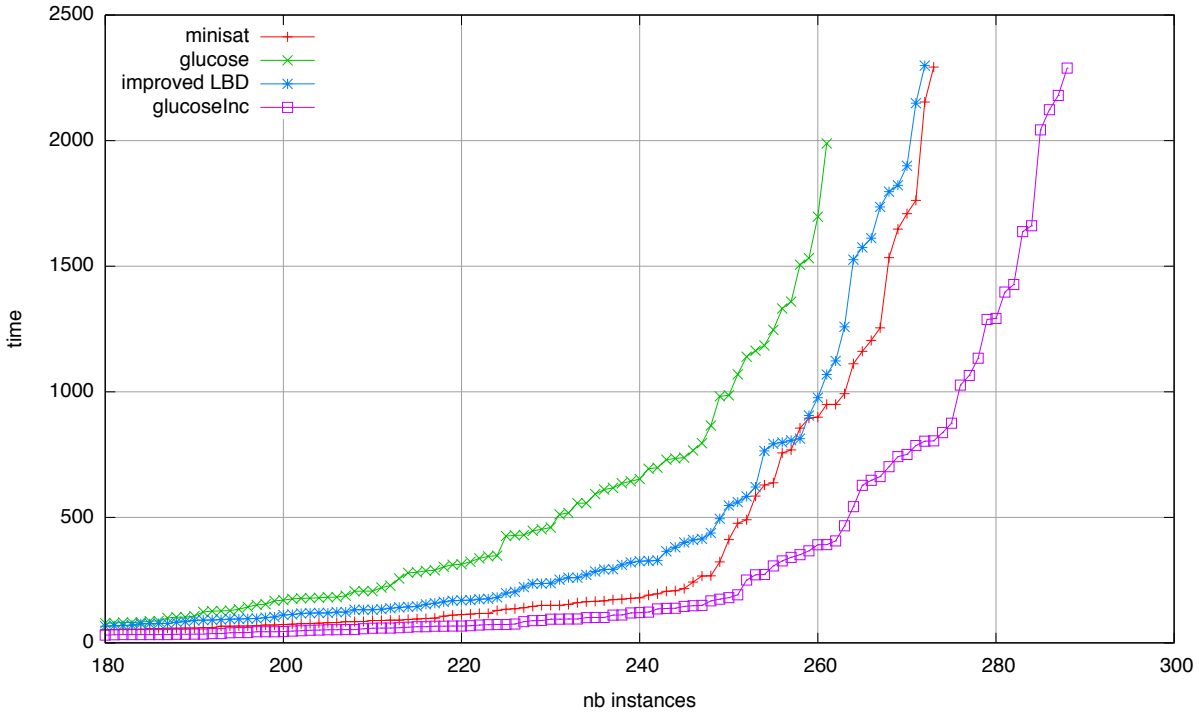


FIGURE 4 – Cactus plot. L'axe de x représente le nombre d'instances. L'axe des y représente le temps nécessaires pour les résoudre si elles sont toutes exécutées en parallèle.

rapport aux lancement précédents.

Références

[1] G. Audemard, J. M. Lagniez, B. Mazure, and L. Saïs. On freezing and reactivating learnt clauses. In *Proc. SAT'11*, pages 188–200, 2011.

[2] G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proc. IJCAI'09*, pages 399–404, 2009.

[3] G. Audemard and L. Simon. Refining restarts strategies for SAT and UNSAT. In *Proc. CP'12*, volume 7514 of *LNCS*, pages 118–126. Springer, 2012.

[4] A. Belov and J. Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *Proc. FMCAD'11*, pages 37–40. FMCAD, 2011.

[5] A. Biere. Adaptive restart strategies for conflict driven sat solvers. In *Proc. SAT'08*, volume 4996 of *LNCS*, pages 28–33. Springer, 2008.

[6] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Proc. DAC'99*, pages 317–320, 1999.

[7] A. Bradley. IC3 and beyond : Incremental, inductive verification. In *proc. of CAV*, 2012.

[8] S. Cook. The complexity of theorem-proving procedures. In *Proc. STOC'71*, pages 151–158. ACM, 1971.

[9] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 1962.

[10] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. SAT'03*, pages 333–336, 2003.

[11] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4) :543 – 560, 2003.

[12] Z. Fu and S. Malik. On solving the partial MAX-SAT problem. In *Proc. SAT'06*, volume 4121 of *LNCS*, pages 252–265. Springer, 2006.

[13] E. Goldberg and Y. Novikov. BerkMin : A fast and robust SAT-solver. In *Proc. DATE'02*, pages 142–149, 2002.

[14] E. Grégoire, B. Mazure, and C. Piette. Extracting muses. In *Proc. ECAI'06*, volume 141 of *Frontiers in Artificial Intel. and Applications*, pages 387–391. IOS Press, 2006.

		Minisat		Glucose		GlucoseInc		
Instance	#C	#SAT calls	avg	#SAT calls	avg	time	#SAT calls	avg
fdmus_b21_96	8541	2103	0.009	2134	0.02	11	2153	0.004
longmult6	8853	706	0.01	1027	0.03	13	748	0.01
dump_vc950	360419	7	135	11	11.5	65	9	7.2
g7n	70492	4791	0.02	4393	0.08	67	4779	0.01

TABLE 2 – Pour des instances représentatives, nous rapportons le nombre de clauses, et, pour chaque moteur SAT incrémental, nous rapportons le nombre d'appels au démonstrateur ainsi que le temps moyen pour chacun de ces appels. Pour GlucoseInc, nous rapportons aussi le temps total nécessaire à la recherche du MUS (pour les autres moteurs SAT, cela est donné dans la table 1

[15] D. LeBerre and L. Simon, editors. *Special Volume on the SAT 2005 competitions and evaluations*, volume 2. Journal on Satisfiability, Boolean Modeling and Computation, 2006.

[16] J. Marques-Silva, I. Lynce, and S. Malik. *Conflict-Driven Clause Learning SAT Solvers*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, page 980. IOS Press, 2009.

[17] M. Moskewicz, C. Conor, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an Efficient SAT Solver. In *Proc. DAC'01*, 2001.

[18] A. Nadel. Boosting minimal unsatisfiable core extraction. In *Proc. FMCAD'10*, pages 221–229, 2010.

[19] A. Nadel and V. Ryvchin. Efficient SAT solving under assumptions. In *Proc. SAT'12*, 2012.

[20] Y. Oh, M.N. Mneimneh, Z.S. Andraus, K.A. Sakallah, and I.L. Markov. AMUSE : a minimally-unsatisfiable subformula extractor. In *Design Automation Conference*, pages 518–523, 2004.

[21] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *Proc. SAT'07*, pages 294–299, 2007.

[22] V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In *Proc. SAT'11*, volume 6695 of *LNCS*, pages 174–187, 2011.

[23] J. P. Marques Silva and K. Sakallah. Grasp – a new search algorithm for satisfiability. In *Proc. CAD'96*, pages 220–227, 1996.

[24] M. Velev and R. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2) :73 – 106, 2003.

[25] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. CAD'01*, pages 279–285, 2001.

Cohérences Locales Paramétrées

Amine Balafrej[1,2] **Christian Bessiere**[1] **Remi Coletta**[1] **El Houssine Bouyakhf**[2]

[1]LIRMM, Université de Montpellier, France
[2]LIMIARF, University Mohammed V Rabat-Agdal, Morocco
{balafrej,bessiere,coletta}@lirmm.fr bouyakhf@fsr.ac.ma

Résumé

Les solveurs de contraintes maintiennent uniformément le même niveau de cohérence locale (généralement la cohérence d'arc) indépendamment de l'instance du problème à résoudre. Nous proposons le concept de *cohérence locale paramétrée*, une approche originale qui permet d'ajuster le niveau de cohérence locale en fonction de l'instance du problème à résoudre et en fonction de la partie du problème dans laquelle s'effectue la propagation. Nous n'utilisons pas comme paramètre l'une des caractéristiques de l'instance, comme le font les portefeuilles de solveurs. Nous utilisons comme paramètre la *stabilité* des valeurs, une caractéristique basée sur l'état de l'algorithme de cohérence d'arc durant son exécution. Les cohérences locales paramétrées appliquent à une valeur, la cohérence d'arc ou un autre niveau de cohérence plus élevé, selon que la stabilité de cette valeur est en dessus ou en dessous d'un seuil donné. L'approche que nous proposons permet d'obtenir un bon compromis entre la capacité d'une cohérence locale à supprimer des valeurs et le coût en temps pour atteindre ce niveau de cohérence dans un réseau de contraintes. Nous validons notre approche sur différents problèmes de la dernière compétition des solveurs de contraintes.

1 Introduction

Les propriétés de cohérence locale constituent un des points forts de la programmation par contraintes (PPC). En effet, la propagation des contraintes en appliquant une de ces propriétés permet aux solveurs de réduire l'espace de recherche en supprimant les valeurs localement inconsistantes. La cohérence d'arc (AC) [5] est la plus célèbre et la plus ancienne propriété de cohérence locale. Elle a la bonne propriété de supprimer des valeurs localement inconsistantes sans modifier la structure du réseau de contraintes. La cohérence d'arc est aussi le niveau standard de cohérence locale maintenue par les solveurs de contraintes. Plusieurs autres propriétés de cohérence locale, ne modifiant pas la structure du réseau de contraintes, qui sont plus fortes que l'AC ont été proposées (telles que la max-cohérence de chemin restreinte (maxRPC) [4] ou la singleton cohérence d'arc (SAC) [4]). Bien qu'elles soient capables de supprimer plus de valeurs inconsistantes que l'AC, ces cohérences sont rarement utilisées en pratique en raison du coût élevé de leur maintien durant la recherche. Cependant, dans certains problèmes, le maintien de la cohérence d'arc devient un surcoût en raison du grand nombre de révisions inefficaces des contraintes, ce qui est pénalisant en temps CPU. Par exemple, dans [6], Stergiou observe que lors de la résolution de l'instance scen11, du problème d'allocation de fréquences de liaison radio (RLFAP) avec un algorithme qui maintient la cohérence d'arc, seulement 27 parmi les 4102 contraintes du problème étaient identifiées comme causant un domaine vide et 1921 contraintes n'ont supprimé aucune valeur. Le choix de la bonne propriété de cohérence locale pour résoudre efficacement un problème de satisfaction de contraintes consiste à trouver un bon compromis entre la capacité de cette cohérence à supprimer des valeurs inconsistantes, et le coût des algorithmes qui permettent de l'atteindre dans un réseau de contraintes. Stergiou suggère d'exploiter la puissance des cohérences fortes pour réduire l'espace de recherche, tout en évitant le coût élevé de leur maintien dans l'ensemble du réseau. Il propose une approche basée sur des heuristiques, qui exploitent des événements qui peuvent se produire durant la propagation, pour adapter dynamiquement le niveau de cohérence locale (AC ou maxRPC) appliqué aux contraintes individuellement. Cette méthode permet d'élaguer plus que la cohérence d'arc et moins que la max-cohérence de chemin restreinte. Mais l'élagage obtenu n'est pas caractéristique d'une propriété de cohérence locale. Nous pouvons converger vers différentes fermetures selon l'ordre

de propagation des contraintes.

Dans ce papier nous définissons la notion de stabilité pour les valeurs. Une notion originale qui n'est pas basée sur les caractéristiques de l'instance à résoudre, mais basée sur l'état de l'algorithme de propagation de la cohérence d'arc pendant son exécution. En se basant sur cette notion, nous proposons les *cohérences paramétrées*, une approche originale qui permet d'ajuster le niveau de cohérence locale utilisé pour résoudre une instance donnée. Une cohérence paramétrée p-LC maintient un niveau intermédiaire de cohérence entre l'AC et une autre cohérence LC plus forte que la cohérence d'arc. Le pouvoir d'élagage de p-LC dépend du paramètre p. Cette approche nous permet de trouver un bon compromis entre la puissance d'élagage d'une cohérence locale et le coût en terme de temps CPU des algorithmes qui permetent de l'atteindre. Nous appliquons le concept de cohérence paramétrée p-LC aux deux propriétés de cohérence locale plus fortes que l'AC les plus connues, qui ne modifient pas la structure du réseau, à savoir maxRPC et SAC. Nous décrivons l'algorithme p-maxRPC3, basé sur maxRPC3 [1], qui réalise la p-max cohérence de chemin restreinte ainsi que l'algorithme p-SAC1, qui réalise la p-singleton cohérence d'arc. Nous évaluons expérimentalement l'intérêt d'utiliser la cohérence locale paramétrée. Et nous montrons qu'en faisant le bon choix du paramètre p, nous profitons du coût réduit de calcul de la cohérence d'arc et l'efficacité d'élagage de LC. Dans le meilleur des cas, un solveur qui maintient un niveau de cohérence intermédiaire p-LC explore le même nombre de nœuds qu'en maintenant LC avec un nombre de vérification de contraintes aussi réduit que lors du maintien d'AC, ce qui entraine une baisse de temps CPU de résolution comparé aux temps obtenus utilisant AC ou LC.

2 Préliminaires

Un *réseau de contraintes* est défini par un ensemble de n variables $X = \{x_1, ..., x_n\}$, un ensemble de domaines ordonnés $D = \{D(x_1), ..., D(x_n)\}$ où $D(x_i)$ est l'ensemble de valeurs possibles pour la variable x_i, et un ensemble de e contraintes $C = \{c_1, ..., c_e\}$. Chaque contrainte c_k est définie par une paire $(var(c_k), sol(c_k))$, où $var(c_k)$ est un sous-ensemble ordonné de X, et $sol(c_k)$ est l'ensemble de combinaisons des valeurs (uplets) que peuvent prendre simultanément les variables de $var(c_k)$. Une contrainte peut être spécifiée en extension par la liste des uplets la satisfaisant, ou en intention par une formule mathématique qui est la fonction caractéristique de la contrainte. Dans la suite, nous nous limitons aux contraintes binaires car certaines propriétés de cohérence locale que nous utilisons sont définies uniquement dans le cas binaire. Cependant, les notions que nous introduisons peuvent être étendues au cas non-binaires. Nous utilisons la notation c_{ij} pour désigner la contrainte impliquant x_i et x_j, et $\Gamma(x_i)$ pour désigner l'ensemble des variables x_j impliquées avec x_i dans une contrainte.

Une valeur $v_j \in D(x_j)$ est appelée *support arc-consistant (support AC)* de $v_i \in D(x_i)$ sur c_{ij} si le tulpe $(v_i, v_j) \in sol(c_{ij})$. Une valeur $v_i \in D(x_i)$ est dite *arc consistante (AC)* si et seulement si pour toute variable $x_j \in \Gamma(x_i)$, v_i possède un support AC $v_j \in D(x_j)$ sur c_{ij}. Un domaine $D(x_i)$ est AC s'il n'est pas vide et toutes les valeurs dans $D(x_i)$ sont AC. Un réseau est AC si tous les domaines dans D sont AC. Si la réalisation de la cohérence d'arc dans un réseau de contraintes N conduit à un domaine vide, on dit que N est arc inconsistant.

Un uplet $(v_i, v_j) \in D(x_i) \times D(x_j)$ est *consistant de chemin (PC)* si et seulement si pour toute variable tierce x_k il existe une valeur $v_k \in D(x_k)$ telle que v_k est un support AC de v_i et v_j. Dans ce cas, v_k est appelée *témoin* de la cohérence de chemin pour (v_i, v_j).

Une valeur $v_j \in D(x_j)$ est appelée *support maxRPC* de $v_i \in D(x_i)$ sur c_{ij} si et seulement si v_j est un support AC de v_i sur c_{ij} et le uplet (v_i, v_j) est consistant de chemin. Une valeur $v_i \in D(x_i)$ est maxRPC sur une contrainte c_{ij} si et seulement si $\exists v_j \in D(x_j)$ support maxRPC de v_i sur c_{ij}. Une valeur $v_i \in D(x_i)$ est maxRPC si et seulement si, pour toute variable $x_j \in \Gamma(x_i)$ v_i possède un support maxRPC $v_j \in D(x_j)$ sur c_{ij}. Un domaine $D(x_i)$ est maxRPC s'il n'est pas vide et toutes les valeurs dans $D(x_i)$ sont maxRPC. Un réseau est maxRPC si tous les domaines dans D sont maxRPC.

Étant donné un réseau de contraintes $N = (X, D, C)$, une valeur $v_i \in D(x_i)$ est dite *Singleton Arc consistante (SAC)* si et seulement si le réseau $N|_{x_i=v_i}$ est arc consistant. Un domaine $D(x_i)$ est SAC s'il n'est pas vide et toutes les valeurs dans $D(x_i)$ sont SAC. Un réseau est SAC si tous les domaines dans D sont SAC.

Une propriété de cohérence locale LC_1 est dite plus forte (ou plus puissante) qu'une autre propriété de cohérence locale LC_2 ($LC_2 \preceq LC_1$) si LC_2 est vérifiée dans tout réseau de contraintes où LC_1 est vérifiée, et qu'il existe des réseaux de contraintes où LC_2 est vérifiée mais pas LC_1.

3 La Cohérence Paramétrée

Dans cette section nous présentons une approche originale qui permet de paramétrer une propriété de cohérence locale LC plus forte que la cohérence d'arc. La force (puissance) de cette cohérence paramétrée est

fonction d'un paramètre p, elle dégénère à la cohérence d'arc lorsque le paramètre est égal à 0, à la cohérence forte LC lorsque le paramètre est égal à 1, et se positionne entre les deux lorsque le paramètre est compris entre 0 et 1. L'idée derrière cela est d'être capable d'ajuster le niveau de cohérence locale en fonction de l'instance du problème à résoudre en espérant qu'un tel niveau de cohérence adapté supprime nettement plus de valeurs inconsistantes que la cohérence d'arc tout en ayant un coût en terme de temps d'exécution inférieur au coût de maintien de LC.

La cohérence paramétrée est basée sur le concept de la stabilité des valeurs. Nous avons besoin de définir un concept qui évalue combien une valeur est loin d'être la dernière dans son domaine. Dans ce qui suit, $rank(v, S)$ désigne la position de la valeur v dans l'ensemble ordonné de valeurs S.

Définition 1 (Distance à la fin pour une valeur) *Pour une valeur $v_i \in D(x_i)$, la distance à la fin est donnée par :*

$$\Delta(x_i, v_i) = (|D_o(x_i)| - rank(v_i, D_o(x_i)))/|D_o(x_i)|,$$

où $D_o(x_i)$ est le domaine initial de x_i

Nous pouvons observer que la première valeur de $D_o(x_i)$ a la distance $(|D_o(x_i)| - 1)/|D_o(x_i)|$, et la dernière valeur a la distance 0. Ainsi, $\forall v_i \in D(x_i), 0 \leq \Delta(x_i, v_i) < 1$.

Nous pouvons maintenant donner la définition de ce que nous appelons la stabilité paramétrée d'une valeur pour la cohérence d'arc.

Définition 2 (p-stabilité pour AC) *Une valeur $v_i \in D(x_i)$ est p-stable pour AC sur c_{ij} si et seulement si v_i possède un support AC $v_j \in D(x_j)$ sur c_{ij}, tel que $\Delta(x_j, v_j) \geq p$. une valeur $v_i \in D(x_i)$ est p-stable pour AC si et seulement si $\forall x_j \in \Gamma(x_i)$, v_i est p-stable pour AC sur c_{ij}.*

Nous sommes maintenant prêts pour donner une première définition de la cohérence locale paramétrée. Cette première définition peut être appliquée sur n'importe quelle cohérence locale LC pour qui la cohérence d'une valeur sur une contrainte est bien définie. C'est le cas par exemple pour toutes les cohérences dites triangle-based [2].

Définition 3 (Constraint-based p-LC) *Soit LC une cohérence locale plus forte que l'AC pour qui la cohérence d'une valeur sur une contrainte est définie. Une valeur $v_i \in D(x_i)$ est constraint-based p-LC sur c_{ij} si et seulement si elle est p-stable pour AC sur c_{ij}, ou elle est LC sur c_{ij}. Une valeur $v_i \in D(x_i)$ est constraint-based p-LC si et seulement si $\forall c_{ij}$, v_i est constraint-based p-LC sur c_{ij}. Un réseau de contraintes est constraint-based p-LC, si et seulement si toutes les valeurs dans tous les domaines dans D sont constraint-based p-LC.*

Théorème 4 *Soit LC une cohérence locale plus forte que l'AC pour qui la cohérence d'une valeur sur une contrainte est définie. Soit p_1 et p_2 deux paramètres dans [0..1]. Si $p_1 < p_2$ alors AC $\preceq$ constraint-based p_1-LC $\preceq$ constraint-based p_2-LC $\preceq$ LC.*

Preuve. Supposons qu'il existe deux paramètres p_1 et p_2 tels que $0 \leq p_1 < p_2 \leq 1$, et supposons qu'il existe un réseau de contrainte N qui est p_2-LC et qui contient une valeur (x_i, v_i) qui est p_1-LC inconsistante. Soit c_{ij} la contrainte sur laquelle (x_i, v_i) est p_1-LC inconsistante. Donc,

1. $\nexists v_j \in D(x_j)$ support AC de (x_i, v_i) sur c_{ij} tel que $\Delta(x_j, v_j) \geq p_1$. Alors, v_i n'est pas p_2-stable pour AC sur c_{ij}.
2. v_i n'est pas LC sur c_{ij}.

(1) et (2) impliquent que v_i n'est pas p_2-LC, et N n'est pas p_2-LC. ∎

Pour les cohérences pour lesquelles la cohérence d'une valeur sur une contrainte n'est pas définie, nous donnons une deuxième définition de la cohérence paramétrée.

Définition 5 (Value-based p-LC) *Soit LC une cohérence locale plus forte que l'AC. Une valeur $v_i \in D(x_i)$ est value-based p-LC si et seulement si elle est p-stable pour AC ou elle est LC. Un réseau de contraintes est value-based p-LC si et seulement si toutes les valeurs dans tous les domaines dans D sont value-based p-LC.*

Théorème 6 *Soit LC une cohérence locale plus forte que AC. Soit p_1 et p_2 deux paramètres dans [0..1]. Si $p_1 < p_2$ alors AC $\preceq$ value-based p_1-LC $\preceq$ value-based p_2-LC $\preceq$ LC.*

Preuve. Supposons qu'il existe deux paramètres p_1, p_2 tel que $0 \leq p_1 < p_2 \leq 1$, et supposons qu'il existe un réseau de contraintes N qui est p_2-LC et qui contient une valeur (x_i, v_i) qui est p_1-LC inconsistante. v_i est p_1-LC incohérence signifie que :

1. v_i n'est pas p_1-stable pour AC : $\exists c_{ij}$ où v_i n'est pas p_1-stable pour AC. Donc $\nexists v_j \in D(x_j)$ support AC de (x_i, v_i) sur c_{ij} tel que $\Delta(x_j, v_j) \geq p_1$. Alors, v_i n'est pas p_2-stable pour AC sur c_{ij}, donc v_i n'est pas p_2-stable pour AC.
2. v_i est LC inconsistante.

(1) et (2) impliquent que v_i n'est pas p_2-LC, et donc N n'est pas p_2-LC. ∎

Pour les deux définitions de p-LC, nous avons la propriété suivante pour les cas extrêmes (p=0, p=1).

Corollaire 7 *Soit LC_1 et LC_2 deux propriétés de cohérence locale plus fortes que l'AC, telles que la cohérence LC_1 d'une valeur sur une contrainte est définie. Nous avons : constraint-based 0-LC_1 = AC et constraint-based 1-LC_1 = LC_1. value-based 0-LC_2 = AC et value-based 1-LC_2 = LC_2.*

4 maxRPC Paramétrée : p-maxRPC

Dans cette section nous appliquons le concept de cohérence paramétrée à maxRPC pour obtenir p-maxRPC, une cohérence qui permet d'atteindre des niveaux de cohérence intermédiaires entre AC et maxRPC.

Définition 8 (p-maxRPC) *Une valeur, un réseau sont p-maxRPC si et seulement si ils sont constraint-based p-maxRPC.*

A partir du théorème 4 et du corollaire 7 nous obtenons le corollaire suivant.

Corollaire 9 *Pour tous paramètres p_1 et p_2 tels que $0 \leq p_1 < p_2 \leq 1$, $AC \preceq p_1\text{-}maxRPC \preceq p_2\text{-}maxRPC \preceq maxRPC$. $0\text{-}maxRPC = AC$ et $1\text{-}maxRPC = maxRPC$.*

Dans ce qui suit, nous décrivons uniquement les changements que nous avons apportés à l'algorithme maxRPC3 pour concevoir p-maxRPC3, un algorithme qui réalise p-maxRPC.

maxRPC3 utilise une liste de propagation Q qui contient les variables dont le domaine a changé. Il utilise aussi deux autres structures de données : LastAC et LastPC. Pour chaque valeur (x_i, v_i) $\text{LastAC}_{x_i,v_i,x_j}$ est utilisé pour stocker le plus petit support AC de (x_i, v_i) sur c_{ij} et $\text{LastPC}_{x_i,v_i,x_j}$ est utilisé pour stocker le plus petit support PC de (x_i, v_i) sur c_{ij} (c-à-d. le plus petit support AC (x_j, v_j) de (x_i, v_i) sur c_{ij} tel que (v_i, v_j) est PC). Cet algorithme consiste en deux phases : une phase d'initialisation et une phase de propagation.

Dans la phase d'initialisation maxRPC3 vérifie si chaque valeur (x_i, v_i) possède un support maxRPC (x_j, v_j) sur chaque contrainte c_{ij}. Sinon, il supprime v_i de $D(x_i)$ et insère x_i dans Q. Pour vérifier si une valeur (x_i, v_i) possède un support maxRPC sur une contrainte c_{ij}, maxRPC3 recherche d'abord un support AC (x_j, v_j) de (x_i, v_i) sur c_{ij}, puis vérifie si (v_i, v_j) est PC. Dans cette dernière étape nous avons modifié maxRPC3 et nous ne vérifions la cohérence de chemin de (v_i, v_j) que si la distance $\Delta(x_j, v_j)$ est inférieure au seuil p.

La phase de propagation consiste à propager l'effet des suppressions. Tant que Q contient des variables, maxRPC3 extrait une variable x_i de Q et vérifie pour chaque valeur (x_j, v_j) de chaque variable voisine $x_j \in \Gamma(x_i)$ si elle n'est pas devenue maxRPC-inconsistante à cause des suppressions de valeurs dans $D(x_i)$.

Dans cette phase aussi, nous avons modifié maxRPC3 pour vérifier si les valeurs sont encore p-maxRPC au lieu de vérifier si elles sont maxRPC. Dans p-maxRPC3, le plus petit support p-maxRPC de (x_j, v_j) sur c_{ij} correspond au plus petit support AC si (x_j, v_j) est p-stable pour AC sur c_{ij}. Sinon, il correspond au plus petit support PC. Ainsi, p-maxRPC3 vérifie si le dernier support p-maxRPC (le dernier support connu) de (x_j, v_j) sur c_{ij} appartient toujours au domaine de x_i. Sinon, il cherche le prochain support AC (x_i, v_i) sur c_{ij}, et vérifie si (v_i, v_j) est PC dans le cas où $\Delta(x_i, v_i) < p$. Si aucun support p-maxRPC n'existe, p-maxRPC3 supprime la valeur et insère la variable x_j dans la liste Q. Si la valeur (x_j, v_j) n'a pas était supprimée dans la phase précédente, p-maxRPC3 vérifie la subsistance du témoin (x_i, v_i) pour chaque paire (v_j, v_k) tel que (x_k, v_k) est le support p-maxRPC de (v_j) sur c_{jk} et $\Delta(x_k, v_k) < p$. Sinon, il recherche le prochain support p-maxRPC de (v_j) sur c_{jk}, et supprime (x_j, v_j) de $D(x_j)$ si ce support n'existe pas puis insère la variable x_j dans Q.

5 SAC Paramétrée : p-SAC

Cette section présente la singleton arc cohérence paramétrée (p-SAC), une instanciation du concept générique de cohérence paramétrée à la la singleton arc cohérence, pour qui la cohérence d'une valeur sur une contrainte n'est pas définie.

Définition 10 (p-SAC) *Une valeur, un réseau, sont p-SAC si et seulement s'ils sont value-based p-SAC.*

A partir du théorème 6 et du corollaire 7 nous obtenons le corollaire suivant.

Corollaire 11 *Pour tous paramètres p_1 et p_2 tels que $0 \leq p_1 < p_2 \leq 1$, $AC \preceq p_1\text{-}SAC \preceq p_2\text{-}SAC \preceq SAC$. $0\text{-}SAC = AC$ et $1\text{-}SAC = SAC$.*

Nous proposons maintenant un algorithme pour réaliser la singleton arc cohérence paramétrée. Sa fonction principale est décrite dans Algorithme 1. Elle commence par un appel à la fonction `InstrumentedAC`, qui applique simplement la cohérence d'arc et insère dans la liste P toutes les valeurs (x_i, v_i) qui ne sont pas p-stable pour AC (ligne 3). Ensuite, elle entre dans une boucle. Dans la première ligne de la boucle (ligne 7), elle fait appel à la procédure `Customized-SAC` qui vérifie SAC pour chaque valeur (x_i, v_i) dans P, et supprime v_i de $D(x_i)$ si elle est SAC-inconsistante.

Algorithm 1: p-SAC1(X, D, C)

```
1  begin
2      P ← ∅
3      if ¬InstrumentedAC(X, D, C, P) then
4          return false
5      while P ≠ ∅ do
6          Change ← false
7          if ¬CustomizedSAC(X, D, C, P, Change) then
8              return false;
9          if Change then
10             foreach x_i ∈ X, v_i ∈ D(x_i) do
11                 if v_i is not p-stable for AC then
12                     P ← P ∪ (x_i, v_i)
13     return true
```

Algorithm 2: CustomizedSAC($X, D, C, P, Change$)

```
1  begin
2      while P ≠ ∅ do
3          pick and delete (x_i, v_i) from P
4          if v_i ∈ D(x_i) ∧ ¬AC(X, D, C ∪ {x_i = v_i}) then
5              remove v_i from D(x_i)
6              Change ← true
7              if D(x_i) = ∅ or ¬propagateAC(X, D, C, {x_i}) then
8                  return false
9      return true
```

Chaque fois qu'une valeur SAC-inconsistante est supprimée, le flag $Change$ est positionné à vrai et la cohérence d'arc est rétablie à nouveau dans le réseau de contraintes. Quand `CustomizedSAC` n'échoue pas et retourne $Change = vrai$, toutes les valeurs qui ne sont plus p-stables pour AC sont mises dans P pour vérifier si elles sont SAC dans l'itération suivante. L'algorithme termine quand un domaine vide est détecté dans `InstrumentedAC` ou `CustomizedSAC`, ou quand `CustomizedSAC` retourne $Change = faux$, ou quand `InstrumentedAC` retourne $vrai$ et P vide.

La procédure `CustomizedSAC`, présentée dans Algorithme 2, procède comme suit : tant que P n'est pas vide, elle prend une valeur (x_i, v_i) de P (ligne 3). Si la valeur v_i est encore dans $D(x_i)$, `CustomizedSAC` vérifie si v_i est SAC (ligne 4) et la supprime de $D(x_i)$ si elle est SAC-inconsistante (ligne 5). Quand v_i est supprimée, le flag Change est positionné à vrai (ligne 6) puis la procédure `propagateAC` est appelée pour rétablir la cohérence d'arc (ligne 7). Si un domaine vide est détecté, la procédure `CustomizedSAC` retourne faux (ligne 8). Elle retourne vrai quand P devient vide (ligne 9).

6 Évaluation Expérimentale

Nous avons implémenté les algorithmes réalisant p-maxRPC et p-SAC comme décrits dans la section précédente, en plus de maxRPC3, SAC1 et AC2001 [3]. Tous ces algorithmes sont implémentés dans notre propre solveur de contraintes binaires et sont maintenus pendant la recherche. Nous avons testé ces algorithmes dans plusieurs classes de problèmes de la dernière compétition internationale de solveurs de contraintes 09 [1]. Nous avons sélectionné les problèmes contenant uniquement des contraintes binaires. Pour résoudre un problème utilisant un niveau de cohérence p-LC, cette cohérence est d'abord établie dans une étape de prétraitement, puis maintenue pendent la recherche. Pour isoler l'effet de la propagation, nous utilisons l'ordre lexicographique pour les variables et les valeurs. Nous avons fixé une heure comme limite de temps CPU. Nous avons réalisé nos expérimentations sur une plateforme 12-core sur une machine GenuineIntel avec 16 Go de RAM fonctionnant à 2.92 GHz.

Nous avons expérimenté le concept de cohérence paramétrée p-LC dans les cas où LC correspond à maxRPC ou SAC, et nous l'avons utilisé pour résoudre des instances de problèmes en maintenant les différents niveaux intermédiaires de cohérence p-LC, allant de la cohérence d'arc (0-LC) à la cohérence forte (1-maxRPC ou 1-SAC), en variant p de 0 à 1 avec un pas de 0.1. Pour chaque instance résolue, nous avons enregistré le paramètre associé au meilleur temps CPU obtenu. Les performances ont été mesurées en termes de temps CPU en secondes, de nombre de nœuds visités (NODE) et de nombre de contraintes testées (CCK). Pour les deux cohérences expérimentées (maxRPC et SAC), les résultats sont données sous la forme "temps CPU(p)", où (p) est le paramètre pour lequel p-LC a donnée le meilleur résultat.

Les tables 1, 2 et 3 comparent les performances de maintien de p-maxRPC et p-SAC aux performances de AC, maxRPC et SAC sur les instances du problème d'allocation de fréquences de liaison radio (RLFAP), les instances du problème Geom, et les instances du problème des Queens-Knights. La comparaison de p-maxRPC à maxRPC et AC dans les problèmes RLFAP et Geom montre, pour la plupart des instances de ces problèmes, l'existence d'un paramètre p où la résolution avec p-maxRPC est plus rapide que AC et maxRPC. Nous constatons la même chose en comparant les performances de p-SAC à SAC et AC sur les instances des problèmes RLFAP et Geom. En général, pour la plupart des instances des problèmes expérimentés, à l'exception du problème Queens-Knights, nous constatons l'existence d'un paramètre p où p-LC

1. http ://cpai.ucc.ie/09/

TABLE 1 – Performances (temps cpu, nœuds et ccks) de p-maxRPC et p-SAC sur les instances du problème RLFAP.

		AC	p-maxRPC	maxRPC	p-SAC	SAC
scen1-f8	cpu(s)	time-out	**1.39(0.2)**	6.1	time-out	time-out
	#nodes	-	927	917	-	-
	#ccks	-	1397440	26932990	-	-
scen2-f24	cpu(s)	time-out	**0.13(0.3)**	0.65	52.77(0.1)	531.29
	#nodes	-	201	201	202	200
	#ccks	-	296974	3462070	8751216	170233715
scen3-f10	cpu(s)	time-out	**0.89(0.5)**	2.8	2162.94(0.5)	time-out
	#nodes	-	469	408	407	-
	#ccks	-	874930	13311797	161605804	-
scen6-w1	cpu(s)	**0.02**	**0.02(0.2)**	0.08	0.09(0.0)	259.74
	#nodes	211	211	211	211	200
	#ccks	295128	296541	904900	295128	233056049
scen6-w2	cpu(s)	0.11	**0.01(1.0)**	**0.01**	0.1(0.7)	0.18
	#nodes	40	0	0	0	0
	#ccks	411071	85769	85769	417870	431405
scen7-w1-f4	cpu(s)	0.08	**0.06(0.2)**	0.14	0.31(0.0)	983.9
	#nodes	424	419	406	424	400
	#ccks	509989	559375	1319246	509989	209684190
scen7-w1-f5	cpu(s)	time-out	**0.04(0.2)**	0.08	1.3(0.2)	9.36
	#nodes	-	0	0	0	0
	#ccks	-	478795	1087223	806395	2020584
scen10-w1-f3	cpu(s)	time-out	**0.05(0.5)**	0.1	3.37(0.2)	13.27
	#nodes	-	0	0	0	0
	#ccks	-	1040434	1714477	1377050	2764334

TABLE 2 – Performances (temps cpu, nœuds et ccks) de p-maxRPC et p-SAC sur les instances du problème Geom

		AC	p-maxRPC	maxRPC	p-SAC	SAC
geo50-20-d4-75-26	cpu(s)	111.48	17.8(1.0)	15.07	**3.19(0.7)**	13.33
	#nodes	477696	3768	3768	51	51
	#ccks	96192822	40784017	40784017	3336287	9304263
geo50-20-d4-75-30	cpu(s)	1.4	**0.14(0.9)**	0.24	0.82(0.1)	10.95
	#nodes	5098	55	55	59	50
	#ccks	1163409	351258	894200	196147	9709063
geo50-20-d4-75-38	cpu(s)	1800.13	**779.01(0.6)**	1017.08	1464.56(0.3)	1901.09
	#nodes	3870476	314454	166210	4118134	4582
	#ccks	1528095910	1445180475	2494969079	1080827574	1350511787
geo50-20-d4-75-43	cpu(s)	1671.35	**1264.36(0.5)**	1530.02	2546.05(0.0)	time-out
	#nodes	4118134	555259	279130	4118134	-
	#ccks	1160664461	1801402535	3898964831	1160664461	-
geo50-20-d4-75-46	cpu(s)	1732.22	**371.3(0.6)**	517.35	1576.13(0.1)	time-out
	#nodes	3682394	125151	64138	253045	-
	#ccks	1516856615	584743023	1287674430	851048532	-
geo50-20-d4-75-78	cpu(s)	**0.13**	0.22(0.6)	0.31	0.17(0.0)	9.77
	#nodes	389	122	84	389	55
	#ccks	170210	351104	1077835	170210	7951262
geo50-20-d4-75-84	cpu(s)	404.63	0.44(0.6)	0.56	**0.41(0.1)**	10.18
	#nodes	2581794	513	333	206	50
	#ccks	293092144	800657	1606047	444784	7887350

est plus rapide que AC et LC. En fait, chaque fois que p-LC améliore AC et LC, il parvient à trouver un compromis entre le nombre de nœuds visités (la puissance de la cohérence p-LC) et le nombre de CCK produit en maintenant p-LC.

Les Figures 1 et 2 illustrent les performances (CPU, NODE et CCK) de p-maxRPC pour des valeurs de p allant de 0 à 1 avec un pas de 0.1. La Figure 1 montre un exemple où le maintien de p-maxRPC résout le problème plus rapidement que AC et maxRPC pour les valeurs de p allant de 0.3 à 0.8. Nous observons que p-maxRPC est plus rapide que AC et maxRPC lorsqu'il réduit la taille de l'espace de recherche aussi bien que maxRPC (même nombre de nœuds visités) avec un nombre de CCK proche du nombre de CCK produit par AC. La Figure 2 montre une instance où p-maxRPC échoue à améliorer le temps CPU de AC et maxRPC en même temps. Elle nous montre que, sauf pour p=1, p-maxRPC est deux à trois fois plus rapide que maxRPC. p-maxRPC améliore maxRPC car il arrive à réduire considérablement le nombre de CCK. Cependant, il n'a pas pu améliorer AC car le nombre de CCK produit par p-maxRPC reste élevé comparé au nombre de CCK produit par AC, alors que la réduction du nombre de nœuds visités n'est pas significative comparée au nombre de nœuds visités par AC.

Pour les deux figures 1 et 2 nous observons que le temps CPU pour 1-maxRPC (respectivement 0-maxRPC) est supérieur au temps CPU pour maxRPC (respectivement AC), bien que les deux cohérences sont équivalentes. En général p-LC est légèrement moins performante que LC (respectivement AC) pour p=1 (respectivement p=0) car elle effectue des tests et des calculs supplémentaires liés à la distance des valeurs. Pour p=0, cette différence est également explicable par le fait que p-LC maintient des structures de données que AC n'utilise pas.

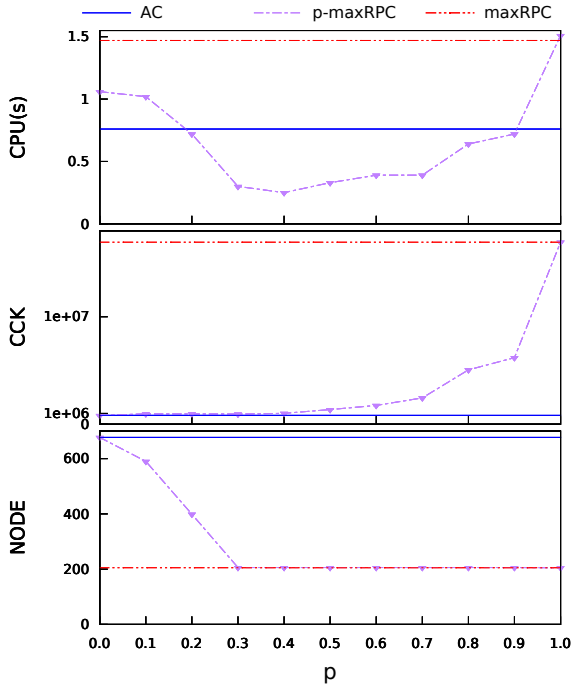


FIGURE 1 – Instance où p-maxRPC améliore AC et maxRPC

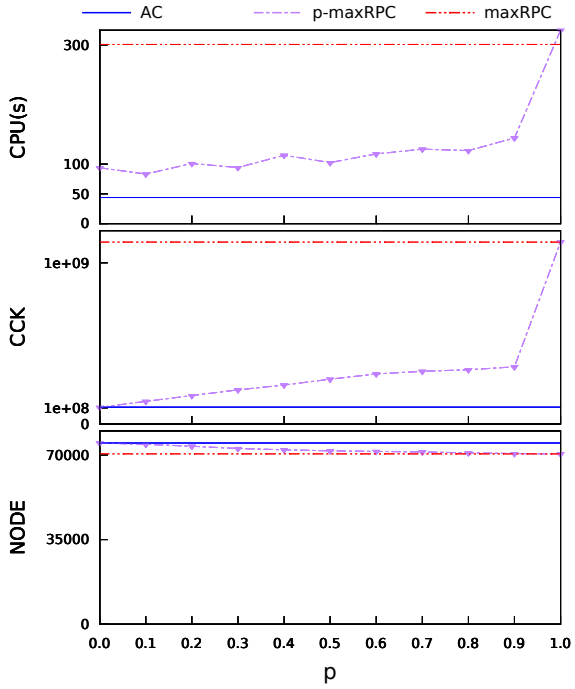


FIGURE 2 – Instance où p-maxRPC échoue à améliorer AC

7 Conclusion

Nous avons introduit la notion de *stabilité* des valeurs pour la cohérence d'arc, une notion basée sur la profondeur des supports AC de ces valeurs dans leur domaine. Nous avons utilisé cette notion pour proposer la *cohérence paramétrée*, une technique qui permet de définir des niveaux de cohérence locale intermédiaires entre la cohérence d'arc et une cohérence locale plus forte que l'AC. La force (puissance) des niveaux intermédiaires résultants est croissante et est fonction du paramètre p. Nous avons instancié l'approche générique de cohérence paramétrée en utilisant la max-cohérence de chemin restreinte et la singleton arc cohérence. Nous avons montré expérimentalement sur de

TABLE 3 – Performances (temps cpu, nœuds et ccks) de p-maxRPC et p-SAC sur les instances du problème Queens-Knights

		AC	p-maxRPC	maxRPC	p-SAC	SAC
queensKnights-10-5-add	cpu(s)	27.14	30.79(0.2)	98.44	0.09(1.0)	**0.06**
	#nodes	82208	81033	78498	0	0
	#ccks	131098933	148919686	954982880	235110	235110
queensKnights-10-5-mul	cpu(s)	43.89	83.27(0.1)	300.74	0.34(1.0)	**0.2**
	#nodes	74968	74414	70474	0	0
	#ccks	104376698	140309576	1128564278	432537	432537
queensKnights-15-5-add	cpu(s)	time-out	time-out	time-out	0.56(1.0)	**0.32**
	#nodes	-	-	-	0	0
	#ccks	-	-	-	1172468	1172468
queensKnights-8-5-add	cpu(s)	1.08	1.48(0.2)	4.48	**0.03(1.0)**	0.04
	#nodes	6656	6629	6502	0	0
	#ccks	6205191	7104271	44551267	94216	94216
queensKnights-8-5-mul	cpu(s)	1.77	3.08(0.1)	11.15	0.11(1.0)	**0.05**
	#nodes	5920	5920	5638	0	0
	#ccks	4599166	5194507	46125818	174627	174627

nombreux exemples de problèmes que le concept de cohérence paramétrée est viable. Notre prochain objectif est d'être en mesure d'adapter le paramètre au cours de la recherche pour n'importe quelle instance de problème. Grâce aux techniques d'apprentissage automatique, le solveur doit être capable de sélectionner la cohérence d'arc ou une cohérence locale plus forte, en fonction de l'instance du problème à résoudre, en fonction de la partie du problème dans laquelle s'effectue la propagation et en fonction de l'évolution de l'arbre de recherche développé par le solveur. Cette méthode générique devrait être valable pour toute propriété de cohérence locale et pour n'importe quel type de problème.

Références

[1] Thanasis Balafoutis, Anastasia Paparrizou, Kostas Stergiou, and Toby Walsh. New algorithms for max restricted path consistency. *Constraints*, 16(4) :372–406, 2011.

[2] Christian Bessiere. Constraint propagation. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 3. Elsevier, 2006.

[3] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artif. Intell.*, 165(2) :165–185, 2005.

[4] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *IJCAI (1)*, pages 412–417. Morgan Kaufmann, 1997.

[5] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1) :99–118, 1977.

[6] Kostas Stergiou. Heuristics for dynamically adapting propagation in constraint satisfaction problems. *AI Commun.*, 22 :125–141, August 2009.

La recherche locale pour la pré-programmation d'environnements urbains durables

Bruno Belin[1] **Marc Christie**[2] **Charlotte Truchet**[1]

[1] Université de Nantes, Laboratoire d'Informatique de Nantes Atlantique UMR 6241, Nantes, France
[2] Université de Rennes 1, IRISA/INRIA Rennes Bretagne Atlantique, Rennes, France
{bruno.belin,charlotte.truchet}@univ-nantes.fr marc.christie@irisa.fr

Abstract

L'un des grands enjeux des politiques urbaines actuelles réside dans la conception de villes nouvelles durables basée sur une approche systémique. Cette approche systémique consiste à intégrer simultanément les contraintes issues des différentes problématiques urbaines (transports, taux d'emploi, équipements, étalement urbain, mixité, etc.) Ces problématiques sont naturellement exprimées par les urbanistes sous une forme déclarative. En s'appuyant sur un modèle urbain utilisé lors de la phase préparatoire de conception (pré-programmation urbaine), nous proposons d'exprimer ces problématiques comme des contraintes et de les intégrer au sein d'un outil d'aide à la décision pour la pré-programmation urbaine. Dans ce contexte, l'objectif est de déterminer rapidement des solutions de qualité acceptable pour des problèmes de grande taille, sans nécessairement trouver la solution optimum. Compte tenu de cette spécificité, nous résolvons le problème avec une méthode dédiée de recherche locale et proposons une extension distribuée de l'algorithme.

1 Introduction

A la variété des villes forgées depuis des millénaires, le XX^{e}siècle a voulu imposer un modèle unique en créant un monde désarticulé composé de grands objets utilitaires que sont les barres ou les tours, vite entourés d'autoroutes [1]. A la ville des continuités a succédé la ville des disjonctions à l'origine des ghettos où la voiture devint reine, et en condamnant la densité de la ville, Le Corbusier [1] a notamment favorisé l'étalement urbain sans fin. Mais tout comme la ville, l'homme universel n'existe pas et chaque culture entretient avec son milieu un rapport symbolique plus important que ses fonctions matérielles de base (habiter, travailler, circuler, se cultiver le corps et l'esprit).

Alors comment recréer les conditions d'un équilibre entre les hommes, l'environnement et la ville dans un monde où plus de 50% de la population vit en agglomération et où les projections des Nations Unies [2] prévoient un taux d'urbanisation mondial de l'ordre de 70% pour 2050 ?

Dans ce contexte difficile à appréhender, nous participons à la création d'un outil d'aide à la décision pour la pré-programmation d'environnements urbains durables. Ce travail s'inscrit dans le cadre du projet FUI SUSTAINS [2], qui rassemble des urbanistes, des experts du secteur de l'énergie et des équipes autour de l'optimisation combinatoire et l'interaction homme-machine. Ce projet propose la réalisation d'un outil interactif d'aide à la décision permettant de répartir et de positionner les différentes entités composant une pré-programmation (habitats, industries, équipements, commerces) sur un territoire afin d'éclairer les choix des décideurs (urbanistes et élus). Cet article se focalise sur la problématique de placement des éléments urbains à l'aide de techniques d'optimisation combinatoire.

Nous proposons une modélisation du problème de pré-programmation d'environnements urbains durables sous la forme d'un problème d'optimisation sous contraintes ainsi qu'une méthode dédiée de recherche locale pour le résoudre.

La programmation par contraintes (PPC) est un paradigme de programmation qui permet de modéliser et de résoudre des problèmes combinatoires de façon déclarative et générique : "L'utilisateur décrit son problème, l'ordinateur le résout" [3]. Un très grand nombre de problèmes combinatoires réels et théoriques peuvent être modélisés par des problèmes de satisfaction de contraintes

1. Architecte et urbaniste, élabore une doctrine nouvelle publiée sous le nom de la Charte d'Athènes en 1941 qui prescrit un zoning strict, c'est-à-dire un classement des grandes fonctions urbaines par catégorie.

2. http ://www.sustains.fr/

(Constraint Satisfaction Problem ou CSP). Un CSP est défini par un ensemble V de variables, un ensemble D de domaines de définition qui encadrent les valeurs que pourront prendre les variables appartenant à V, et un ensemble C de contraintes qui conditionnent les valeurs que pourront prendre les variables appartenant à V. Un problème d'optimisation sous contraintes est un problème pour lequel on cherche parmi l'ensemble de toutes les solutions réalisables (c-à-d celles qui satisfont les contraintes dures du problème) la meilleure solution selon une fonction qui définit un objectif donné. Pour chaque propriété (ou contrainte souple) urbaine identifiée, nous définissons une fonction de coût pour représenter le degré de violation de la propriété, et intégrons une contrainte dure de placement d'éléments urbains. Pour résoudre le problème, notre algorithme de recherche locale cherche à minimiser la somme pondérée des coûts associés aux contraintes souples. Différents choix de voisinage pour la recherche locale sont comparés (multi-candidats, multi-permutations) et une version distribuée de l'algorithme est également présentée à la fin de l'article.

2 Pré-programmation et modèle urbain

La pré-programmation urbaine consiste à organiser spatialement les différentes entités constituant une ville (îlots, quartiers, centralités, équipements induits, logements, commerces) en intégrant les dimensions environnementales, sociales, de transport et énergétiques, le tout sous contraintes de taux d'emploi [3], de densités de population et de topologie de l'environnement. La pré-programmation urbaine permet aux décideurs et aménageurs de s'accorder sur les grandes orientations d'un projet avant même d'engager une étude urbaine complète et détaillée. Jusqu'à présent, la solution initiale utilisée comme base de travail par les décideurs reposait sur un processus manuel et approximatif incapable de prendre en compte l'ensemble des contraintes relatives à un développement urbain durable. Le projet SUSTAINS vise donc à proposer (i) une solution initiale optimisée sur la base d'un ensemble de contraintes définies par les urbanistes (décrites dans la suite de ce document) et (ii) une manipulation interactive intelligente de ces solutions (*i.e.* en tenant compte des contraintes). L'enjeu consiste à réduire en partie la phase de pré-programmation qui pouvait prendre jusqu'à six mois de travail (nombreux allers-retours entre cabinets d'urbanistes, décideurs ou réunions publiques, etc).

3. Le taux d'emploi est calculé en rapportant le nombre d'emplois au nombre total d'individus habitant la ville et en âge de travailler. La situation idéale se rapproche d'un taux d'emploi égal à "1", minimisant ainsi les déplacements des usagers qui disposent, dans ce cas, d'un travail et d'un logement au sein même de la ville.

2.1 Modèle urbain

En urbanisme, la modélisation consiste à simplifier la réalité du monde, de façon à mieux comprendre comment les décisions et les événements interagissent les uns avec les autres. Elle permet alors de tester des solutions susceptibles d'influencer ou d'orienter les décisions politiques et les stratégies pouvant conduire à un futur souhaitable [4].

Le modèle urbain, tel qu'il est proposé par les urbanistes d'Epamarne [4], repose sur les notions centrales que sont les *îlots*, les *intensités* et les *formes urbaines* et intègre, dans une démarche systémique, les contraintes urbaines majeures liées au développement urbain durable.

2.1.1 Îlot

En pré-programmation, le niveau de granularité le plus fin retenu par les urbanistes est l'îlot urbain. Les îlots sont représentés dans un quadrillage orthogonal avec une dimension de 80 mètres de coté. Un piéton peut facilement faire le tour d'un îlot, ce dernier étant déterminé par la voirie. A un îlot est associé une forme urbaine (habitat, commerce, industrie, école,...). Les îlots s'intègrent dans un quartier, espace urbain organisé autour d'un lieu structurant et qu'un piéton peut facilement parcourir (300 mètres de rayon).

2.1.2 Forme urbaine

Dans notre contexte, la forme urbaine correspond à un type d'utilisation dominant qui est fait de l'espace et du bâti pour un îlot complet. Une forme urbaine dite "dominante" peut intégrer dans sa déclaration une certaine mixité fonctionnelle (proportion de résidence, de commerce, ...). Les formes urbaines (au nombre de 19 dans l'étude actuelle) se décomposent en trois grandes familles :

- habitat : maison individuelle, maison de ville, habitat intermédiaire, habitat collectif
- activité économique : activité industrielle, artisanale, commerciale, tertiaire
- équipement public : école maternelle, primaire, collège, lycée, équipement sportif, etc.

2.1.3 Intensité

L'intensité urbaine est l'état d'un espace urbain assurant :

- un niveau de densité urbaine pour une ville compacte limitant l'étalement urbain
- une mixité fonctionnelle permettant d'agir sur les besoins de déplacement consommateurs d'énergie
- un maillage viaire hiérarchisé pour accueillir tout type de mobilité

4. http ://etablissements.epa-marnelavallee.fr/epamarne_fre

Le niveau d'animation d'un quartier est le reflet de l'intensité urbaine. Une intensité urbaine forte correspond à un quartier vivant, dense, mixte et pour lequel la marche à pied est le moyen de déplacement le plus simple pour accéder à toutes les fonctions urbaines essentielles. Ce niveau d'intensité urbaine se trouve au centre ville, alors qu'une intensité urbaine faible est plutôt liée à un quartier à dominante pavillonnaire. Six niveaux d'intensité différents sont définis : de "1" pour l'intensité la plus faible jusqu'à "6" pour celle la plus élevée. Ce sont les urbanistes et les décideurs qui fixent les différents niveaux d'intensité sur le territoire à aménager.

2.2 Problématique

Le modèle urbain détermine dans un premier temps le type et le nombre de formes urbaines à répartir sur la ville par niveau d'intensité en fonction du taux d'emploi spécifié et des emprises (surface au sol) des niveaux d'intensité. Cette répartition des formes urbaines par niveau d'intensité participe à la mixité fonctionnelle. La problématique consiste alors, dans un deuxième temps, à répartir spatialement les formes urbaines (calculées par le modèle urbain) sur l'ensemble des îlots du territoire en respectant les ventilations par niveau d'intensité et les contraintes de placement décrites en détail dans la section 3. Les différentes contraintes que nous exploitons ont été formalisées à partir des descriptions fournies par des urbanistes.

3 Modélisation

Dans cette section, nous décrivons le problème complet en reprenant les contraintes exprimées informellement par les urbanistes et leur traduction en fonctions de coût (sauf pour certaines contraintes dures qui seront directement exprimées à l'aide de la notion de voisinage).

3.1 Modèle de base

Nous représentons la ville par un carroyage régulier où chaque cellule de la grille correspond à un îlot urbain. L'enjeu consiste, dès lors, à répartir "au mieux" sur l'ensemble de la grille les différentes formes urbaines fournies par le modèle urbain. Nous disposons donc :

- d'un nombre de formes urbaines à répartir sur la ville, ce nombre étant ventilé par niveau d'intensité ;
- de cellules, en nombre équivalent au nombre total de formes urbaines, chaque cellule étant associée à un niveau d'intensité urbaine donné

Pour la répartition des formes urbaines sur la grille, il faut respecter les correspondances d'intensité entre les formes urbaines d'une part et les cellules d'autre part. La grille est vue comme une matrice rectangulaire de cellules de taille $x * y$. On note $V_{l,c}$ la variable forme urbaine de la cellule correspondant à la ligne l, colonne c :

$V_{1,1}$	...	$V_{1,y}$
...		...
$V_{x,1}$	...	$V_{x,y}$

Chaque variable modélise un type de forme urbaine, repéré par un numéro correspondant à son code : (1) Maison individuelle, (2) Habitat collectif R+2 (deux étages), ..., (19) Espace de respiration. Dans notre modèle, on s'intéresse à un problème d'optimisation sous contraintes où certaines contraintes sont traduites en fonctions de coût alors que d'autres restreignent les mouvements possibles lors de la résolution. Lorsqu'une forme urbaine est assignée à une cellule, un coût pour cette cellule peut être calculé à partir de chaque contrainte applicable. La somme de ces coûts permet d'obtenir un coût total pour la cellule en question. L'objectif consiste à minimiser la somme des coûts sur l'ensemble des cellules. On cherche alors à déterminer la forme urbaine $V_{l,c}$ en minimisant le coût global $Cost$ issu du coût de chaque contrainte auquel on applique un poids pondéré w_{ct} lié à chaque contrainte ct :

$$Cost = \sum_{l=1}^{x} \sum_{c=1}^{y} \sum_{ct=1}^{z} \left(w_{ct} * Cost_{ct} \left(V_{l,c} \right) \right)$$

où : z correspond au nombre de contraintes.

Dès lors où le coût global $Cost$ de la configuration courante est connu, il est possible d'impacter sur ce coût global tout changement opéré sur une variable $V_{l,c}$ de façon incrémentale en considérant sa valeur avant et après changement.

3.2 Contraintes

3.2.1 Correspondance d'intensité

La modèle urbain nous transmet :

- des formes urbaines non localisées ventilées par niveau d'intensité,
- un niveau d'intensité associé à chaque cellule de la grille.

A partir de ces éléments, il faut que toute affectation d'une forme urbaine à une cellule respecte la correspondance d'intensité entre la cellule et la forme urbaine. On note $C_{l,c}$ la constante intensité associée à la cellule repérée par la ligne l et la colonne c :

$C_{1,1}$	...	$C_{1,y}$
...		...
$C_{x,1}$	...	$C_{x,y}$

Chaque constante $C_{l,c}$ contient une intensité urbaine, identifiée par un code :

- 1 : Intensité urbaine la plus faible (zone résidentielle)
- 2 : ...
- 6 : Intensité urbaine la plus forte (centre ville)

Les constantes (correspondant aux différents niveaux d'intensité) sont fixées par l'utilisateur. Cette contrainte "dure" de correspondance d'intensité s'exprime en filtrant les domaines pour éviter une permutation entre deux zones d'intensité différente. Cette contrainte n'est donc pas traduite sous forme de coût.

3.2.2 Interaction entre formes urbaines

Les contraintes d'interactions correspondent aux préférences de localisation des formes urbaines entre elles. Entre deux formes urbaines, les valeurs d'interaction possibles sont 0 (double attraction), 10 (simple attraction), 20 (neutre), 50 (simple répulsion) et 100 (double répulsion). On note la matrice d'interaction I, avec $I_{p,q}$ la valeur d'interaction entre la forme urbaine p et la forme urbaine q. Cette matrice est asymétrique : il peut donc arriver que $I_{p,q} \neq I_{q,p}$. Pour une forme urbaine positionnée en $V_{l,c}$, on évalue le coût d'interaction uniquement avec ses cellules voisines immédiates, c'est à dire avec :

$$\mathscr{V}_{l,c} = \left\{V_{i,j}, i \in [l-1, l+1], j \in [c-1, c+1], (i \neq l \wedge j \neq c)\right\}$$

Le résultat est divisé par le nombre de cellules voisines $|\mathscr{V}_{l,c}|$ pour prendre en compte de façon uniforme les cellules situées ou non en bordure de grille.

$$Cost_1\left(V_{l,c}\right) = \frac{\sum_{v \in \mathscr{V}_{l,c}} I_{V_{l,c},v}}{|\mathscr{V}_{l,c}|}$$

NB : Ne pas confondre $\mathscr{V}_{l,c}$ l'ensemble des cellules voisines avec le "voisinage" l'ensemble des configurations possibles par rapport à la configuration courante.

3.2.3 Zone de taille critique

Pour la création des zones industrielles ou artisanales, il est nécessaire d'atteindre une taille critique pour que cette opération puisse être réalisable (fonctionnelle, rentable, ...), la taille critique étant déterminée par les urbanistes pour chaque forme urbaine à regrouper (activité artisanale, activité industrielle). Il n'y a pas de taille maximale pour un regroupement. D'autre part, bien qu'elle ne soit pas exprimée explicitement par les urbanistes, la zone doit bénéficier d'une structure suffisamment compacte. Soit :

- f une forme urbaine appartenant à l'ensemble $\mathscr{F}_1$ des formes urbaines à regrouper,
- tc^f la taille critique pour f,
- $\mathscr{V}_{l,c}^f$ l'ensemble des éléments $V_{l',c'}$ de $\mathscr{V}_{l,c}$ tel que $V_{l',c'} = f$,
- s un coefficient constant $(0 < s \leq 1)$ permettant de calculer la proportion, pour une cellule donnée, de ses cellules voisines devant appartenir au même groupe. Ce coefficient permet de contrôler la structure des groupes et intervient au niveau de chaque cellule. Dans notre contexte, $s = 0,5$.

On note :

- $G_{l,c}$ le groupe associé à $V_{l,c}$ tel que :
  $G_{l,c} = Si\ \left(V_{l,c} \in \mathscr{F}_1\right) \left\{\forall V_{l',c'} \in \mathscr{V}_{l,c}^{V_{l,c}}, V_{l',c'} \cup G_{l',c'}\right\}\ Sinon\ \varnothing$
- $\mathscr{C}_{l,c}$ le coût associé au groupe $G_{l,c}$ tel que :
  $\mathscr{C}_{l,c} = max\left(tc^{V_{l,c}} - \left|G_{l,c}\right|, 0\right)$

Le coût associé à la variable $V_{l,c}$ peut alors s'écrire ainsi :

$$Cost_2\left(V_{l,c}\right) = \begin{cases} max\left(\left(s * \left|\mathscr{V}_{l,c}\right|\right) - \left|\mathscr{V}_{l,c}^{V_{l,c}}\right|, 0\right) + \mathscr{C}_{l,c} & V_{l,c} \in \mathscr{F}_1 \\ 0 & V_{l,c} \notin \mathscr{F}_1 \end{cases}$$

3.2.4 Accessibilité

Cette contrainte porte sur une seule forme urbaine : les espaces de respiration. Elle spécifie qu'en tout point de la ville, on puisse accéder à pied à un espace de respiration en moins de 15 minutes, ce qui correspond à une distance approximative de 1,25 km pour une vitesse moyenne de 5 km/h (ou le parcours d'une quinzaine de cellules). Nous proposons deux versions complémentaires de cette contrainte. Une version "globale" qui pénalise les cellules non couvertes de façon proportionnelle à leur éloignement à tout espace vert et une version "locale" qui pénalise de façon uniforme les cellules non couvertes. La première version est plus pertinente que la seconde si le nombre d'espaces de respiration à répartir sur la ville est insuffisant pour couvrir la totalité de la grille, mais son temps de calcul est lié à la taille de la grille. A l'inverse, la seconde version peut être calculée de façon "locale" et elle reste pertinente si le modèle urbain fournit suffisamment d'espaces de respiration pour couvrir l'ensemble du territoire.

Contrainte d'accessibilité "globale" A ce niveau, on mesure à quel point la contrainte est violée en quantifiant la violation au lieu de simplement l'observer. La distance entre cellules est mesurée en distance de Manhattan, plus simple à calculer que la distance Euclidienne et plus pertinente pour un déplacement piétonnier en ville. On note $\mathscr{D}^{Man}_{V_{l,c},V_{l',c'}}$ la distance de Manhattan entre deux cellules liées aux positions (l,c) et (l',c'). Les coûts correspondants sont illustrés sur le tableau ci-dessous, en supposant qu'il n'y ait qu'un seul espace de respiration (repéré par *"ER")* dans la grille et que la distance de couverture d'un espace de respiration, au dessous de laquelle il n'y a pas de coût, soit fixée à une distance $d = 4$ cellules (au lieu de 15 cellules normalement, ceci pour simplifier la représentation) :

4	3	2	1	0	0	0
3	2	1	0	0	0	0
2	1	0	0	0	0	ER
3	2	1	0	0	0	0
4	3	2	1	0	0	0

La fonction de coût associée à cette contrainte correspond alors à la somme de toutes les violations pour toutes

les cellules. Pour la calculer, il est nécessaire de connaître les positions des espaces de respiration dans la configuration courante, que l'on notera $ER_1, \ldots, ER_m$ (en supposant qu'il y ait m espaces de respiration). $ER_1 = (l', c')$ signifie que le premier espace de respiration correspond à la variable $V_{l',c'}$. La fonction de coût associée à la contrainte d'accessibilité globale vaut alors :

$$Cost_3\left(V_{l,c}\right) = min_{1 \leq \alpha \leq m}\left(max\left(\mathscr{D}^{Man}_{V_{l,c},ER_\alpha} - d, 0\right)\right)$$

Contrainte d'accessibilité "locale" Dans cette version, on s'intéresse aux cellules qui ne sont couvertes par aucun espace de respiration. Pour un espace de respiration repéré par $V_{l,c}$, sa zone de couverture correspond au voisinage que l'on notera $\mathscr{V}^d_{l,c}$, c'est à dire l'ensemble des variables $V_{i,j}$ tel que $i \in [l-d, l+d],\ j \in [c-d, c+d],\ (i \neq l \wedge j \neq c)$ avec d correspondant à la distance de couverture retenue. En lien avec ce voisinage, la notation $Cover\left(V_{l',c'}, V_{l,c}\right)$ retournera la valeur 1 si $V_{l',c'} \in \mathscr{V}^d_{l,c}$ sinon 0. Contrairement à la version globale, on marque dans le tableau ci-dessous la zone de couverture associée à chaque espace de respiration en actualisant le nombre d'espaces de respiration qui recouvrent une même cellule. La notion est illustrée dans le tableau suivant, en supposant qu'il y ait deux espaces de respiration (repérés par *"ER_1" et "ER_2"*) dans la grille et que $d = 4$:

0	0	1	1	1	1	ER_1
1	1	2	2	2	1	1
1	1	2	2	2	1	1
1	1	2	2	2	1	1
1	1	2	2	2	1	1
ER_2	1	1	1	1	0	0

Les cellules du tableau qui contiennent le chiffre 1 sont couvertes par ER_1 ou ER_2, celles qui contiennent le chiffre 2 sont couvertes à la fois par ER_1 et ER_2. Seul le nombre de cellules non couvertes, c'est à dire celles identifiées par le chiffre 0, sont comptabilisées pour déterminer le coût d'une configuration. La fonction de coût associée à cette contrainte correspond alors au nombre total de cellules non couvertes. Pour la calculer, il est nécessaire, comme pour la version globale, de connaître les positions des espaces de respiration dans la configuration courante, que l'on notera de la même manière. La fonction de coût associée à la contrainte d'accessibilité locale vaut alors :

$$Cost_3\left(V_{l,c}\right) = Si\left(\sum_{1 \leq \alpha \leq m} Cover\left(V_{l,c}, ER_\alpha\right) = 0\right) 1\, Sinon\, 0$$

3.2.5 Séparation

Il s'agit d'une contrainte liée aux *espaces tampons* et *équipements bâtis* : pour certaines formes urbaines particulières positionnées à proximité les unes des autres, il est stipulé qu'elles doivent satisfaire les conditions suivantes :

– ne pas être contiguës (ne pas directement se toucher),
– être séparées soit par un *espace tampon*, soit par un *équipement bâti*.

Les formes urbaines suivantes sont considérées comme des *espaces tampons* : équipement sportif et espace de respiration alors que les formes urbaines école maternelle, école primaire, collège, lycée, équipement administratif et équipement technique sont considérées comme des *équipements bâtis.* Par exemple :

– entre une maison individuelle et un habitat collectif R+7, on doit trouver un *espace tampon*.
– entre une maison individuelle et un habitat collectif R+4, on doit trouver un *espace bâti*.

Soit la forme urbaine repérée par $V_{l,c}$, on s'intéresse uniquement aux variables voisines présentes dans l'ensemble $\mathscr{V}^d_{l,c}$ avec $d = 2$. Dans la grille ci-dessous, on représente les cellules voisines de $V_{4,4}$ correspondant à $\mathscr{V}^2_{4,4}$:

$V_{2,2}$	$V_{2,3}$	$V_{2,4}$	$V_{2,5}$	$V_{2,6}$
$V_{3,2}$	$V_{3,3}$	$V_{3,4}$	$V_{3,5}$	$V_{3,6}$
$V_{4,2}$	$V_{4,3}$	$V_{4,4}$	$V_{4,5}$	$V_{4,6}$
$V_{5,2}$	$V_{5,3}$	$V_{5,4}$	$V_{5,5}$	$V_{5,6}$
$V_{6,2}$	$V_{6,3}$	$V_{6,4}$	$V_{6,5}$	$V_{6,6}$

Si on note S la matrice de séparation, $S_{p,q}$ correspond à l'ensemble des formes urbaines autorisées pour séparer la forme urbaine p de la forme urbaine q. Cette matrice est symétrique, on aura donc $S_{p,q} = S_{q,p}$. On peut avoir $S_{p,q} = \varnothing$ lorsqu'il n'y a pas de contrainte particulière de séparation entre les deux formes urbaines p et q.

Pour chaque cellule $V_{i,j}$ située dans le voisinage $\mathscr{V}^1_{l,c}$, on doit vérifier $S_{V_{l,c},V_{i,j}} = \varnothing$. En effet, lorsqu'une séparation est imposée entre deux formes urbaines, celles-ci ne doivent pas être contiguës. D'autre part, pour chaque cellule $V_{i,j}$ située dans le voisinage $\left(\mathscr{V}^2_{l,c} - \mathscr{V}^1_{l,c}\right)$, si $S_{V_{l,c},V_{i,j}} \neq \varnothing$, il faut vérifier que les formes urbaines positionnées sur les cellules qui séparent $V_{l,c}$ et $V_{i,j}$ appartiennent bien à $S_{V_{l,c},V_{i,j}}$.

La notation $Sep\left(V_{l,c}, V_{i,j}\right)$ désigne les formes urbaines positionnées sur les cellules qui séparent $V_{l,c}$ et $V_{i,j}$. Cet ensemble peut contenir un ou deux éléments comme le montre les tableaux ci-dessous qui indiquent les variables de séparation $\left\{f_{sep1}, f_{sep2}\right\}$ entre $V_{4,4}$ et ses cellules voisines $V_{i,j}$ situées à une distance stricte $d = 2$:

$V_{l,c}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$
$V_{i,j}$	$V_{2,2}$	$V_{2,3}$	$V_{2,4}$	$V_{2,5}$	$V_{2,6}$	$V_{3,6}$	$V_{4,6}$	$V_{5,6}$
f_{sep_1}	$V_{3,3}$	$V_{3,3}$	$V_{3,4}$	$V_{3,5}$	$V_{3,5}$	$V_{3,5}$	$V_{4,5}$	$V_{5,5}$
f_{sep_2}		$V_{3,4}$		$V_{3,4}$		$V_{4,5}$		$V_{4,5}$

$V_{l,c}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$	$V_{4,4}$
$V_{i,j}$	$V_{6,6}$	$V_{6,5}$	$V_{6,4}$	$V_{6,3}$	$V_{6,2}$	$V_{5,2}$	$V_{4,2}$	$V_{3,2}$
f_{sep_1}	$V_{5,5}$	$V_{5,5}$	$V_{5,4}$	$V_{5,3}$	$V_{5,3}$	$V_{5,3}$	$V_{4,3}$	$V_{3,3}$
f_{sep_2}		$V_{5,4}$		$V_{5,4}$		$V_{4,3}$		$V_{4,3}$

Avec $\mathscr{F}_2$ et $\mathscr{F}_3$, deux ensembles constitués de formes urbaines, $\mathscr{F}_3$ pouvant être égal à $\varnothing$, la notation

$ErrSep(\mathscr{F}_2, \mathscr{F}_3)$ retourne le nombre de formes urbaines présentes dans l'ensemble $\mathscr{F}_2$ mais inexistantes dans $\mathscr{F}_3$ si $\mathscr{F}_3 \neq \varnothing$ sinon retourne 0. La fonction de coût peut alors s'exprimer ainsi :

$$Cost_4(V_{l,c}) = \sum_{\alpha \in \mathscr{V}^1_{l,c}} \left(Si\left(S_{V_{l,c},\alpha} \neq \varnothing\right) 1 \, Sinon \, 0\right) + \sum_{\alpha \in \left(\mathscr{V}^2_{l,c} - \mathscr{V}^1_{l,c}\right)} \left(ErrSep\left(Sep\left(V_{l,c},\alpha\right), S_{V_{l,c},\alpha}\right)\right)$$

3.2.6 Éloignement

Cette contrainte spécifie un éloignement minimum de plusieurs cellules entre deux formes urbaines particulières (éloignement de 4, de 3 ou de 2 cellules). Par exemple, entre une maison individuelle et un bâtiment tertiaire R+7, on veut un éloignement minimum de 4 cellules. Pour cette contrainte, nous utilisons des distances Euclidiennes qui permettent de mesurer un éloignement en ligne droite entre deux cellules. On note E la matrice d'éloignement, tel que $E_{p,q}$ identifie l'éloignement minimum entre les formes urbaines p et q. Cette matrice est symétrique, on aura donc $E_{p,q} = E_{q,p}$. On peut avoir $E_{p,q} = 0$ lorsqu'il n'y a pas de contrainte particulière de distance minimum entre les deux formes urbaines p et q. Soit $\mathscr{D}^{Euc}_{p,q}$ la distance Euclidienne exprimée en nombre de cellules qui sépare les cellules p et q, la fonction de coût peut alors s'exprimer ainsi :

$$Cost_5(V_{l,c}) = \sum_{\alpha \in \mathscr{V}^4_{l,c}} \left(max\left(E_{V_{l,c},\alpha} - \mathscr{D}^{Euc}_{V_{l,c},\alpha}, 0\right)\right)$$

3.2.7 Constitution de place

Le principe est le suivant : lorsqu'il y a, à proximité immédiate d'un lycée ou d'un collège, certaines formes urbaines (Maison individuelle ou de ville, habitat intermédiaire ou collectif, bâtiment tertiaire), on souhaite constituer une "place" autour du collège ou du lycée en positionnant, à coté de l'édifice, un nombre plus ou moins important d'espaces de respiration.

Par exemple, s'il n'y a autour d'un collège que des maisons individuelles, des maisons de ville ou de l'habitat intermédiaire, on voudra alors constituer une petite place en positionnant à coté du collège au moins un espace de respiration. Par contre, si on trouve, à proximité du collège, de l'habitat collectif ou des bâtiments tertiaires, on souhaitera constituer une place plus importante avec au moins deux espaces de respiration contigus au collège.

La présence de plusieurs collèges (ou lycées) dans le même voisinage sera également pénalisée afin de favoriser la répartition de ces formes urbaines sur la grille.

On note P la matrice qui contient le paramétrage de constitution des places (ou lieux), tel que $P_{p,q}$ identifie le nombre minimum d'espaces de respiration à positionner autour de la forme urbaine p compte tenu de la proximité de la forme urbaine q. Cette matrice est asymétrique, donc $P_{p,q}$ peut être différent de $P_{q,p}$. On peut également avoir $P_{p,q} = 0$ lorsqu'il n'y a pas de conditions particulières entre les deux formes urbaines p et q. Concernant la proximité liée à une forme urbaine $V_{l,c}$ de référence, seules les cellules voisines représentées par l'ensemble $\mathscr{V}^4_{l,c}$ sont considérées. Soit :

- f une forme urbaine appartenant à l'ensemble $\mathscr{F}_4$ des formes urbaines concernées par la création d'une place dans leur voisinage immédiat,
- $\mathscr{V}^{ER}_{l,c}$ l'ensemble des éléments $V_{l',c'}$ de $\mathscr{V}_{l,c}$ tel que $V_{l',c'} = ER$ ("ER" étant l'identifiant des Espaces de Respiration),
- $\mathscr{V}^{4,f}_{l,c}$ l'ensemble des éléments $V_{l',c'}$ de $\mathscr{V}^4_{l,c}$ tel que $V_{l',c'} = f$,

Si $V_{l,c} \in \mathscr{F}_4$, son coût est constitué de deux parties :

1. une pénalité pour chaque forme urbaine $f = V_{l,c}$ présente dans le voisinage $\mathscr{V}^{4,f}_{l,c}$, pénalité d'autant plus importante que les distances sont réduites,
2. un coût correspondant au nombre d'espaces de respiration manquants dans la constitution de la place autour de $V_{l,c}$

$$Cost_6(V_{l,c}) = Si\left(V_{l,c} \in \mathscr{F}_4\right) \sum_{\alpha \in \mathscr{V}^{4,V_{l,c}}_{l,c}} \left(5 - \mathscr{D}^{Euc}_{V_{l,c},\alpha}\right) + max\left(max_{\alpha \in \mathscr{V}^4_{l,c}}\left(P_{V_{l,c},\alpha}\right) - \left|\mathscr{V}^{ER}_{l,c}\right|, 0\right) Sinon \, 0$$

4 Résolution et résultats

Il est souvent extrêmement difficile de résoudre des modèles de positionnement de façon optimale alors même que les modèles les plus classiques sont NP-complet et deviennent difficilement surmontables pour des grandes instances de problèmes [5]. Les paradigmes les mieux adaptés aujourd'hui à la résolution d'un problème comme le nôtre, de nature combinatoire, proviennent de la Programmation Par Contraintes (PPC) et de la Recherche Locale (RL). La PPC effectue en général une recherche énumérative complète en inférant des réductions de l'espace de recherche à partir des contraintes du problème et permet donc de prouver l'optimalité d'une solution ; mais les temps de calcul peuvent devenir rédhibitoires quand la taille du problème est trop grande. Au contraire, la RL parcourt l'espace de recherche en se dirigeant vers les régions "prometteuses" grâce à une heuristique mais en exploitant les contraintes de façon passive : pénalité dans le coût pour les solutions qui violent les contraintes et impossibilité de garantir l'optimalité du résultat. La RL permet d'obtenir de bonnes solutions pour les problèmes de grandes tailles ou très difficiles et elle est souvent utilisée quand les autres méthodes échouent à cause de la trop grande taille ou du manque de structure d'un problème.

Suite aux échecs rencontrés sur de petites instances partielles du problème (Implémentation de la contrainte "Interaction entre formes urbaines" sur une grille 16x16) avec des méthodes complètes de recherche (Mise en œuvre du solver CHOCO [5]), nous proposons l'utilisation de méthodes incomplètes du type RL. Nos travaux s'inspirent plus particulièrement de la méthode *Adaptive Search* qui a prouvé son efficacité sur des instances de grandes tailles et de natures différentes [6]. Il s'agit d'une méthode de résolution de CSP générique et applicable à une large classe de contraintes qui tire parti de la structure du problème en terme de contraintes et de variables pour guider efficacement la recherche. Cette méthode utilise un espace mémoire dynamique dans l'esprit de la recherche Tabou afin d'éviter une stagnation dans un minimum local ou un cycle.

Dans un premier temps, l'algorithme génère une solution initiale complète en plaçant aléatoirement chaque forme urbaine fournie par le modèle urbain sur une cellule vierge de la grille en respectant néanmoins la correspondance d'intensité entre la cellule et la forme urbaine de sorte que la totalité des formes urbaines provenant du modèle urbain soit répartie sur la grille et que chaque cellule soit associée à une et une seule forme urbaine compatible avec son niveau d'intensité. Ensuite, l'algorithme effectue une série de permutations (échange des formes urbaines) entre deux cellules sélectionnées pour améliorer le coût global de la grille, une permutation ne pouvant être réalisée qu'entre deux cellules de même intensité : ce qui permet de satisfaire à tout moment la contrainte d'intensité.

Toutes nos expérimentations ont été effectuées sur un même ordinateur portable HP EliteBook doté d'un processeur Intel i7-2630QM composé de 8 cœurs et cadencé à 2 GHz avec une mémoire vive de 4 Go dans un environnement Ubuntu 11.10. Chaque résultat correspond à une moyenne réalisée sur 10 exécutions successives en partant d'une même configuration initiale. L'application a été développée en C++ et s'appuie sur le framework *EasyLocal++*[8] [6]. Concernant la contrainte d'accessibilité (section 3.2.4), la version globale a été désactivée au profit de la version locale qui offre de bien meilleurs temps de résolution (Exemple : Passage de 4 min à quelques millisecondes pour la recherche d'une meilleure permutation sur une grille 64x64).

4.1 Algorithme de base

Notre méthode réalise une succession de permutations (échange des valeurs entre deux variables), de manière à améliorer le plus possible la fonction de coût globale à chaque itération. Dans [6], les auteurs qualifient cette opération de "réparation itérative" consistant à réduire en priorité l'anomalie sur la plus mauvaise variable.

5. http ://www.emn.fr/z-info/choco-solver/
6. http ://satt.diegm.uniud.it/EasyLocal++/

Le processus se termine lorsque plus aucune permutation ne permet d'améliorer la fonction de coût globale.

Algorithme :

- V : ensemble des variables,
- C : ensemble des contraintes,
- p^{best} : meilleure permutation trouvée,
- $\triangle^{best}$: delta sur le coût global relatif à p^{best},
- max : nombre de candidats à sélectionner (constante).

```
Realiser une assignation aléatoire des
    variables de V (Cf §4)
Repeter
  Calculer les coûts liés aux contraintes de
       C et projeter les coûts sur chaque
       variable de V en considérant
       uniquement les contraintes où elles
       apparaissent
  Selectionner le sous-ensemble
      V' = {v_1, ..., v_max} de variables parmi V,
      non présentes dans la liste tabou et
      ayant les coûts les plus importants
  △^best ← 0, p^best ← ∅
  Pour_chaque candidat v ∈ V' Faire
  Debut
    candidat_tabu ← true
    Pour_chaque permutation p possible à
         partir de v Faire
    Debut
      Evaluer la variation du coût △_p liée à
           la permutation p
      Si (△_p < 0) Alors
        candidat_tabu ← false
        Si (△_p < △^best) Alors
          △^best ← △_p
          p^best ← p
          numberOfBest ← 1
        Sinon Si (△_p = △^best) Alors
          Si (Random(0, numberOfBest) = 0) Alors
            p^best ← p
          Fin Si
          numberOfBest ← +1
        Fin
      Fin
    Fin
    Si (candidat_tabu) Alors
      Inserer v dans la liste tabou
    Fin
  Fin
  Si (p^best ≠ ∅) Alors
    Echanger les valeurs des variables
        impliquées dans p^best
  Sinon
    Gerer la liste Tabou (Cf §4.2)
  Fin
Tant_que condition de fin non remplie
```

Pour trouver la meilleure permutation à chaque itération et afin de ne pas avoir à évaluer l'ensemble des permutations possibles, nous utilisons une méta-heuristique proche

de celle proposée dans *Adaptive Search* qui consiste à utiliser la structure du problème pour guider la recherche. Pour ce faire, nous introduisons la notion de "*bon candidat"* (ou plus simplement de "*candidat*") pour trouver *"la meilleure permutation*". Il s'agit d'une variable présentant un coût important par rapport aux autres variables et donc susceptible d'améliorer de façon significative la fonction de coût globale en permutant sa valeur avec une autre variable. *Adaptive Search* identifie un seul *candidat* à chaque itération et recherche, à partir de ce *candidat*, la meilleure permutation possible. Pour ce qui nous concerne, nous considérons que le meilleur candidat n'est peut être pas celui qui permettra de réaliser la meilleure permutation. Compte tenu de cette hypothèse, nous évaluons, à chaque itération, les meilleurs candidats (nombre de candidats paramétrable) à partir desquels nous recherchons la meilleure permutation avec l'ensemble des cellules de la grille. Nous exploitons également un espace mémoire dynamique pour éliminer des évaluations suivantes les candidats précédemment sélectionnés mais incapables de produire un bénéfice.

4.2 Gestion de la liste Tabou

Contrairement à une recherche Tabou classique, nous stockons dans la liste Tabou les "mauvais" candidats sous la forme <*variable*> et non avec des couples <*variable, valeur*>. Cette liste est de type *FIFO* [7] avec une taille limitée, de sorte que les plus anciens éléments soient retirés de la liste lorsqu'elle est pleine. Lorsqu'aucun des candidats sélectionnés pendant une itération n'est en mesure de produire le moindre bénéfice sur la fonction de coût globale, la liste *Tabou* est agrandie (application d'un pourcentage d'augmentation sur sa taille courante) sans toutefois pouvoir dépasser une taille maximale et on relance alors une nouvelle itération. Lorsque la liste Tabou a atteint sa taille maximale et qu'elle ne peut plus être agrandie, on vide la liste Tabou au lieu de l'agrandir et on relance une nouvelle itération dans la limite d'un nombre d'itérations infructueuses sans extension possible de la liste Tabou.

Algorithme

```
Si (taille_tabu < taille_max_tabu) Alors
  Augmenter la taille de la liste Tabou en
      fonction % paramétré
Sinon Si
    nbre_iter_without_improve = max_iter_without_improve
    Alors
  Condition de fin remplie
Sinon
  Vider la liste Tabou
  nbre_iter_without_improve ← +1
Fin
```

7. First In First Out

NB : Pour pouvoir augmenter la taille de la liste Tabu, il faut qu'elle soit pleine (nombre éléments = taille_tabu).

Résultats

Dans la table 1, nous comparons à intervalle de temps régulier, l'évolution du coût global entre une résolution avec liste tabou à taille fixe et une résolution avec liste tabou à taille variable. Nous voyons que ces paramètres liés à la liste tabou sont sensibles et qu'ils influent de façon conséquente sur les coûts. Pour obtenir les meilleurs résultats avec une liste tabou à taille variable, nous avons identifié de façon empirique que la borne supérieure devait être très légèrement inférieure au nombre de variables du problème (Par exemple : max :1000 pour une grille 32x32 qui compte 1024 cellules).

Temps (s)	Coût global		% gain
	Tabou fixe	Tabou variable	
10	97259	97047	0,22
20	63517	63527	-0,02
30	52453	52369	0,16
40	46922	38653	17,62
50	44326	30588	30,99
60	43000	26718	37,87
70	42162	25574	39,76
80	41975	25285	40,64
90	41707	24757	40,86
100	41510	24550	41,29
110	41501	24365	41,44
120	41474	24287	41,45

TABLE 1 – Comparaison des coûts à intervalle de temps régulier entre une résolution avec liste tabou à taille fixe (255) et liste tabou à taille variable (mini : 255, maxi : 1000, % incrémentation : 100%) sur une grille 32x32 avec sélection d'un seul candidat à chaque itération

Néanmoins, lorsque la taille fixe est définie avec la borne maximale (1000, et non avec la borne minimale 255), on obtient d'aussi bons résultats qu'une résolution avec une taille variable. Pour la liste à taille variable, si on fixe la borne minimale avec une petite valeur (10) et un pourcentage de variation à 25 %, les résultats restent similaires. Cette liste tabou permet d'écarter des évaluations les mauvais candidats incapables de produire le moindre bénéfice lorsqu'on échange leurs valeurs avec une autre cellule de la grille. Elle permet ainsi d'économiser du temps de calcul, temps proportionnel au nombre de candidats évalués à chaque itération et à la taille de la grille. Au début de la procédure de résolution, le nombre de mauvais candidats est faible mais il augmente tout au long du processus de résolution. Pour ne pas se priver d'un mauvais candidat pouvant devenir bon au fil des itérations, il est conseillé de limiter la taille de cette liste tabou pour qu'il puisse être réévalué périodiquement. Cependant, la taille de la liste doit

être proportionnelle au nombre de mauvais candidats pour ne pas pénaliser les temps de résolution.

4.3 Optimisations

Nos fonctions de coût sont définies par contrainte et permettent d'évaluer chaque variable. Nous pouvons ainsi calculer le coût de chaque variable et le mémoriser dans un cache. En effet, on y fait très souvent référence pour identifier les bons candidats ou pour évaluer le coût incrémental d'une permutation. Lorsqu'une permutation est réalisée sur la grille, ce qui est relativement rare en comparaison du nombre d'évaluations, on vide le cache impacté par la permutation (uniquement pour les variables "locales" concernées).

4.4 Adaptations de la méta-heuristique

En complément du mode multi-candidats (section 4.1), nous proposons un mode multi-permutations qui consiste à identifier les meilleures permutations sur une configuration donnée. Au lieu de ne retenir que la meilleure permutation pour le candidat en cours d'évaluation, nous retenons les n meilleures permutations (n : nombre paramétrable) relatives aux différents candidats évalués. Il n'y a pas de surcoût significatif puisque les permutations doivent être évaluées entre chaque candidat et l'ensemble des cellules de la grille, juste une empreinte mémoire correspondant à la liste des meilleures permutations. Une fois cet ensemble identifié, chaque permutation provenant de la liste est réévaluée pour déterminer le gain. Si ce dernier est acceptable, la permutation est appliquée (échange des valeurs entre les deux variables concernées). Pour qu'une permutation soit acceptée, il faut qu'elle puisse encore produire un bénéfice (après l'application des permutations précédentes mieux classées) et que la variation relative à son bénéfice initial (celui qui a permis de la sélectionner) ne soit pas trop importante (variation paramétrable).

Cand-Perm	10s	20s	30s	40s	50s
1-1	97047	63527	52369	38653	30588
1-24	**84943**	**56143**	**45800**	**33113**	**27561**
6-1	210444	172823	146546	125523	106699
6-24	155755	101670	69208	54920	43110

TABLE 2 – Comparaison des modes candidat unique, multi-candidats et multi-permutations (avec maxdeviation=100). La codification choisie pour Cand-Perm indique respectivement le nombre maximum de candidats et de permutations.

Sur le tableau de comparaison (2), il ressort clairement que le mode multi-candidats (Cand-Perm : 6-1) est très pénalisant. En effet, le coût engendré par l'évaluation successive des permutations pour chaque candidat est excessif par rapport à la qualité des permutations qui peuvent être retrouvées. Toutefois, on constate que le mode multi-permutations (Cand-Perm : 1-24) améliore légèrement les performances du mode *Adaptive Search* (Cand-Perm : 1-1) et lorsqu'il est associé au mode multi-candidats, le mode multi-permutations (Cand-Perm : 6-24) se révèle très efficace en permettant de compenser une bonne partie du surcoût engendré par le mode multi-candidats.

4.5 Mode distribué

Pour obtenir plus rapidement de bonnes solutions sur des grilles de grandes tailles, nous proposons de paralléliser la recherche des meilleures permutations [9]. Notre motivation de base est la suivante : le mode multi-permutations est efficace, principalement lorsqu'il opère avec des candidats multiples. Or, si la recherche des meilleures permutations par candidat peut être répartie sur différents processus, nous devrions pouvoir obtenir des gains significatifs. Dans cette implémentation, nous proposons une version basique où la grille globale est décomposée en 4 parties uniformes.

❶	❷
❸	❹

TABLE 3 – Décomposition de la grille globale ou sous-espaces pour paralléliser la recherche des meilleures permutations

On confie cette grille globale à dix processus et chaque processus (*Slave*) prend en charge l'une des configurations suivantes pour effectuer la recherche des meilleures permutations pour un seul candidat (par sous-espace) :

Slave	sous-espace	*Slave*	sous-espace
1	1	6	1 ; 3
2	2	7	1 ; 4
3	3	8	2 ; 3
4	4	9	2 ; 4
5	1 ; 2	10	3 ; 4

TABLE 4 – Répartition des sous-espaces confiés à chaque S*lave*

La grille complète est transmise à chaque processus. Cependant, chaque *Slave* recherche les meilleures permutations possibles uniquement sur le et les sous-espaces qui lui sont attribués. De cette manière, les fonctions de coût liées à chaque contrainte ne sont pas impactées car elles peuvent continuer à opérer sur l'ensemble de la grille si nécessaire. Dans la mesure où cette recherche des meilleures permutations ne modifie pas l'état de la grille, les traitements peuvent être exécutés en parallèle sans aucune difficulté et sans changer les procédures. Le problème se résume alors à limiter les couples de cellules à évaluer. Pour un sous-espace unique, le processus cherche les permutations relatives à cet espace uniquement. Lorsque deux sous-espaces

sont confiés à un processus, on veut alors évaluer les permutations entre ces deux espaces uniquement, et non les permutations à l'intérieur de chaque espace. De cette façon, on couvre l'ensemble des permutations possibles sur la grille globale. Chaque processus *Slave* est synchronisé avec un processus *Master* qui collecte l'ensemble des permutations, les évalue et les applique (idem modes multi-permutations & multi-candidats) dans son contexte local. Le processus *Master* transmet alors les permutations qui ont été jouées à chacun des *Slaves* pour une synchronisation de chaque sous-espace distribué. Une fois synchronisés, les *Slaves* peuvent rechercher de nouvelles permutations en parallèle. Les processus *Slaves* peuvent être localisés sur un même ordinateur ou sur des machines différentes. La synchronisation entre les différents processus se fait par échange de messages en utilisant *Apache Active MQ* [8] et la librairie ActiveMQ-CPP [9].

Résultats

Dans le tableau (5), il apparaît clairement que la recherche distribuée des permutations par candidat est efficace (améliore le coût) pour des instances de taille importante. En effet, malgré le coût des échanges de messages entre les *Slaves* et le *Master*, le mode distribué obtient de bons résultats plus rapidement qu'avec le mode *Adaptive Search*. Néanmoins, sur de petites instances (8x8, 16x16), nos tests ont révélé que le mode distribué était moins performant.

Taille	Proc. //	Coût global				
		10s	20s	30s	40s	50s
32x32	1	97047	63527	52369	38653	30588
	10	55469	34973	29692	27978	26947
	% gain	*42,84*	*44,95*	*43,30*	*27,62*	*11,90*
40x40	1	267826	187843	139825	115967	105888
	10	152481	99727	81593	69752	57691
	% gain	*43,07*	*46,91*	*41,65*	*39,85*	*45,52*
48x48	1	501027	401539	340903	288436	248901
	10	346914	222811	164951	143063	130542
	% gain	*30,76*	*44,51*	*51,61*	*50,40*	*47,55*
56x56	1	763724	665907	599610	540228	491394
	10	604916	453623	357257	291693	247670
	% gain	*20,79*	*31,88*	*40,42*	*46,01*	*49,60*
64x64	1	1109363	1020866	947762	882787	830168
	10	939044	778496	663769	564895	491348
	% gain	*15,35*	*23,74*	*29,96*	*36,01*	*40,81*

TABLE 5 – Comparaison de l'évolution des coûts à chaque intervalle de temps entre la méthode *Adaptive Search* et le mode Distribué (chaque *Slave* exploite à chaque itération un seul candidat et les 10 meilleures permutations)

8. Middleware Orienté Messages, http ://activemq.apache.org/
9. http ://activemq.apache.org/cms/using-activemq-cpp.html

5 Conclusion

Nous avons présenté une modélisation CSP pour un problème combinatoire d'affectation de formes urbaines à des îlots dans le cadre de la réalisation d'un outil d'aide à la décision pour les urbanistes. Cette modélisation intègre différentes contraintes spatiales (de voisinage, de distance et de regroupement) entre formes urbaines. Nous avons proposé et implémenté des variations de l'algorithme *Adaptive Search* pour résoudre efficacement différentes instances du problème. Nous avons également proposé une approche de résolution distribuée qui montre de bonnes performances (en temps et en qualité). Les résultats ont donné lieu à une permière évaluation par les urbanistes impliqués dans le projet SUSTAINS qui retrouvent une organisation graphique cohérente de l'espace urbain.

Basés sur ces premiers résultats, les travaux de recherche vont se focaliser sur (i) l'identification et le calcul d'indicateurs de performance permettant d'évaluer les propriétés d'une ville durable, (ii) l'intégration de nouvelles contraintes exprimées par les urbanistes et (iii) la manipulation interactive des solutions avec maintien des contraintes.

Références

[1] Serge SALAT, Les villes et les formes : Sur l'urbanisme durable, Editions Hermann, 2011

[2] United Nations, Department of Economic and Social Affairs, World urbanization prospects : the 2007 revision, Urban and Rural Areas, 2007

[3] Eugene C Freuder, In pursuit of the Holy Grail, Constraints 2(1), 1997

[4] Jean-Philippe ANTONI, Modéliser la ville : Formes urbaines et politiques de transport, Economica, 2010

[5] S. Moujahed, Approche multi-agents auto-organisée pour la résolution de contraintes spatiales dans les problèmes de positionnement mono et multi-niveaux, Thèse de doctorat, Université de Technologie de Belfort-Montbéliard, 2007

[6] Philippe Codognet, Daniel Diaz, Yet Another Local Search Method for Constraint Solving, 2001

[7] Pascal Van Hentenryck, Laurent Michel, Constraint-Based Local Search, MIT Press, 2009

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

[9] El-Ghazali Talbi : Metaheuristics - From Design to Implementation. Wiley 2009, isbn 978-0-470-27858-1, pages 460-502

Compilation de requêtes à préférences en problèmes de contraintes qualitatives

Jean-François Condotta[1] **Souhila Kaci**[2]

[1]Université Lille-Nord de France, Artois, CRIL-CNRS, UMR 8188
Rue de l'université, F-62307
Lens, France
condotta@cril.fr

[2]Université Montpellier 2, LIRMM-CNRS, UMR 5506
161 rue Ada, F-34095
Montpellier, France
kaci@lirmm.fr

Résumé

Les préférences comparatives sont les ingrédients de base des logiques conditionnelles pour la représentation compacte de préférences d'utilisateurs. Ces préférences peuvent être strictes ou non et obéir à différentes sémantiques. Dans la littérature, des algorithmes ont été développés afin de calculer une relation de préférence sur des alternatives étant donné un ensemble de préférences et une ou plusieurs sémantiques. Ces algorithmes en utilisant des principes du raisonnement non monotone (plus particulièrement, les principes de minimum et de maximum de spécificité) veillent à ce que les relations de préférence soient des préordres complets. Cependant, la limitation principale de ces logiques concernent les requêtes de préférences lorsque deux alternatives sont comparées. En effet, étant donné deux alternatives ayant la même préférence par rapport à la relation de préférence, il n'y a aucune indication permettant de savoir si cette égalité résulte d'une égalité des deux préférences ou d'une incomparabilité des deux alternatives qui a été réduite en une égalité par les principes de spécificité. D'autre part, les préférences comparatives et les sémantiques associées peuvent être traduites en problèmes de satisfaction de contraintes qualitatives à partir desquels nous pouvons avoir un ordre précis entre deux alternatives. Dans cet article, nous étudions cette passerelle et proposons une compilation des requêtes de préférences basées sur des logiques conditionnelles en problèmes de contraintes qualitatives.

1 Introduction

Les préférences sont utiles dans de nombreux problèmes de la vie quotidienne, guidant les humains à prendre des décisions des plus élémentaires jusqu'à des décisions organisationnelles et professionnelles complexes. Les préférences sont particulièrement importantes dans le cadre du raisonnement non monotone, dans les systèmes multi-agents, pour la prise de décision, etc.
Il est communément admis qu'ordonner la totalité des alternatives est simplement irréalisable. Ceci, du fait que l'ensemble des alternatives est généralement trop grand. Heureusement, en pratique, nous avons les préférences posées sur des descriptions partielles d'alternatives. Ces préférences peuvent être données sous différents formats. Cependant, elles se réfèrent implicitement ou explicitement à des préférences de la forme "préférer X à Y". Nous pouvons aussi rencontrer des préférences comparatives conditionnelles de la forme "si Z alors préférer X à Y". Ces préférences conditionnelles permettent d'exprimer des préférences générales et des préférences plus spécifiques. Par exemple, on

peut préférer *le poisson* à *la viande* mais cette préférence peut être inversée (i.e., préférer *la viande* au *poisson*) quand *du vin rouge* est servi.

Les logiques conditionnelles sont des langages de représentation de préférences qui supportent la première forme de préférences. Elles utilisent différents principes de complétion afin de calculer une relation de préférence induite par un ensemble de préférences. Elles utilisent des éléments du raisonnement non monotone, nommément les principes de spécificité minimale et maximale, pour calculer une relation de préférence (qui est un préordre complet) sur des alternatives étant donné un ensemble de préférences. Les logiques conditionnelles ont été étendues pour traiter des préférences comparatives non strictes. Plus précisément, on peut établir que "X est au moins aussi préféré que Y" (conditionnellement ou inconditionnellement). Ainsi, des préférences de la forme "X et Y sont autant préférés" peuvent être également formulées dans ces logiques. Comme les préférences sont exprimées sur des descriptions partielles d'alternatives, chaque préférence comparative conduit à la comparaison de deux ensembles d'alternatives. Différentes manières sont possibles pour réaliser une telle comparaison. Elles sont appelées sémantiques de préférences. Elles expriment plus ou moins des contraintes fortes sur la relation de préférence associée à l'ensemble des préférences comparatives posées.

Dans la proposition originale des logiques conditionnelles, seules des préférences strictes sont considérées. Deux alternatives ayant une égale préférence par rapport à la relation de préférence est interprété comme deux alternatives originellement incomparables. Les principes de spécificité renforcent leur égalité puisqu'elles sont préférées à toutes les alternatives de rang inférieur dans le préordre et moins préférées à toutes les alternatives de rang supérieur par rapport à ces principes. Cependant, ceci n'est plus le cas avec les logiques conditionnelles étendues, i.e. lorsque des préférences non strictes sont présentes. En fait, la raison d'une égalité entre deux alternatives est perdue. Nous ne savons plus si elles sont incomparables ou également préférées par rapport à des préférences non strictes. Cette information est particulièrement importante dans des systèmes de recommandation par exemple.

Fort heureusement, la représentation de préférences et les problèmes de satisfaction de contraintes sont des thèmes proches rendant possible la résolution du problème décrit précédemment. Plus précisément, des préférences comparatives et leur sémantique associée peuvent être traduites en problèmes de satisfaction de contraintes qualitatives dans lesquels nous pouvons avoir un ordre précis sur deux alternatives. Dans cet article, nous étudions cette passerelle et présentons une compilation des requêtes à préférences basées sur des logiques conditionnelles en problèmes de contraintes qualitatives.

Après des rappels nécessaires donnés dans la section 2, nous présentons les logiques conditionnelles en section 3. Puis nous donnons un exemple soulignant les limitations de ces logiques dans la section 4. Après cela, nous présentons les problèmes de satisfaction de contraintes sur des ensembles partiellement ordonnés en section 5 avant de proposer un encodage des requêtes à préférences basées sur des logiques conditionnelles logiques conditionnelles en problèmes de contraintes qualitatives en section 6. Pour terminer, nous concluons.

2 Rappels

Soit $V = \{V_1, \ldots, V_h\}$ un ensemble de h variables, chaque variable V_i prend ses valeurs dans un domaine $Dom(V_i)$. Une alternative possible, dénotée par ω, est le résultat d'une attribution d'une valeur de $Dom(V_i)$ à la variable V_i pour chacune des variables $V_i \in V$. Ω dénote l'ensemble de toutes les alternatives possibles. Nous supposons que cet ensemble est fixé et fini. Soit $\mathcal{L}$ un langage basé sur V. $Mod(\alpha)$ dénote l'ensemble des alternatives satisfaisant la formule α (construite sur $\mathcal{L}$). Cet ensemble est appelé *α-alternatives*. Une relation de préférence $\succeq$ sur $\mathcal{X}$= $\{x, y, z, \ldots\}$ est une relation binaire $\subseteq \mathcal{X} \times \mathcal{X}$ telle que $x \succeq y$ lorsque "x est au moins aussi préféré que y". $x \approx y$ signifie que $x \succeq y$ et $y \succeq x$ sont satisfaits i.e., x et y sont autant préférés. Enfin, $x \sim y$ signifie que ni $x \succeq y$ et ni $y \succeq x$ sont satisfaits, i.e., x et y sont incomparables. La notation $x \succ y$ exprime que x est strictement préféré à y. Nous avons $x \succ y$ si $x \succeq y$ est satisfait et $y \succeq x$ ne l'est pas. $\succeq$ est un préordre sur $\mathcal{X}$ ssi $\succeq$ est réflexive et transitive, i.e., $\forall x \in \mathcal{X}$, $x \succeq x$ est satisfait et $\forall x, y, z \in \mathcal{X}$, si $x \succeq y$ et $y \succeq z$ alors $x \succeq z$. Nous supposons qu'une relation de préférence est un préordre. $\succeq$ est complet ssi $\forall x, y \in \mathcal{X}$, $x \succeq y$ ou $y \succeq x$ est satisfait. $\succeq$ est cyclique ssi $\exists x, y \in \mathcal{X}$ tels que $x \succ y$ et $y \succ x$ sont satisfaits. Dans le cas contraire, il est acyclique.
Étant données une relation de préférence $\succeq$ et une formule α, l'ensemble des α-alternatives maximalement (resp. minimalement) préférées est dénoté par $\mathsf{max}(\alpha, \succeq)$ (resp. $\mathsf{min}(\alpha, \succeq)$) et est défini par :

$$\mathsf{max}(\alpha, \succeq) = \{\omega | \omega \in Mod(\alpha), \nexists \omega' \in Mod(\alpha), \omega' \succ \omega\},$$
$$\mathsf{min}(\alpha, \succeq) = \{\omega | \omega \in Mod(\alpha), \nexists \omega' \in Mod(\alpha), \omega \succ \omega'\}.$$

Pour des raisons pratiques, un préordre complet $\succeq$ peut également être représenté par une partition bien

ordonnée de Ω. C'est une représentation équivalente, dans le sens où chaque préordre correspond à une partition ordonnée et vice versa.

Définition 1 (Partition) *Une séquence d'ensembles d'alternatives de la forme $(E_1, \ldots, E_n)$ est une partition de Ω ssi*

(i) $\forall i$, $E_i \neq \emptyset$,

(ii) $E_1 \cup \ldots \cup E_n = \Omega$, et

(iii) $\forall i, j$, $E_i \cap E_j = \emptyset$ pour $i \neq j$.

Une partition de Ω est ordonnée si et seulement si elle est associée avec un préordre $\succeq$ sur Ω tel que ($\forall \omega, \omega' \in \Omega$ avec $\omega \in E_i, \omega' \in E_j$ nous avons $i \leq j$ ssi $\omega \succeq \omega'$).

Définition 2 *[12] Soient $\succeq$ et $\succeq'$ deux préordres complets sur Ω représentés respectivement par $(E_1, \ldots, E_n)$ et $(E'_1, \ldots, E'_{n'})$. Nous dirons que $\succeq$ est moins spécifique que $\succeq'$, noté par $\succeq \sqsubseteq \succeq'$, ssi $\forall \omega \in \Omega$, si $\omega \in E_i$ et $\omega \in E'_j$ alors $i \leq j$. $\succeq$ appartient à l'ensemble des préordres minimalement (resp. maximalement) spécifiques, parmi un ensemble de préordres complets, si et seulement si, il n'existe pas de préordre dans l'ensemble qui est strictement moins (resp. plus) spécifique que $\succeq$. Si $\succeq$ est l'unique préordre complet minimalement (resp. maximalement) spécifique alors il est appelé le préordre le moins spécifique (resp. le plus spécifique).*

3 Logiques conditionnelles

Dans cette section nous présentons quelques rappels concernant les logiques conditionnelles. Pour plus de détails, le lecteur peut se reporter à [4, 3, 2, 11, 6].

3.1 Préférences comparatives

L'ingrédient de base des logiques conditionnelles sont les préférences comparatives qualitatives de la forme "préférer α à β". Le gestion de telles préférences est facile lorsque α et β font chacune référence à une alternative. Cependant, cette tâche devient complexe lorsque α et β référencent des ensembles d'alternatives, en particulier lorsqu'elles partagent des alternatives. Afin de prévenir cette situation, von Wright [10] interprète "préférer α à β" comme un problème de choix entre entre $\alpha \wedge \neg\beta$ et $\beta \wedge \neg\alpha$. Nous avons ainsi "préférer α à β" qui conduit à préférer les $\alpha \wedge \neg\beta$-alternatives aux $\beta \wedge \neg\alpha$-alternatives. Des situations particulières apparaissent lorsque $\alpha \wedge \neg\beta$ (resp. $\beta \wedge \neg\alpha$) est une contradiction ou n'est pas réalisable. En pareil cas il est remplacé par α (resp. β). Nous reportons le lecteur à [10] pour plus de détails. Pour des raisons de simplicité, nous supposons que $\alpha \wedge \neg\beta$ et $\beta \wedge \neg\alpha$ sont cohérents et réalisables.

Remarque 1 *On pourrait se demander si "préférer poisson à vin rouge" est une préférence puisque nous comparons la valeur de deux variables différentes, nommément plat principal (i.e., poisson) et vin (i.e., vin rouge). C'est en fait une déclaration d'importance. C'est à dire qu'il est plus important pour un individu d'avoir un menu composé d'un poisson et sans vin rouge plutôt qu'un menu composé de vin rouge sans poisson. Ainsi, les menus composés de poisson et de vin blanc seront préférés aux menus composés de viande et de vin rouge. La clause "préférer α à β" est une déclaration de préférence lorsque α et β font référence aux valeurs d'une même variable e.g. "préférer poisson à viande". Que "préférer α à β" corresponde à une déclaration de préférence ou une déclaration d'importance, cela revient à préférer les $\alpha \wedge \neg\beta$-alternatives aux $\beta \wedge \neg\alpha$-alternatives. Pour cette raison, nous ne faisons pas la distinction entre une déclaration de préférence et une déclaration d'importance.*

Notons que des préférences comparatives peuvent s'exprimer par rapport à un contexte. Elles sont alors de la forme "si γ, préférer α à β" qui s'entend pour "préférer α à β lorsque γ est satisfait". Cela signifie également que nous comparons les $\gamma \wedge \alpha \wedge \neg\beta$-alternatives et les $\gamma \wedge \beta \wedge \neg\alpha$-alternatives, ce qui correspond à "préférer $\gamma \wedge \alpha$ à $\gamma \wedge \beta$". Sans perte de généralité, nous nous focalisons sur des déclarations de la forme "préférer α à β".

Pour terminer, les logiques conditionnelles ont été étendues pour gérer des préférences comparatives non strictes de la forme "α est au moins aussi préféré que β". Que la préférence de α sur β soit stricte ou non, nous devons comparer les $\alpha \wedge \neg\beta$-alternatives et les $\beta \wedge \neg\alpha$-alternatives.

Différentes manières ont été proposées dans le littérature pour comparer deux ensembles d'objets. Elles sont appelées sémantiques de préférence.

3.2 Sémantiques de préférence

Nous dénotons par $\alpha \rhd \beta$ (resp. $\alpha \trianglerighteq \beta$) une déclaration de préférence comparative "préférer α à β" ("α est au moins aussi préféré que β"). Une sémantique de préférence décrit la manière dont les $\alpha \wedge \neg\beta$-alternatives et les $\beta \wedge \neg\alpha$-alternatives sont rangées par rapport à la relation de préférence $\succeq$ (à construire). Plus précisément, elle exprime une contrainte sur $\succeq$ afin d'avoir $\alpha \rhd \beta$ (resp. $\alpha \trianglerighteq \beta$) satisfait par $\succeq$. Différentes approches ont été étudiées pour comparer deux ensembles d'objets. Elles ont conduit à différentes sémantiques de préférence. Nous rappelons les sémantiques les plus utilisées [4, 2, 9]. Grossièrement, ces sémantiques comparent deux ensembles d'alternatives (nommément les $\alpha \wedge \neg\beta$-alternatives et les $\beta \wedge \neg\alpha$-alternatives) sur la

base de leurs alternatives maximalement et minimalement préférées par rapport à une relation de préférence donnée.

Définition 3 *Soit $\succeq$ une relation de préférence. Considérons $\alpha \rhd \beta$ (resp. $\alpha \trianglerighteq \beta$).*

- **Sémantique forte** *[4, 3, 11]*
  $\succeq$ *satisfait* $\alpha \rhd \beta$ *(resp.* $\alpha \trianglerighteq \beta$*) suivant* la sémantique forte *, noté par* $\succeq \models_{st} \alpha \rhd \beta$ *(resp.* $\succeq \models_{st} \alpha \trianglerighteq \beta$*), ssi* $\forall \omega \in \mathsf{min}(\alpha \wedge \neg\beta, \succeq)$, $\forall \omega' \in \mathsf{max}(\beta \wedge \neg\alpha, \succeq)$, $\omega \succ \omega'$ *(resp.* $\omega \succeq \omega'$*).*
- **Sémantique optimiste** *[8, 4]*
  $\succeq$ *satisfait* $\alpha \rhd \beta$ *(resp.* $\alpha \trianglerighteq \beta$*) suivant* la sémantique optimiste *, noté par* $\succeq \models_{opt} \alpha \rhd \beta$ *(resp.* $\succeq \models_{opt} \alpha \trianglerighteq \beta$*), ssi* $\forall \omega \in \mathsf{max}(\alpha \wedge \neg\beta, \succeq)$, $\forall \omega' \in \mathsf{max}(\beta \wedge \neg\alpha, \succeq)$, $\omega \succ \omega'$ *(resp.* $\omega \succeq \omega'$*).*
- **Sémantique pessimiste** *[2]*
  $\succeq$ *satisfait* $\alpha \rhd \beta$ *(resp.* $\alpha \trianglerighteq \beta$*) suivant* la sémantique pessimiste *, noté par* $\succeq \models_{pes} \alpha \rhd \beta$ *(resp.* $\succeq \models_{pes} \alpha \trianglerighteq \beta$*), ssi* $\forall \omega \in \mathsf{min}(\alpha \wedge \neg\beta, \succeq)$, $\forall \omega' \in \mathsf{min}(\beta \wedge \neg\alpha, \succeq)$, $\omega \succ \omega'$ *(resp.* $\omega \succeq \omega'$*).*
- **Sémantique opportuniste** *[9]*
  $\succeq$ *satisfait* $\alpha \rhd \beta$ *(resp.* $\alpha \trianglerighteq \beta$*) suivant* la sémantique opportunistes *, noté par* $\succeq \models_{opp} \alpha \rhd \beta$ *(resp.* $\succeq \models_{opp} \alpha \trianglerighteq \beta$*), ssi* $\forall \omega \in \mathsf{max}(\alpha \wedge \neg\beta, \succeq)$, $\forall \omega' \in \mathsf{min}(\beta \wedge \neg\alpha, \succeq)$, $\omega \succ \omega'$ *(resp.* $\omega \succeq \omega'$*).*

Lorsqu'il n'y a pas d'ambiguïté, par abus de notation, nous écrirons $\alpha \rhd_* \beta$ (resp. $\alpha \trianglerighteq_* \beta$), avec $* \in \{st, opt, pes, opp\}$, pour signifier que $\alpha \rhd \beta$ (resp. $\alpha \trianglerighteq \beta$) est interprété suivant la sémantique correspondante, nommément $*$. Nous dirons donc que $\succeq$ satisfait $\alpha \rhd_* \beta$ (resp. $\alpha \trianglerighteq_* \beta$) pour signifier que $\succeq \models_* \alpha \rhd \beta$ (resp. $\succeq \models_* \alpha \trianglerighteq \beta$).

Définition 4 *Un ensemble de préférences du type $*$, dénoté par $\mathcal{P}_*$, est un ensemble de préférences de la forme $\{p_i \rhd_* q_i, p_i \trianglerighteq_* q_i | i = 1, \ldots, n\}$, avec $* \in \{st, opt, pes, opp\}$. Une relation de préférence acyclique $\succeq$ est un modèle de $\mathcal{P}_*$ si et seulement si $\succeq$ satisfait chaque préférence $p_i \rhd_* q_i$ (resp. $p_i \trianglerighteq_* q_i$) dans $\mathcal{P}_*$. Un ensemble de préférences $\mathcal{P}_*$ est cohérent s'il admet un modèle.*

3.3 Des ensembles de préférences aux relations de préférence

Généralement, nous devons traiter plusieurs préférences comparatives exprimées par un individu. Il existe principalement deux types de requêtes dans le cadre de la représentation des préférences : soit on recherche les alternatives maximalement préférées, soit on compare deux alternatives. Dans de nombreuses applications (dans le cadre de requêtes de base données par exemple), les individus sont plus concernés par les alternatives préférées. Dans le cas où ces alternatives ne sont pas satisfaisantes (les menus préférés sont trop chers par exemple), nous devons calculer les alternatives préférées parmi les alternatives restantes, et ainsi de suite. Afin de prendre en compte ces différentes considérations, nous associons un préordre complet à un ensemble de préférences. Différents préordres complets peuvent satisfaire un ensemble de préférences étant donné une sémantique. Cependant, il est largement accepté que, pour des prises de décison, il est plus pratique de caractériser un *unique* préordre complet [4]. Le principe de spécificité a été habituellement utilisé afin de caractériser un tel préordre. Proposition 1 résume des résultats existants sur l'unicité des modèles (qui sont des préordres complets) pour chacune des sémantiques.

Proposition 1 *[6]*

- *(1) Le modèle le moins spécifique de $\mathcal{P}_{\rhd_{opt}} \cup \mathcal{P}_{\trianglerighteq_{opt}}$ (resp. $\mathcal{P}_{\rhd_{st}} \cup \mathcal{P}_{\trianglerighteq_{st}}$) existe.*
- *(2) Le modèle le plus spécifique de $\mathcal{P}_{\rhd_{pes}} \cup \mathcal{P}_{\trianglerighteq_{pes}}$ (resp. $\mathcal{P}_{\rhd_{st}} \cup \mathcal{P}_{\trianglerighteq_{st}}$) existe.*
- *(3) Le modèle le plus (resp. le moins) spécifique de $\mathcal{P}_{\rhd_{opt}} \cup \mathcal{P}_{\trianglerighteq_{opt}}$ (resp. $\mathcal{P}_{\rhd_{pes}} \cup \mathcal{P}_{\trianglerighteq_{pes}}$) n'existe pas.*
- *(4) Le modèle le moins / le plus spécifique n'existe pas pour $\mathcal{P}_{\rhd_{opp}} \cup \mathcal{P}_{\trianglerighteq_{opp}}$.*

Pour des préférences comparatives uniquement non strictes, à la fois existent le modèle le moins spécifique et le modèle le plus spécifique, et il existe la relation de préférence triviale pour laquelle toutes les alternatives sont équivalentes. Ainsi, la notion de spécificité maximale et minimale pour les ensembles de préférences consistant seulement à des préférences comparatives non strictes n'est pas très utile. D'autre part, les auteurs de [6] ont montré que des sémantiques peuvent être utilisées ensemble tout en préservant l'unicité des modèles. Plus précisément, le modèle le moins (resp. le plus) spécifique de $\mathcal{P}_{\rhd_{st}} \cup \mathcal{P}_{\trianglerighteq_{st}} \cup \mathcal{P}_{\rhd_{opt}} \cup \mathcal{P}_{\trianglerighteq_{opt}}$ (resp. $\mathcal{P}_{\rhd_{st}} \cup \mathcal{P}_{\trianglerighteq_{st}} \cup \mathcal{P}_{\rhd_{pes}} \cup \mathcal{P}_{\trianglerighteq_{pes}}$) existe.

Dans cet article, nous cherchons des modèles uniques, aussi nous ne considérerons pas la sémantique opportuniste. Nous ne donnons pas d'algorithmes pour calculer les uniques modèles et reportons le lecteur à [6].

4 Exemple et problème

Dans cette section, nous soulevons un problème qui se pose dans le contexte de requêtes à préférences avec des logiques conditionnelles.

Exemple 1 *Supposons qu'un individu souhaite planifier ses vacances. Elle/il exprime ses préférences sur la base de trois variables : P (pour la période) qui est soit W (hiver) ou S (été), , D (pour la destination) qui est soit M ou B (resp. montagne et plage) et L (pour le lieu) qui vaut soit H ou A (resp. hôtel et appartement). Ainsi, nous avons* $Dom(P) = \{W, S\}$, $Dom(D) = \{M, B\}$, $Dom(L) = \{H, A\}$ *et* $\Omega = \{\omega_0 : HMW, \omega_1 : HMS, \omega_2 : HBW, \omega_3 : HBS, \omega_4 : AMW, \omega_5 : AMS, \omega_6 : ABW, \omega_7 : ABS\}$. *L'individu exprime cinq préférences :*

- (i) *elle/il préférerait voyager en été plutôt qu'en hiver,*
- (ii) *si la destination est la montagne alors elle/il préférerait voyager en hiver plutôt qu'en été,*
- (iii) *si elle/il voyage en hiver alors elle/il préférerait louer un appartement plutôt qu'aller à l'hôtel,*
- (iv) *si elle/il va à l'hôtel alors elle/il préférerait la plage à la montagne et,*
- (v) *si elle/il va à l'hôtel alors été et hiver sont autant préférés.*

Formellement nous écrivons $\mathcal{P}_* = \{s_1 : S \rhd_* W, s_2 : M \wedge W \rhd_* M \wedge S, s_3 : W \wedge A \rhd_* W \wedge H, s_4 : H \wedge B \rhd_* H \wedge M, s_5 : H \wedge S \trianglerighteq_* H \wedge W, s_6 : H \wedge W \trianglerighteq_* H \wedge S\}$. *Remarquons que* s_5 *et* s_6 *pris ensemble expriment la préférence (*v*). Nous associons à chaque préférence* $s_i : p_i \rhd_* q_i$ *un couple* $C_i = (L(s_i), R(s_i))$ *tel que* $L(s_i) = Mod(p_i \wedge \neg q_i)$ *et* $R(s_i) = Mod(q_i \wedge \neg p_i)$. *Ainsi, nous avons* $\mathcal{L}(\mathcal{P}_*) = \{C_1 = (\{\omega_1, \omega_3, \omega_5, \omega_7\}, \{\omega_0, \omega_2, \omega_4, \omega_6\}), C_2 = (\{\omega_0, \omega_4\}, \{\omega_1, \omega_5\}), C_3 = (\{\omega_4, \omega_6\}, \{\omega_0, \omega_2\}), C_4 = (\{\omega_2, \omega_3\}, \{\omega_0, \omega_1\}), C_5 = (\{\omega_1, \omega_3\}, \{\omega_0, \omega_2\}), C_6 = (\{\omega_0, \omega_2\}, \{\omega_1, \omega_3\})\}$.
*Soit *=opt. La relation de préférence associée est* $\succeq = (\{\omega_7\}, \{\omega_4, \omega_6\}, \{\omega_2, \omega_3, \omega_5\}, \{\omega_0, \omega_1\})$. *Nous avons* ω_4 *et* ω_6 *(resp.* $\omega_2, \omega_3, \omega_5$ *et* ω_0, ω_1*) qui sont autant préférés. Cependant, seuls* ω_2 *et* ω_3 *sont autant préférés du fait de* s_5 *et* s_6. *Toutes les autres égalités sont des incomparabilités transformées en égalités à cause du principe de spécificité. Cette transformation peut être importante dans des systèmes de recommandation expliquant si les alternatives préférées (ou n'importe quel couple d'alternatives) sont en fait autant préférées ou incomparables.*

5 Problèmes de satisfaction de contraintes qualitatives sur des ensembles partiellement ordonnés

Dans cette section, nous introduisons des problèmes de satisfaction de contraintes qualitatives qui permettront de raisonner sur des un ensemble de préférences. Pour cela, nous faisons quelques rappels sur les relations de l'algèbre des points et introduisons des relations que nous appellerons relations $\mathsf{min/max}$.

L'algèbre des points [5], AP en abrégé, est un formalisme qualitatif [7] proposé pour la représentation et le raisonnement sur des informations temporelles à l'aide de contraintes qualitatives. Étant donné un ordre partiel $\geq$ sur un ensemble T, dénoté par $(T, \geq)$, AP considère quatre relations de base :

- $\forall x, y \in T, x > y$ ssi $x \geq y$ et non $(y \geq x)$;
- $\forall x, y \in T, x < y$ ssi $y \geq x$ et non $(x \geq y)$;
- $\forall x, y \in T, x = y$ ssi $x \geq y$ et $y \geq x$;
- $\forall x, y \in T, x \| y$ ssi non $x \geq y$ et non $y \geq x$.

L'ensemble de ces relations de base sera dénoté par $\mathsf{B_{AP}}$ dans la suite. Remarquons que ces relations sont complètes et mutuellement exclusives (*i.e.* tout couple d'éléments x et y appartenant à T satisfait une et une seule relation de $\mathsf{B_{AP}}$). En réalisant un parallèle avec une relation de préférence $\succeq$, les relations de base précédentes correspondent à une stricte préférence, à l'égalité et l'incomparabilité.

Une relation (complexe) de AP correspond à une union de relations de base de $\mathsf{B_{AP}}$. Elle est représentée par ce sous-ensemble de relations de base. Par exemple, l'ensemble $\{\|, =\}$ correspond à la relation $\| \cup =$. Elle est satisfaite par deux éléments x et y si et seulement si $x \| y$ ou $x = y$. $2^{\mathsf{B_{AP}}}$ dénotera l'ensemble des 16 relations de AP.

Maintenant, nous avons besoin de traduire la définition 3 dans le contexte des problèmes de satisfaction de contraintes. Étant donné un ordre partiel $(T, \geq)$ et un sous-ensemble $X \subseteq T$, nous définissons $\mathsf{min}(X)$ et $\mathsf{max}(X)$ de la manière suivante : $\mathsf{min}(X) = \{x \in X : \nexists y \in X$ tel que $x \geq y$ et non $(y \geq x)\}$ et $\mathsf{max}(X) = \{x \in X : \nexists y \in X$ tel que $y \geq x$ et non $(x \geq y)\}$. Nous définissons maintenant 8 relations binaires, appelées relations de base $\mathsf{min/max}$, pouvant être satisfaites par des sous-ensembles de T. Chacune de ces relations de base sera dénotée par rel^{op1}_{op2} avec $rel \in \{\geq, >\}$ et $op1, op2 \in \{\mathsf{min}, \mathsf{max}\}$. Elles sont définies de la manière suivante :

- pour $op1, op2 \in \{\mathsf{min}, \mathsf{max}\}$, $\forall X, Y \in 2^T$, $X >^{op1}_{op2} Y$ ssi $\forall x \in op1(X)$ et $\forall y \in op2(Y)$, $x \geq y$ et non $(y \geq x)$;
- pour $op1, op2 \in \{\mathsf{min}, \mathsf{max}\}$, $\forall X, Y \in 2^T$, $X \geq^{op1}_{op2} Y$ ssi $\forall x \in op1(X)$ et $\forall y \in op2(Y)$, $x \geq y$.

Dans la suite, nous noterons par $\mathsf{B_{min/max}}$ l'ensemble composé de ces 8 relations de base, $\mathsf{B_{min/max}} = \{>^{\mathsf{min}}_{\mathsf{min}}, >^{\mathsf{min}}_{\mathsf{max}}, >^{\mathsf{max}}_{\mathsf{min}}, >^{\mathsf{max}}_{\mathsf{max}}, \geq^{\mathsf{min}}_{\mathsf{min}}, \geq^{\mathsf{min}}_{\mathsf{max}}, \geq^{\mathsf{max}}_{\mathsf{min}}, \geq^{\mathsf{max}}_{\mathsf{max}}\}$.

Nous sommes maintenant prêt à définir les problèmes de satisfaction de contraintes qualitatives que nous appelons problèmes de contraintes qualitatives sur des ensembles partiellement ordonnés ($\mathsf{PCQ_{POS}}$ en abrégé). Un $\mathsf{PCQ_{POS}}$ $\mathcal{Q}$ permet de spécifier des configurations possibles d'éléments partiellement ordonnés représentés par un ensemble de variables $\mathcal{V}$. Chaque

$\mathsf{PCQ_{POS}}$ est composé de deux types de contraintes qualitatives. Le premier type est représenté par une fonction c spécifiant l'ordre relatif de chaque couple d'éléments de $\mathcal{V}$ au moyen d'une relation de AP. Les contraintes de second type sont collectées dans un ensemble $\mathcal{C}$ et s'appliquent à des sous-ensembles de $\mathcal{V}$. Plus précisément, chaque contrainte de $\mathcal{C}$ contraint l'ordre relatif entre les éléments minimums et maximums de deux ensembles d'éléments. Elle est de la forme $V\ R\ V'$ avec $V, V' \in 2^{\mathcal{V}}$ et $R \subseteq \mathsf{B}_{\mathsf{min/max}}$. Dans la suite, de telles contraintes seront appelées contraintes min/max sur $\mathcal{V}$ et correspondront à l'ensemble dénoté par $\mathcal{C}^{\mathcal{V}}_{\mathsf{min/max}}$. Formellement, un $\mathsf{PCQ_{POS}}$ est défini de la manière suivante :

Définition 5 *Un* $\mathsf{PCQ_{POS}}$ $\mathcal{Q}$ *est un triplet* $(\mathcal{V}, c, \mathcal{C})$ *où :*

- $\mathcal{V}$ *est un ensemble non-vide de variables ;*
- c *est une fonction associant une relation* $c(v, v') \in 2^{\mathsf{B_{AP}}}$ *à chaque couple* (v, v') *de* $\mathcal{V} \times \mathcal{V}$. c *est telle que* $c(v, v) \subseteq \{=\}$ *et* $c(v, v') = (c(v', v))^{-1}$ *pour tout* $v, v' \in V$ *(où* $^{-1}$ *dénote l'opération inverse de* AP*) ;*
- $\mathcal{C}$ *est un sous-ensemble fini de* $\mathcal{C}^{\mathcal{V}}_{\mathsf{min/max}}$. *Nous supposerons que pour chaque* min/max *contrainte* $V\ R\ V' \in \mathcal{C}$, V *et* V' *sont des ensembles non vides.*

Nous dirons qu'un $\mathsf{PCQ_{POS}}$ $\mathcal{Q} = (\mathcal{V}, c, \mathcal{C})$ est atomique ssi pour tout $v, v' \in \mathcal{V}$, la relation $c(v, v')$ est définie par une relation singleton de AP. De plus, un $\mathsf{PCQ_{POS}}$ $\mathcal{Q}' = (\mathcal{V}', c', \mathcal{C}')$ est un sous-$\mathsf{PCQ_{POS}}$ de $\mathcal{Q}$ lorsque $\mathcal{V} = \mathcal{V}'$, pour tout $v, v' \in \mathcal{V}$ $c'(v, v') \subseteq c(v, v')$, $|\mathcal{C}| = |\mathcal{C}'|$ et pour chaque min/max contrainte $X\ R'\ Y \in \mathcal{C}'$, il existe $X\ R\ Y \in \mathcal{C}$ avec $R' \subseteq R$.

Définition 6 *Soit un* $\mathsf{PCQ_{POS}}$ $\mathcal{Q} = (\mathcal{V}, c, \mathcal{C})$. *Une interprétation* π *de* $\mathcal{Q}$ *est un couple* $(f, (T, \geq))$ *où* f *est une application surjective de* $\mathcal{V}$ *vers* T *et* $(T, \geq)$ *est un ordre partiel. Une interprétation* $\pi = (f, (T, \geq))$ *de* $\mathcal{Q}$ *est une solution de* $\mathcal{Q}$ *ssi :*

- *pour chaque couple de variables* $v, v' \in \mathcal{V}$, $f(v)\ c(v, v')\ f(v')$, *i.e. le couple* $(f(v), f(v'))$ *satisfait une relation de base de* $\mathsf{B_{AP}}$ *appartenant à* $c(v, v')$ *;*
- *pour chaque contrainte* $V\ R\ V' \in \mathcal{C}$, $f(V)$ *et* $f(V')$ *satisfont au moins une relation de base* min/max *appartenant à* R.

D'autre part, deux $\mathsf{PCQ_{POS}}$ définis sur un même ensemble de variables sont équivalents ssi ils admettent les mêmes solutions. Étant donné un $\mathsf{PCQ_{POS}}$ $\mathcal{Q} = (\mathcal{V}, c, \mathcal{C})$, le couple $(\mathcal{V}, c)$ correspond à un problème de satisfaction de contraintes de AP. Ainsi, les $\mathsf{PCQ_{POS}}$ peuvent être vus comme une généralisation particulière des problèmes de satisfaction de contraintes qualitatives de l'algèbre de points. De plus, nous pouvons noter que les contraintes spécifiées par la fonction c peuvent être exprimées par des contraintes appartenant à $\mathcal{C}^{\mathcal{V}}_{\mathsf{min/max}}$ et contraignant des ensembles singletons. Par exemple, la contrainte $v\ \{>, =\}\ v'$ peut être formulée de manière équivalente par la min/max contrainte $\{v\} \geq^{\mathsf{min}}_{\mathsf{min}} \{v'\}$. Comme autre exemple, considérons la contrainte de AP $v\ \{=\}\ v'$. Cette contrainte peut être exprimée par la conjonction des deux min/max contraintes $\{v\} \geq^{\mathsf{min}}_{\mathsf{min}} \{v'\}$ et $\{v'\} \geq^{\mathsf{min}}_{\mathsf{min}} \{v\}$. Ainsi, chaque $\mathsf{PCQ_{POS}}$ $\mathcal{Q} = (\mathcal{V}, c, \mathcal{C})$ peut être exprimé de manière équivalente par un $\mathsf{PCQ_{POS}}$ $\mathcal{Q}' = (\mathcal{V}, c', \mathcal{C}')$ tel que $c'(v, v') = \mathsf{B_{AP}}$ pour chaque couple de variables distinctes (v, v'). Néanmoins, c permet d'exprimer explicitement les relations d'ordre possibles entre chaque couple de variables de $\mathcal{V}$. Étant donné un $\mathsf{PCQ_{POS}}$ $\mathcal{Q} = (\mathcal{V}, c, \mathcal{C})$, différents problèmes peuvent se poser :

- *Le problème de la cohérence* : décider si $\mathcal{Q}$ est cohérent ou non, *i.e.* est-ce que $\mathcal{Q}$ admet au moins une solution ? À partir du fait que les $\mathsf{PCQ_{POS}}$ généralisent les problèmes de contraintes de AP, nous pouvons établir que le problème de la cohérence des $\mathsf{PCQ_{POS}}$s est un problème NP-complet.

- *Le problème de trouver une solution de* $\mathcal{Q}$. Ce problème revient à caractériser un équivalent sous-$\mathsf{PCQ_{POS}}$ $\mathcal{Q}'$ de $\mathcal{Q}$ tel que $\mathcal{Q}'$ est cohérent et atomique. À partir d'un tel $\mathsf{PCQ_{POS}}$ $\mathcal{Q}'$ nous pouvons facilement déduire une solution de $\mathcal{Q}$.

- *Le problème de l'étiquetage minimal* : trouver pour chaque couple de variables $(v, v') \in \mathcal{V} \times \mathcal{V}$ et chaque contrainte $X\ R\ Y \in \mathcal{C}$ l'ensemble des relations de base de $c(v, v')$ et de R pouvant être satisfaites, *i.e.* l'ensemble des relations de base de $c(v, v')$ et de R impliquées dans au moins une solution. Ce problème consiste à calculer le $\mathsf{PCQ_{POS}}$ minimal de $\mathcal{Q}$, *i.e.* l'unique sous-$\mathsf{PCQ_{POS}}$ de $\mathcal{Q}$ équivalent à $\mathcal{Q}$ et ayant la propriété de minimalité (chaque relation de base de chacune de ses contraintes est satisfiable).

Une méthode pour caractériser un atomique et cohérent sous-$\mathsf{PCQ_{POS}}$ d'un $\mathsf{PCQ_{POS}}$ $\mathcal{Q} = (\mathcal{V}, c, \mathcal{C})$ (dans le cas où il existe) consiste en une recherche avec retour arrière sur la composante AP de $\mathcal{Q}$ (*i.e.* le problème de contraintes qualitatives de AP défini par $\mathcal{N} = (\mathcal{V}, c)$). À chaque étape de la recherche, une contrainte $c(v, v')$ non définie par une relation singleton est sélectionnée et définie par une relation composée d'une de ses relations de base. De plus, la méthode de la chemin-cohérence est réalisée afin de filtrer l'espace de recherche et d'assurer la cohérence du réseau

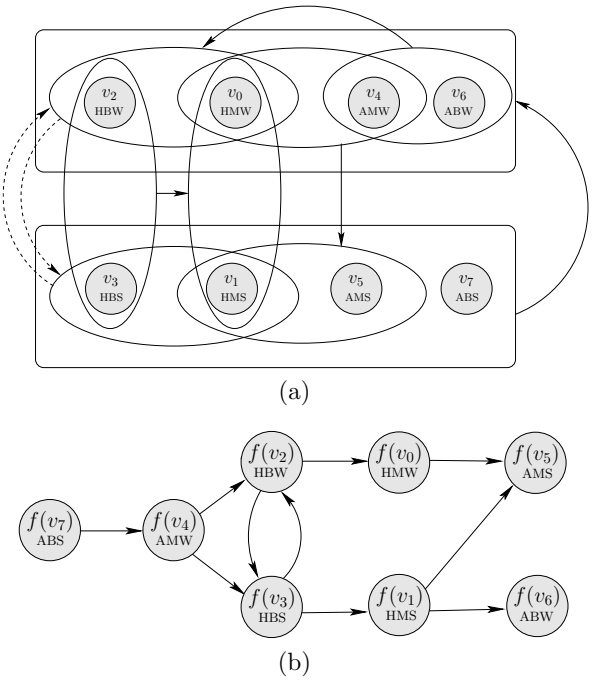


FIGURE 1 – (a) Les contraintes $\mathcal{C}$ du $\mathsf{PCQ}_{\mathsf{POS}}$ $\mathcal{Q} = (\mathcal{V}, c, \mathcal{C}) = \mathsf{PCQ}_{\mathsf{POS}}(\Omega, \mathcal{P})$ avec $\mathcal{P}$ l'ensemble de préférences sur Ω de l'exemple 1, (b) une solution $(f, \geq)$ de $\mathcal{Q}$.

atomique de contraintes qualitatives de AP partiellement construit. Notons que dans le cadre des formalismes qualitatifs, la méthode de la chemin-cohérence est une méthode polynomiale habituellement utilisée pour supprimer des relations de base non satisfiables à l'aide de l'opération de composition, voir par exemple [1]. Durant la recherche, pour assurer que le réseau atomique de contraintes qualitatives de AP construit peut être étendu en un sous-$\mathsf{PCQ}_{\mathsf{POS}}$ de $\mathcal{Q}$ atomique et cohérent, nous devons enlever les relations de base non satisfiables de chaque contrainte appartenant à $\mathcal{C}$.

6 Des préférences aux contraintes qualitatives

Dans cette section, nous allons montrer que les $\mathsf{PCQ}_{\mathsf{POS}}$ peuvent être utilisés de manière naturelle pour représenter et raisonner sur des ensembles de préférences. Considérons un ensemble d'alternatives Ω et un ensemble de préférences $\mathcal{P} = \{s_0, \ldots, s_k\}$. Nous dénoterons par $*(s_i)$, avec $i \in \{0, \ldots, k\}$, la sémantique associée à la formule s_i ($*(s_i) \in \{st, opt, pes, opp\}$) et par $\mathsf{rel}(s_i)$ la stricte/non stricte relation de préférence utilisée dans s_i ($\mathsf{rel}(s_i) \in \{\rhd, \trianglerighteq\}$). Nous définissons $\mathsf{PCQ}_{\mathsf{POS}}(\Omega, \mathcal{P})$ par le $\mathsf{PCQ}_{\mathsf{POS}}$ $(\mathcal{V}, c, \mathcal{C})$ où :

- à chaque $w_i \in \Omega$ est associée une variable $v_i \in \mathcal{V}$. Ainsi, pour $\Omega = \{w_0, \ldots, w_k\}$, $\mathcal{V}$ est défini par l'ensemble $\{v_1, \ldots, v_k\}$.
- Pour chaque variable $v, v' \in \mathcal{V}$ avec $v \neq v'$, $c(v, v')$ est par la relation universelle de AP, *i.e.* la relation B_{AP}. Pour chaque $v \in \mathcal{V}$, $c(v, v)$ est la relation $\{=\}$.
- Pour chaque préférence $s_i \in \mathcal{P}$ est définie une min/max contrainte $V_i \ R_i \ V_i' \in \mathcal{C}$. V_i et V_i' correspondent aux deux sous-ensembles de variables de $\mathcal{V}$ représentant les deux sous-ensembles d'alternatives reliés par s_i. R_i est un ensemble composé d'une unique relation de base de $\mathsf{B}_{\mathsf{min/max}}$. Cette relation dépend de la sémantique $*(s_i)$ et de la stricte / non stricte relation de préférence $\mathsf{rel}(s_i)$. Plus précisément, pour $*(s_i) = st$ (resp. $*(s_i) = opt$, $*(s_i) = pes$, $*(s_i) = opp$), pour le cas où $\mathsf{rel}(s_i)$ est $\rhd$, R_i est défini par $\{>^{\mathsf{min}}_{\mathsf{max}}\}$ (resp. $\{>^{\mathsf{max}}_{\mathsf{max}}\}$, $\{>^{\mathsf{min}}_{\mathsf{min}}\}$, $\{>^{\mathsf{max}}_{\mathsf{min}}\}$). Dans le cas où $\mathsf{rel}(s_i)$ est $\trianglerighteq$, R_i est défini par $\{\geq^{\mathsf{min}}_{\mathsf{max}}\}$ (resp. $\{\geq^{\mathsf{max}}_{\mathsf{max}}\}$, $\{\geq^{\mathsf{min}}_{\mathsf{min}}\}$, $\{\geq^{\mathsf{max}}_{\mathsf{min}}\}$) .

Exemple 2 *(Exemple 1 poursuivi) Considérons de nouveau Ω et $\mathcal{P}$. Définissons le $\mathsf{PCQ}_{\mathsf{POS}}$ $\mathcal{Q} = (\mathcal{V}, c, \mathcal{C})$ par $\mathcal{Q} = \mathsf{PCQ}_{\mathsf{POS}}(\Omega, \mathcal{P})$. Dans la figure 1.a sont représentées les* min/max *contraintes de l'ensemble $\mathcal{C}$. Un arc plein correspond à l'ensemble $\{>^{\mathsf{max}}_{\mathsf{max}}\}$ tandis qu'un arc pointillé correspond à $\{\geq^{\mathsf{max}}_{\mathsf{max}}\}$. Par exemple, nous avons les deux* min/max *contraintes $\{v_2, v_3\}\{>^{\mathsf{max}}_{\mathsf{max}}\}\{v_0, v_1\}$ et $\{v_0, v_2\}\{\geq^{\mathsf{max}}_{\mathsf{max}}\}\{v_1, v_3\}$ qui appartiennent à l'ensemble $\mathcal{C}$.*

Maintenant, nous allons étudier comment raisonner sur un ensemble de préférences à partir de sa représentation en $\mathsf{PCQ}_{\mathsf{POS}}$. Tout d'abord, nous pouvons clairement établir que la cohérence (ou la non cohérence) d'un ensemble de préférences peut être décidée par la cohérence ou non du $\mathsf{PCQ}_{\mathsf{POS}}$ correspondant. En effet, nous avons la propriété suivante :

Proposition 2 *Soit Ω un ensemble d'alternatives et $\mathcal{P}$ un ensemble de préférences. $\mathcal{P}$ est cohérent si et seulement si $\mathsf{PCQ}_{\mathsf{POS}}(\Omega, \mathcal{P})$ est cohérent.*

Pour illustration, une solution $(f, (T, \geq))$ de $\mathsf{PCQ}_{\mathsf{POS}}(\Omega, \mathcal{P})$, avec $\mathcal{P}$ l'ensemble de préférences donné dans l'exemple 1, est représenté par un graphe orienté dans la figure 1.b. Un arc $(f(v), f(v'))$ représente le fait que $f(v) \geq f(v')$. $f(v) \geq f(v')$ n'est pas représenté dans le cas où v et v' sont les mêmes variables ou quand il peut être déduit par transitivité à partir des arcs déjà présents. À partir de la proposition précédente, nous pouvons affirmer que $\mathcal{P}$ est un ensemble cohérent de préférences. Remarquons

également qu'à partir de la solution $(f, (T, \geq))$ nous pouvons facilement définir un modèle de $\mathcal{P}$.

Comme précédemment remarqué, pour certaines sémantiques de préférence, nous pouvons caractériser des modèles uniques suivant des principes de spécificité. Maintenant, nous allons montrer comment ces modèles peuvent être caractérisés à partir de la résolution de $\mathsf{PCQ}_{\mathsf{POS}}$. Plus spécifiquement, nous proposons un algorithme permettant de calculer le modèle le moins spécifique (resp. le plus spécifique) d'un ensemble de préférences étant donné une sémantique forte ou une sémantique optimiste (resp. sémantique pessimiste) dans le cas où il existe, *i.e.* l'ensemble des préférences est cohérent.

Algorithm 1: specificModel(Ω,$\mathcal{P}$,kind)

In : Un ensemble d'alternatives Ω, un ensemble $\mathcal{P}$ de préférences, le type du modèle unique avec kind $\in$ {leastSpecific, mostSpecific}.
Out : Un unique modèle représenté par une séquence $(E_0, \ldots, E_n)$ ou Inconsistent.

```
1  begin
2    if kind = leastSpecific then
3      rel ← {>, =} ;
4    else
5      rel ← {<, =};
6    Q = (V, c, C) ← PCQ_POS(Ω, P) ; k ← 0 ;
     V' ← ∅ ;
7    while V ≠ V' do
8      E_k ← ∅ ;
9      for v_i ∈ V \ V' do
10       for v ∈ V \ (V' ∪ {v_i}) do
11         c(v_i, v) ← rel ; c(v, v_i) ← rel^{-1};
12       if consistent(Q) then
13         E_k ← E_k ∪ {w_i} ; V' ← V' ∪ {v_i};
14       else
15         for v ∈ V \ (V' ∪ {v_i}) do
16           c(v_i, v) ← B_AP ; c(v, v_i) ← B_AP;
17     if E_k = ∅ then
18       return Inconsistent;
19     k ← k + 1;
20   if kind = leastSpecific then
21     return (E_0, ..., E_{k-1}) ;
22   else
23     return (E_{k-1}, ..., E_0) ;
```

Considérons la fonction specificModel prenant comme paramètres un ensemble d'alternatives Ω, un ensemble de préférences $\mathcal{P}$ et le type de l'unique modèle que nous souhaitons. Grossièrement, en considérant $\mathcal{Q} = \mathsf{PCQ}_{\mathsf{POS}}(\Omega, \mathcal{P})$, cette fonction construit la partition de Ω correspondant à ce modèle. Pour le modèle le moins (resp. le plus) spécifique, les strates de plus bas (resp. de plus haut) rangs sont considérées en premier. Pour une alternative ω_i, son appartenance à une strate donnée est décidée par un test de cohérence d'un sous-$\mathsf{PCQ}_{\mathsf{POS}}$ de $\mathcal{P}$ contraignant la position de ω_i dans la strate. Cela est réalisé en ajoutant des contraintes de précédence sur v_i définies par $\{<, =\}$ (resp. $\{>, =\}$). Nous pouvons formellement prouver la propriété suivante :

Proposition 3 *Soit Ω un ensemble d'alternatives. Soit $\mathcal{P}$ un ensemble de préférences interprétées suivant une sémantique donnée $*$. L'appel de la fonction* specificModel *avec $\Omega, \mathcal{P}$ et* leastSpecific *(resp.* mostSpecific*) comme paramètres retourne pour la sémantique $* \in \{st, opt\}$ (resp. $* \in \{st, pes\}$) le moins spécifique modèle (resp. le plus spécifique modèle) dans le cas où il existe.*

Habituellement, pour des prises de décision, il est plus commode de considérer un modèle unique. Néanmoins, du fait qu'un tel modèle est un préordre complet sur Ω, nous ne pouvons pas distinguer les situations d'égalité et les situations d'incomparabilité entre deux alternatives appartenant à la même partition (*i.e.* deux alternatives $\omega, \omega' \in \Omega$ égales par rapport au moins/plus spécifique modèle. Un moyen de remédier à cela est de voir le moins/plus spécifique modèle comme un ensemble de modèles particuliers et de le considérer comme un ensemble de contraintes complémentaires. Cela nous conduit à considérer une nouvelle traduction d'un ensemble de préférences en $\mathsf{PCQ}_{\mathsf{POS}}$ prenant en compte le modèle le moins/le plus spécifique. Nous la dénoterons par $\mathsf{PCQ}'_{\mathsf{POS}}$. Soit $\mathcal{P}$ un ensemble de préférences et une séquence $\succeq = (E_0, \ldots, E_n)$ correspondant au modèle le moins / le plus spécifique de $\mathcal{P}$. $\mathsf{PCQ}'_{\mathsf{POS}}(\Omega, \mathcal{P}, \succeq)$ est défini par le $\mathsf{PCQ}_{\mathsf{POS}}$ $(\mathcal{V}, c, \mathcal{C})$ où $\mathcal{V}$ et $\mathcal{C}$ sont respectivement l'ensemble de variables et l'ensemble de min/max contraintes de $\mathsf{PCQ}_{\mathsf{POS}}(\Omega, \mathcal{P})$ et c est défini pour chaque $v_i, v_j \in \mathcal{V}$ par :

- pour la cas $i = j$, $c(v_i, v_i)$ est la relation $\{=\}$;
- pour le cas $i \neq j$ et $v_i, v_j \in E_k$ avec $k \in \{0, \ldots, n\}$, $c(v_i, v_j)$ est la relation $\{=, \|\}$;
- pour le cas $i \neq j$, $v_i \in E_k$, $v_j \in E_l$ avec $k, l \in \{0, \ldots, n\}$ et $k < l$ (resp. $l > k$), $c(v_i, v_j)$ is $\{>\}$ (resp. $\{<\}$).

La considération de $\mathcal{Q} = \mathsf{PCQ}'_{\mathsf{POS}}(\Omega, \mathcal{P}, \succeq)$ nous permet par exemple de déterminer les alternatives mises ensemble dans une même strate dans $\succeq$ du fait qu'elles sont autant préférées à cause d'une égale préférence.

Exemple 3 *Soit $\mathcal{Q}$ le $\mathsf{PCQ}_{\mathsf{POS}}$ défini par $\mathcal{Q} = \mathsf{PCQ}'_{\mathsf{POS}}(\Omega, \mathcal{P}, \succeq)$ avec Ω, $\mathcal{P}$ et $\succeq$ correspondant respectivement à l'ensemble d'alternatives, l'ensemble de*

$c(v, v')$	v_7	v_4	v_6	v_2	v_3	v_5	v_0	v_1
v_7	$\{=\}$	$\{>\}$	$\{>\}$	$\{>\}$	$\{>\}$	$\{>\}$	$\{>\}$	$\{>\}$
v_4	$\{<\}$	$\{=\}$	$\{=, \|\|\}$	$\{>\}$	$\{>\}$	$\{>\}$	$\{>\}$	$\{>\}$
v_6	$\{<\}$	$\{=, \|\|\}$	$\{=\}$	$\{>\}$	$\{>\}$	$\{>\}$	$\{>\}$	$\{>\}$
v_2	$\{<\}$	$\{<\}$	$\{<\}$	$\{=\}$	$\{=\}$	$\{=, \|\|\}$	$\{>\}$	$\{>\}$
v_3	$\{<\}$	$\{<\}$	$\{<\}$	$\{=\}$	$\{=\}$	$\{=, \|\|\}$	$\{>\}$	$\{>\}$
v_5	$\{<\}$	$\{<\}$	$\{<\}$	$\{=, \|\|\}$	$\{=, \|\|\}$	$\{=\}$	$\{>\}$	$\{>\}$
v_0	$\{<\}$	$\{<\}$	$\{<\}$	$\{<\}$	$\{<\}$	$\{<\}$	$\{=\}$	$\{=, \|\|\}$
v_1	$\{<\}$	$\{<\}$	$\{<\}$	$\{<\}$	$\{<\}$	$\{<\}$	$\{=, \|\|\}$	$\{=\}$

TABLE 1 – La fonction c.

préférences et le modèle le moins spécifique donné dans l'exemple 1. La résolution du problème de l'étiquetage minimal de $\mathcal{Q}$ fournit le $\mathsf{PCQ}_{\mathsf{POS}}$ minimal $\mathcal{Q}_{min} = (\mathcal{V}, c, \mathcal{C})$ avec la fonction c représenté par la table 1. La relation $c(v, v')$ est donnée par l'entrée dont la ligne correspond à v et la colonne correspond à v'. En examinant ces relations nous pouvons par exemple remarquer que pour les modèles $\mathcal{P}$ les alternatives ω_2 et ω_3 ne peuvent pas être incomparables ($c(v_2, v_3) = \{=\}$).

7 Conclusion et perspectives

Les préférences sont exprimées en logique conditionnelle avec des préférences comparatives strictes ou non strictes. Elles peuvent être interprétées suivant différentes sémantiques. Des principes de spécificité sont utilisés pour caractériser un modèle unique étant donnés un ensemble de préférences et une sémantique. La principale limitation de ces logiques réside dans les requêtes de préférences. Plus précisément, deux alternatives autant préférées par rapport à ce modèle unique ne peuvent pas être distinguées. Nous ne savons pas si elles sont autant préférées à cause de préférences non strictes ou d'incomparabilité. Cette distinction pourrait être importante dans des systèmes de recommandations par exemple. Dans cet article, nous résolvons ce problème par encodage des logiques de préférences et leurs algorithmes associés dans le cadre des problèmes de satisfaction de contraintes qualitatives.

Une analyse de notre encodage nous indique que les $\mathsf{min/max}$ contraintes résultant de la traduction d'un ensemble de préférences en un $\mathsf{PCQ}_{\mathsf{POS}}$ sont définies par des ensembles singletons de $\mathsf{B}_{\mathsf{min/max}}$. Ceci provient du fait que les préférences utilisent un unique opérateur de comparaison ($\rhd$ or $\trianglerighteq$). Étant donné le pouvoir expressif des $\mathsf{PCQ}_{\mathsf{POS}}$, nous pouvons étendre les logiques conditionnelles et considérer des préférences exprimant par exemple un choix entre $\rhd$ ou $\trianglerighteq$ (i.e., $\alpha \rhd \beta$ ou $\beta \rhd \alpha$). Ce type de préférence peut être exprimé à l'aide de relations non singletons de $\mathsf{B}_{\mathsf{min/max}}$. Un travail futur consiste en la définition et l'étude d'algorithme pour résoudre efficacement un $\mathsf{PCQ}_{\mathsf{POS}}$.

Références

[1] F. D. Anger, D. Mitra, and R. V. Rodríguez. Satisfiability in nonlinear time : Algorithms and complexity. In *Proceedings of the Twelfth International Florida Artificial Intelligence Research Society Conference (FLAIRS'99)*, pages 406–411, 1999.

[2] S. Benferhat, D. Dubois, S. Kaci, and H. Prade. Bipolar representation and fusion of preferences in the possibilistic logic framework. In *KR'02*, pages 421–432, 2002.

[3] S. Benferhat and S. Kaci. A possibilistic logic handling of strong preferences. In *International Fuzzy Systems Association (IFSA'01)*, pages 962–967, 2001.

[4] C. Boutilier. Toward a logic for qualitative decision theory. In *KR'94*, pages 75–86, 1994.

[5] M. Broxvall and P. Jonsson. Point algebras for temporal reasoning : Algorithms and complexity. *Artificial Intelligence.*, 149(2) :179–220, 2003.

[6] S. Kaci and L. van der Torre. Reasoning with various kinds of preferences : Logic, nonmonotonicity and algorithms. *Annals of Operations Research*, 163(1) :89–114, 2008.

[7] G. Ligozat and J. Renz. What is a qualitative calculus ? a general framework. In Chengqi Zhang, Hans W. Guesgen, and Wai-Kiang Yeap, editors, *Proceedings of 8th Pacific Rim International Conference on Artificial Intelligence (PRICAI'04), Auckland, New Zealand*, volume 3157 of *Lecture Notes in Computer Science*. Springer, 2004.

[8] J. Pearl. System Z : A natural ordering of defaults with tractable applications to default reasoning. In *TARK'90*, pages 121–135, 1990.

[9] L. van der Torre and E. Weydert. Parameters for utilitarian desires in a qualitative decision theory. *Applied Int*., 14(3) :285–301, 2001.

[10] G. H. von Wright. *The Logic of Preference*. University of Edinburgh Press, 1963.

[11] N. Wilson. Extending CP-nets with stronger conditional preference statements. In *AAAI'04*, pages 735–741, 2004.

[12] R.R. Yager. Entropy and specificity in a mathematical theory of evidence. *Int. J. of General Systems*, 9 :249–260, 1983.

Un modèle général pour la classification non supervisée sous contraintes d'utilisateur

Thi-Bich-Hanh Dao, Khanh-Chuong Duong, Christel Vrain

LIFO, Université d'Orléans, 45067 Orléans cedex 02, 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 et depuis le début des années 2000, on tend à intégrer dans ce processus des contraintes afin de mieux modéliser les attentes des utilisateurs. Cependant, les systèmes développés sont en général adaptés à certains types de contraintes seulement. Nous proposons un modèle général pour la classification non supervisée sous contraintes en Programmation par Contraintes. Il permet d'intégrer des contraintes, qu'elles portent sur des instances ou sur les groupes recherchés. Différents critères d'optimisations sont intégrés au modèle : la minimisation du diamètre maximal, la maximisation de la marge entre les clusters ou la minimisation de la somme des dissimilarités intracluster. Des expériences sur des ensembles de données classiques montrent l'intérêt de notre approche.

Abstract

Cluster analysis is an important task in Data Mining and since the early 2000s user-defined constraints have been integrated in order to better model real-world applications. Well-known algorithms are extended to handle user-constraints, however usually for one kind of constraints. We present a general model for constrained clustering in Constraint Programming, which allows to integrate directly different kinds of user-constraints. Different criteria are included: minimizing the maximal diameter, maximizing the separation between clusters or minimizing the within-cluster sum of dissimilarities. Experiments on classical datasets show the interest of our approach.

1 Introduction

La problématique de classification non supervisée (aussi appelée clustering) a déjà été longuement étudiée. C'est un champ de recherche difficile pour plusieurs raisons : le choix de la mesure de dissimilarité entre les objets dépendant principalement de l'application mais influant fortement sur les résultats, la définition du critère à optimiser, la taille de l'espace de recherche avec pour conséquence la nécessité de définir des heuristiques conduisant souvent à un optimum local. Poser des contraintes sur la solution recherchée permet d'une part, de modéliser plus finement les applications réelles et d'autre part de restreindre la taille de l'espace de recherche. Néanmoins, la plupart des algorithmes classiques n'ont pas été développés pour la classification non supervisée sous contraintes et doivent être adaptés, si possible, pour prendre en compte les contraintes posées par l'utilisateur. Développer des solveurs généraux applicables à une grande variété de problèmes pose de nouveaux défis.

D'autre part, des avancées récentes en Programmation par Contraintes (PPC) ont rendu ce paradigme beaucoup plus puissant. Plusieurs travaux [10, 9, 4] ont étudié l'intérêt de la PPC pour modéliser des problèmes de fouille de données et ont montré l'apport de la déclarativité inhérente à la PPC.

Dans des travaux récents [5], nous avons proposé une approche en PPC pour la classification non supervisée sous contraintes d'utilisateur avec comme critère la minimisation du diamètre maximal des classes. L'intérêt de notre approche est de fournir un modèle déclaratif permettant de spécifier le problème de classification non supervisée et d'intégrer facilement des contraintes d'utilisateur. Dans ce papier, nous généralisons le modèle avec plusieurs critères d'optimisation : minimiser le diamètre maximal, maximiser la marge entre les classes ou minimiser la somme des dissimilarités intraclasse. Dans notre modèle, nous faisons l'hypothèse que nous disposons d'une mesure de dissimilarité entre les paires d'objets et que le nombre k de classes est fixé (en théorie aucune limite n'est donnée sur la valeur de k, mais plus k est grand, plus la complexité est élevée).

Nous montrons que notre cadre intègre naturellement des contraintes d'utilisateur connues, par exemple des contraintes sur les instances (must-link, cannot-link) ou des contraintes sur les classes (capacité maximale ou minimale, séparation, diamètre). Nous avons également généralisé une contrainte sur les classes, permettant de modéliser une notion de densité.

Il est reconnu en PPC que le choix de variables et de contraintes est fondamental, mais que la stratégie de recherche est tout aussi importante pour améliorer l'efficacité. Des améliorations du modèle sont étudiées, reposant sur des résultats théoriques ce qui permet d'alléger le modèle sans changer la sémantique. De plus, parmi les critères intégrés, celui de la somme des dissimilarités intraclasse se représente par une somme linéaire. Afin d'avoir une propagation plus efficace pour ce critère, nous avons implanté un algorithme de filtrage qui exploite des connaissances lors d'une affectation partielle de variables. Des expérimentations sur des bases de données classiques montrent que notre approche est capable de traiter ces bases en un temps raisonnable et surtout, que la possibilité de combiner différents types de contraintes d'utilisateur permet d'obtenir des solutions intéressantes. Contrairement à la plupart des travaux existants dans la classification non supervisée sous contraintes d'utilisateur, notre modèle permet de trouver un optimum global.

Des travaux récents [14, 16] ont déjà proposé d'utiliser la PPC pour la classification non supervisée conceptuelle dans des bases de données transactionnelles. Le problème est formalisé comme la recherche d'un ensemble de k-motifs fréquents, deux à deux non recouvrants, dont l'ensemble couvre toutes les données : les transactions sont vues comme des objets et les motifs comme des définitions en intension des classes. Plusieurs critères d'optimisation sont considérés comme maximiser la taille minimale d'une classe ou minimiser la différence entre les tailles des classes. Une approche de modélisation par programmation linéaire sur des entiers a été proposée dans [17]. Dans cette approche un ensemble de clusters candidat doit être connu à l'avance, et le modèle permet de trouver un clustering formé par le meilleur sous-ensemble parmi les candidats. Cette approche est expérimentée pour la classification non supervisée conceptuelle, les clusters candidat sont des motifs fréquents. Notons que ces approches sont donc adaptées à des bases de données qualitatives alors que notre approche peut traiter tout type de données dès lors que l'on dispose d'une mesure de dissimilarité entre les données. Un cadre SAT pour la classification non supervisée sous contraintes est proposé en [8], mais uniquement pour un problème à 2 classes ($k = 2$) : il traite les contraintes sur les instances (must-link et cannot-link) et des contraintes sur les classes (diamètre des clusters, séparation entre les clusters). Son algorithme converge vers un optimum global. Notre approche est plus générale dans la mesure où différents autres critères sont traités et le nombre de classes n'est pas limité à 2.

Le papier est organisé comme suit. Dans la section 2, nous rappelons des notions sur la classification non supervisée et celle sous contraintes d'utilisateur. La section 3 est dédiée à la présentation de notre modèle et la section 4 aux expérimentations. La conclusion et une discussion sur les travaux futurs sont données dans la section 5.

2 Préliminaires

2.1 Classification non supervisée

La classification non supervisée (ou clustering) consiste à regrouper les données dans des classes (ou clusters), de manière à regrouper dans un même cluster les données similaires et à séparer les données distantes dans des clusters différents. Etant donnés une base de données de n points (objets) $\mathcal{O} = \{o_1, \ldots, o_n\}$ d'un espace $\mathcal{X}$ et une mesure de dissimilarité $d(o_i, o_j)$ entre deux points o_i et o_j, l'objectif est de regrouper les points en différentes classes de telle manière que la partition obtenue optimise un critère donné. Le problème de clustering peut être donc formulé comme un problème d'optimisation.

Les problèmes de clustering sont variés, dépendant de différents critères, comme la structure souhaitée (une partition, une hiérarchie, des classes recouvrantes, ...), ou encore le critère à optimiser. Dans ce papier, nous nous intéressons à la recherche d'une partition des points en k classes $C_1, \ldots, C_k$ telle que : (1) pour tout $c \in [1, k]$, $C_c \neq \emptyset$, (2) $\cup_c C_c = \mathcal{O}$, (3) pour tout $c \neq c'$, $C_c \cap C_{c'} = \emptyset$, et (4) un critère E est optimisé. Le critère optimisé peut être, entre autres :

- Minimisation des moindres carrés : $E = \sum_{c=1}^{k} \sum_{o_i \in C_c} d(m_c, o_i)^2$, où m_c est le centre de chaque cluster C_c.
- Minimisation de la somme de dissimilarités intra-cluster : $E = \sum_{c=1}^{k} \sum_{o_i, o_j \in C_c} d(o_i, o_j)^2$. Ce critère, standardisé en divisant par la taille de chaque cluster, est équivalent au critère des moindres carrés dans le cas de la distance euclidienne.
- Minimisation de l'erreur absolue : $E = \sum_{c=1}^{k} \sum_{o_i \in C_c} d(o_i, r_c)$, où r_c est le représentant du cluster C_c.
- Minimisation du diamètre maximal : $E = max_{c \in [1,k], o_i, o_j \in C_c}(d(o_i, o_j))$. E est le diamètre maximal des clusters, où le diamètre d'un cluster est la distance maximale entre chaque paire

de ses points.
– Maximisation du séparateur minimal : $E = min_{c<c' \in [1,k], o_i \in C_c, o_j \in C_{c'}}(d(o_i, o_j))$. E est le séparateur minimal des clusters, où le séparateur de deux clusters $C_c, C_{c'}$ est la valeur minimale des distances $d(o_i, o_j)$, avec $o_i \in C_c$ et $o_j \in C_{c'}$.

L'algorithme k-means représente chaque cluster par la moyenne des points du cluster et tend à minimiser le critère des moindres carrés, alors que l'algorithme k-médoïdes choisit parmi les points un représentant de chaque cluster et cherche à minimiser le critère d'erreur absolue. A chaque itération, les deux méthodes réduisent la valeur du critère jusqu'à un optimum, en général local.

L'algorithme FPF (Furthest Point First) [13] minimise le diamètre maximal des clusters. L'algorithme commence par choisir un point comme le représentant du premier cluster et affecte tous les points à ce cluster. A l'itération suivante, le point le plus loin du premier représentant est choisi comme représentant du second cluster. Les points qui sont plus proches du second représentant que du premier sont réaffectés au second cluster. L'algorithme réitère les 2 étapes : choisir comme nouveau représentant le point le plus loin des représentants existants et réaffecter les points. Il s'arrête après k itérations, ayant ainsi formé k clusters. La complexité en temps est $O(nk)$. Si d_{opt} est l'optimum global alors l'algorithme garantit de trouver un clustering dont le diamètre maximal des clusters d vérifie $d_{opt} \leq d \leq 2d_{opt}$, si la mesure utilisée satisfait l'inégalité triangulaire. De plus, dans [13] il a été prouvé que trouver d tel que $d \leq (2 - \epsilon)d_{opt}$ est NP-Difficile pour tout $\epsilon > 0$, lorsque les points sont dans un espace à 3 dimensions. L'algorithme modifié (M-FPF) proposé dans [12] trouve une solution identique à FPF tout en étant plus rapide en temps.

La plupart des méthodes de partitionnement trouvent des groupes sous forme sphérique. Des approches, comme par exemple DBSCAN [11] basée sur la notion de densité et ne reposant sur aucun critère d'optimisation, permettent de découvrir des groupes de forme arbitraire. DBSCAN nécessite deux paramètres d'entrée : une distance ϵ et un nombre minimum de points $MinPts$. Un point est un noyau si son voisinage de rayon ϵ contient au moins $MinPts$ points. La méthode considère un à un tous les points non affectés à un cluster, si le point est un noyau, un nouveau cluster est créé et les points dans son voisinage y sont ajoutés. L'algorithme étend ensuite le cluster de façon récursive en y ajoutant tout le voisinage des points noyaux du cluster. Cette méthode se base donc seulement sur des paramètres de densité ϵ et $MinPts$, il n'y a pas de critère à optimiser et le nombre de classes k n'est pas fixé à l'avance. Dans OPTICS [1], un algorithme qui peut être vu comme une généralisation de DBSCAN, la valeur de ϵ est remplacée par une borne maximale. Cet algorithme permet d'identifier ϵ lorsque le nombre k est fixé.

2.2 Classification non supervisée sous contraintes d'utilisateurs

Les contraintes définies par l'utilisateur sur la solution recherchée permettent de modéliser plus finement des applications réelles. Ce sont des contraintes portant sur des instances (des points) ou sur des clusters.

Les contraintes portant sur des instances les plus utilisées sont les contraintes "must-link" et "cannot-link", introduites pour la première fois dans [21] . Ces contraintes sont posées sur des paires d'objets individuels. Une contrainte must-link indique que deux points o_i et o_j doivent être dans le même cluster :

$$\forall c \in [1, k],\ o_i \in C_c \Leftrightarrow o_j \in C_c$$

Une contrainte cannot-link indique que deux points ne peuvent pas appartenir au même cluster :

$$\forall c \in [1, k],\ \neg(o_i \in C_c \wedge o_j \in C_c)$$

Les contraintes portant sur des clusters imposent des conditions sur la forme, la taille ou d'autres caractéristiques. Une contrainte sur la taille peut être une contrainte de capacité minimale, qui indique que chaque cluster doit avoir au moins α points, permettant ainsi d'éviter des clusters trop petits :

$$\forall c \in [1, k],\ |C_c| \geq \alpha,$$

ou encore une contrainte de capacité maximale, qui indique que chaque cluster doit avoir au plus β points :

$$\forall c \in [1, k],\ |C_c| \leq \beta.$$

Comme contrainte sur la forme, l'utilisateur peut limiter le diamètre des clusters :

$$\forall c \in [1, k], \forall o_i, o_j \in C_c,\ d(o_i, o_j) \leq \gamma.$$

Il existe d'autres types de contraintes sur des clusters comme par exemple la δ-contrainte et la ϵ-contrainte [7]. Une δ-contrainte ou contrainte de séparation impose que chaque paire de points appartenant à deux groupes différents soit à une distance d'au moins δ :

$$\forall c < c' \in [1, k], \forall o_i \in C_c, o_j \in C_{c'}, d(o_i, o_j) \geq \delta.$$

Une ϵ-contrainte impose que chaque point o_i ait dans un rayon ϵ un autre point du même cluster :

$$\forall c \in [1, k], \forall o_i \in C_c, \exists o_j \in C_c, o_j \neq o_i \ et\ d(o_i, o_j) \leq \epsilon.$$

Cette contrainte est basée sur la notion de densité du DBSCAN. Dans la même idée que DBSCAN, nous proposons une nouvelle contrainte de densité plus générale que ϵ-contrainte, qui impose que chaque point o_i doit avoir dans un rayon de ϵ au moins $MinPts$ points qui sont dans le même cluster que o_i.

Depuis une dizaine d'années, différents travaux ont développé des extensions des algorithmes classiques aux contraintes must-link et cannot-link, comme par exemple COBWEB [21], k-means [22, 3], classification non supervisée hierarchique [6] ou spectrale [15, 23], etc. L'extension est faite en modifiant la mesure de dissimilarité, la fonction objectif ou la stratégie de recherche. Cependant à notre connaissance il n'existe pas de solution générale pour étendre un algorithme traditionnel en y intégrant simultanément différents types de contraintes.

3 Modélisation des problèmes de clustering sous contraintes

Nous disposons d'une collection de n points et d'une mesure de dissimilarité entre ces points. Sans perte de généralité nous supposons que les points sont indexés et nommés par leur indice. Nous considérons le cas où le nombre de clusters k est connu à l'avance.

3.1 Modèle

Variables Nous fixons pour chaque cluster un représentant, qui est le point de plus petit indice de ce cluster. Nous introduisons donc une variable entière $I[c]$ pour chaque $c \in [1, k]$, dont la valeur est le représentant du cluster c. Le domaine de chaque $I[c]$ est $[1, n]$, car tout point peut être représentant d'un cluster.

Affecter un point à un cluster devient associer ce point au représentant du cluster. Nous introduisons pour chaque point $i \in [1, n]$ une variable entière $G[i]$ du domain $[1, n]$ qui donne le représentant associé.

Nous introduisons également une variable qui représente la valeur du critère à optimiser (D s'il s'agit du diamètre maximal, S la séparation minimale, V la somme des dissimilarités intra-cluster). C'est une variable entière, car nous utilisons des solveurs sur des variables entières. Le domaine de D et de S est l'intervalle formé par la distance minimale et la distance maximale entre chaque paire de points. Le domaine de V est borné supérieurement par la somme des distances au carré de toutes les paires de variables. Dans nos expérimentations, lorsque la distance n'est pas entière, elle est multipliée par 100 et seule la partie entière est conservée.

Modélisation d'une partition La relation entre les points et leur cluster est exprimée par ces contraintes :

- Le représentant d'un représentant est lui-même :

$$\forall c \in [1, k],\ G[I[c]] = I[c].$$

- Le représentant d'un point doit être parmi les représentants : la valeur de chaque $G[i]$ doit être présente une fois dans I

$$\forall i \in [1, n],\quad \#\{c \mid I[c]{=}G[i]\} = 1.$$

- Le représentant doit être d'indice minimal, l'indice de chaque point est donc supérieur ou égale à celui du représentant : $\forall i \in [1, n],\quad G[i] \leq i$.

Un même ensemble de clusters pourrait être représenté de différentes manières, selon l'ordre des clusters. Les contraintes suivantes permettent d'éviter cette symétrie :

- Les représentants sont en ordre croissant : $\forall c < c' \in [1, k],\ I[c] < I[c']$.
- Le représentant du premier cluster est le premier point : $I[1] = 1$.

Modélisation de différents critères d'objectif S'il s'agit de minimiser le diamètre maximal des clusters, les contraintes suivantes sont ajoutées :

- Deux points à une distance supérieure au diamètre maximal doivent être dans des clusters différents. Ceci est représenté par les contraintes réifiées suivantes : $\forall i < j \in [1, n]$,

$$d(i, j) > D \rightarrow G[j] \neq G[i] \tag{1}$$

- Le diamètre maximal est minimisé : minimise D.

S'il s'agit de maximiser la séparation minimale entre clusters :

- Deux points à une distance inférieure à la séparation minimale doivent être dans le même cluster. Ceci est représenté par les contraintes réifiées suivantes : $\forall i < j \in [1, n]$,

$$d(i, j) < S \rightarrow G[j] = G[i] \tag{2}$$

- La séparation est maximisée : maximise S.

Si le critère est de minimiser la somme des dissimilarités, minimise V avec :

$$V = \sum_{i,j \in [1,n]} (G[i] == G[j]) d(i, j)^2 \tag{3}$$

Modélisation des contraintes définies par l'utilisateur Plusieurs contraintes définies par l'utilisateur, sur des clusters ou sur des points, peuvent s'ajouter directement au modèle. Pour les contraintes portant sur les clusters :

– La capacité minimale α des clusters : $\forall c \in [1,k]$,

$$\#\{i \mid G[i]=I[c]\} \geq \alpha$$

– La capacité maximale β des clusters : $\forall c \in [1,k]$,

$$\#\{i \mid G[i]=I[c]\} \leq \beta$$

– La δ-contrainte indique que la séparation entre deux clusters doit être au moins δ, ainsi deux points à une distance inférieure à δ doivent être dans le même cluster. Pour chaque $i < j \in [1,n]$ tels que $d(i,j) < \delta$, nous posons la contrainte : $G[i] = G[j]$.
– La contrainte de diamètre indique que le diamètre de chaque cluster doit être au plus γ, ainsi deux points à une distance supérieure à γ doivent être dans des clusters différents. Pour chaque $i < j \in [1,n]$ tels que $d(i,j) > \gamma$, nous posons : $G[j] \neq G[i]$.
– La contrainte de densité indique que chaque point i doit avoir dans un rayon de ϵ, au moins $MinPts$ points dans le même cluster que i. Donc, pour chaque $i \in [1,n]$, l'ensemble de points qui sont dans un rayon de ϵ est calculé, sur lequel la contrainte suivante est posée :

$$\#\{j \mid d(i,j) \leq \epsilon, G[j]=G[i]\} \geq MinPts$$

Pour les contraintes portant sur les couples de points :
– Une contrainte must-link sur i,j se traduit par $G[i] = G[j]$.
– Une contrainte cannot-link : $G[i] \neq G[j]$.

L'ajout de ces contraintes implique également des contraintes sur D ou S, par exemple $G[i] = G[j]$ implique $D \geq d(i,j)$ et $G[i] \neq G[j]$ implique $S \leq d(i,j)$.

Stratégie de recherche Les variables sont choisies dans l'ordre des variables de I puis celles de G. Cet ordre indique que les représentants de cluster doivent être identifiés en premier, puis le solveur essaye de déterminer l'affectation de points aux clusters.

Les variables de I sont choisies de $I[1]$ à $I[k]$, c'est-à-dire du premier au dernier représentant. Puisqu'un représentant doit avoir l'indice minimal dans le cluster, les valeurs pour chaque $I[c]$ sont choisies dans l'ordre croissant.

Les variables de G sont choisies dans l'ordre croissant sur la taille du domaine restant. Pour le choix de valeur pour chaque $G[i]$, l'indice du représentant le plus proche est choisi en premier.

3.2 Améliorations du modèle

Les solveurs de PPC réalisant une recherche exhaustive, ce modèle permet de trouver la solution optimale. Afin d'améliorer l'efficacité du modèle, différents aspects sont considérés.

Amélioration de la recherche en réordonnant les points Afin d'identifier les représentants, les variables de I sont choisies dans l'ordre $I[1], \ldots, I[k]$. Puisqu'un représentant doit avoir l'indice minimal, pour chaque variable, les valeurs (indices possibles) sont choisies dans l'ordre croissant. L'indice des points a donc une grande importance. Par conséquent, les points sont réordonnés de façon à ce que ceux qui sont plus probables d'être représentants aient un indice plus faible. Nous utilisons l'heuristique FPF pour réordonner les points. On applique l'algorithme FPF en prenant $k = n$ (autant de classes que de points). Chaque point est donc choisi par l'algorithme FPF selon l'heuristique décrite en section 2, l'ordre de choix dans FPF donne l'ordre des points. Une autre heuristique utilisée est présentée dans [5].

Amélioration du modèle pour le critère du diamètre maximal Dans le cas sans contraintes définies par l'utilisateur, avec un k fixé, il est prouvé dans [13] que le diamètre d_{FPF} calculé par l'algorithme FPF vérifie $d_{opt} \leq d_{FPF} \leq 2d_{opt}$, avec d_{opt} le diamètre optimal. Cette connaissance implique des bornes pour la variable D, à savoir $[d_{FPF}/2, d_{FPF}]$. De plus, pour chaque couple de points i,j :
– si $d(i,j) < d_{FPF}/2$, la contrainte réifiée (1) sur i,j ne sera pas posée,
– si $d(i,j) > d_{FPF}$, la contrainte (1) est remplacée par une contrainte cannot-link $G[j] \neq G[i]$.

Ce résultat permet de supprimer plusieurs contraintes réifiées sans modifier la sémantique du modèle. Puisque les contraintes réifiées nécessitent la gestion de variables supplémentaires, leur suppression permet d'améliorer l'efficacité du modèle.

Dans le cas où s'ajoutent des contraintes d'utilisateur, le diamètre optimal est en général supérieur à d_{opt}. La borne supérieure du domaine de D n'est plus valable, par contre nous avons toujours la borne inférieure $d_{FPF}/2$. Ainsi, pour chaque couple i,j, si $d(i,j) < d_{FPF}/2$, la contrainte (1) ne sera pas posée.

Algorithme de filtrage pour la somme des dissimilarités La contrainte (3) peut se réaliser par une contrainte linéaire sur des variables booléennes $V = \sum_{i<j} B_{ij} d(i,j)^2$, avec $B_{ij} = (G[i] == G[j])$. Cependant ces contraintes, considérées indépendamment, n'offrent pas suffisamment de propagation. Par exemple, prenons 4 points de 1 à 4, avec $k = 2$ et une affectation partielle où $G[1]=1$ et $G[2]=G[3]=2$. Nous avons trois variables booléennes B_{14}, B_{24}, B_{34} où $B_{i4} = (G[i] == G[4])$, avec $i \in [1,3]$. Supposons que pendant la recherche par *branch-and-bound*, la borne supérieure de V est mise à 4, donc avec la minimisation de V la contrainte $B_{14} + 2B_{24} + 3B_{34} < 4$ s'est

ajoutée. On constate que $B_{24} = B_{34}$ car $G[2] = G[3]$ et ne peuvent donc pas valoir 1, sinon la contrainte serait violée. On devrait conclure que le point 4 ne peut pas être dans le même cluster que les points 2 et 3, et supprimer ainsi la valeur 2 du domaine de $G[4]$. Ce filtrage n'est malheureusement pas fait pendant la propagation de la contrainte de somme. C'est le cas où des variables qui interviennent dans la contrainte de somme apparaissent également dans d'autres contraintes.

En PPC, des filtrages plus efficaces de la contrainte de somme ont été proposés, lorsqu'elle est considérée avec d'autres contraintes. Dans le cas d'une contrainte somme $y = \sum x_i$ avec des contraintes d'inégalité $x_j - x_i \leq c$, un algorithme peut réduire le domaine des x_i lorsque les bornes de y sont réduites [20]. Des algorithmes de réduction de bornes du domaine de variables sont proposés pour le cas d'une contrainte somme avec des contraintes d'ordre $x_i \leq x_{i+1}$ [18] ou avec une contrainte *alldifferent*$(x_1, \ldots, x_n)$ [2]. Cependant la contrainte (3) n'en fait pas partie. Un algorithme générique de réduction de borne qui exploite une contrainte somme et un ensemble de contraintes est proposé dans [19]. Dans notre cas, le domaine restant d'une variable $G[i]$ est constitué de valeurs qui sont des indices de représentants et n'est donc pas en général un intervalle continu. Nous avons donc implanté un algorithme de filtrage qui tient compte des connaissances de l'affectation partielle.

Supposons que l'ensemble des points soit séparé en deux ensembles disjoints K (les points i dont $G[i]$ est instanciée) et U (les points i dont $G[i]$ est non instanciée). Supposons que la contrainte (3) par le *branch-and-bound* devienne

$$\sum_{i \in K, j \in U} (G[i] == G[j]) a_{ij} + \sum_{i,j \in U} (G[i] == G[j]) a_{ij} < v$$

avec a_{ij} des constantes. L'idée est que pour chaque point inconnu $j \in U$, si l'affectation à un cluster c avec des points connus crée une valeur supérieure ou égale à v, alors la valeur c sera supprimée du domaine de $G[j]$. L'algorithme est comme suit :

pour chaque $j \in U$
  pour chaque *valeur c du domaine de* $G[j]$
    $m \leftarrow \sum_{i \in K, G[i]=c} a_{ij}$
    si $m \geq v$ **alors**
      supprimer c du domaine de $G[j]$

Cet algorithme n'est bien entendu pas complet car il ne tient compte que de la partie de la somme qui relie les points inconnus aux points connus. Il permet cependant d'améliorer l'efficacité du modèle pour pouvoir traiter la somme des dissimilarités avec un plus grand nombre de points.

4 Expérimentations

Des algorithmes classiques de classification non supervisée sont développés pour un critère particulier ou pour des types particuliers de contraintes définis par l'utilisateur. A cause de la complexité de la classification non supervisée, la plupart des algorithmes classiques se contentent d'un optimum local. De plus, la performance de ces méthodes diminue souvent lors de l'ajout de contraintes définies par l'utilisateur. Notre modèle est extensible à différents types de contraintes d'utilisateurs tout en traitant différents critères et garantit de trouver un optimum global.

Nous avons implanté notre modèle en utilisant la bibliothèque de programmation par contraintes Gecode version 3.7.3 [1]. Les expérimentations sont réalisées sur un processeur 2.4GHz Core i5 Intel sous Ubuntu 12.04. Toutes les contraintes sont réalisées directement par des contraintes de la bibliothèque Gecode, comme des contraintes `element`, `count` ou des contraintes réifiées. Nous présentons dans ce qui suit des résultats d'expérimentations sur des bases classiques et l'intérêt de la possibilité de combiner différents types de contraintes.

4.1 Clustering sans contraintes d'utilisateur

Nous avons expérimenté notre modèle en considérant différents schémas de classification, faisant varier le critère d'optimisation et considérant ou non des contraintes d'utilisateur. Nous considérons 5 bases de données du répertoire de données de UCI [2] :

Base de données	#Objets	#Classes
iris	150	3
ionosphere	351	2
synthetic control	600	6
vehicle	846	4
yeast	1484	10

Nous avons expérimenté avec k le nombre de classes connues. Le temps en secondes sur ces bases avec des critères de minimisation du diamètre maximal et de maximisation de la séparation minimale sont donnés dans le tableau suivant :

Base de données	Diamètre	Séparation
iris	0.1s	0.3s
ionosphere	0.8s	7.4s
synthetic control	24.6s	102.8s
vehicle	36.7s	308.6s
yeast	4211.2s	$>$ 2 heures

Le critère de diamètre est plus efficace dans tous les cas. Cela s'explique par notre stratégie de réordonner

1. http ://www.gecode.org
2. http ://archive.ics.uci.edu/ml/

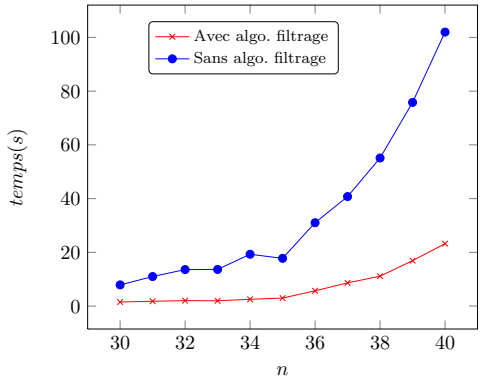


FIGURE 1 – Critère de somme des dissimilarités

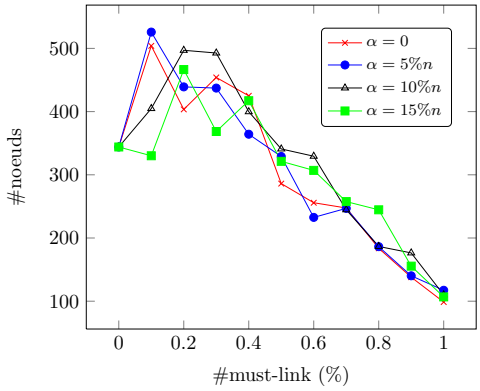


FIGURE 2 – Must-link et contrainte de capacité

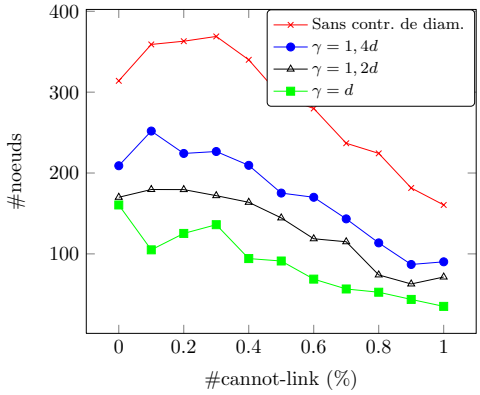


FIGURE 3 – Cannot-link et contrainte de diamètre

des points, appropriée pour ce critère : les k représentants de la solution optimale minimisant le diamètre maximal sont souvent les k premiers points après réordonnancement. Cette remarque n'est plus valable avec le critère de séparation. De plus, nous avons exploité des résultats théoriques pour améliorer le critère de diamètre.

Concernant le critère de minimisation de la somme des dissimilarités intraclasse, la propagation de contraintes est moins efficace. Il faut en général instancier suffisamment de points pour que la contrainte de somme puisse filtrer le domaine des variables restantes. Des expérimentations sont réalisées sur la base `iris`, avec un échantillon composé de 30 à 40 points. Figure 1 présente le temps en secondes nécessaire avec ou sans utilisation de l'algorithme de filtrage que nous avons implanté.

4.2 Clustering avec contraintes d'utilisateur

Concernant la classification non supervisée avec des contraintes d'utilisateur, nous présentons des expérimentations sur la base `iris`. Figure 2 présente le résultat de test où sont pris en compte le critère de minimisation du diamètre maximal, des contraintes must-link et une contrainte de capacité minimale. Le nombre de contraintes must-link varie de 0% à 1% du nombre de paires de points. La capacité minimale α testée est de 0 à 15% du nombre de points. Pour comparer les performances, nous choisissons de présenter le nombre de nœuds de l'arbre de recherche, car le temps est proche de 0 seconde. Nous constatons que des contraintes must-link améliore en général la performance, ce qui n'est pas le cas pour la contrainte de capacité.

Figure 3 présente le résultat de tests où sont pris en compte la maximisation de la séparation minimale, des contraintes cannot-link et une contrainte de diamètre. Le nombre de contraintes must-link varie de 0% à 1% du nombre de paires de points. Le paramètre γ de la contrainte de diamètre est de 1 à 1,4 fois le diamètre maximal d des classes réelles de la base. On constate que la contrainte cannot-link et la contrainte de diamètre augmente la performance.

Dans les tests de la Figure 4,on tient compte du critère de minimisation du diamètre maximal, d'une contrainte de séparation et d'une contrainte de densité. Le paramètre δ de la séparation varie de 0 à 10% de la distance maximale d entre deux points. Pour la contrainte de densité, $MinPts = 4$ et ϵ varie de 20% à 60% de d. On constate que la contrainte de séparation améliore la performance plus que la contrainte de densité. En effet, les états de recherche engendrés ne violent pas de contrainte de densité. En conséquence, la propagation de contrainte de densité réduit peu de nœuds dans l'arbre de recherche.

Nous avons également expérimenté le critère de minimisation de la somme des dissimilarités avec des contraintes must-link et cannot-link, dans les mêmes conditions que les tests de la Figure 1 avec notre

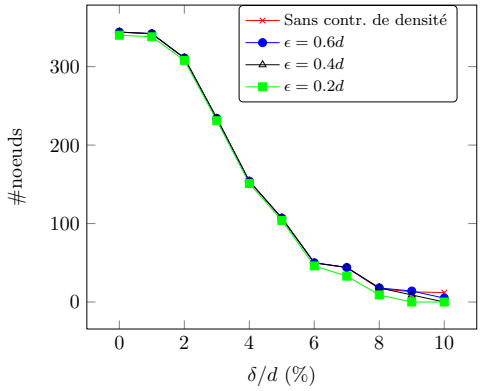


FIGURE 4 – Contraintes de séparation et de densité

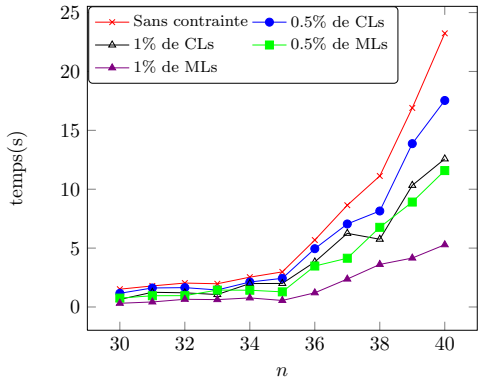


FIGURE 5 – Somme de dissimilarités avec ML et CL

contrainte globale. Le nombre de contraintes must-link ou cannot-link ajoutées varie de 0.5% à 1% du nombre de paires de points. Les résultats sont donnés en Figure 5. On constate que l'ajout des contraintes must-link améliore plus la performance car il a un impact plus direct sur la contrainte globale sur la somme.

4.3 Qualité de la solution

La possibilité de combiner différents types de contraintes permet de détailler plus finement la solution recherchée et en général permet d'améliorer la qualité de la solution trouvée. Nous considérons trois bases de données 2D similaires à celles utilisées dans [11]. Les données sont représentées dans la figure 6.

Dans la première base, on constate quatre groupes de diamètres différents. La deuxième base est plus difficile parce que les groupes ont des formes différentes. Avec le critère de diamètre maximal, le solveur trouve des groupes de diamètres plutôt homogènes, donc ce critère seul ne convient pas très bien, ainsi que l'illustre la Figure 7. L'ajout d'une contrainte de séparation,

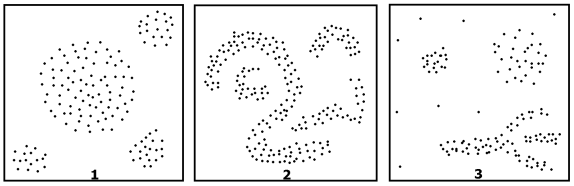


FIGURE 6 – Bases de données

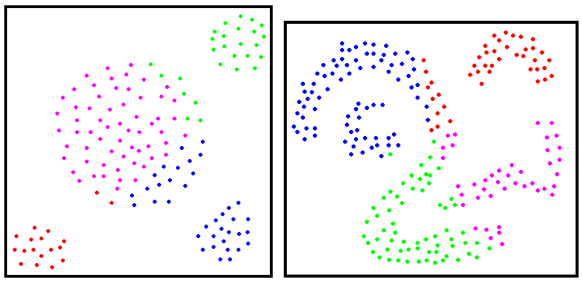


FIGURE 7 – Critère de diamètre

avec le paramètre δ valant 5% de la distance maximale entre paires de points, améliore nettement la qualité de la solution, comme le présente Figure 8. Notons que le critère de maximisation de la séparation minimale permet aussi de trouver cette solution.

La troisième base contient des points séparés (outliers). Notre modèle ne détecte pas encore les outliers, donc ces points sont aussi classés. Le critère de diamètre (Figure 9) ou de séparation (Figure 10) ne permet pas de trouver une solution de bonne qualité. Cependant, la qualité de la solution est nettement améliorée lorsqu'une contrainte de densité est ajoutée, avec $MintPts = 4$ et ϵ valant 25% de la distance maximale entre paires de points (Figure 11).

5 Conclusion

Nous présentons dans ce papier un modèle en PPC pour la classification non supervisée et celle

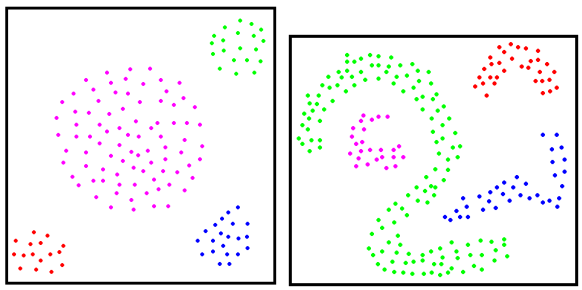


FIGURE 8 – Meilleure solution

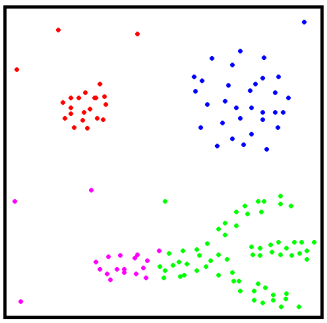


FIGURE 9 – Minimiser le diamètre maximal

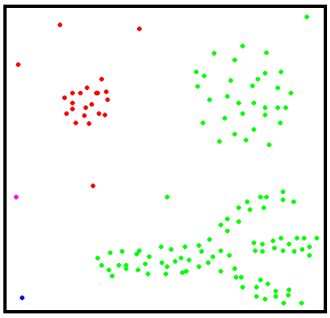


FIGURE 10 – Maximiser la séparation minimale

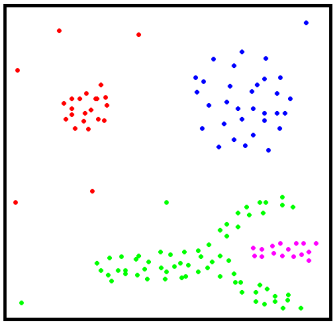


FIGURE 11 – Critère de séparation et contrainte de densité

sous contraintes d'utilisateur. Le modèle est générique dans le sens où il s'adapte à différents critères de classification et les contraintes d'utilisateurs les plus connues sont intégrées directement. Parmi des contraintes d'utilisateurs intégrées, nous généralisons une nouvelle contrainte portant sur la densité des clusters. Le modèle trouve un optimum global qui optimise un critère comme minimiser le diamètre maximal des clusters, maximiser le séparateur minimal entre clusters ou minimiser la somme des dissimilarités intra-classe. Du fait que le modèle se base sur la distance, des données qualitatives ou quantitatives peuvent être considérées. Des expérimentations sur des jeux de données classiques montrent l'intérêt de notre approche.

Le modèle actuel est plus efficace en traitant des critères qui peuvent se traduire par une relation entre deux points, comme le diamètre maximal ou la séparation minimale. Lorsque le critère se représente par une somme reliant un ensemble de points, comme la somme des dissimilarités intraclasse, la performance reste encore à améliorer. L'amélioration de l'efficacité constitue un objectif des travaux futurs afin de passer à l'échelle. Un des aspects envisagés est d'étudier une propagation de contraintes plus efficace pour exploiter des contraintes utilisateur.

Enfin, nous souhaitons également étendre le cadre pour traiter d'autres tâches de clustering, comme par exemple lorsque le nombre de classes n'est pas fixé à l'avance, ou la classification recouvrante.

Références

[1] Mihael Ankerst, Markus M. Breunig, Hans peter Kriegel, and Jörg Sander. Optics : Ordering points to identify the clustering structure. In *ACM SIGMOD international conference on Management of data*, pages 49–60. ACM Press, 1999.

[2] Nicolas Beldiceanu, Mats Carlsson, Thierry Petit, and Jean-Charles Régin. An o(nlog n) bound consistency algorithm for the conjunction of an alldifferent and an inequality between a sum of variables and a constant, and its generalization. In *ECAI 2012 - 20th European Conference on Artificial Intelligence*, pages 145–150, 2012.

[3] M. Bilenko, S. Basu, and R. J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *Proceedings of the 21st ICML*, pages 11–18, 2004.

[4] Patrice Boizumault, Bruno Crémilleux, Mehdi Khiari, Samir Loudni, and Jean-Philippe Métivier. Discovering Knowledge using a Constraint-based Language. *CoRR*, abs/1107.3407, 2011.

[5] Thi-Bich-Hanh Dao, Khanh-Chuong Duong, and Christel Vrain. Une approche en PPC pour la classification non supervisée. In *13e EGC*, 2013.

[6] I. Davidson and S. S. Ravi. Agglomerative hierarchical clustering with constraints : Theoretical and empirical results. *Proceedings of the 9th European Conf. on Principles and Practice of Knowledge Discovery in Databases*, pages 59–70, 2005.

[7] Ian Davidson and S. S. Ravi. Clustering with Constraints : Feasibility Issues and the k-Means Algorithm. In *Proc. 5th SIAM Data Mining Conference*, 2005.

[8] Ian Davidson, S. S. Ravi, and Leonid Shamis. A SAT-based Framework for Efficient Constrained Clustering. In *SDM*, pages 94–105, 2010.

[9] L. De Raedt, T. Guns, and S. Nijssen. Constraint Programming for Data Mining and Machine Learning. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[10] Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 204–212, 2008.

[11] Martin Ester, Hans P. Kriegel, Jorg Sander, and Xiaowei Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Second International Conference on Knowledge Discovery and Data Mining*, pages 226–231, Portland, Oregon, 1996. AAAI Press.

[12] F. Geraci, M. Pellegrini, M. Maggini, and F. Sebastiani. Cluster Generation and Cluster Labelling for Web Snippets : a Fast and Accurate Hierarchical Solution. In *Proceedings of the 13th Int. Conf. on String Processing and Information Retrieval*, pages 25–36, 2006.

[13] T. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38 :293–306, 1985.

[14] Tias Guns, Siegfried Nijssen, and Luc De Raedt. k-Pattern set mining under constraints. *IEEE Transactions on Knowledge and Data Engineering*, 2011.

[15] 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. IEEE, June 2008.

[16] Jean-Philippe Métivier, Patrice Boizumault, Bruno Crémilleux, Mehdi Khiari, and Samir Loudni. Constrained Clustering Using SAT. In *IDA 2012, LNCS 7619*, pages 207–218, 2012.

[17] Marianne Mueller and Stefan Kramer. Integer linear programming models for constrained clustering. In *Discovery Science DS 2010*, pages 159–173, 2010.

[18] Thierry Petit, Jean-Charles Régin, and Nicolas Beldiceanu. A (n) bound-consistency algorithm for the increasing sum constraint. In *Principles and Practice of Constraint Programming CP 2011*, pages 721–728, 2011.

[19] Jean-Charles Régin and Thierry Petit. The objective sum constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems CPAIOR 2011*, pages 190–195, 2011.

[20] Jean-Charles Régin and Michel Rueher. Inequality-sum : a global constraint capturing the objective function. *RAIRO - Operations Research*, 39(2) :123–139, 2005.

[21] K. Wagstaff and C. Cardie. Clustering with instance-level constraints. In *Proceedings of the 17th ICML*, pages 1103–1110, 2000.

[22] K. Wagstaff, C. Cardie, S. Rogers, and S. Schroedl. Constrained k-means clustering with background knowledge. In *Proc. of the 18th ICML*, pages 577–584, 2001.

[23] Xiang Wang and Ian Davidson. Flexible constrained spectral clustering. In *KDD '10 : 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 563–572, 2010.

Extension des cohérences WCSP aux tuples

Djamel E. Dehani Christophe Lecoutre Olivier Roussel

CRIL - CNRS UMR 8188,
Université Lille Nord de France, Artois,
rue de l'universite,
62307 Lens cedex, France
{dehani,lecoutre,roussel}@cril.fr

Résumé

Dans cet article, nous présentons un nouveau type de propriétés pour les réseaux de contraintes pondérées (WCNs pour Weighted Constraint networks). Il s'agit de la cohérence de tuples (TC) dont l'établissement sur un WCN est effectué grâce à une nouvelle opération appelée `TupleProject`. Nous proposons également une version "optimale" de cette propriété, OTC, qui peut être perçue comme une généralisation de OSAC (Optimal Soft Arc Consistency). Le principe sous-jacent à OTC est d'appliquer de manière itérative l'opération `TupleProject` afin de factoriser un coût qui maximise la borne inférieure $w_\emptyset$, sur la base de transferts de coûts entre tuples de différentes contraintes d'arité quelconque.

1 Introduction

Le problème de satisfaction de contraintes pondérées (WCSP) consiste, pour un réseau constitué de variables et de contraintes souples, à trouver une affectation de valeur à chaque variable qui soit optimale, c'est à dire une assignation complète de coût minimal parmi toutes les assignations complètes possibles. Il s'agit d'un problème NP-difficile. Un certain nombre de travaux ont étés menés pour adapter au cadre WCSP des propriétés (et algorithmes efficaces) définies pour le cadre CSP, généralement dans le but de filtrer l'espace de recherche, comme par exemple la cohérence de nœud (NC) ou la cohérence d'arc (AC) [9, 10]. Un certain nombre d'algorithmes de plus en plus sophistiqués ont été proposés au cours des années pour approcher la cohérence d'arc souple idéale : FDAC, EDAC, VAC et OSAC (voir [4]).

Les algorithmes évoqués ci-dessus utilisent des opérations de transfert de coûts, appelées transformations préservant l'équivalence (EPTs). Celles-ci permettent la projection (ou extension) de coûts entre contraintes binaires et contraintes unaires ainsi que la projection de coûts entre contraintes unaires et la contrainte particulière, $w_\emptyset$, représentant un coût à apposer à toute instanciation complète. Nous proposons dans cet article d'étudier une nouvelle opération (EPT), appelée `TupleProject`. Nous l'utilisons pour définir de nouvelles propriétés basées sur le transfert de coûts entre contraintes, y compris lorsque ces contraintes sont d'arité supérieure ou égale à 2. Une nouvelle propriété, la cohérence de tuples (TC), est identifiée, ainsi qu'une version "optimale", OTC, qui peut être perçue comme une généralisation de OSAC (Optimal Soft Arc Consistency).

Pour illustrer l'intérêt du transfert de coûts entre tuples de différentes contraintes (non unaires), considérons un WCN contenant 4 variables x, y, z et t avec pour chaque variable deux valeurs possibles a et b, et 3 contraintes w_{xy}, w_{xyz} et w_{xyt} définies par la figure 1 (pour chaque tuple possible, le coût est donné dans la colonne de droite). Ce WCN est arc-cohérent (AC) et même arc-cohérent souple optimal (OSAC) car il n'existe pas de transformation (classique) permettant d'augmenter la borne $w_\emptyset$ ($w_\emptyset$ fournit un minorant du coût du réseau). En revanche, nous remarquons pour la contrainte w_{xyz} que l'instanciation $\{(x, a), (y, b)\}$ a nécessairement un coût de 1. Par conséquent, ce coût peut être déplacé (sans perte d'équivalence) vers le tuple $\{(x, a), (y, b)\}$ de la contrainte w_{xy} à partir de la contrainte w_{xyz}. De même, pour la contrainte w_{xyt}, nous remarquons que l'instanciation $\{(x, b), (y, a)\}$ a également nécessairement un coût de 1. Par conséquent, ce coût peut être déplacé vers le tuple $\{(x, b), (y, a)\}$ de la contrainte w_{xy} à partir de la contrainte w_{xyz}. On obtient alors le réseau de la figure 2. Nous sommes capables maintenant de transférer un coût de 1 sur les valeurs de x ou y

depuis la contrainte w_{xy}, ensuite, de le transférer vers $w_\emptyset$. Le WCN obtenu (voir figure 3) est équivalent au problème d'origine et a maintenant un minorant $w_\emptyset = 1$ et toutes les contraintes sont uniformément nulles.

x	y	w_{xy}
a	a	1
a	b	0
b	a	0
b	b	1

$w_\emptyset$
0

x	y	z	w_{xyz}
a	a	a	0
a	a	b	0
a	b	a	1
a	b	b	1
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

x	y	t	w_{xyt}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	1
b	a	b	1
b	b	a	0
b	b	b	0

FIGURE 1 – Réseau initial

x	y	w_{xy}
a	a	1
a	b	1
b	a	1
b	b	1

$w_\emptyset$
0

x	y	z	w_{xyz}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

x	y	t	w_{xyt}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

FIGURE 2 – Réseau obtenu après les premiers transferts

x	y	w_{xy}
a	a	0
a	b	0
b	a	0
b	b	0

$w_\emptyset$
1

x	y	z	w_{xyz}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

x	y	t	w_{xyt}
a	a	a	0
a	a	b	0
a	b	a	0
a	b	b	0
b	a	a	0
b	a	b	0
b	b	a	0
b	b	b	0

FIGURE 3 – Réseau final

2 Préliminaires

Un *réseau de contraintes* (CN) P est défini par un ensemble fini de variables, noté $vars(P)$, et un ensemble fini de contraintes, noté $cons(P)$. Chaque variable x a un domaine associé noté $dom(x)$, qui contient l'ensemble fini des valeurs pouvant être assignées à x ; d représente la taille du plus grand domaine. Chaque contrainte c_S porte sur un ensemble ordonné $S \subseteq vars(P)$ de variables appelé *portée* de c_S. Elle est définie par une relation contenant l'ensemble des tuples autorisés pour les variables de S. L'arité d'une contrainte est le nombre de variables dans sa portée. Une contrainte *unaire* (resp., *binaire*) porte sur 1 (resp., 2) variable(s), et une contrainte *non-binaire* porte sur plus de deux variables. Une *instanciation* I d'un ensemble $X = \{x_1, \ldots, x_p\}$ de variables est un ensemble $\{(x_1, a_1), \ldots, (x_p, a_p)\}$ tel que $\forall i \in 1..p, a_i \in dom(x_i)$; chaque a_i est noté $I[x_i]$. Une solution de P est une instanciation complète de P (i.e., l'assignation d'une valeur à chaque variable) qui satisfait toutes les contraintes. Pour plus d'information sur les réseaux de contraintes, voir [7, 11].

Un *réseau de contraintes pondéré* (WCN) W est formé par un ensemble fini de variables $vars(W)$, un ensemble fini de contraintes souples $cons(W)$, et une valeur k qui est soit un entier naturel strictement positif soit $+\infty$. Chaque contrainte *souple* $w_S \in cons(W)$ porte sur un ensemble ordonné S de variables (sa portée) et est définie par une fonction de coût de $l(S)$ vers $\{0, \ldots, k\}$, où $l(S)$ est le produit cartésien des domaines des variables présentes dans S ; pour toute instanciation $I \in l(S)$, on notera le coût de I dans w_S par $w_S(I)$. Pour simplifier, pour toute contrainte (souple) w_S, un couple (x, a) avec $x \in S$ et $a \in dom(x)$ est appelé une *valeur* de w_S. Une instanciation de coût k (noté aussi $\top$) est interdite. Autrement, elle est autorisée avec le coût correspondant (0, noté aussi $\bot$, est complètement satisfaisant). Les coûts sont combinés par l'opérateur binaire $\oplus$ défini par :

$$\forall \alpha, \beta \in \{0, \ldots, k\}, \alpha \oplus \beta = min(k, \alpha + \beta)$$

Dans cet article, nous supposons les réseaux normalisés, c'est à dire qu'il n'existe pas deux contraintes de même portée. L'objectif du problème de satisfaction de contraintes pondéré (WCSP) est, pour un WCN donné, de trouver une instanciation complète de coût minimal. Pour plus d'information sur les contraintes pondérées, voir [1, 12].

Différentes variantes de la cohérence d'arc souple pour le cadre WCSP ont étés proposées durant ces dix dernières années. Il s'agit de la cohérence d'arc souple (AC*) [9, 10], la cohérence d'arc directionnelle complète (FDAC) [2], la cohérence d'arc directionnelle existentielle (EDAC) [6], la cohérence d'arc virtuelle (VAC) [3] et la cohérence d'arc souple optimale (OSAC) [4]. Tous les algorithmes proposés pour atteindre ces différents niveaux de cohérence utilisent des opérations de transfert de coûts (ou transformations préservant l'équivalence) décrites dans la section suivante.

Pour finir, nous appellerons tuple d'une contrainte w_S tout tuple de $l(S)$, et dans un souci de simplicité, nous sup-

poserons donné un WCN W, et nous appellerons simplement contrainte toute contrainte souple de W.

3 EPTs

Certaines opérations permettent le transfert de coûts entre différentes contraintes. Elle sont appelées transformations préservant l'équivalence (EPTs). Nous introduisons d'abord les EPTs classiques, puis nous proposons une opération originale.

3.1 EPTs classiques

Les EPTs classiques sont les opérations `Project` et `UnaryProject` utilisées dans de nombreux algorithmes. Elles permettent de factoriser des coûts entre contraintes de portées différentes tout en garantissant l'équivalence. Pour factoriser les coûts, il faut être capable de les soustraire. Cela se fait à l'aide de l'opérateur $\ominus$ défini comme suit :

$$\forall \alpha, \beta \in \{0, \ldots, k\} \mid \alpha \geq \beta : \alpha \ominus \beta = \begin{cases} \alpha - \beta : \alpha \neq k \\ k : \alpha = k \end{cases}$$

L'algorithme 1 décrit l'opération `Project` qui factorise un coût α depuis les tuples de la contrainte w_S vers la valeur (x, a) de la contrainte w_x.

Algorithm 1: `Project`(w_x, a, w_S, α)

Pré-condition: $x \in S \wedge 0 < \alpha \leq \min_{t \in l(S), t[x]=a} w_S(t)$

1 $w_x(a) \leftarrow w_x(a) \oplus \alpha$
2 **pour chaque** $t \in l(S)$ *t.q.* $t[x] = a$ **faire**
3 $\quad w_S(t) \leftarrow w_S(t) \ominus \alpha$

L'algorithme 2 décrit l'opération `UnaryProject` qui factorise un coût α de la contrainte w_x vers $w_\emptyset$.

Algorithm 2: `UnaryProject`(w_x, α)

Pré-condition: $0 < \alpha \leq \min_{a \in dom(x)} w_x(a)$

1 $w_\emptyset \leftarrow w_\emptyset \oplus \alpha$
2 **pour chaque** $a \in dom(x)$ **faire**
3 $\quad w_x(a) \leftarrow w_x(a) \ominus \alpha$

3.2 Projection entre tuples

Nous proposons une nouvelle opération (EPT) qui généralise les EPT précédentes : la projection entre tuples notée `TupleProject`. L'idée est de pouvoir transférer un coût α entre des tuples de différentes contraintes et d'arité quelconque. La définition de la projection entre tuples est la suivante.

Définition 1 *Soient w_S et $w_{S'}$ deux contraintes telles que $S \subset S'$. L'opération* `TupleProject`$(w_S, t, w_{S'}, \alpha)$ *consiste à projeter un coût α sur un tuple t de w_S depuis les tuples t' de $w_{S'}$ tels que $t \subset t'$, avec $0 < \alpha \leq \min_{t' \in l(S'), t \subset t'} w_{S'}(t')$. Il s'agit :*

- *d'ajouter α à $w_S(t)$, à l'aide de l'opérateur $\oplus$;*
- *de soustraire α de $w_{S'}(t')$ à l'aide de l'opérateur $\ominus$, et ceci pour chaque $t' \in l(S')$ tel que $t \subset t'$.*

Dans l'exemple de la figure 1, la projection sur le tuple $t = \{(x, a), (y, b)\}$ d'un coût 1 peut se faire grâce à la fonction `TupleProject`$(w_{xy}, t, w_{xyz}, 1)$.

Clairement, `TupleProject` préserve l'équivalence et généralise `Project` et `UnaryProject`. En effet, l'opération `Project` correspond au cas $|S| = 1$ et $|S'| > 1$ tandis que l'opération `UnaryProject` correspond au cas $|S| = 0$ et $|S'| = 1$.

L'EPT `TupleProject` est également décrite par l'algorithme 3 qui factorise un coût α depuis les tuples de $w_{S'}$ vers le tuple t de w_S.

Algorithm 3: `TupleProject`$(w_S, t, w_{S'}, \alpha)$

Pré-condition: $S \subset S' \wedge t \in l(S)$
Pré-condition: $0 < \alpha \leq \min_{t' \in l(S'), t \subset t'} w_{S'}(t')$

1 $w_S(t) \leftarrow w_S(t) \oplus \alpha$
2 **pour chaque** $t' \in l(S') \mid t \subset t'$ **faire**
3 $\quad w_{S'}(t') \leftarrow w_{S'}(t') \ominus \alpha$

La définition 1 présente le transfert de coûts entre deux contraintes dans le cas particulier où le scope de l'une est inclus dans le scope de l'autre mais l'objectif de nos transferts est plus large. En effet, cette définition aboutit très naturellement à des transferts de coûts entre contraintes qui partagent des variables, mais sans relation d'inclusion. De fait, quand deux contraintes w_S et $w_{S'}$ partagent des variables, on peut décomposer les transferts en deux étapes : de S vers $S \cap S'$ et ensuite de $S \cap S'$ vers S' (ces deux transferts obéissant à la définition 1).

4 La cohérence de tuples TC

Avant de définir ce type de cohérence, nous introduisons la notion de *sur-contrainte* d'un ensemble de variables.

Définition 2 *Soit $S \subseteq vars(W)$ un ensemble de variables de W. Une sur-contrainte de S est une contrainte $w_{S'}$ telle que $S \subset S'$. L'ensemble des sur-contraintes de S est noté $\Gamma(S) = \{w_{S'} \in cons(W) \mid S \subset S'\}$.*

Définition 3 *Soient $S \subseteq vars(W)$ un ensemble de variables de W, t un tuple de $l(S)$, et $w_{S'}$ une contrainte de $\Gamma(S)$. L'ensemble $\sigma_{w_{S'}}(t) = \{t' \in l(S') \mid t \subset t'\}$ contient tous les tuples définis sur S' et qui étendent t.*

Définition 4 *Soient $S \subseteq vars(W)$ un ensemble de variables de W, t un tuple de $l(S)$, et $w_{S'}$ une contrainte de $\Gamma(S)$. On dit que $t' \in \sigma_{w_{S'}}(t)$ est un support de t sur $w_{S'}$ ssi $w_{S'}(t') = 0$.*

Quand un tuple n'a pas de support, une factorisation est possible et permet d'en créer un. Soit une contrainte w_S de $cons(W)$ et un tuple t de w_S, un support de t sur une contrainte $w_{S'} \in \Gamma(S)$ est créé en procédant comme suit :

- chercher le tuple $t'' = argmin_{t' \in \sigma_{w_{S'}}(t)} w_{S'}(t')$ qui va devenir le support de t sur $w_{S'}$;
- et exécuter $\texttt{TupleProject}(w_S, t, w_{S'}, w_{S'}(t''))$.

Pour illustrer ce dernier point, nous reprenons notre exemple introductif (voir figure 1). Sur ce WCN, il est clair que le tuple $t = \{(x, a), (y, b)\}$ n'a pas de support sur la contrainte w_{xyz}. Pour créer un support à ce tuple, il suffit d'appliquer l'EPT $\texttt{TupleProject}(w_{xy}, t, w_{xyz}, 1)$.

Définition 5 *Soit un ensemble de variables $S \subseteq vars(W)$. Un tuple $t \in l(S)$ est tuple-cohérent (TC) ssi il possède un support sur chaque contrainte $w_{S'} \in \Gamma(S)$. On dit que S est TC ssi tous les tuples de $l(S)$ sont TC. On dit que W est TC ssi tout ensemble de variables $S \subseteq vars(W)$ est TC.*

Chercher à rendre TC un WCN n'est pas toujours réaliste car, dans certains cas, cela peut nécessiter l'introduction d'une contrainte pour tout sous-ensemble possible de variables du problème, ce qui est exponentiel. C'est la raison pour laquelle nous proposons d'étudier une forme limitée de cette cohérence, notée TC_r, en ne cherchant des supports que pour les tuples d'arité inférieure ou égale à une arité limite r donnée. En pratique, r aura une petite valeur, ce qui nous permettra de n'introduire, si nécessaire, que des contraintes d'arité faible.

Définition 6 *Pour tout entier strictement positif r, W est TC_r si $\forall S \subseteq vars(W)$ tel que $|S| \leq r$, S est TC.*

Avec la propriété TC_r, chaque fois qu'un tuple t défini sur un ensemble de variables S d'arité $i \leq r$ n'admet pas de support sur une contrainte, une factorisation doit être effectuée vers t. S'il n'existe pas de contrainte w_S dans le WCN W, une nouvelle contrainte doit être ajoutée à W, en initialisant les coûts de tous ses tuples à 0 (avant la projection vers t) : $\forall t \in l(S), w_S(t) = 0$. Il est à noter que pour $r = 1$, TC_r est équivalent à AC^*. En effet, dans ce cas, l'établissement de TC_r consiste à créer pour chaque valeur de chaque variable un support sur les contraintes qui portent sur cette variable.

Établir TC_r peut être réalisé à l'aide d'un premier algorithme appelé $TC1_r$ (voir algorithme 4). Nous commençons d'abord par établir la propriété TC pour les tuples d'arité r, puis les tuples d'arité $r-1$, etc. De fait, nous établissons la propriété TC par ordre décroissant d'arité, car

Algorithm 4: $\text{TC1}_r(W : \text{WCN})$

```
1 pour i ← r à 0 faire
2     pour chaque S ⊆ vars(W) tel que |S| = i faire
3         pour chaque t ∈ l(S) faire
4             pour chaque w_S' ∈ Γ(S) faire
5                 α ← min_{t'∈l(S'),t⊂t'} w_S(t')
6                 si α > 0 alors
7                     si w_S ∉ cons(P) alors
8                         ajouter la contrainte w_S de
                          coût uniformément nul à W
9                     TupleProject(w_S, t, w_S', α)
```

les transferts de coûts s'effectuent de contraintes d'arité j vers des contraints d'arité i avec $j > i$. L'intérêt est qu'un support établi pour un tuple t d'une contrainte d'arité i reste valide lorsque la recherche de supports s'effectue aux niveaux inférieurs à i.

Proposition 1 *Appliqué à tout WCN W, l'algorithme $TC1_r$ produit un WCN qui est TC_r et qui est équivalent à W.*

Preuve. L'algorithme n'applique que des EPTs, donc le réseau obtenu est équivalent au réseau initial. Les lignes 3 à 9 construisent un support (le tuple qui a le coût α dans $w_{S'}$) pour tout ensemble de variables S tel que $|S| = i$. On peut noter que seuls les coûts des tuples de $l(S)$ peuvent augmenter. Comme la boucle externe est décroissante, une fois un support établi, son coût ne peut plus jamais augmenter et reste donc égal à 0. Autrement dit, une fois établi, un support n'est jamais remis en cause par l'algorithme. La boucle énumérant toutes les arités de r à 0, un support est établi pour tout ensemble S tel que $|S| \leq r$. □

L'algorithme 4 nécessite de considérer pour tout ensemble de variables S toutes les contraintes de $\Gamma(S)$. Une autre stratégie pour établir TC_r consiste à ajouter au préalable toutes les contraintes d'arité inférieure ou égale à r qui n'existeraient pas déjà dans le réseau. Cet ajout systématique permet de simplifier les factorisations en exploitant des propriétés de transitivité et reste envisageable tant que r est faible. Nous obtenons un second algorithme $TC2_r$ (voir algorithme 5).

La figure 4 illustre le principe de ce second algorithme. Soit S_1, S_2 et S_3 trois ensembles de variables tels que $S_1 \subset S_2 \subset S_3$ et $r \geq |S_3| = |S_2| + 1 = |S_1| + 2$. Lorsqu'une factorisation est possible de S_3 vers S_1, elle peut s'effectuer en deux étapes : de S_3 vers S_2 puis de S_2 vers S_1.

Algorithm 5: TC2$_r$(W : WCN)

1 compléter W avec toutes les contraintes manquantes d'arité inférieure ou égale à r
2 **pour chaque** $w_S \in cons(W)$ *tel que* $|S| = r$ **faire**
3 **pour chaque** $t \in l(S)$ **faire**
4 **pour chaque** $w_{S'} \in \Gamma(S)$ **faire**
5 $\alpha \leftarrow \min_{t' \in l(S'), t \subset t'} w_S(t')$
6 **si** $\alpha > 0$ **alors**
7 TupleProject$(w_S, t, w_{S'}, \alpha)$
8 **pour** $i \leftarrow r-1$ ***à*** 0 **faire**
9 **pour chaque** $w_S \in cons(P)$ *tel que* $|S| = i$ **faire**
10 **pour chaque** $t \in l(S)$ **faire**
11 **pour chaque** $w_{S'} \in \Gamma(S)$ *t.q.* $|S'| = i+1$ **faire**
12 $\alpha \leftarrow \min_{t' \in l(S'), t \subset t'} w_S(t')$
13 **si** $\alpha > 0$ **alors**
14 TupleProject$(w_S, t, w_{S'}, \alpha)$

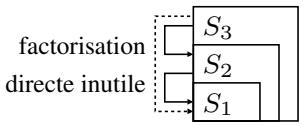


FIGURE 4 – Exploitation de la transitivité des factorisations

Proposition 2 *Appliqué à tout WCN W, l'algorithme TC2$_r$ produit un WCN qui est TC$_r$ et qui est équivalent à W.*

Preuve. La première instruction de l'algorithme assure que toutes les contraintes d'arité inférieure ou égale à r sont présentes dans le réseau. Aussi, quand on peut appliquer TupleProject de $w_{S'}$ d'arité a' vers w_S d'arité a avec $a < a' \leq r$, on a alors la garantie qu'il existe une succession de contraintes w_{S_i} pour $i \in a..a'$ telles que $S_{a'} = S'$, $S_a = S$, $\forall i \in a..a'-1, S_i \subset S_{i+1}$ et $|S_i| = i$. De ce fait, on peut remplacer une factorisation directe de $w_{S'}$ vers w_S par une succession de factorisations de $w_{S_{i+1}}$ vers w_{S_i}. □

Avec l'algorithme TC2$_r$, il faut donc d'abord ajouter les contraintes w_S qui portent sur les différents sous-ensembles de variables $S \subseteq vars(P)$ tel que $|S| \leq r$ et $w_S \notin cons(P)$. Pour illustrer ce point, prenons un WCN W comportant seulement deux contraintes ternaires w_{xyz} et w_{wyz}. L'ensemble des contraintes du WCN W' équivalent à W sur lequel nous appliquerons la propriété TC$_r$ avec $r = 2$ est donné par la figure 5.

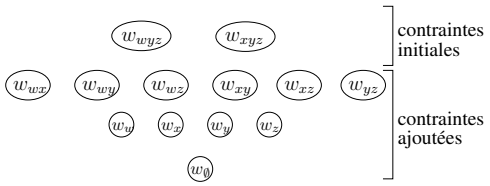


FIGURE 5 – Ajout des contraintes d'arité inférieure ou égale à $r = 2$

En fait, il est possible d'éviter l'introduction de certaines contraintes inutiles. Sur l'exemple précédent, il est inutile d'introduire w_{xw} car elle n'a pas de sur-contrainte. La définition 7 formalise cette remarque.

Définition 7 *Un ensemble de variables $S \subseteq vars(W)$ est dit* isolé *ssi* $\Gamma(S) = \emptyset$.

Clairement, si un ensemble de variables S est isolé alors il est TC. Ainsi, lorsqu'un ensemble de variables S est isolé et que la contrainte w_S est manquante, il n'est pas nécessaire d'ajouter cette contrainte au réseau. Notre réseau de contraintes de la figure 5 peut donc être simplifié en supprimant la contrainte w_{wx}.

Une variante de TC$_r$ consiste à ne pas ajouter de contraintes au réseau, et à simplement créer des supports pour les tuples des contraintes existantes d'arité inférieure ou égale à r. Comme il s'agit d'une version affaiblie de TC$_r$, nous appellerons cette variante TC$_r^w$ (weak TC$_r$). Établir TC$_r^w$ peut s'effectuer sur la base de l'algorithme $TC1_r$ (voir algorithme 4) en s'interdisant simplement l'ajout de contrainte et en ne parcourant que les ensembles S qui sont le scope de contraintes existantes.

5 La cohérence de tuples optimale (OTC)

Nous proposons maintenant une méthode cherchant à tirer le meilleur parti de l'opération TupleProject, tout comme OSAC exploite au mieux les EPTs classiques.

Il est facile de montrer que l'application itérée de TupleProject n'est pas confluente, ce qui signifie qu'on peut obtenir plusieurs résultats différents selon l'ordre dans lequel on effectue les opérations. D'une part, le WCN obtenu après factorisation n'est pas unique car les coûts peuvent être factorisés de différentes manières, et d'autre part, la borne inférieure $w_\emptyset$ qui est obtenue peut également être différente.

Dans cette section, nous développons la même stratégie que pour OSAC [5], à savoir rechercher un ensemble de projections TupleProject qui peuvent être appliquées simultanément de manière à augmenter le plus possible le minorant $w_\emptyset$. Cette recherche est codée sous la forme d'un

problème linéaire. De manière à permettre la résolution de ce problème en temps polynomial, nous autorisons la factorisation de coûts rationnels et pas seulement entier. Nous considérons donc dans ce qui suit que les coûts du WCN sont rationnels.

Comme pour OSAC, on ne cherche pas à déterminer dans quel ordre appliquer les opérations `TupleProject`, ni à vérifier si à chaque étape le WCN obtenu est valide (tous les coûts restent positifs). On considère que les opérations sont effectuées simultanément et notre seule exigence est que le WCN final soit valide et équivalent au réseau initial.

Définition 8 *Étant donné un WCN W, un ensemble d'opérations `TupleProject` est valide si et seulement si, appliquées simultanément, ces opérations transforment W en un WCN W' qui est valide (tous les coûts restent positifs) et équivalent (toute instanciation complète a le même coût dans W et W').*

La cohérence de tuple optimale (OTC) généralise OSAC car elle utilise l'opération `TupleProject` qui subsume les opérations $Project$ et $UnaryProject$ utilisées dans OSAC.

Définition 9 *Un WCN W est tuple-cohérent optimal (OTC) s'il n'existe pas d'ensemble d'opérations `TupleProject` valide qui, appliqué à W, permette d'augmenter le minorant $w_\emptyset$.*

La méthode OTC peut être déclinée de deux manières : soit en n'ajoutant pas de contrainte comme dans OSAC, soit en complétant initialement le réseau avec des contraintes supplémentaires. Dans les deux cas, il est nécessaire en pratique de limiter l'arité des tuples sur lesquels porte $TupleProject$ afin de conserver une complexité raisonnable. Nous nommerons OTC$_r^w$ (weak OTC$_r$) la version qui n'ajoute pas de contrainte, et OTC$_r$ la version qui ajoute des contraintes. Dans les deux cas, r est l'arité maximale considérée par la méthode. Nous présentons ces deux méthodes dans les sections qui suivent.

5.1 Cohérence de tuple optimale restreinte faible OTC$_r^w$

Dans cette version, on s'interdit de modifier l'ensemble de contraintes. Par ailleurs, pour des raisons d'efficacité, on restreindra les opérations $TupleProject$ au cas où au moins l'une des contraintes est d'arité inférieure ou égale à r. Cette version est nommée OTC$_r^w$.

Pour trouver l'ensemble d'opérations `TupleProject` valide qui maximise $w_\emptyset$, on peut définir un problème de programmation linéaire (PL) comme suit. Nous notons par $\alpha_t^{S'}$ le coût transféré depuis une contrainte $w_{S'}$ vers le tuple t de la contrainte w_S avec $t \in l(S)$ par l'opération `TupleProject`$(w_S, t, w_{S'}, \alpha_t^{S'})$. En utilisant la notation $t[S]$ pour représenter la projection du tuple t sur le scope S, le problème linéaire obtenu est donné en figure 6.

La première inéquation traite des contraintes d'arité strictement supérieure à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les contraintes d'arité inférieure ou égale à r doit être au final positif ou nul.

La seconde inéquation traite des contraintes d'arité inférieure ou égale à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les contraintes d'arité inférieure plus les transferts depuis les contraintes d'arité supérieure doit être au final positif ou nul.

La fonction d'objectif maximise le transfert de coûts vers $w_\emptyset$ qui est la somme des transferts $\alpha_{w_\emptyset}^x$ de la contrainte unaire portant sur x vers $w_\emptyset$.

On notera que dans le système linéaire, les variables α_t^S peuvent être soit positives, soit négatives. Une valeur positive correspond à une projection d'un coût depuis plusieurs tuples vers un seul, tandis qu'une valeur négative correspond à une extension, c'est à dire le transfert d'un coût depuis un tuple unique vers plusieurs tuples. De ce fait, OTC ne se contente pas d'établir des supports mais alterne les opérations de projection et d'extension de manière à obtenir la meilleure borne $w_\emptyset$.

Lorsque les coûts sont rationnels, ce système linéaire se résout en temps polynomial [8].

5.2 Cohérence de tuple optimale restreinte OTC$_r$

Une seconde déclinaison de OTC consiste à ajouter au réseau de départ toutes les contraintes d'arité inférieure ou égale à r et ensuite effectuer les opérations $TupleProject$ en se restreignant au cas ou au moins l'une des contraintes est d'arité inférieure ou égale à r. Cette méthode est nommée OTC$_r$.

Dans ce cas, en utilisant la transitivité des factorisations, le système linéaire peut être simplifié. Ce système est présenté en figure 7.

La première inéquation traite des contraintes d'arité strictement supérieure à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les contraintes d'arité égale à r doit être au final positif ou nul.

La deuxième inéquation traite des contraintes d'arité égale à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les contraintes d'arité immédiatement inférieure plus les transferts depuis les contraintes d'arité plus grande que r doit être au final positif ou nul.

La troisième inéquation traite des contraintes d'arité inférieure à r. Pour chaque tuple de ces contraintes, le coût initial de ce tuple moins les transferts de coûts vers les

$$\begin{cases} \max : \sum\limits_{x \in vars(P)} \alpha^x_{w_\emptyset} \\ \forall w_S \in cons(W), \text{ t.q. } |S| > r, \forall t \in l(S) : w_S(t) - \sum\limits_{\substack{S' \subset S \\ |S'| \leq r}} \alpha^S_{t[S']} \geq 0 \\ \forall w_S \in cons(W), \text{ t.q. } |S| \leq r, \forall t \in l(S) : w_S(t) - \sum\limits_{S' \subset S} \alpha^S_{t[S']} + \sum\limits_{w_{S'} \in \Gamma(S)} \alpha^{S'}_t \geq 0 \end{cases}$$

FIGURE 6 – Système linéaire généré par OTC^w_r

$$\begin{cases} \max : \sum\limits_{x \in vars(P)} \alpha^x_{w_\emptyset} \\ \forall w_S \in cons(W) \text{ t.q. } |S| > r, \forall t \in l(S) : w_S(t) - \sum\limits_{\substack{S' \subset S, \\ |S'| = r}} \alpha^S_{t[S']} \geq 0 \\ \forall w_S \in cons(W) \text{ t.q. } |S| = r, \forall t \in l(S) : w_S(t) - \sum\limits_{\substack{S' \subset S, \\ |S| = |S'|+1}} \alpha^S_{t[S']} + \sum\limits_{w_{S'} \in \Gamma(S)} \alpha^{S'}_t \geq 0 \\ \forall w_S \in cons(W) \text{ t.q. } |S| < r, \forall t \in l(S) : w_S(t) - \sum\limits_{\substack{S' \subset S, \\ |S| = |S'|+1}} \alpha^S_{t[S']} + \sum\limits_{\substack{S \subset S', \\ |S'| = |S|+1}} \alpha^{S'}_t \geq 0 \end{cases}$$

FIGURE 7 – Système linéaire généré par OTC_r

contraintes d'arité immédiatement inférieure plus les transferts depuis les contraintes d'arité immédiatement supérieure doit être au final positif ou nul.

La fonction d'objectif maximise le transfert de coûts vers $w_\emptyset$ qui est la somme des transferts $\alpha^x_{w_\emptyset}$ de la contrainte unaire portant sur x vers $w_\emptyset$.

Il est clair que OTC_r (avec $r \geq 1$) est également une généralisation de OSAC, et donc OTC_r obtient un minorant $w_\emptyset$ au moins aussi grand que OSAC.

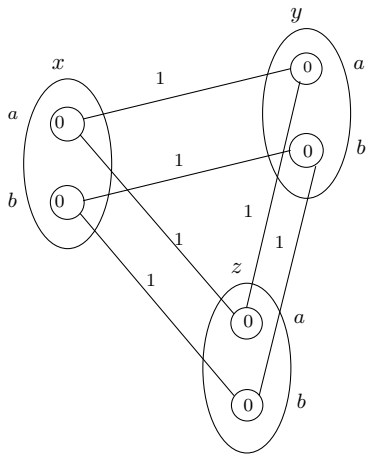


FIGURE 8 – Un WCN OTC_2 cohérent mais non OTC_3 cohérent

L'intérêt de OTC_r est illustré par l'exemple de la figure 8. Ce réseau est OTC_2 et donc OSAC. Par contre, il n'est pas OTC_3 car l'introduction de la contrainte ternaire w_{xyz} va permettre d'augmenter le minorant $w_\emptyset$. En effet, une fois la contrainte w_{xyz} introduite (avec des coûts nuls au départ), on peut transférer le coût de chaque tuple de chaque contrainte binaire vers les tuples correspondant de la contrainte ternaire. Par exemple, le coût 1 assigné au tuple $\{(x,a),(y,a)\}$ est étendu aux deux tuples de $\{(x,a),(y,a),(z,a)\}$ et $\{(x,a),(y,a),(z,b)\}$ de w_{xyz}. À ce moment, les contraintes binaires ont toutes des coûts nuls et les coûts de la contrainte ternaire sont donnés par la figure 9.

x	y	z	w_{xyz}
a	a	a	3
a	a	b	1
a	b	a	1
a	b	b	1
b	a	a	1
b	a	b	1
b	b	a	1
b	b	b	3

FIGURE 9 – La contrainte w_{xyz} après les opérations d'extension

À ce stade, il est direct de factoriser un coût de 1 depuis w_{xyz} vers chacun des tuples de w_{xy} (par exemple), puis de factoriser ce même coût vers w_x (par exemple) pour enfin factoriser 1 vers $w_\emptyset$.

Cet exemple montre que dans certains cas, l'ajout de surcontraintes dans un WCN permet d'augmenter le minorant

$w_\emptyset$.

Il est à noter que le réseau obtenu par OTC$_r$ n'est pas nécessairement TC$_r$, tout comme le réseau obtenu par OSAC n'est pas nécessairement AC.

6 OSAC vs OTC

Dans cette section, nous revenons sur OSAC et OTC, cherchant à établir des rapports entre ces deux cohérences.

Proposition 3 *OSAC est équivalent à OTC$_1^w$.*

Preuve. Le système linéaire généré par OSAC est identique au système linéaire généré par OTC$_1^w$. □

Proposition 4 *Sur l'ensemble des WCNs binaires, OSAC est équivalent à OTC$_r^w$, $\forall r \geq 1$.*

Preuve. Dans le cas des réseaux binaires, que ce soit avec OSAC ou OTC$_r^w$, les seuls transferts possibles sont effectués depuis les contraintes binaires vers les contraintes unaires, puis, depuis les contraintes unaires vers $w_\emptyset$. Aussi, le système linéaire défini par OSAC est le même que celui défini par OTC$_r^w$, sous condition que $r \geq 1$. □

Appliquer OSAC (respectivement, OTC$_r^w$) sur un WCN W permet de calculer un minorant que nous noterons $w_\emptyset^{OSAC}(W)$ (respectivement, $w_\emptyset^{OTC_r^w}(W)$).

Proposition 5 *$\forall r \geq 2$, OTC$_r^w$ est strictement plus fort que OSAC, signifiant que :*

- *pour tout WCN W, $w_\emptyset^{OTC_r^w}(W) \geq w_\emptyset^{OSAC}(W)$*
- *il existe des WCNs W t.q. $w_\emptyset^{OTC_r^w}(W) > w_\emptyset^{OSAC}(W)$*

Preuve. Pour OTC$_r^w$, le système généré contient les inéquations correspondant aux transferts générés par OSAC plus des inéquations pour des transferts potentiels entres tuples. Cela garantit que la valeur du minorant obtenu après l'application de OTC$_r^w$ est supérieure ou égale à celle obtenue après l'application de OSAC. L'exemple introductif à ce papier démontre le caractère strict de cette relation entre OTC$_r^w$ et OSAC. □

Pour finir, nous nous intéressons au cas des WCNs binaires acycliques (i.e., des WCNs dont le graphe de contraintes est acyclique).

Proposition 6 *Soit W un WCN binaire acyclique. Si W est VAC (virtual arc-consistent) alors la valeur de $w_\emptyset$ représente le coût de la solution optimale.*

Preuve. Pour établir VAC, un CN P est construit à partir du WCN W en transformant chaque contrainte souple en une contrainte dure, où seuls les tuples de coût 0 sont autorisés. Comme par hypothèse, W est VAC, cela signifie que W n'est pas détecté incohérent par un algorithme établissant la cohérence d'arc (AC). D'autre part, le graphe de contraintes de W est le même que celui de P, et celui-ci est acyclique. Tout CN acyclique qui est AC est connu pour admettre au moins une solution S. On en déduit que W admet S comme solution de coût 0, auquel il faut ajouter la valeur de $w_\emptyset$. □

Corollaire 1 *Soit W un WCN binaire acyclique. Si W est OSAC alors la valeur de $w_\emptyset$ représente le coût de la solution optimale (puisque un réseau OSAC est nécessairement VAC [3]).*

De ce dernier corollaire, nous pouvons retenir que si le graphe de contraintes est acyclique alors OSAC permet de factoriser le coût optimal vers $w_\emptyset$. Par ailleurs, si on veut établir OTC sur un WCN binaire acyclique, il n'est pas nécessaire d'ajouter de sur-contraintes dans le but d'essayer d'obtenir un minorant plus grand que celui obtenu par OSAC. En revanche, si le WCN n'est pas acylique, l'ajout de sur-contraintes peut éventuellement produire un minorant plus important.

7 Conclusion

Dans cet article, nous avons étudié quelques pistes concernant l'utilisation d'une opération de transfert de coûts généralisant les opérations classiques. Celle-ci permet le transfert de coûts entre deux contraintes toutes deux non unaires. Cela nous a permis de définir une nouvelle cohérence, TC, ainsi que sa version optimisée, OTC. Nous prévoyons d'expérimenter ces nouvelles propriétés dans un avenir proche.

Remerciements

Ce travail bénéficie du soutien du CNRS et d'OSEO dans le cadre du projet ISI Pajero.

Références

[1] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based CSPs and valued CSPs : Frameworks, properties, and comparison. *Constraints*, 4(3) :199–240, 1999.

[2] M. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3) :311–342, 2003.

[3] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted CSP. In *Proceedings of AAAI'08*, pages 253–258, 2008.

[4] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8) :449–478, 2010.

[5] M.C. Cooper, S. de Givry, and T. Schiex. Optimal soft arc consistency. In *Proceedings of IJCAI'07*, pages 68–73, 2007.

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

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

[8] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4) :373–396, 1984.

[9] J. Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of AAAI'02*, pages 48–53, 2002.

[10] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2) :1–26, 2004.

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

[12] P. Meseguer, F. Rossi, and T. Schiex. Soft constraints. In *Handbook of Constraint Programming*, chapter 9, pages 281–328. Elsevier, 2006.

Solveurs CP(FD) vérifiés formellement

Catherine Dubois[1,2] **Arnaud Gotlieb**[2,3]

[1] CEDRIC-ENSIIE, Évry, France
[2] Inria, France
[3] Certus V&V Center, SIMULA RESEARCH LAB., Lysaker, Norway
dubois@ensiie.fr arnaud@simula.no

Résumé

Les solveurs de contraintes sont utilisés pour résoudre des problèmes d'optimisation, de planification, d'ordonnancement, etc. Leur utilisation dans des domaines critiques réclame un certain degré de confiance et requiert un examen sceptique des résultats fournis par un solveur, tout particulièrement lorsque ce dernier assure qu'un problème n'a pas de solution. Nous proposons de développer un solveur pour les domaines finis - CP(FD) - correct par construction. Nous avons implanté le solveur à l'aide de l'outil de preuve Coq et l'avons prouvé correct par rapport à sa spécification elle-même exprimée en Coq. Il implante l'algorithme AC3 (et AC2001) et met en œuvre une consistance d'arcs. Le mécanisme d'extraction de Coq permet d'obtenir un solveur écrit en OCaml, formellement vérifié, le premier à notre connaissance. Ce solveur peut être utilisé pour résoudre un problème ou encore pour vérifier les résultats d'un autre solveur non vérifié formellement. Ces résultats ont été publiés récemmment dans les actes de la conférence internationale FM 2012 [1]. Nous présentons ici un résumé de ces travaux et étudions leur extension à la consistance de bornes.

Abstract

Constraint solvers are used to solve problems coming from optimization, scheduling, planning, etc. Their usage in critical applications implies a more skeptical regard on their implementation, especially when the result is that a constraint problem has no solution, i.e., unsatisfiability. In this paper, we propose an approach aiming to develop a correct-by-construction finite domain based -CP(FD)-solver. We developed this solver within the Coq proof tool and proved its correctness in Coq. It embeds the algorithm AC3 (and AC2001) and uses arc-consistency. The Coq extraction mechanism allows us to provide a finite domain based solver written in OCaml formally verified, the first one to our knowledge. This solver can be used directly or as *a second shot* solver to verify results coming from an untrusted solver. This result has recently been published in FM 2012 [1]. In this paper, we present a summary of this result and extend it to deal with another local-consistency property, namely, bound-consistency.

1 Introduction

Les solveurs de contraintes à domaines finis (CP(FD) dans la suite) ont pour but de rechercher des solutions à des problèmes exprimés sous la forme de triplets (X, C, D) où X est un ensemble de variables, C un ensemble de contraintes et D associe à chaque variable son domaine, ici un ensemble fini de valeurs possibles. Une solution affecte à chaque variable de X une valeur de son domaine de telle sorte que les contraintes soient toutes satisfaites. Lorsqu'un solveur fournit une solution ou répond UNSAT indiquant ainsi que le problème soumis n'a pas de solution, avons nous confiance dans ces résultats ? Il est souvent facile de vérifier le résultat dans le premier cas, en exécutant les contraintes avec la ou les solutions fournies. Ceci peut néanmoins nécessiter de mettre en œuvre un programme qui fait cette vérification. Mais lorsque le problème est déclaré insatisfaisable i.e. sans solution, comment faire ? La vérification *a posteriori* précédente n'est plus possible. La confiance dans un solveur est cruciale lorsque ce dernier est utilisé dans des applications sensibles voire critiques . Notons également que les solveurs CP(FD) sont fortement utilisés dans des outils de vérification de logiciels [2] ou de génération de tests [4], eux-mêmes utilisés pour vérifier des applications critiques. La complexité des solveurs actuels rend impossible la vérification formelle directe de leur code. Une technique alternative consisterait à faire produire par le solveur une trace post-mortem, qu'un vérificateur (vérifié formellement) pourrait ensuite analyser

pour certifier la correction des résultats fournis. Cette approche requiert cependant de développer un langage approprié de traces ou d'identifier le bon degré d'abstraction des traces car les langages de trace existants ont été développés pour la mise au point des solveurs à contraintes [5] et non pour la preuve formelle. Nous proposons une autre approche qui consiste à développer un solveur certifié formellement, i.e. prouvé correct. Pour cela, nous utilisons l'outil de preuve Coq [8] pour spécifier, programmer et prouver la correction du solveur. Grâce au mécanisme d'extraction de Coq, de ce développement est extrait automatiquement un solveur écrit en OCaml, formellement vérifié (ou certifié) par construction. Ce solveur peut être utilisé directement pour résoudre un problème, il peut aussi être utilisé pour vérifier les résultats d'un autre solveur non vérifié formellement. Nous avons appliqué cette approche pour fournir un solveur CP(FD) formellement certifié s'appuyant sur la consistance d'arc pour des contraintes binaires. A notre connaissance il s'agit du premier solveur CP(FD) formellement vérifié. Ces résultats ont été publiés dans [1]. Ces questions de confiance dans les résultats des solveurs ont été abordées pour les solveurs SAT et SMT avec des approches à base de traces [3] ou avec des approches similaires à la nôtre [6, 7].

Dans la section 2, nous présentons brièvement l'outil de preuve Coq et la méthodologie suivie. Puis la section 3 résume les points principaux de la formalisation et de la preuve de correction des algorithmes de filtrage et de propagation (voir [1] pour plus de détails). Nous avons également développé un processus d'énumération élémentaire qui, avec les processus de filtrage et de propagation, fournit un solveur correct et complet. Enfin dans la section 4, nous étudions l'extension des spécifications et preuves précédentes à la consistance de bornes.

2 Coq et la méthodologie utilisée

L'outil de preuve Coq [8] permet d'écrire des spécifications et d'énoncer des théorèmes dans un langage proche des mathématiques fortement expressif, puis de prouver ces théorèmes et enfin de vérifier ces preuves. Il ne s'agit pas d'un outil de preuve automatique, il requiert l'interaction avec l'utilisateur mais offre toutefois des procédures de décision qui peuvent être utilisées pour prouver automatiquement certaines formules. C'est aussi un langage de programmation fonctionnel typé qui propose, entre autres, des fonctions simples ou récursives, des types inductifs et du *pattern-matching*. Il impose que toutes les fonctions terminent, que tous les cas d'un pattern-matching soient traités. La programmation en Coq induit donc une programmation plus sûre. Nous avons utilisé ce langage pour implanter les différentes fonctions (par exemple la fonction `revise` qui supprime du domaine d'une variable les valeurs impossibles pour une contrainte donnée) qui composent le solveur comme si nous l'avions écrit directement dans un langage fonctionnel, OCaml par exemple. Nous avons défini des spécifications, la consistance d'arc par exemple, et décrit en Coq des propriétés concernant les fonctions précédentes. Par exemple, *l'application de* `revise` *sur les variables qui apparaissent dans une contrainte assure l'arc-consistance pour cette contrainte*. Puis nous avons prouvé ces propriétés.

Il est possible d'exécuter directement dans Coq les fonctions écrites en Coq. Néanmoins on n'obtient pas l'efficacité que l'on pourrait obtenir avec un programme écrit directement en OCaml. On peut utiliser le mécanisme d'extraction de Coq pour obtenir un programme OCaml sémantiquement équivalent. Ce mécanisme permet d'extraire le contenu calculatoire d'une preuve, en effaçant ce dernier. Dans notre cas, les propriétés et les preuves sont effacées, les fonctions composant le solveur sont traduites en OCaml.

3 Solveur CF(FD) pour consistance d'arc

Nous résumons ici le développement réalisé en Coq (fonctions, propriétés et preuves) présenté dans [1]. Un point important concerne la généricité de ce développement. Le solveur est en effet paramétré par le type des variables et des valeurs et aussi par le langage de contraintes. En Coq, ces types sont abstraits, supposés équipés d'une égalité décidable. On suppose également que la sémantique des contraintes est donnée par une fonction d'interprétation qui évalue la contrainte en fonction des valeurs de ses deux variables. On requiert aussi la définition d'une fonction qui associe à toute contrainte ses deux variables. Ces types et fonctions sont à définir directement en OCaml pour pouvoir utiliser le solveur extrait dans un contexte particulier.

Un système de contraintes (csp) est alors défini par un enregistrement composé d'une liste finie de variables (champ `X`), d'une liste finie de contraintes (`C`) et d'une table (`D`) associant à chaque variable son domaine, ici une liste finie de valeurs (dans la suite nous utilisons la notation pointée usuelle pour accéder aux différents champs). Un prédicat spécifie la bonne formation d'un csp : le domaine de définition de `D` est exactement `X`, les variables apparaissant dans les contraintes sont exactement celles de `X`, enfin les contraintes sont normalisées. Les deux premières contraintes sont généralement implicites dans la littérature, elles doivent être explicitées lorsqu'il s'agit de preuve formelle. De nombreux théorèmes prendront en

hypothèse la bonne formation du csp.

Il est classique de représenter un csp par un graphe de contraintes symétrique. Nous avons adopté cette vue dans notre formalisation Coq. Ainsi chaque contrainte c de variables x et y donne lieu à deux arcs, l'un reliant le sommet x au sommet y étiqueté par c (noté dans la suite $c(x, y)$) et l'autre reliant y à x également étiqueté par c (noté $c(y, x)$).

Nous avons implanté le solveur à l'aide de 3 fonctions récursives, l'une réalise le filtrage (`revise`), une autre la propagation (`ac3` qui appelle la précédente) et la dernière réalise l'énumération (`labeling` qui utilise `ac3`). Nous décrivons les deux premières à la figure 1. Ces fonctions suivent les présentations usuelles de la littérature adaptées au paradigme fonctionnel. La fonction `ac3` met en œuvre une récursion plus complexe que celle de `revise`, elle requiert explicitement une preuve de terminaison alors que la preuve de terminaison de `revise` est faite automatiquement par Coq. Hormis quelques fonctions auxiliaires et la fonction d'énumération non présentée ici, le code du solveur est complet. Le code extrait en OCaml correspond à cette partie à laquelle est ajoutée la traduction en OCaml des fonctions de la librairie utilisées (comme celles manipulant les listes et les tables).

Il nous faut exprimer la notion de consistance locale mise en œuvre dans ce processus de résolution, à savoir ici la consistance d'arc. Nous formalisons la propriété, sous la forme du prédicat `loc_consistent` $c\ x\ y\ D$, de la manière suivante : l'arc $c(x, y)$ est arc-consistant pour D si et seulement si pour toute valeur u de $D(x)$, il existe au moins une valeur (appelée support) v dans $D(y)$ telle que $c[x/u, y/v]$ (i.e la contrainte quand on a remplacé x par u et y par v) est satisfaite.

La correction du filtrage consiste à montrer que la consistance locale est établie après une étape de filtrage (voir le théorème *revise_loc_consistent* à la figure 1). Quant à la complétude, elle exprime que si on dispose d'une solution compatible avec une table de domaines D et si une étape de filtrage amène à réduire un domaine alors la solution initiale est toujours compatible avec les nouveaux domaines. On ne perd pas de solution ! La preuve repose sur un lemme qui consiste à montrer que les valeurs éliminées du domaine touché par le filtrage ne peuvent pas être éléments d'une solution. La correction d'`ac3` consiste à montrer que l'étape de propagation conduit à des domaines tels que tous les arcs du graphe des contraintes sont arc-consistants. La preuve de correction repose sur la correction du filtrage et aussi sur des lemmes relatifs à la fonction `visitagain`. Par exemple on démontre (A) que si l'arc $c(x, y)$ a provoqué une réduction du domaine de x alors la consistance des arcs qui ne sont pas dans `visitagain` $x\ y\ g$ n'est pas affectée.

La complétude de `ac3` consiste à montrer que cette fonction ne perd pas de solution ; elle utilise la complétude du filtrage.

En nous limitant au filtrage et à la propagation, le code Coq complet (définitions et preuves) représente environ 5300 lignes. L'adaptation à AC2001 requiert principalement de modifier la fonction `revise` de manière à lui faire gérer une table qui garde les supports. Les deux développements sont en fait deux instances d'un même développement modulaire, permettant de partager une grande partie du code.

4 Vers un solveur certifié pour la consistance de bornes

Lorsque les domaines des variables sont trop grands, la recherche d'un support pour chaque valeur d'un domaine peut devenir trop coûteuse. Une alternative est alors d'utiliser la consistance de bornes qui consiste à raisonner sur les bornes des domaines. Ainsi rechercher la consistance de bornes pour un arc $c(x, y)$ consiste à trouver la borne inférieure (resp. supérieure) du domaine de x, m_x (resp. M_x), telle que la contrainte soit vérifiée. Le domaine de chaque variable x est dorénavant représenté par le couple (m_x, M_x). La fonction de filtrage est redéfinie de manière à modifier éventuellement les bornes du domaine d'une variable. Quant à la fonction de propagation, elle reste identique, à l'adaptation près de la fonction `visitagain` : en effet après le filtrage de l'arc $c(x, y)$, la propagation requiert de visiter également l'arc inverse $c(y, x)$ (en plus de ceux nécessités par l'arc-consistance). Les preuves de correction et de complétude de la propagation restent identiques, si on peut fournir les preuves de correction et de complétude de la fonction de filtrage ainsi que celle de la propriété (A) énoncée à la section précédente.

Plus généralement, on obtient une formalisation (spécification et preuve) modulaire et générique de la fonction de propagation, paramétrée par la représentation des domaines, la notion de consistance locale et la fonction de filtrage. Elle peut être instanciée pour la consistance d'arc ou la consistance de bornes.

Jusque là, nous n'avons fait aucune hypothèse quant à la forme des contraintes. Or, dans le filtrage de $c(x, y)$, le domaine de y doit être parcouru pour trouver un support pour m_x et M_x, ce qui n'est pas efficace. En nous plaçant dans le cadre d'un langage de contraintes numériques, nous proposons de rendre ce filtrage plus efficace en utilisant des fonctions de projection qui permettent de mettre à jour les bornes des domaines des variables, sans parcourir les domaines. Afin de réutiliser les développements précédents nous conservons la méthode de filtrage arc par arc. Nous pouvons réutiliser toutes les définitions et les preuves

- x et y sont les variables de c, de domaine resp. dx et dy
- *revise c x y dx dy = (b, dx')*
si $b = true$, dx' est le domaine de x après filtrage ($dx' \subsetneq dx$), sinon $dx' = dx$.

```
Function revise c x y dx dy {...} :=
    match dx with
      nil ⇒ (false, dx)
    | v ::r ⇒ let (b, d) := revise c x y r dy in
              if exists_support c v dy
              then (b, v ::d)
              else (true, d)
    end.
```

avec *exists_support c v dy* = true si il existe $t \in dy$ tel que $c[x/u, y/t]$ soit satisfaite, false sinon.

```
Theorem revise_loc_consistent : ∀ csp c x y ,
    c ∈ csp.C → x ∈ (vars_of c) → y ∈ (vars_of c) →
    ∀ dx dy dx' b,
    csp.D(x) = Some dx → csp.D(y) = Some dy →
    revise c x y dx dy = (b, dx') →
      loc_consistent x y c (csp.D(x) ← dx').
```

avec $(D(x) \leftarrow dx')$, la table obtenue à partir en D en modifiant le domaine de x qui devient dx'.

- g : graphe des contraintes, D : table des domaines, qu : liste des arcs à visiter (*worklist*)
- Si découverte d'un domaine vide, alors *ac3 g D qu* = *None* sinon, *ac3 g D qu* = *Some D'* : la propagation se termine avec les domaines D'

```
Function ac3 g D qu { ...} :=
match qu with
| nil ⇒ Some (D)
| (x, c, y) : :r ⇒
  match D(x), D(y) with
  | Some dx, Some dy ⇒
    let (b, dx') := revise c x y dx dy in
      if b then
        if is_empty dx' then None
        else ac3 g (D(x) ← dx') (r ⊕ (visitagain x y g))
      else ac3 g D r
  | _, _ ⇒ None
  end
end
```

avec $\oplus$ = concaténation stricte de deux listes et *visitagain x y g* = liste des arcs à revisiter = liste de tous les arcs de la forme $c'(z, x)$ avec $z \neq y$
Appel initial : qu = liste de tous les arcs de g.

FIGURE 1 – `revise` et `ac3` en Coq

concernant la propagation moyennant les preuves requises sur le filtrage, ici reportées au niveau de chaque fonction de projection. Il convient en particulier de montrer que chaque fonction de projection assure la consistance de bornes et ne perd pas de solution. Ces preuves sont actuellement en cours.

5 Conclusion et perspectives

Ce papier résume les travaux que nous avons menés sur le développement d'un solveur de contraintes sur les domaines finis, formellement certifié en Coq. A notre connaissance, c'est la première fois qu'un tel développement est proposé. La difficulté de ces travaux réside dans la spécification méticuleuse de toutes les fonctions du solveur, et dans les preuves de correction et complétude associées. Une perspective à court terme de ce travail concerne le traitement de contraintes ternaires et N-aires, qui nécessite une révision partielle de la spécification. A plus long terme, nous envisageons d'utiliser ce solveur certifié dans le contexte de la vérification formelle de code.

Références

[1] M. Carlier, C. Dubois, and A. Gotlieb. A certified constraint solver over finite domains. In *Formal Methods (FM)*, volume 7436 of *LNCS*, pages 116–131, Paris, 2012.

[2] H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv : A constraint-programming framework for bounded program verification. *Constraints Journal*, 15(2) :238–264, 2010.

[3] A. Van Gelder. Verifying rup proofs of propositional unsatisfiability. In *Int. Symp. on Artificial Intelligence and Mathematics (ISAIM), Fort Lauderdale, USA*, 2008.

[4] A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Computational Logic (CL)*, pages 399–413, London, 2000.

[5] L. Langevine, P. Deransart, and M. Ducassé. A generic trace schema for the portability of cp(fd) debugging tools. In *Int. Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2003), Budapest, Hungary, Selected Papers*, volume 3010 of *LNCS*. Springer, 2004.

[6] S. Lescuyer and S. Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *Emerging Trends - Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[7] F. Maric. Formal verification of a modern sat solver by shallow embedding into isabelle/hol. *Theor. Comput. Sci.*, 411(50) :4333–4356, 2010.

[8] The Coq Development Team. *Coq, version 8.4*. Inria, August 2012. http://coq.inria.fr/.

Une approche basée sur les contraintes pour résoudre le problème d'arbre recouvrant de coût minimum avec contraintes de degré

Jean-Guillaume Fages[1] **Xavier Lorca**[1] **Lous-Martin Rousseau**[2]

[1] Ecole des Mines de Nantes, France
[2] CIRRELT - Ecole Polytechnique de Montréal, Canada
{jean-guillaume.fages,xavier.lorca}@mines-nantes.fr, louis-martin.rousseau@cirrelt.net

Résumé

Cet article présente un modèle de Programmation par Contraintes pour résoudre le problème de recherche d'un Arbre Recouvrant de Poids Minimum sous Contraintes de Degré (ARPMCD). Cette approche générale, basée sur une relaxation Lagrangienne, est compétitive avec la méthode dédiée de Da Cunha et Lucena qui fait état de l'art. De plus, elle améliore la résolution des instances à coûts aléatoires d'un à deux ordres de grandeur.

Abstract

This paper introduces a Constraint Programming model for solving the Degree Constrained Minimum Spanning Tree Problem. This general approach, based on a Lagrangian relaxation, is competitive with the state-of-the-art dedicated method of Da Cunha and Lucena and improves the resolution of random instances by one to two orders of magnitude.

1 Introduction

Etant donné un graphe non orienté $G = (V, E)$ muni d'une fonction de coût $f : E \rightarrow \mathbb{Z}$ et d'une fonction $d_{max} : V \rightarrow \mathbb{N}$, le problème de recherche d'un Arbre Recouvrant de Poids Minimum sous Contraintes de Degré (ARPMCD) consiste à trouver un Arbre Recouvrant de Poids Minimum (ARMP) dans G, tel que tout nœud $v \in V$ a, au plus, $d_{max}(v)$ voisins. Ce problème NP-difficile [9] peut survenir dans de nombreuses applications industrielles impliquant des réseaux. Les plus communes sont les application de conception de réseaux de télécommunication où des modules doivent être connectés ensemble, chacun d'eux ayant un nombre limité de ports de connexion disponibles. De telles restrictions peuvent être dues aux composants matériels utilisés mais aussi à des mesures de sécurité sur le réseau.

Plusieurs méthodes, exactes et heuristiques, ont été proposées pour résoudre le problème de l'ARPMCD [1, 2, 5, 6, 11, 16]. Actuellement, la meilleure approche de l'état de l'art est la méthode « *Relax and Cut and Branch and Cut* » (RCBC) [6], qui implique une relaxation Lagrangienne renforcée par l'ajout de coupes (de type « *Blossom* ») et d'une procédure de recherche locale qui démarre à chaud un algorithme de « *Branch and Cut* ». Il est important de remarquer que le problème d'ARPMCD n'est en général qu'un sous-problème des préoccupations industrielles, qui peuvent inclure de nombreuses contraintes annexes qu'il serait bon d'intégrer au modèle. Cependant, la recherche locale, proposée par [6], n'est pas effectuée par un solveur générique mais par un algorithme dédié intégré au cœur même de la relaxation Lagrangienne. D'une redoutable efficacité, une telle approche détériore les capacités d'évolutions de leur modèle. A l'opposé, la Programmation par Contraintes est une technologie qui permet une prise en compte aisée des spécificités de ce type d'applications [18].

Ce papier introduit une approche en Programmation par Contraintes qui permet de fait une intégration facile des contraintes annexes, pour résoudre le problème d'ARPMCD. La force du modèle proposé repose sur la complémentarité de ses composants : il inclut une optimisation du bas vers le haut et un propagateur de relaxation Lagrangienne, dont le filtrage est directement relié à la borne supérieure de la variable objectif, ce qui forme une combinaison puis-

sante. Ces raisonnements basés sur les coûts sont renforcés en considérant les propriétés structurelles du problème. Des propagateurs basés sur la structure du graphe réduisent l'espace de recherche alors que l'heuristique de branchement tend à créer de plus en plus de structure. Ce modèle améliore la résolution des instances dîtes aléatoire d'un à deux ordres de grandeur.

Cet article est organisé comme suit : Tout d'abord, la section 2 introduit le modèle de Programmation par Contraintes. Ensuite, la section 3 le compare à l'approche de l'état de l'art. Finalement, une brève conclusion est donnée en section 4.

2 Le modèle

Le problème d'ARPMCD peut être formalisé comme suit :

$$\begin{array}{lll} \textit{minimize} & \sum_{(i,j)\in G'} f(i,j) & (1)\\ \textit{subject to :} & arbre(G') & (2)\\ & |\{(i,j)\in G'\}| \leq d_{max}(i) \quad ,\forall i \in V & (3)\\ & G' = (V, E' \subseteq E) & (4) \end{array}$$

Notre modèle implique deux variables : une variable entière, $z \in \mathbb{Z}$, représente la fonction objectif (1) ; une variable graphe [7, 12, 13], notée G', représente la solution du problème, *i.e.* un arbre recouvrant. La définition du domaine initial de G' (4) indique que le graphe solution doit contenir tous les nœuds de G et un sous ensemble de ses arêtes. Une contrainte globale de graphe $arbre$ (2) assure que G' forme bien un arbre (nécessairement recouvrant puisque tous les nœuds sont obligatoires). Cette contrainte empêche G' de contenir des cycles avec un propageur similaire à celui de la contrainte $noCycle$ [4] et assure la connexité de G' grâce à un propagateur, basé sur l'algorithme de Tarjan [19], qui détecte et force les isthmes. Les contraintes de degré (3) sont prises en compte par une simple contrainte de cardinalité. Finalement, le lien entre G' et z est fait via une contrainte de type $scalaire$ (1).

2.1 Raisonnement sur les faibles degrés

En règle générale, plus le degré maximum est faible, plus l'instance est difficile [1, 6, 11]. Cependant, lorsque le degré maximum de quelques nœuds vaut un, cela apporte une certaine structure au graphe, qui peut alors être exploitée dans le cadre de la Programmation par Contraintes. On remarque ainsi que, si $|V| > 2$, deux nœuds dont le degré maximum vaut 1 ne peuvent pas être connectés par une arête, car la contrainte de degré les déconnecterait du reste du graphe (voir figure 1(b)). Cette règle peut être généralisée aux autres nœuds du graphe : considérons qu'à chaque fois qu'une arête $(i,j) \in G'$, $d_{max}(i) = 1$ et $d_{max}(j) \neq 1$, est forcée, le nœud i (alors saturé) est retiré du graphe et le degré maximum de j est décrémenté de 1. Alors, s'il reste encore au moins trois nœuds dans le graphe, dès que $d_{max}(j)$ devient égal à 1, on peut filtrer toutes les arêtes $(j,k) \in G'$ telles que $d_{max}(k) = 1$. La figure 1 illustre cette règle de filtrage : en appliquant cette procédure au graphe de la figure 1(c), celui-ci devient isomorphe au graphe de la figure 1(b). On en déduit que l'arête (D,E) est impossible. Nous ajoutons donc à notre modèle un propagateur qui applique ce raisonnement pour filtrer des arêtes impossibles.

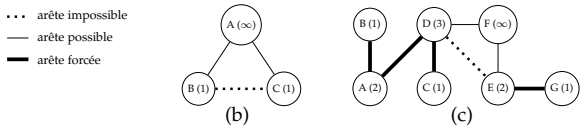


FIGURE 1 – Observation structurelle sur les degrés maximum des nœuds (affichés entre parenthèses).

2.2 Relaxation Lagrangienne

Malheureusement, la contrainte $scalaire$ maintenant la variable objectif ne va ni donner une bonne borne inférieure pour z, ni déclencher beaucoup de filtrage sur G'. Nous pourrions améliorer ce modèle en y ajoutant la contrainte *arbre recouvrant pondéré* [14, 15], plus puissante, mais cela ne serait toujours pas suffisant pour espérer battre les meilleures approches actuelles. Pour cette raison, nous introduisons un nouveau propagateur, *redondant* au modèle d'un point vue sémantique, qui applique la relaxation Lagrangienne du problème d'ARPMCD. En effet, des travaux similaires [3, 8] sur le problème du Voyageur De Commerce, qui peut être vu comme un cas particulier du problème d'ARPMCD, ont montré que la relaxation Lagrangienne basée sur un recouvrement de graphe par un arbre est très efficace. Cette relaxation est un processus itératif dans lequel chaque itération consiste à résoudre le problème d'ARPM et mettre à jour la fonction de coût en fonction des violations des contraintes de degré. Chaque itération pouvant donner lieu au filtrage décrit en [14, 15]. De plus amples informations sur cette relaxation peuvent être trouvée dans [1, 5].

Notre propagateur applique cette relaxation dans sa forme la plus classique, *i.e.* avec le simple algorithme de sous-gradient utilisé dans [1, 5]. Nous donnons en revanche quelques conseils macroscopiques qui permettent d'en obtenir une interprétation en Programmation par Contraintes qui soit pertinente et efficace : On pourrait chercher à filtrer entre chaque itération, ce qui serait long et perturberait la convergence de la méthode, tout comme on pourrait attendre la fin de la convergence et ne filtrer qu'une seule fois le domaine de G'. D'expérience, il est préférable d'opter pour un compromis en ne filtrant que toutes les k itérations (paramètre variable mais généralement compris entre 10 et 50). Une piste évoquée dans [17] consiste a enregistrer le filtrage que l'on peut effectuer à chaque itération, mais ne l'appliquer qu'une fois la convergence atteinte. Cette technique ne s'est pas avérée bénéfique sur ce problème car la complexité des algorithmes d'ARPM dépend du nombre d'arêtes dans G' : plus le filtrage est appliqué tôt, plus chaque itération est rapide. Il est crucial de rendre ce propagateur incrémental en conservant les

multiplicateurs Lagrangiens d'une propagation à l'autre : lorsque le propagateur est réveillé et qu'il exécute une relaxation Lagrangienne, les multiplicateurs Lagrangiens de la premiére itération sont ceux obtenus après convergence de la derniére propagation. On bénéficie ainsi du travail déjà effectué. Aussi, il est utile de distinguer le premier réveil du propagateur des autres appels. La première relaxation Lagrangienne requiert un grand nombre d'itérations pour converger au plus près de l'optimum. De plus, pour gagner en flexibilité durant l'algorithme du sous-gradient, il vaut mieux considérer un graphe initial complet, *i.e.* autoriser temporairement l'utilisation des arêtes déjà filtrées par les autres propagateurs (du moment que leur coût est défini). Par la suite, il est préférable de diminuer le nombre d'itérations et de ne considérer que les arêtes du domaine de G'. Enfin, puisque l'algorithme de sous-gradient nécessite une estimation supérieure à l'objectif pour converger, ce propagateur doit rester inactif tant qu'aucune solution n'a été trouvée.

2.3 Branchement

L'intervalle entre la première solution et l'optimum peut contenir un nombre considérable de solutions intermédiaires. Aussi, atteindre l'optimum en appliquant une optimisation par le haut (approche par défaut des solveurs de contraintes) risque de prendre un temps conséquent. Cependant, la relaxation Lagrangienne donne une borne inférieure de très bonne qualité (bien souvent à moins de 1% de la valeur optimale). Notre approche cherche donc une première solution, puis réinitialise la recherche en effectuant une optimisation par le bas : on cherche une solution de coût égal à la borne inférieure courante. Si aucune solution n'existe, on incrémente de un la borne inférieure. Il en découle que la deuxième solution trouvée est nécessairement optimale. Le gros de la recherche va se dérouler dans des régions non réalisables, aussi il est conseillé d'opter pour une approche de type « *fail-first* » [10] qui va tendre à provoquer les échecs. Pour cela, notre heuristique de branchement cherche à forcer les arêtes en dehors du support de la relaxation Lagrangienne. Cela fera travailler le propagateur Lagrangien et générera du filtrage. Nous renforçons enfin cette heuristique en cherchant à saturer les nœuds dont le degré maximum est à 1 aussi tôt que possible afin d'entretenir le propagateur défini en section 2.1. Pour résumer, notre heuristique se décompose comme suit :

1. trouver une première solution en forçant des arêtes de faible coût,
2. réinitialiser la recherche,
3. brancher sur z en l'instanciant à sa borne inférieure,
4. brancher sur G' en forçant des arêtes en dehors du support Lagrangien et qui, si possible, relient un nœud dont le degré maximum vaut 1.

3 Etude expérimentale

Cette section présente des expériences effectuées dans le but de positionner notre approche en Programmation par Contraintes par rapport à l'état de l'art. Elle démontre que notre modèle est compétitif avec la méthode dédiée de [6] et qu'elle offre de bonnes perspectives de passage à l'échelle sur de nombreuses instances.

Notre modèle a été implémenté en `Java`, avec le solveur `CHOCO 3.0`, et testé sur un Macbook pro sous OS X 10.7.2 avec un processeur Intel core i7 à 2.7 GHz et 8GB de mémoire DDR3. Une limite de 2 GB a été imposée sur la machine virtuelle Java (`JVM`). Malheureusement, le code de l'approche de l'état de l'art [6] n'est pas disponible, donc nous reportons directement les résultats de leur article. Leur code ayant été implémenté en `C`, l'écart de temps d'exécution des langages devrait compenser l'écart de performance des machines. Notre code est inclu dans la librairie d'exemples de `CHOCO 3.0`, il est donc accessible à tous.

Nous considérons deux jeux d'instances, nommés ANDINST et DR, qui font parti des plus difficiles de la littérature. Ils impliquent des graphes initiaux complets ayant de 100 à 2000 nœuds pour les instances ANDINST et de 100 à 600 nœuds pour les instances DR. Dans les instances de type ANDINST, chaque nœud est associé a un point du plan Euclidien, placé aléatoirement. La fonction de coût est la distance Euclidienne séparant les nœuds d'une arête. Le degré maximum de chaque nœud est ensuite choisi aléatoirement dans l'intervalle $[1, 4]$. Pour les instances de type DR, le coût de chaque arête a été choisi aléatoirement dans l'intervalle $[1, 1000]$, avec une distribution de probabilités uniforme, et le degré maximum de chaque nœud est choisi dans $[1, 3]$. Du point de vue des restrictions de degré, les instances DR sont donc plus difficiles.

3.1 Instances Euclidiennes : ANDINST

La figure 2 montre que l'approche par contraintes est compétitive avec l'état de l'art sur la plupart des instances Euclidiennes. Elle résout à l'optimum des instances impliquant jusqu'à deux millions d'arêtes, ce qui atteste d'une bonne aptitude au passage à l'échelle. Cependant, elle ne parvient pas à résoudre toutes ces instances dans le temps imparti (10^4 secondes). Il a été montré dans [6] qu'utiliser une relaxation linéaire augmentée de coupes de type « *Blossom* » apportait des gains significatifs sur les instances Euclidiennes. On pourrait donc s'attendre à améliorer l'approches par contraintes sur ce type d'instances, en ajoutant un propagateur qui applique cette relaxation.

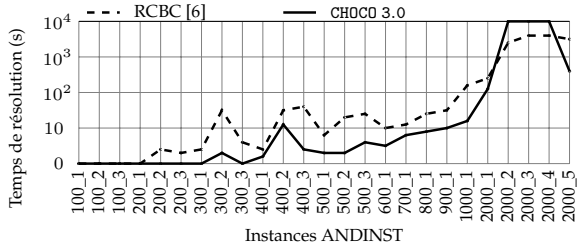


FIGURE 2 – Positionnement de l'approche par contraintes sur les instances Euclidiennes.

3.2 Instances aléatoires : DR

Concernant les instances aléatoires (figure 3), notre approche améliore les résultats antérieurs d'un à deux ordres de grandeur. Cela démontre la pertinence d'un modèle basé sur les contraintes pour résoudre cette classe d'instance, qui était dîte difficile à résoudre [6]. La relaxation Lagrangienne, qui n'était utilisée que dans une phase amont, devrait plutôt être appliquée durant tout le processus de recherche, comme le fait notre approche. Aussi, l'utilisation de recherche locale et de programmation linéaire a plus tendance à ralentir qu'à aider la résolution de ces instances.

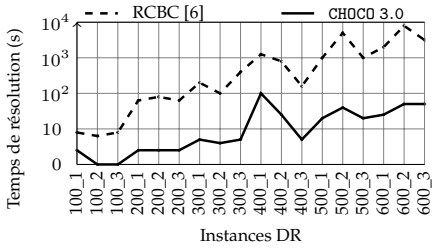


FIGURE 3 – Positionnement de l'approche par contraintes sur les instances aléatoires.

4 Conclusion

Nous avons proposé une première approche en Programmation par Contraintes pour résoudre le problème d'ARPM sous contraintes de degré. Ce modèle est plus flexible que la meilleure approche de l'état de l'art et il améliore significativement la résolution des instances aléatoires. Cette approche est donc appropriée pour concevoir une application industrielle amenée à évoluer.

Références

[1] Rafael Andrade, Abilio Lucena, and Nelson Maculan. Using lagrangian dual information to generate degree constrained spanning trees. *Discrete Applied Mathematics*, 154(5) :703–717, 2006.

[2] Markus Behle, Michael Jünger, and Frauke Liers. A primal branch-and-cut algorithm for the degree-constrained minimum spanning tree problem. In *WEA*, pages 379–392, 2007.

[3] Pascal Benchimol, Willem-Jan Van Hoeve, Jean-Charles Régin, Louis-Martin Rousseau, and Michel Rueher. Improved filtering for Weighted Circuit Constraints. *Constraints*, To appear.

[4] Yves Caseau and François Laburthe. Solving Small TSPs with Constraints. In *ICLP*, pages 316–330, 1997.

[5] Alexandre Salles da Cunha and Abilio Lucena. Lower and upper bounds for the degree-constrained minimum spanning tree problem. *Networks*, 50(1) :55–66, 2007.

[6] Alexandre Salles da Cunha and Abilio Lucena. A hybrid relax-and-cut/branch and cut algorithm for the degree-constrained minimum spanning tree problem. Technical report, Universidade Federal do Rio de Janeiro, 2008.

[7] Grégoire Dooms, Yves Deville, and Pierre Dupont. CP(Graph) : Introducing a Graph Computation Domain in Constraint Programming. In *CP*, volume 3709, pages 211–225, 2005.

[8] Jean-Guillaume Fages and Xavier Lorca. Improving the Asymmetric TSP by Considering Graph Structure. Technical report, Ecole des Mines de Nantes 12/4/INFO, 2012.

[9] Michael R. Garey and David S. Johnson. *COMPUTERS AND INTRACTABILITY*. W. H. Freeman and Company, victor klee edition, 1979.

[10] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. In *Proceedings of the 6th international joint conference on Artificial intelligence - Volume 1*, IJCAI'79, pages 356–364. Morgan Kaufmann Publishers Inc., 1979.

[11] Mohan Krishnamoorthy, Andreas T. Ernst, and Yazid M. Sharaiha. Comparison of algorithms for the degree constrained minimum spanning tree. *J. Heuristics*, 7(6) :587–611, 2001.

[12] Claude Le Pape, Laurent Perron, Jean-Charles Régin, and Paul Shaw. Robust and Parallel Solving of a Network Design Problem. In *CP*, volume 2470, pages 633–648, 2002.

[13] Jean-Charles. Régin. Tutorial : Modeling Problems in Constraint Programming. In *CP*, 2004.

[14] Jean-Charles Régin. Simpler and Incremental Consistency Checking and Arc Consistency Filtering Algorithms for the Weighted Spanning Tree Constraint. In *CPAIOR*, volume 5015, pages 233–247, 2008.

[15] Jean-Charles Régin, Louis-Martin Rousseau, Michel Rueher, and Willem Jan van Hoeve. The Weighted Spanning Tree Constraint Revisited. In *CPAIOR*, volume 6140, pages 287–291, 2010.

[16] Celso C. Ribeiro and Maurício C. Souza. Variable neighborhood search for the degree-constrained minimum spanning tree problem. *Discrete Applied Mathematics*, 118(1-2) :43–54, 2002.

[17] Meinolf Sellmann. Theoretical foundations of cp-based lagrangian relaxation. In *CP*, pages 634–647, 2004.

[18] Helmut Simonis. *Handbook of Constraint Programming, Constraint Applications in Networks*. Elsevier, victor klee edition, 2006.

[19] Robert E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972.

Compacité pratique des diagrammes de décision valués : normalisation, heuristiques et expérimentations*

H. Fargier[1], P. Marquis[2], N. Schmidt[12]

[1] IRIT, 118 route de Narbonne, 31062 Toulouse Cedex
[2] CRIL, rue Jean Souvraz, 62307 Lens Cedex
{fargier,schmidt}@irit.fr marquis@cril.fr

Résumé

Les diagrammes de décision valués (VDDs) sont particulièrement intéressants pour la compilation de problèmes de satisfaction de contraintes valuées (VCSPs). L'intérêt des différents langages de la famille VDD (en particulier, les langages ADD, SLDD, AADD) est qu'ils admettent des algorithmes en temps polynomial pour des traitements (comme l'optimisation) qui ne sont pas polynomiaux à partir des VCSPs de départ. Comme l'efficacité pratique de tels traitements dépend de la taille du VDD compilée obtenu, il est important d'obtenir une forme la plus compacte possible. Nous décrivons dans cet article quelques résultats issus de nos travaux sur la compacité expérimentale des VDDs. Nous présentons un compilateur ascendant de VCSPs en SLDD_+ et $\text{SLDD}_\times$, un jeu d'heuristiques d'ordonnancement des variables, ainsi que des procédures de traduction des langages SLDD_+ et $\text{SLDD}_\times$ vers les langages ADD et AADD. Les différents langages cibles et les heuristiques ont été testés sur deux familles de jeux d'essai, des VCSPs additifs représentant des problèmes de configuration de voitures avec fonctions de coût, et des VCSPs multiplicatifs représentant des réseaux bayésiens. Il apparaît que, bien que le langage AADD soit strictement plus succinct en théorie que SLDD_+ et $\text{SLDD}_\times$, ces deux langages conviennent bien en pratique quand il s'agit de compiler des problèmes de nature purement additive (respectivement purement multiplicative).

Abstract

Valued decision diagrams (VDDs) prove valuable data structures for compiling valued constraint satisfaction problems (VCSPs). Indeed, languages from the VDD family (especially, ADD, SLDD, AADD) benefit from polynomial-time algorithms for some tasks of interest (e.g., the optimization one) for which no polynomial-time algorithm exists when the input is the VCSP considered at start. Since the practical efficiency of such tasks depends in practice on the size of the compiled VDD, it is important to look for diagrams which as as compact as possible. In this paper we present some results issued from our work on the practical compactness of VDDs. We present a VCSP-to-SLDD_+ bottom-up compiler and a VCSP-to-$\text{SLDD}_\times$ bottom-up compiler, several variable ordering heuristics, and some translation procedures from SLDD_+ and $\text{SLDD}_\times$ to ADD and to AADD. The target languages and the heuristics under consideration have been tested on two families of benchmarks, additive VCSPs representing car configuration problems with cost functions and multiplicative VCSPs representing Bayesian nets. It turns out that even if the AADD language is strictly more succinct (from the theoretical side) than any of SLDD_+ or $\text{SLDD}_\times$, those two languages perform well in practice when purely additive (respectively, purely multiplicative) problems are to be compiled.

1 Introduction

Les diagrammes de décision – automates, diagrammes de décision binaires ordonnés (OBDD), diagrammes de décision multivalués (MDD) – sont particulièrement intéressants pour la compilation de problèmes de satisfaction de contraintes, et spécialement pour le type d'application qui nous intéresse, à savoir la compilation de problèmes de configuration de produit [3, 10, 12]. Cela dit, ces diagrammes ne permettent pas tels quels de représenter des fonctions de coût, ou plus généralement des fonctions associant

*Ce travail a bénéficié du support du projet BR4CP ANR-11-BS02-008 de l'Agence Nationale de la Recherche.

une valuation (coût, degré de satisfaction, probabilité, etc.) aux affectations de variables – ils ne permettent pas la compilation de problèmes de satisfaction de contraintes *valuées* (VCSPs). Dans un tel cadre, les diagrammes de décision valués (VDDs) – diagrammes de décision algébriques (ADDs) [4], diagrammes de décision à arcs valués (EVBDDs) [15, 14, 3], *Semiring Labeled Decision Diagrams* (SLDDs) [20], diagrammes de décision algébriques affines (AADDs) [18][17] sont alors des langages cibles pertinents. Les ADDs par exemple ont été utilisés pour la compilation de problèmes de planification [13] ; des travaux en configuration de produit [3, 11] ont proposé d'utiliser les EVBDDs (ou de manière équivalente, les SLDDs additifs – SLDD_+) pour capturer des fonctions de coût ou de préférences fortement additives, c'est-à-dire des cas où la valuation associée à une affectation des variables s'exprime directement par un VCSP dont toutes les contraintes souples sont unaires : l'idée est alors de compiler les contraintes dures qui définissent le produit configurable par un MDD, puis d'ajouter les valuations définies par les contraintes unaires directement sur les arcs.

L'intérêt de ces langages de compilation est qu'ils permettent une manipulation efficace de l'ensemble des solutions du VCSP, une fois compilé. La résolution interactive (c'est-à-dire par l'utilisateur) d'un problème de configuration avec fonction coût par exemple se rénduit en effet à des opérations de conditionnement, de propagation et d'optimisation qui sont linéaires dans la taille de la structure compilée. Ce qui est donc important en pratique, c'est d'obtenir une forme compilée la plus compacte possible. Plusieurs facteurs influencent cette compacité :

- la compacité théorique (ou "*succinctness*") qui est relative (elle indique s'il est possible de séparer exponentiellement ou non un langage d'un autre) et qui se focalise sur le pire cas ;
- la canonicité, c'est-à-dire la capacité de chaque langage à offrir pour chaque VCSP une forme canonique – ceci permet de reconnaître et de fusionner efficacement (par exemple, par un mécanisme de cache) les sous-diagrammes équivalents ;
- les heuristiques d'ordonnancement des variables choisies pour construire le diagramme de décision.

Les travaux de Sanner et Mc Allester ont montré que le langage AADD offre une forme canonique et est plus performant que le langage ADD du point de vue de la compacité théorique comme du point de vue pratique. Dans des travaux récents [**?**], nous avons montré que la propriété de canonicité à l'œuvre dans les AADD pouvaient être étendue, algébriquement, au langage SLDD, et qu'en théorie au moins, le langage AADD est plus succinct que SLDD_+ (resp. $\text{SLDD}_\times$), c'est-à-dire le langage des SLDDs fondé sur le semi-anneau commutatif $\langle \mathbb{R}^+ \cup \{+\infty\}, 0, +\infty, +, \min \rangle$ (resp. sur le semi-anneau commutatif $\langle \mathbb{R}^+, 1, 0, \times, \max \rangle$).

Nous décrivons dans la suite quelques résultats issus de nos travaux sur la compacité expérimentale des VDDs. Nous présentons un compilateur ascendant de VCSPs en SLDD_+ et $\text{SLDD}_\times$, un jeu d'heuristiques d'ordonnancement des variables, ainsi que des procédures de traduction des langages SLDD_+ et $\text{SLDD}_\times$ vers les langages ADD et AADD. Les différents langages cibles et les heuristiques ont été testés sur deux familles de jeux d'essai, des VCSPs additifs représentant des problèmes de configuration de voitures avec fonctions de coût, et des VCSPs multiplicatifs représentant des réseaux bayésiens. Il apparaît que, quoique le langage AADD soit strictement plus succinct en théorie que SLDD_+ et $\text{SLDD}_\times$, ces deux langages conviennent bien en pratique quand il s'agit de compiler des problèmes de nature purement additive (respectivement purement multiplicative).

2 Diagrammes de décision valués

Soit $X = \{x_1, \ldots, x_n\}$ un ensemble de variables où chaque $x_i \in X$ prend ses valeurs dans un domaine discret D_x ; on note D_X l'ensemble des affectations $\vec{x}$ de X. Un diagramme de décision α est une structure de données permettant de représenter une fonction f_α qui associe à chaque affectation $\vec{x} = \{(x_i, d_i) \mid d_i \in D_{x_i}, i = 1, \ldots, n\}$ un élément d'un ensemble E de valuations. E est à la base d'une structure de valuation $\mathcal{E}$ qui peut être plus ou moins riche d'un point de vue algébrique. Dans le formalisme ADD, aucune hypothèse n'est faite sur E (bien que l'on considère généralement que $E = \mathbb{R}$). Pour le langage AADD, $E = \mathbb{R}^+$.

Définition 1 *Un* diagramme de décision valué *(VDD) est un graphe orienté et acyclique avec une seule racine, où chaque nœud N est étiqueté par une variable $x \in X$: si $D_x = \{d_1, \ldots, d_k\}$, alors N a k arcs sortants $a_1, \ldots, a_k$, tels que chaque a_i est valué par $val(a_i) = d_i$.*
Les variables étiquetant les nœuds de tout chemin de la racine à une feuille sont toutes distinctes. Les nœuds N (resp. les arcs a) peuvent également être étiquetés par une valeur $\phi(N)$ (resp. $\phi(a)$) de E. On note $In(N)$ (respectivement $Out(N)$) les arcs entrants dans (respectivement issus de) N.

Les diagrammes de décision valués sont généralement *ordonnés* : un ordre total $<$ sur X est choisi de manière à ce que la suite des variables associées aux nœuds rencontrés sur chaque chemin de la racine vers une feuille soit compatible avec cet ordre.

Un diagramme de décision valué est dit *sous forme réduite* s'il ne contient pas de nœuds isomorphes (deux nœuds étiquetés par la même variable et dont les arcs sortants sont identiques, c'est-à-dire pointent sur les mêmes nœuds en portant la même valeur du domaine de leur variable et la même valuation ϕ). Tout VDD possède une unique forme réduite, qu'il est possible d'obtenir en temps linéaire en sa taille, soit par une procédure de réduction remontant de la/des feuille/s vers la racine, soit simplement par un mécanisme de cache (une "*unique table*"). Dans la suite, nous supposons que les diagrammes de décision considérés sont sous forme réduite.

Les ADDs sont la généralisation aux valuations non booléennes des OBDDs, les deux nœuds terminaux *true* et *false* étant remplacés par autant de nœuds que de valeurs de E associées à une affectation au moins.

Définition 2 *Un ADD est un VDD ordonné dont seuls les nœuds terminaux sont valués (les arcs ne le sont pas). Un ADD α associe à chaque affectation $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x}) \in E$ définie par :*

- *si α est un nœud terminal N étiqueté par un élément $\phi(N)$ de E, alors $f_\alpha(\vec{x}) = \phi(N)$;*
- *sinon, la racine N de α est étiquetée par $x \in X$; soient d la valeur de x dans $\vec{x}$, $a = (N, M)$ l'arc issu de N tel que $v(a) = d$, et β le ADD de racine M dans α ; on a pour tout $\vec{x} \in D_X$: $f_\alpha(\vec{x}) = f_\beta(\vec{x})$.*

Les nœuds terminaux reprenant l'ensemble des valeurs possibles de la fonction représentée, le nombre de nœuds d'un ADD croît avec le cardinal de l'ensemble de ces valeurs (l'image de la fonction représentée). Ainsi, la fonction $f(x_1, \ldots, x_n) = \Sigma_{i=1}^{n} 2^{i-1} x_i$ sur $\{0,1\}^n$, qui est représentable en espace polynomial par un VCSP fortement additif, prend 2^n valeurs différentes, d'où une taille exponentielle pour les ADDs qui la représentent.

Dans les SLDDs (*Semiring Labeled Decision Diagrams*) tels que définis dans [20], la structure de valuation doit être un semi-anneau $\mathcal{E} = \langle E, \otimes, \oplus, 1_s, 0_s \rangle$ – 1_s dénotant l'élément neutre de l'opérateur $\otimes$ et 0_s dénotant l'élément neutre de l'opérateur $\oplus$, absorbant pour $\otimes$. L'opérateur $\oplus$ n'a aucune influence pour la définition d'un SLDD en tant que représentation d'une fonction de D_X dans E ($\oplus$ est utilisé lorsque l'on veut calculer une valuation optimale, ou lorsque l'on veut éliminer une ou plusieurs variables). Pour cette raison, nous utilisons dans la suite une définition un petit peu plus générale que celle de [20], exigeant simplement une structure de monoïde pour $\mathcal{E} = \langle E, \otimes, 1_s \rangle$: $\otimes$ est une loi interne à E, associative, et qui possède un élément neutre 1_s.

Définition 3 *Un SLDD α sur X est un VDD avec une unique racine et un unique nœud terminal, dont les arcs sont étiquetés par des éléments de E où $\mathcal{E} = \langle E, \otimes, 1_s \rangle$ est un monoïde. Un SLDD associe à chaque affectation $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x})$ appartenant à E définie par :*

- *si α est le nœud terminal alors $f_\alpha(\vec{x}) = 1_s$;*
- *sinon, la racine N de α est étiquetée par $x \in X$; soient $d \in D_x$ la valeur de x dans $\vec{x}$, a l'arc issu de N tel que $v(a) = d$, M son extrémité et β le SLDD de racine M dans α ; on a pour tout $\vec{x} \in D_X$: $f_\alpha(\vec{x}) = \phi(a) \otimes f_\beta(\vec{x})$.*

À des fins de normalisation, on peut associer à α une valeur $\phi_0 \in E$ (son "offset"). La fonction "augmentée" que représente α est définie par, pour tout $\vec{x} \in D_X$, $f_{\alpha,\phi_0}(\vec{x}) = \phi_0 \otimes f_\alpha(\vec{x})$.

Deux monoïdes sont particulièrement intéressants : $\mathcal{E} = \langle \mathbb{R}^+ \cup \{+\infty\}, +, 0 \rangle$ pour tous les problèmes dont les valuations sont de nature additive (coûts) et $\mathcal{E} = \langle \mathbb{R}^+, \times, 1 \rangle$ pour tous les problèmes dont les valuations sont de nature multiplicative (probabilités). Les langages associés sont notés respectivement SLDD_+ et $\text{SLDD}_\times$. Chacun admet un élément absorbant (0 pour $\text{SLDD}_\times$, $+\infty$ pour les SLDD_+), ce qui permet de compiler des VCSPs possédant des contraintes dures : dans un SLDD_+ par exemple, toute affectation $\vec{x}$ telle que $f(\alpha)(\vec{x}) = +\infty$ est considérée comme non admissible car violant une contrainte dure.

Enfin, les diagrammes de décision algébriques affines introduits dans [18][17] permettent d'utiliser conjointement les opérateurs $\times$ et $+$ sur $\mathbb{R}^+$. Dans un SLDD, chaque arc a porte une valeur $\phi(a)$; dans un AADD, les arcs sont étiquetés par des *couples* de valeurs.

Définition 4 *Un AADD α sur X est un VDD ordonné avec une unique racine et un unique nœud terminal, dont les arcs sont étiquetés par des couples d'éléments de $\mathbb{R}^+$. α associe à chaque affectation $\vec{x} \in D_X$ la valeur $f_\alpha(\vec{x}) \in \mathbb{R}^+$ définie par :*

- *si α est le nœud terminal N, $f_\alpha(\vec{x}) = 1$;*
- *sinon, la racine N de α est étiquetée par $x \in X$; soient $d \in D_x$ la valeur de x dans $\vec{x}$, a l'arc issu de N tel que $v(a) = d$, M son extrémité, $\phi(a) = \langle q, f \rangle$ le couple de valeurs associée à a et β le AADD de racine M dans α ; on a pour tout $\vec{x} \in D_X$: $f_\alpha(\vec{x}) = q + f \times f_\beta(\vec{x})$.*

À des fins de normalisation, on attache à α un couple $\langle q_0, f_0 \rangle$ de $\mathbb{R}^+ \times \mathbb{R}^+$ (son "offset"). La fonction "augmentée" que représente α est définie par, pour tout $\vec{x} \in D_X$, $f_{\alpha,\langle q_0,f_0 \rangle}(\vec{x}) = q_0 + f_0 \times f_\alpha(\vec{x})$.

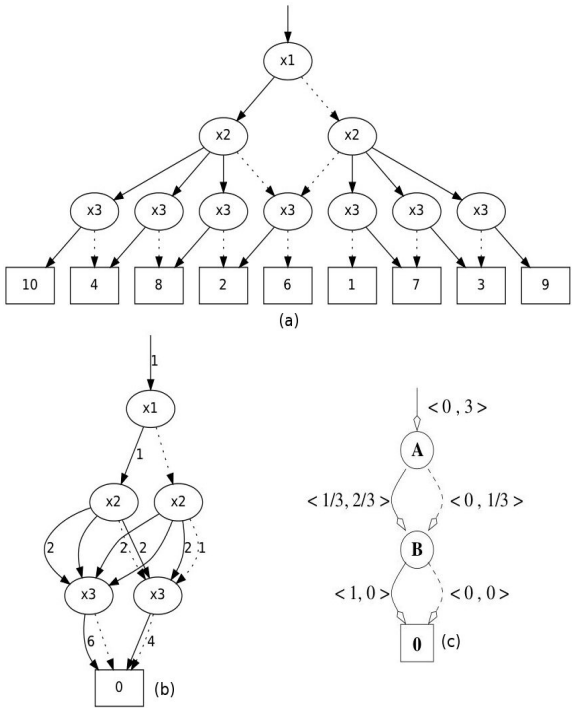


FIGURE 1 – (a) : Exemple de ADD ; (b) Exemple de SLDD_+ ; (c) Exemple de AADD

Un SLDD_+ peut être considéré comme un AADD particulier [1] dont les facteurs multiplicatifs portés par les derniers arcs (aboutissant au puits) sont égaux à 0 alors que tous les autres sont égaux à 1. De la même façon, un $\text{SLDD}_\times$ peut être considéré comme un AADD particulier dont les facteurs additifs sont nuls. Dans la suite, nous supposerons les SLDDs ordonnés selon un ordre $<$ sur X fixé ; cela est nécessaire d'une part pour pouvoir les comparer aux AADDs et aux ADDs qui sont des structures ordonnées, et d'autre part parce que cette propriété garantit une forme réduite canonique (et des opérations de combinaison en temps polynomial). La figure 2 décrit (a) un ADD, (b) un SLDD_+ et (c) un AADD. L'ADD et le SLDD_+ représentent la même fonction ; l'AADD représentant cette fonction serait très similaire au SLDD_+ décrit en (b). L'AADD donné à la figure 2 (c) représente la fonction $f = a + (a + 1)b$.

3 Normalisation

Le langage AADD est muni d'une procédure de normalisation qui permet de garantir, pour chaque fonction à représenter et étant donné un ordre des variables, une unique représentation réduite. Cette propriété joue un rôle important dans l'obtention de forme compacte, puisqu'elle permet de détecter, et donc de fusionner, les sous-diagrammes qui représentent la même sous-fonction.

1. En étendant E à $(\mathbb{R}^+ \cup \{+\infty\}) \times \mathbb{R}^+$.

Algorithme 1 : normalizeAADD$(\alpha, \langle q_0, f_0 \rangle)$

```
input  : Un AADD α et son "offset" ⟨q0, f0⟩
output : Un AADD normalisé équivalent à α
for each node N of α in inverse topological ordering do
    q_min ← min_{a∈Out(N)} q_a;
    range ← max_{a∈Out(N)}(q_a + f_a) − q_min;
    for each a ∈ Out(N) do
        if range > 0 then
            q_a ← (q_a − q_min)/range;
            f_a ← f_a/range;
        else // here, f_a = 0 and q_a = q_min
        q_a ← q_a − q_min;
    for each a ∈ In(N) do
        q_a ← q_a + q_min × range;
        f_a ← f_a × q_range;
q0 ← q0 + q_min × range;
f0 ← f0 × range;
return α;
```

3.1 Normalisation des AADD [17]

Soient un nœud N étiqueté par une variable x dont le domaine contient n valeurs, $Out(N) = \{a_1, a_2, \ldots, a_n\}$ l'ensemble des arcs sortants de N, et pour chaque $i \in 1, \ldots, n$, $\phi(a_i) = \langle q_i, f_i \rangle$ la valuation attachée à cet arc. N est *normalisé* ssi :

- $min(q_1, q_2, ..., q_n) = 0$;
- $max(q_1 + f_1, q_2 + f_2, ..., q_n + f_n) = 1$;
- le sous-AADD β sur lequel pointe un a_i est tel que si $\forall \vec{x}, f(\beta)(\vec{x}) = 0$, alors $f_i = 0$.

Un AADD est normalisé ssi tous ses nœuds le sont.

Tout AADD peut être normalisé en temps linéaire (voir algorithme 1). Cette procédure normalise les nœuds du puits vers la racine ; pour chaque N, elle calcule un "offset interne" $\langle q_{min}, range \rangle$, qui permet de normaliser N et puis est reporté sur les parents de N, ou sur l'offset de l'AADD si N est sa racine. Appliquée depuis le puits vers la racine, cette procédure remonte sur l'offset du diagramme les valeurs minimum (q_0) et maximum ($q_0 + f_0$) de la fonction représentée par l'AADD.

Les SLDD_+ et $\text{SLDD}_\times$ pouvant être vus comme des AADDs, nous pouvons nous appuyer sur l'existence d'une forme normalisée de ces structures pour proposer une définition similaire allégée de leur forme normalisée :

Algorithme 2 : normalizeSLDD(α, ϕ_0)

input : Un SLDD α, et son "offset" ϕ_0
output : Un SLDD normalisé équivalent à α
// For SLDD$_+$ let
// $\otimes = +, \otimes^{-1} = -, \oplus = min, 1_s = 0$
// For SLDD$_\times$ let
// $\otimes = \times, \otimes^{-1} = \oplus = max, 1_s = 1$
for *each node N of* α *in inverse topological ordering* do
  $\phi_m \leftarrow \oplus_{a \in Out(N)} \phi(a)$;
  // Rem: when $\oplus = max$, $\phi_m = 0$ iff
  // $\forall a, \phi(a) = 0$
  for *each* $a \in Out(N)$ do
    if $\phi(a) = \phi_m$ then $\phi(a) \leftarrow 1_s$;
    else $\phi(a) \leftarrow \phi(a) \otimes^{-1} \phi_m$;
  for *each* $a \in In(N)$ do
    $\phi(a) \leftarrow \phi(a) \otimes \phi_m$;
$\phi_0 \leftarrow \phi_0 \otimes \phi_m$;
return α;

Définition 5

- *Un SLDD$_+$ est normalisé ssi pour tout nœud N d'arcs sortants* $Out(N) = \{a_1, a_2, \ldots, a_n\}$, $min(\phi(a_1), \phi(a_2), ..., \phi(a_n)) = 0$.
- *Un SLDD$_\times$ est normalisé ssi pour tout nœud N d'arcs sortants* $Out(N) = \{a_1, a_2, \ldots, a_n\}$, $max(\phi(a_1), \phi(a_2), ..., \phi(a_n)) = 1$.

De même, on peut utiliser une spécialisation de la procédure de [17] pour normaliser un SLDD (voir algorithme 2). Dans un SLDD$_+$ (resp. SLDD$_\times$) normalisé, le minimum (resp. le maximum) de la fonction représentée est ainsi remonté à la racine en offset.

4 Construction de diagrammes de décision valués

À des fins expérimentales, nous avons implémenté un compilateur de VDDs. Ce compilateur permet de compiler d'une part des VCSPs à valuations additives [5] dont les contraintes sont exprimées par des tables tuple↦valuation (selon le format XCSP 2.1 décrit dans [16]) sous la forme de SLDD$_+$; et d'autre part de compiler des réseaux bayésiens sous la forme de SLDD$_\times$ (selon le format XML de [7]). Pour permettre l'obtention de différents types de VDDs, nous avons également implémenté des traductions de SLDD$_+$ et SLDD$_\times$ vers ADD et réciproquement, ainsi que des traductions de SLDD$_+$ et SLDD$_\times$ vers AADD.

4.1 Compilation

Notre approche de la compilation d'un VCSP en SLDD suit un procédé ascendant classique pour la construction de diagrammes de décisions ordonnés, valués ou non [17][6][2]. Nous décrivons ici la procédure d'ajout d'une contrainte valuée à un SLDD$_+$. L'ajout d'une contrainte C à un SLDD$_\times$ suit le même schéma, en remplaçant toute addition par une multiplication, toute soustraction par une division, et toute occurrence de l'élément neutre 0 par une occurrence de l'élément neutre 1.

On détermine tout d'abord un ordre total $<$ des variables selon lesquels le SLDD à construire sera ordonné (la section suivante décrit et compare les heuristiques d'ordonnancement de variables). On crée ensuite un SLDD "blanc ", c'est-à-dire que l'on associe à chaque variable un nœud dont chacun des arcs sortants (il y en a un par valeur dans le domaine de la variable) pointe vers le nœud portant la variable suivante selon l'ordre $<$ avec une valuation $\phi = 0$ (ou $\phi = 1$ pour un SLDD$_\times$). Les contraintes sont ajoutées une à une au SLDD en construction.

Dans un premier temps, lorsque la contrainte à ajouter contient un ϕ_0 par défaut (à appliquer à toutes les affectations qui ne sont pas explicitement dans la table de la contrainte), on met à jour l'étiquette de tout arc a sortant d'un nœud étiqueté par la dernière variable de C (selon $<$) via $\phi(a) \leftarrow \phi(a) + \phi_0$ (cf. algorithme 3). On ajoute ensuite chacun des tuples t de la table au SLDD courant, selon une procédure inspirée de la procédure *Apply*$(+)$ décrite dans [17] [2] – voir algorithme 4. Le SLDD résultant est normalisé et réduit, et devient le nouvel SLDD courant.

Algorithme 3 : AddConstraint(C, ϕ_0, α)

input : Un SLDD ordonné α, une contraint C, une valuation par defautϕ_0
output : Le SLDD α auquel on a ajouté la contrainte C

$x \leftarrow last(Var(C))$ //dernière var. de C selon $<$
for *all a sortant d'un nœud étiqueté par* x do
  $\phi(a) \leftarrow \phi(a) + \phi_0$
$y \leftarrow first(Var(C))$ //première var. de C selon $<$
for $(t, \phi_t) \in Table(C)$ do
  for *nœud* $N \in \alpha$ *t.q.* $Var(N) = y$ do
    $AddTuple(t, \phi_t, N)$
$reduce(normalizeSLDD(\alpha))$

2. Cette procédure étant elle-même une extension de la procédure de conjonction de OBDDs proposée dansr [6] au cas d'autres opérateurs ($\times$, $+$, min, *etc.*) que la conjonction.

Algorithme 4 : AddTuple(t, ϕ_t, α)

```
input  : Un SLDD ordonné α, un n-uplet t, une
         valuation φ_t
output : Le SLDD α auquel on a ajouté le tuple t
// ⊗ = × for SLDD_×; ⊗ = + for SLDD_+
x ← Var(N);
if x ∈ Var(t) then
    let a = (N, M) be the arc in Out(N)
    corresponding to value t[x];
    if x = last(Var(t)) then
        φ(a) ← φ(a) ⊗ φ_t ;
    else
        if M_t = null then
            M_t ← clone(M) // same successors
            // as those of M, reached by
            // copies of the arcs in Out(N)
            Out(a) ← M_t ;
        AddTuple(t, φ_t, M_t);
else
    for a = (N, M) ∈ Out(N) do
        if M_t = null then
            M_t ← clone(M) ;
            Out(a) ← M_t ;
        AddTuple(t, φ_t, M_t) ;
```

4.2 Heuristiques

La taille des diagrammes de décision classiques est évidemment très sensible à l'ordre dans lequel ses variables ont été ordonnées [2][8] – et c'est également le cas pour leurs versions valuées. Nous avons appliqué aux VDDs plusieurs heuristiques proposées pour les MDDs [2][8].

MCF [2] Afin de réduire la taille du diagramme de décision, il peut sembler intéressant de rencontrer les variables les plus contraintes (intervenant dans le plus de contraintes) le plus rapidement possible. On espère ainsi, lorsque le problème contient des contraintes dures, poster le plus tôt possible des arcs portant la valeur absorbante (n-uplet non admissible) – ces arcs joignant directement le puits, cela limite la taille de du VDD. L'heuristique MCF (pour *Most Constrained First*) trie les variables en fonction du nombre de contraintes dures dans lesquelles elles sont utilisées. $MCF(x) = |\{c \mid x \in Var(c)\}|$.

Band-Width [2] a montré que la *Band-Width* du graphe de contraintes permet de borner la taille du meilleur MDD décrivant un CSP classique. Soit $G = (X, C)$ un graphe de contraintes et $O : \{1, \ldots, n\} \mapsto X$ un ordre sur les n variables de X (O associe à chaque rang une variable). La *Band-Width*h d'un ordre O est : $BW(O) = max\{j - i$ t.q. $i < j$ et $\exists C, O[i], O[j] \in Var(C)\}$.
Pour un O donné, on place en tête les variables de plus fort BW dans l'ordre.

Le calcul d'un ordre de *Band-Width* minimum pour un graphe quelconque étant un problème NP-complet, nous utilisons un algorithme glouton pour l'approcher. On choisit les variables de l'ordre itérativement : la première sélectionnée (celle qui sera en tête du VDD) est la variable la plus contrainte. Étant donnée une suite O de k variables sélectionnées, la variable suivante est la variable x non encore sélectionnée qui maximise la quantité : $H_O(x) = max\{|O| - i \mid O[i]$ voisine de $x\}$.

MCS-Inv [19] MCS (pour *Maximum Cardinal Search*) est une méthode introduite dans le cadre de reconnaissance de graphes triangulés. La méthode MCS-Inv reprend le même principe en inversant l'ordre final. Elle permet aux variables fortement contraintes d'être proches des variables avec lesquelles elles sont liées. Notre implémentation utilise un algorithme glouton. On choisit les variables de l'ordre itérativement : la première sélectionnée est la variable la plus contrainte. Étant donnée une suite O de k variables sélectionnées, la variable suivante est la variable x non encore sélectionnée qui maximise la quantité $H_O(x) = \Sigma_{i=1,\ldots,k,O[i]}$ voisine de $_x|S| - i$.

Force [1] Le but de *Force* est de minimiser le *span* induit sur le graphe de contraintes, c'est-à-dire la somme (et non pas, comme pour l'heuristique *Band-Width*, le maximum) des distances séparant les variables appartenant à la même contrainte. $span(O) = \Sigma_C(j - i / i < j$ et $O[i], O[j] \in Var(C))$.
La méthode consiste à calculer, à partir d'un ordre quelconque, le "centre de gravité (COG)" de chacune des contraintes C (en fonction de la position des variables dans l'ordre).

$$COG(C) = \frac{\Sigma_{x \in Var(C)} POS(x)}{|Var(C)|}.$$

On remet à jour ensuite les positions de chacune des variables en fonction des centres de gravité de l'ensemble des contraintes auxquels elles appartiennent.

$$POS(x) = \frac{\Sigma_{C, x \in Var(C)} COG(c)}{|\{C, x \in Var(C)\}|}.$$

Cette procédure part d'un ordre quelconque sur les variable O : à l'initialisation, $POS(x)$ est le rang de x dans O. Elle est répétée autant de fois que nécessaire, jusqu'à arriver à un point fixe, où, d'une itération à l'autre, les estimations de centres de gravités des va-

riables n'évoluent plus. On réordonne ensuite les variables par POS croissant.

4.3 Traductions ADD⇌SLDD⇌AADD

Nous développons ici les principes qui fondent les traductions d'un type de VDD vers chacun des autres (en supposant que tous partagent le même domaine de valuation, typiquement $E = \mathbb{R}^+ \cup \{+\infty\}$). Ces traductions peuvent souvent être vues comme l'application de procédures de normalisation.

Traductions SLDD↦AADD, ADD↦SLDD, ADD↦AADD

La traduction la plus évidente est la traduction d'un SLDD_+ (ou d'un $\text{SLDD}_\times$) en AADD : tout SLDD_+ peut être transformé en AADD en remplaçant, pour chaque arc a à destination du puits son étiquette $\phi(a)$ par le couple $\langle \phi(a), 0 \rangle$, et pour tout autre arc a', son étiquette $\phi(a')$ par le couple $\langle \phi(a'), 1 \rangle$. On normalise ensuite l'AADD obtenu, ce qui permet éventuellement de le réduire et d'obtenir une structure de plus faible taille. De la même façon, tout $\text{SLDD}_\times$ peut être transformé en AADD en remplaçant, pour chaque arc a, son étiquette $\phi(a)$ par le couple $\langle 0, \phi(a) \rangle$. On normalise et réduit ensuite l'AADD obtenu, ce qui permet éventuellement d'obtenir une structure plus compacte que le diagramme original.

De la même façon, on peut toujours transformer un ADD en SLDD_+ ou en $\text{SLDD}_\times$ en reportant les valuations portées par les nœuds terminaux sur leurs arcs entrants (les autres arcs portant la valuation $\phi = 0$ lorsque l'on veut construire un SLDD_+ , la valuation $\phi = 1$ lorsque l'on veut construire un $\text{SLDD}_\times$) ; les nœuds terminaux sont remplacés par le puits du nouveau SLDD. On normalise et réduit ensuite le diagramme obtenu selon les principes de normalisation des SLDD_+ (respectivement des $\text{SLDD}_\times$).

La procédure est la même lorsqu'il s'agit de transformer un ADD en AADD, les valuations ϕ_i des nœuds terminaux étant reportées sur leurs arcs entrants sous la forme d'un couple $\langle 0, \phi_i \rangle$.

Traductions SLDD↦ADD, AADD↦ADD, AADD↦SLDD

Transformer un SLDD en ADD revient à repousser les valuations ϕ vers les arcs du dernier niveau. En quelque sorte, il s'agit d'une normalisation assurant que pour tout a, $\phi(a)$ est égal à l'élément neutre (0 pour les SLDD_+ et 1 pour les $\text{SLDD}_\times$) ; il faut alors, pour porter les valuations, dupliquer le puits en autant de nœuds finaux que de valuations différentes sur ses arcs entrants. La procédure de "normalisation" du

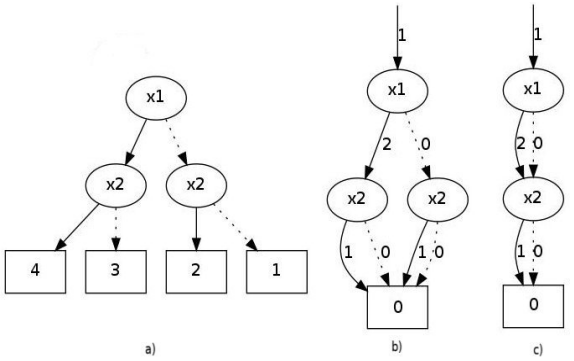


FIGURE 2 – Exemple de traduction d'un ADD (a) en SLDD_+ : en (b) les poids sont remontés sur les arcs et les nœuds sont normalisés ; en (c) les nœuds isomorphes sont fusionnés

SLDD en ADD procède de la racine vers le puits ; plus précisément, pour tout nœud N dont les parents ont été traités :

- remplacer N par autant de copies N_1, ... N_k de N que de valuations différentes $\phi_1, \ldots, \phi_k$ sur les arcs entrant dans N ;
- si N est le puits du diagramme de décision, chaque copie N_i porte la valuation ϕ_i ;
- sinon (N est un nœud interne), pour chaque copie N_i, et chaque arc a de N vers un nœud M, créer un arc a' étiqueté $Val(a)$ de N_i vers M, prenant pour valeur $\phi(a') = \phi(a) + \phi_i$;
- normaliser et réduire le diagramme en fusionnant les nœuds isomorphes.

Le même type de procédure est appliqué pour transformer un AADD en SLDD_+, c'est-à-dire en AADD dont les valuations sont de la forme $\langle q, 1 \rangle$: on crée autant de copies N_1, ..., N_k de N que de facteurs multiplicatifs différents f_1, ..., f_k sur les arcs a_j entrant dans N. Chaque copie N_i est liée aux prédécesseurs de N par des copies a' des arcs a entrant dans N et portant la valuation f_i : chaque arc a' porte la même valeur du domaine de sa variable que a et sa valuation est $\langle q_a, 1 \rangle$; le facteur multiplicatif de a est reporté sur les arcs suivants : pour chaque copie N_i, et chaque arc a de N vers un nœud M, étiqueté par $\langle q, f \rangle$, on crée un arc a' étiqueté $Val(a)$ de N_i vers M, prenant pour valeur $\langle q' \times f_i, f' \times f_i \rangle$.

Pour obtenir un $\text{SLDD}_\times$ plutôt qu'un SLDD_+, on "normalise" le diagramme affine de manière à assurer que toutes les valuations sont de la forme $\langle 0, f \rangle$. Pour cela, on crée autant de copies N_1, ..., N_k de N que de facteurs $q_1 + f_1$, ..., $q_k + f_k$ différents sur les arcs a_j entrant dans N. Chaque copie a' d'un $a \in In(N)$ porte la valuation $\langle 0, f_a \rangle$; le facteur additif est reporté sur

TABLE 1 – Comparaison des heuristiques MCF, *Band-Width*, MCS-Inv et *Force*

	MCF		*Band-Width*		MCS-Inv		*Force*	
Instance	nœuds(arcs)	tps(s)	nœuds(arcs)	tps(s)	nœuds(arcs)	tps(s)	nœuds(arcs)	tps(s)
VCSP↦SLDD$_+$								
SmallPriceOnly	105(239)	< 1	40(120)	< 1	36(108)	< 1	351(2946)	< 1
MedPriceOnly	777(2009)	< 1	312(906)	< 1	169(499)	< 1	3362(40474)	1
BigPriceOnly	-	t-o	97646(251594)	4002	3317(9687)	18	8873 (824180)	499
Small	3100(7564)	1	4349(10451)	2	2344(5584)	1	4503 (12430)	2
Medium	5660(19363)	11	11700(30835)	18	6234(17062)	7	13603(34263)	32
Big	-	t-o	-	t-o	198001(925472)	79043	-	t-o
Bayes↦SLDD$_\times$								
Cancer	13(25)	<1	13(25)	<1	13(25)	<1	13(25)	<1
Asia	36(71)	<1	29(57)	<1	23(45)	<1	25(49)	<1
Car-starts	77(157)	<1	41(83)	<1	41(83)	<1	49(99)	<1
Alarm	-	t-o	5852(14899)	<1	1301(3993)	<1	7054(20134)	<1
Hailfinder25	-	t-o	-	t-o	32718(108083)	8	m-o	-

les arcs suivants : pour chaque copie N_i, et chaque arc a de N vers un nœud M, étiqueté par $\langle q, f\rangle$, on crée un arc a' étiqueté $Val(a)$ de N_i vers M, prenant pour valeur $\langle q' + \frac{q_i}{f_i}, f'\rangle$. Finalement, les étiquettes des arcs entrant dans le puits sont mises sous la forme $\langle q+f, 0\rangle$.

Enfin, pour obtenir un ADD plutôt qu'un SLDD à partir d'un AADD, on crée autant de copies N_i de N que de valeurs $q_i + f_i$ différentes sur les arcs entrant dans N et le facteur $q_i + f_i$ est repoussé sur les arcs sortant des N_i : les étiquettes $\langle q', f'\rangle$ des arcs a' deviennent $\langle q_i + f_i \times q', f_i \times f'\rangle$.

Notons que ces transformations peuvent faire "exploser" le diagramme, ce qui est inévitable dans le pire cas : la fonction $f(x_1, \ldots, x_n) = \Sigma_{i=1}^{n} 2^{i-1} x_i$ sur $\{0,1\}^n$ par exemple, prend 2^n valeurs différentes ; sa représentation par un ADD posséde donc 2^n feuilles, alors qu'elle peut être représentée par un SLDD$_+$ (et donc par un AADD) à $n+1$ nœuds et $2n$ arcs. On peut également montrer que cette fonction ne peut pas être représentée par un SLDD$_\times$ de taille polynomiale. Formellement, le langage AADD est strictement plus succinct que celui des SLDD$_+$, lui-même strictement plus succinct que ADD. Il est également strictement plus succinct que le langage SLDD$_\times$ lui-même strictement plus succinct que ADD [9].
Le nombre d'opérations effectuées lors de la transformation en AADD et d'un AADD vers un autre langage augmente le risque d'erreurs d'arrondis (on trouve en effet plus facilement dans les AADD des additions entre deux valeurs d'ordres de grandeur différents, dont le résultat sera nécessairement arrondi, oubliant la valeur la plus faible). C'est pourquoi dans les faits, nous évitons les traductions de AADD vers SLDD et ADD. Par exemple, transformer un SLDD$_+$ en SLDD$_\times$ (ou inversement), on passe par le langage ADD et non AADD. Même si le passage en ADD n'est pas toujours possible (à cause de l'explosion en espace), cela évite d'obtenir un résultat erroné. Le passage de SLDD$_\times$ vers SLDD$_+$ (en passant par un ADD) peut aussi entraîner des erreurs d'arrondis s'il y a de trop gros écarts d'ordres de grandeur en sortie d'un SLDD$_\times$.

5 Résultats expérimentaux

Dans cette section, nous comparons expérimentalement, d'une part, l'efficacité des différentes heuristiques, et, d'autre part, la compacité pratique des différents types de VDDs décrits dans les sections précédentes.

Nous avons testé nos structures de données et heuristiques sur deux familles de jeux d'essai, des VCSP additifs (codant des problèmes de configuration de véhicule) et des VCSP multiplicatifs (codant des réseaux bayésiens). En ce qui concerne les réseaux bayésiens, nous avons utilisé des jeux d'essai standard au format XML [7]. Les instances de CSP pondérés représentant des problèmes de configuration de voitures nous ont été fournies par Renault – elles sont disponibles via [**?**]. Ces instances sont composées de contraintes dures, définissant les modèles de voitures faisables (la diversité de la gamme, en termes automobiles) ainsi que de contraintes valuées représentant les coûts, le prix d'un véhicule étant la somme des coût spécifiés par les différentes contraintes valuées. Trois jeux d'essai nommés **Small**, **Medium** et **Big** représentent trois modèles différents de voitures (deux citadines et un utilitaire). Les caractéristiques des jeux d'essai sont les suivantes :

- **Small** : #variables=139 ; #taille du domaine max=16 ; #contraintes=176 (incluant 29 de coût)
- **Medium** : #variables=148 ; #taille du domaine max=20 ; #contraintes=268 (incluant 94 de coût)
- **Big** : #variables=268 ; #domaine max=324 ; #contraintes=2157 (incluant 1825 de coût)

Enfin, les problèmes **Small Price Only**, **Medium Price Only** et **Big Price Only** sont constitués des

TABLE 2 – Compilation de problèmes de configuration en $SLDD_+$, ADD, $SLDD_\times$ et AADD.

VCSP	$SLDD_+$		ADD	$SLDD_\times$	AADD
Instance	nœuds (arcs)	temps (s)	nœuds (arcs)	nœuds (arcs)	nœuds (arcs)
SmallPriceonly	36 (108)	< 1	4364 (7439)	3291 (7439)	36 (108)
MediumPriceonly	169 (499)	< 1	37807 (99280)	33595 (99280)	168 (495)
BigPriceonly	3317 (9687)	18	m-o	-	3317 (9687)
Small	2344 (5584)	1	299960 (637319)	14686 (33639)	2344 (5584)
Medium	6234 (17062)	6	752466 (2071474)	129803 (314648)	6234 (17062)
Big	198001 (925472)	79043	m-o	-	198001 (925472)

TABLE 3 – Compilation de réseaux bayésiens en $SLDD_+$, ADD, $SLDD_\times$ et AADD.

Réseaux bayésiens	$SLDD_\times$		ADD	$SLDD_+$	AADD
Instance	nœuds (arcs)	temps (s)	nœuds (arcs)	nœuds (arcs)	nœuds (arcs)
Cancer	13 (25)	< 1	38 (45)	23 (45)	11 (21)
Asia	23 (45)	< 1	415 (431)	216 (431)	23 (45)
Car-starts	41 (83)	< 1	42741 (64029)	19632 (39265)	38 (77)
Alarm	1301 (3993)	< 1	m-o	-	1301 (3993)
Hailfinder25	32718 (108083)	8	m-o	-	32713 (108063)

seules contraintes de coût des instances **Small**, **Medium** et **Big**, respectivement (les contraintes dures sont omises).

5.1 Efficacité spatiale des heuristiques

Pour tester l'efficacité des différents heuristiques, nous avons mesuré, pour chacune d'elles, le temps de compilation de chaque instance en SLDD, ainsi que le nombre de nœuds et d'arcs du diagramme résultant. Les problèmes de configuration avec fonction de coût ont été compilés en $SLDD_+$, les instances de réseaux bayésiens ont été compilés en $SLDD_\times$. Les résultats sont présentés à la table 1. Les notations t-o et m-o signifient respectivement *time-out* (>24h) et *out of memory* (>128Mb). Toutes les expérimentations ont été effectués sur un processeur de 800MHz.

Il apparaît que MCS-Inv est généralement l'heuristique la plus performante, tant du point de vue de la taille du diagramme généré que de celui du temps de calcul. Les résultats de MCF sont mauvais lorsqu'il n'y a que des contraintes souples (problèmes **Price Only** et réseaux bayésiens), car cette heuristique n'est intéressante qu'appliquée à des contraintes dures. Néanmoins, les performances de MCF sont supérieurs à ceux de *Band-width* (au moins pour la taille) sur les problèmes alliant contraintes souples et contraintes dures, et même supérieurs à ceux de MCS-Inv en taille sur le problème **Medium**. Notons également la faiblesse de *Force* en toutes circonstances. Au vu de ces résultats, nous avons choisi d'utiliser MCS-Inv pour la suite des expérimentations.

5.2 Efficacité spatiale des types de VDD

Nous avons voulu aussi comparer la taille des diagrammes obtenus selon les différents langages considérés dans l'article. Les jeux d'essai ont été compilés sous la forme de SLDD ($SLDD_+$ pour les VCSP et $SLDD_\times$ pour les réseaux bayésiens) puis traduits dans les autres langages. Les tables 2 et 3 indiquent les tailles obtenues pour la représentation des problèmes de configuration et des réseaux bayésiens sous la forme de ADD, $SLDD_+$, $SLDD_\times$ et AADD.
Ces expérimentations confirment les hypothèses de compacité des différents langages. En effet, on retrouve bien que le langage ADD (le moins succinct) est toujours moins compact que $SLDD_+$, $SLDD_\times$ et AADD. À l'inverse, l'AADD est toujours au moins aussi compact que $SLDD_+$, $SLDD_\times$ et ADD. Il s'avère qu'un langage offre en pratique une bonne compacité si celui-ci intègre l'opérateur adéquat au type d'instance considéré. Ainsi les langages $SLDD_+$ et AADD qui intègrent l'addition sont plus efficaces spatialement pour la compilation de VCSP dont les contraintes sont de nature additive (portant sur un prix), alors que $SLDD_\times$ et AADD sont plus compacts pour la compilation de réseaux bayésiens, où les contraintes sont de nature multiplicative (tables de probabilités) [3]. . À l'inverse, un opérateur non pertinent n'apporte que peu, voire pas, d'amélioration. Ainsi, la comparaison entre les différents langages de type SLDD et le langage AADD nous montre que l'utilisation d'un deuxième opérateur n'apporte pas, dans les problèmes purement additifs ou purement multiplicatifs, d'amélioration pratique en terme de compacité. L'utilisation d'un AADD par rapport à un $SLDD_+$ (resp. $SLDD_\times$) n'apporte pas de

3. Lors de la compilation de réseaux bayésiens en ADD et AADD, nous obtenons des représentations nettement moins succinctes que celles décrites dans [17]. Ceci s'explique par le choix que nous avons fait de représenter les valeurs réelles avec une précision plus importante. Dans le cas des ADDs, le nombre de valeurs finales possible explose clairement quand la précision augmente.

réelle amélioration pour la compilation des problèmes de configuration avec coût (resp. réseaux bayésiens)

6 Conclusion

Dans cette article nous avons étudié la compilation vers et la traduction entre différents types de diagrammes de décision valués. Du point de vue théorique le langage AADD peut être vu comme une généralisation des langages SLDD_+ et $\text{SLDD}_\times$, langages qui généralisent eux-mêmes le langage ADD : AADD est donc, parmi eux, le langage de compilation le plus succinct. Les expérimentations montrent que, dans la pratique, pour la compilation de problèmes de nature purement additive (respectivement purement multiplicative), seul l'opérateur + (respectivement $\times$) est utile, et que l'utilisation de SLDD_+ (respectivement $\text{SLDD}_\times$) suffit amplement à la compilation du problème original. Nous avons également étudié plusieurs heuristiques, et vérifié que les meilleures tendaient au regroupement des variables incluses dans un même groupe de contraintes. Dans nos travaux futurs, nous prévoyons d'étudier la complexité et d'implémenter des requêtes et des transformations de VDDs, afin de pouvoir utiliser ces structures de données pour traiter efficacement (et avec une garantie de temps de réponse) des problèmes de configuration en ligne avec maintien d'un indicateur de coût minimal.

Références

[1] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. Force : a fast and easy-to-implement variable-ordering heuristic. In *ACM Great Lakes symposium on VLSI*, GLSVLSI '03, 2003.

[2] Jérôme Amilhastre. *Représentation par automate d'ensemble de solutions de problèmes de satisfaction de contraintes*. PhD thesis, Université de Montpellier II, 1999.

[3] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs application to configuration. *Artif. Intell.*, 135(1-2) :199–234, 2002.

[4] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. Algebraic decision diagrams and their applications. In *Proc. of ICCAD'93*, pages 188–191, 1993.

[5] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based csps and valued csps : Frameworks, properties, and comparison. *Constraints*, 4(3) :199–240, 1999.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35 :677–691, 1986.

[7] Fabio Gagliardi Cozman. JavaBayes Version 0.347, Bayesian Networks in Java, User Manual. Technical report, dec 2002. Benchmarks at http ://sites.poli.usp.br/pmr/ltd/ Software/javabayes/Home/node3.html.

[8] Rolf Drechsler. Evaluation of static variable ordering heuristics for mdd construction. In *ISMVL*, pages 254–260, 2002.

[9] Helene Fargier, Pierre Marquis, and Nicolas Schmidt. Semiring labelled decision diagrams, revisited : Canonicity and spatial efficiency issues, irit research report rr–2013-20–fr. Technical report, IRIT, 2013.

[10] Tarik Hadzic. A bdd-based approach to interactive configuration. In *CP*, page 797, 2004.

[11] Tarik Hadzic and Henrik Reif Andersen. A BDD-based polytime algorithm for cost-bounded interactive configuration. In *AAAI*, pages 62–67, 2006.

[12] Tarik Hadzic, Rune Jensen, and Henrik Reif Andersen. Calculating valid domains for bdd-based interactive configuration. *CoRR*, 0704.1394, 2007.

[13] Jesse Hoey, Robert St-Aubin, Alan J. Hu, and Craig Boutilier. SPUDD : Stochastic planning using decision diagrams. In *UAI*, 1999.

[14] Yung-Te Lai, Massoud Pedram, and Sarma B. K. Vrudhula. Formal verification using edge-valued binary decision diagrams. *IEEE Trans. on Computers*, 45(2) :247–255, 1996.

[15] Yung-Te Lai and Sarma Sastry. Edge-valued binary decision diagrams for multi-level hierarchical verification. In *DAC*, pages 608–613, 1992.

[16] Olivier Roussel and Christophe Lecoutre. XML Representation of Constraint Networks : Format XCSP 2.1. Technical report, CoRR abs/0902.2362, feb 2009.

[17] Scott Sanner and David A. McAllester. Affine algebraic decision diagrams (AADDs) and their application to structured probabilistic inference. In *Proc. of IJCAI'05*, pages 1384–1390, 2005.

[18] Paul Tafertshofer and Massoud Pedram. Factored edge-valued binary decision diagrams. *Formal Methods in System Design*, 10(2/3), 1997.

[19] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3) :566–579, July 1984.

[20] Nic Wilson. Decision diagrams for the computation of semiring valuations. In *IJCAI*, 2005.

Raffinement de décompositions arborescentes par fusion de clusters pour guider DGVNS

Mathieu Fontaine[1,2] **Samir Loudni**[1,2] **Patrice Boizumault**[1,2]

[1] Université de Caen Basse-Normandie, UMR 6072 GREYC, F-14032 Caen, France
[2] CNRS, UMR 6072 GREYC, F-14032 Caen, France
{prénom.nom}@unicaen.fr

Résumé

Dans nos précédents travaux, nous avons proposé la méthode DGVNS (Decomposition Guided VNS) permettant d'exploiter les clusters, issus de la décomposition arborescente du graphe de contraintes, pour guider l'exploration des voisinages dans les méthodes de type VNS. Dans cet article, nous proposons deux schémas de raffinement de la décomposition arborescente permettant de limiter la redondance entre clusters, afin de guider DGVNS vers des voisinages plus pertinents. A cet effet, nous proposons deux critères : le premier exploite la taille du plus grand séparateur, alors que le second prend en considération à la fois la taille des séparateurs et la taille des clusters. Les expérimentations menées sur des instances aléatoires (GRAPH) et des instances réelles (RLFAP, SPOT5 et tagSNP) montrent la pertinence et l'intérêt de notre approche.

Abstract

In this paper, we propose two schemes for refining tree decomposition in order to reduce the redundancy between clusters and to guide DGVNS (Guided Decomposition VNS) to more relevant neighborhoods structures. The main idea is to increase the proportion of proper variables in clusters by merging those having few proper variables in the tree decomposition. For this aim, we propose two criteria to merge clusters : the first one uses the size of the largest separator, while the second one takes into account both the size of the separators and the size of clusters. Experiments conducted on several difficult instances (RLFAP, SPOT5, GRAPHand tagSNP) show the relevance and the interest of strengthening the quality of neighborhoods in DGVNS.

1 Introduction

La notion de décomposition arborescente, proposée par Robertson et Seymour [24], vise à découper un problème en sous-problèmes (*clusters*) constituant un graphe acyclique. Chaque cluster correspond à un sous-ensemble de variables fortement connectées. Chaque sous-problème, étant plus petit que le problème original, devient plus facile à résoudre. L'intérêt d'exploiter les propriétés structurelles d'un problème a été attestée dans divers domaines : pour vérifier la satisfiabilité dans SAT [1, 14, 23] pour résoudre les CSP (CTE [7]), dans les réseaux bayésiens (AND/OR graph search [21]), en bases de données relationnelles [12, 13], pour les problèmes d'optimisation sous contraintes (BTD [27], Lc-BTD$^+$ [5], RDS-BTD [25], DB [19]). Toutes ces propositions exploitent la décomposition arborescente dans des méthodes de recherche complète.

Dans nos précédents travaux [9, 10], nous avons proposé la méthode DGVNS (Decomposition Guided VNS) permettant d'exploiter les clusters issus de la décomposition arborescente du graphe de contraintes, pour guider l'exploration des voisinages dans les méthodes de type VNS. Les expérimentations menées sur des instances aléatoires (GRAPH) et des instances réelles (CELAR, SPOT5 et tagSNP) montrent la pertinence de notre approche comparée à VNS/LDS+CP [20] ou à ID-Walk [22]. Cependant, pour la plupart de ces instances, les décompositions obtenues par MCS [1] comportent de nombreux clusters qui se chevauchent très fortement. De plus, ces clusters contiennent peu ou pas de variables propres. Cette forme de redondance entre clusters limite l'effort de diversification de DGVNS, en considérant plusieurs fois des voisinages très proches.

Dans cet article, nous proposons deux raffinements de la décomposition arborescente, permettant de limiter la redondance entre clusters, afin de guider

1. Maximum Cardinality Search, [26].

DGVNS vers des voisinages plus pertinents. L'idée sous-jacente est d'augmenter la proportion de variables propres dans les clusters, en fusionnant ceux qui sont globalement redondants dans la décomposition arborescente. A cet effet, nous proposons deux critères. Le premier consiste à fusionner les clusters partageant plus de variables qu'un seuil maximal fixé. Le second, basé sur la notion *d'absorption*, consiste à fusionner les clusters ayant un taux d'absorption (i.e. proportion de variables partagées) supérieur à un seuil maximal fixé. Les expérimentations menées sur des instances aléatoires (GRAPH) et des instances réelles (RLFAP, SPOT5 et tagSNP) montrent la pertinence et l'intérêt de fusionner les clusters globalement redondants pour renforcer la qualité des voisinages explorés par DGVNS.

La section 2 présente le contexte de nos travaux. La section 3 détaille nos deux schémas de raffinement de la décomposition arborescente. Les sections 4 et 5 présentent les jeux de test et les résultats expérimentaux. Enfin, nous concluons et présentons des perspectives.

2 Contexte et Définitions

2.1 Réseau de fonctions de coût

Un réseau de fonctions de coût (CFN) est un couple (X, W) où $X=\{x_1, \ldots, x_n\}$ est un ensemble de n variables et W est un ensemble de e fonctions de coût. Chaque variable $x_i \in X$ a un domaine fini D_i de valeurs qui peuvent lui être affectées. La taille du plus grand domaine est notée d. Une affectation de x_i à la valeur $a \in D_i$ est notée (x_i, a). Pour un sous-ensemble de variables $S \subseteq X$, on note D^S le produit cartésien des domaines des variables de S. Pour un n-uplet donné t, $t[S]$ représente la projection du n-uplet t sur l'ensemble de variables S. Une affectation *complète* $t=(a_1, ..., a_n)$ est une affectation de toutes les variables ; dans le cas contraire, on l'appelle affectation *partielle*. Une fonction de coût $w_S \in W$, de portée $S \subseteq X$, est une fonction $w_S : D^S \mapsto [0, k_\top]$ où $k_\top$ est un coût entier maximum (fini ou non) utilisé pour représenter les affectations interdites (exprimant des contraintes dures). Pour capturer fidèlement les contraintes dures, les coûts sont combinés par l'addition bornée $\oplus$, définie par $\alpha \oplus \beta = \min(k_\top, \alpha + \beta)$. Le problème consiste à trouver une affectation complète t de l'ensemble des variables minimisant la combinaison des fonctions de coût $\bigoplus_{w_S \in W} w_S(t[S])$.

2.2 Décomposition arborescente

Le graphe de contraintes d'un CFN est un graphe $G=(X,E)$ composé d'un sommet par variable et il existe une arête $\{u, v\} \in E$ si, et seulement si, $\exists\, w_S \in W, u, v \in S$.

Définition 1 *Une décomposition arborescente [24] de $G=(X, E)$ est un couple (C_T, T) où $T = (I, A)$ est un arbre avec pour ensemble de nœuds I et pour ensemble d'arêtes A, et $C_T = \{C_i \mid i \in I\}$ est une famille de sous-ensembles de X (appelés clusters) telle que :*
- $\cup_{i \in I} C_i = X$,
- $\forall (u, v) \in E$, $\exists\, C_i \in C_T$ *t.q.* $u, v \in C_i$,
- $\forall i, j, k \in I$, *si* j *est sur le chemin de* i *à* k *dans* T, *alors* $C_i \cap C_k \subseteq C_j$.

Nous notons $\texttt{parent}(C_i)$ (resp. $\texttt{fils}(C_i)$) le parent (resp. l'ensemble des fils) de C_i dans T.

Définition 2 *L'intersection entre deux clusters est appelée séparateur et notée $sep(C_i, C_j)$. Deux clusters C_i et C_j sont adjacents ssi $sep(C_i, C_j) \neq \emptyset$. Les variables appartenant à un et un seul cluster sont appelées variables propres.*

Définition 3 *Un graphe de clusters, pour une décomposition arborescente (C_T, T), est un graphe non-orienté $G_T = (C_T, E_T)$ dont les sommets sont les éléments de C_T et il existe une arête $(C_i, C_j) \in E_T$ entre les sommets C_i et C_j ssi $sep(C_i, C_j) \neq \emptyset$.*

La largeur d'une décomposition arborescente $T=(I, A)$ est définie par $w^-(T)=\max_{i \in I}(|C_i|-1)$. La largeur de décomposition $tw(G)$ d'un graphe G est la plus petite largeur de toutes les décompositions arborescentes possibles de G. Calculer la largeur de décomposition d'un graphe est un problème NP-complet [2]. Cependant, des heuristiques reposant sur la notion de *triangulation* de graphe [2] permettent le calcul de décompositions approchées. Elles fournissent un majorant de la largeur de décomposition.

Dans cet article, nous utilisons l'heuristique *Maximum Cardinality Search* (MCS) [26] qui constitue un bon compromis entre la largeur de la décomposition obtenue et le temps nécessaire à son calcul [17].

2.3 Decomposition Guided VNS

DGVNS (Decomposition Guided VNS) [9, 10] étend le principe de VNDS [15] (Variable Neighborhood Decomposition Search), en exploitant le graphe de clusters pour guider l'exploration de grands voisinages.

Les voisinages sont obtenus en désaffectant une sous-partie de la solution courante selon une heuristique de choix de variables. La reconstruction de la solution sur les variables désinstanciées est effectuée par une recherche arborescente partielle, LDS (*Limited discrepancy search*, [16]), aidée par la propagation des contraintes (CP) basée sur un calcul de minorants.

2. Un graphe est cordal ou triangulé si et seulement si tous ses cycles de taille supérieure à quatre ont une corde (une arête connectant deux sommets non-adjacents du cycle).

Algorithm 1: Pseudo-code de DGVNS

```
   function DGVNS(X, W, k_init, k_max, δ_max);
   begin
1      let G be the constraints graph of (X, W) ;
2      let (C_T, T) be a tree decomposition of G ;
       let C_T = {C_1, C_2, ..., C_p} ;
3      S ← genInitSol() ;
4      k ← k_init ;
5      i ← 1 ;
6      while (k < k_max) ∧ (notTimeOut) do
7          C_s ← CompleteCluster(C_i, k) ;
8          X_un ← Hneighborhood(C_s, N_{k,i}, S) ;
9          A ← S\{(x_i, a) | x_i ∈ X_un} ;
10         S' ← Rebuild(A, X_un, δ_max, f(S), S) ;
11         NeighbourhoodChangeDGVNS(S, S', k, i);
12     return S ;
   procedure NeighbourhoodChangeDGVNS (S, S', k, i);
   begin
13     if f(S') < f(S) then
14         S ← S' ;
15         k ← k_init, i ← succ(i) ;
16     else k ← k + 1 ; i ← succ(i) ;
```

Définition 4 *Soit G_T=($\mathcal{C}_T$,E_T) le graphe de clusters associé à G. Soient $C_i \in \mathcal{C}_T$ un cluster de G_T et $k \in [1 \dots n]$ la dimension du voisinage. La structure de voisinage $N_{k,i}$ désigne l'ensemble des combinaisons de k variables parmi C_i.*

L'algorithme 1 présente le pseudo-code de DGVNS. Tout d'abord, il construit une décomposition arborescente de G (ligne 2), puis génère aléatoirement une solution initiale S (fonction genInitSol, ligne 3). Afin de favoriser les mouvements dans des régions fortement liées, DGVNS se base sur les structures de voisinages $N_{k,i}$ (cf. Définition 4). En effet, le concept de cluster permet d'exhiber ces régions, de part sa taille (plus petite que le problème initial), et de part la forte connexion entre les variables qu'il contient. Ainsi, l'ensemble de variables candidates C_s à désaffecter sont sélectionnées à partir du cluster C_i. Si ($k > |C_i|$), C_s est étendu aux variables des clusters C_j voisins de C_i afin de prendre en compte la topologie du graphe de clusters. Ce traitement est réalisé par la fonction CompleteCluster(C_i, k) (ligne 7). De plus, grâce à la forte connexion entre les variables de C_s, l'étape de reconstruction pourra bénéficier d'un plus fort filtrage et d'un meilleur calcul de minorants. Un sous ensemble X_{un} de k variables est sélectionné aléatoirement dans C_s parmi les variables en conflit par l'heuristique de voisinage Hneighborhood (line 8). Une affectation partielle $\mathcal{A}$ est générée à partir de la solution courante S en désaffectant les k variables de X_{un} (ligne 9). Ensuite, ces variables sont reconstruites (ligne 10) par une recherche arborescente partielle LDS [16], aidée par la propagation de contraintes (CP) (voir [20] pour plus de détails). La recherche s'arrête dès que la dimension maximale de voisinage k_{max} ou le *TimeOut* est atteint (ligne 6).

La procédure NeighborhoodChangeDGVNS contrôle les mécanismes d'intensification et de diversification de DGVNS (cf. Algorithme 1). Soit p le nombre total de clusters, *succ* une fonction de succession [3], et $N_{k,i}$ la structure de voisinage courante. Si LDS+CP ne trouve par de meilleure solution S' dans le voisinage de S, DGVNS cherche des améliorations dans $N_{(k+1),succ(i)}$ (la structure de voisinage où $(k+1)$ variables de C_s seront désaffectées) (ligne 16). Tout d'abord, la diversification réalisée par le déplacement du cluster C_i au cluster $C_{succ(i)}$ permet de favoriser l'exploration de nouvelles parties de l'espace de recherche et de rechercher de meilleures solutions dans celles-ci. Ensuite, quand un optimum local est atteint dans le voisinage courant, l'augmentation de la dimension du voisinage k permet aussi de diversifier la recherche en explorant de plus grandes régions.

Quand une solution de meilleure qualité S' est trouvée par LDS+CP dans le voisinage $N_{k,i}$, S' devient la solution courante (ligne 14), k est réinitialisé à k_{init} et le prochain cluster est considéré (ligne 15). En effet, rester dans le même cluster rend plus difficile la découverte de nouvelles solutions : se déplacer sur un nouveau cluster permet la diversification de la recherche autour de la nouvelle solution S'.

3 Raffinements par fusion de clusters

Comme nous l'avons indiqué dans la Section 1, la plupart des décompositions arborescentes, obtenues par MCS sur les différents problèmes étudiés, comportent de nombreux clusters contenant peu ou pas de variables propres, car se chevauchant très fortement. Cette forme de redondance entre clusters limite l'effort de diversification de DGVNS, en considérant, de manière répétée, des voisinages très proches.

Afin de limiter la redondance entre clusters, et donc de renforcer la qualité des structures de voisinage explorées par DGVNS, nous proposons de fusionner les clusters redondants dans la décomposition arborescente. L'idée sous-jacente est d'augmenter la proportion de variables propres dans les clusters. À cet effet, nous proposons deux critères : (*i*) la taille du plus grand séparateur (s_{max}), et (*ii*) le taux d'absorption maximal (Ab_{max}). Nous désignerons par DGVNS-*smax* (resp. DGVNS-*abs*) la méthode DGVNS exploitant la décomposition arborescente obtenue par le critère s_{max} (resp. le critère Ab_{max}). Notons que toutes les opérations de fusion sont effectuées à l'issue de la décomposition (cf. ligne 2, algorithme 1).

3. si $i < p$ alors $succ(i) = i + 1$ sinon $succ(p) = 1$.

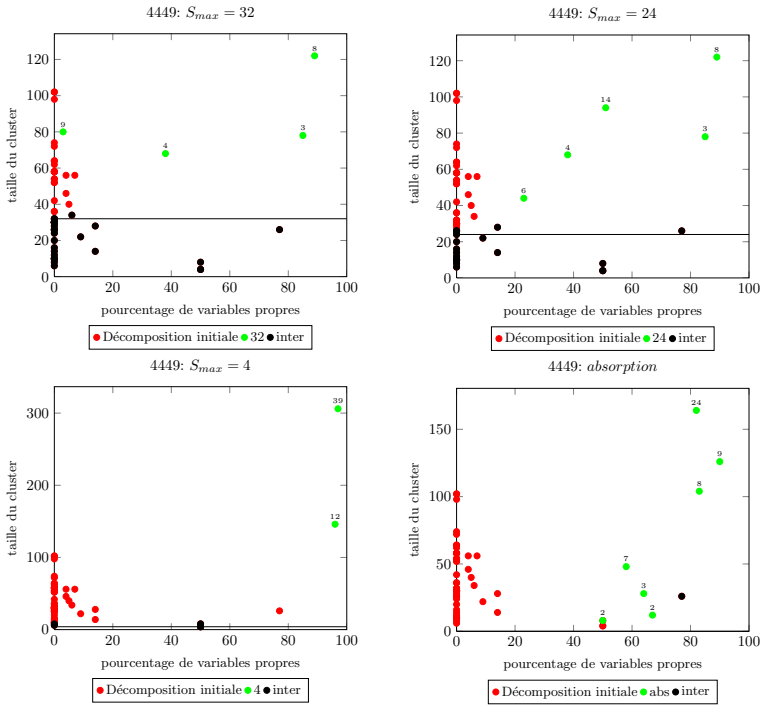


FIGURE 1 – Comparaison des décompositions obtenues par les deux méthodes de fusion sur une instance `tagSNP`.

3.1 Fusion de clusters basée sur s_{max}

La problématique de fusion de clusters est au coeur des méthodes complètes exploitant les décompositions arborescentes [6]. Dans [18], les auteurs proposent de fusionner les clusters en se basant sur le critère s_{max} afin d'augmenter le degré de liberté pour les heuristiques de choix de variables au sein des clusters. Dans [25], les auteurs utilisent le même principe pour réduire la complexité spatiale de la méthode utilisée.

Afin de générer de nouvelles décompositions à partir de celles obtenues par `MCS`, nous avons suivi le schéma proposé dans [18], en bornant la taille maximale des séparateurs s_{max}. Tout d'abord, nous calculons une décomposition arborescente en fixant s_{max} à $+\infty$ (cas où la taille des séparateurs n'est pas limitée). Ensuite, en partant des feuilles de la décomposition produite par `MCS`, nous fusionnons les clusters C_i qui partagent plus de s_{max} variables avec leur parent $\texttt{parent}(C_i)$. Nous obtenons ainsi une nouvelle décomposition, dans laquelle aucun séparateur ne contient plus de s_{max} variables.

Cette approche souffre cependant de sa rigidité. En effet, le critère de fusion ne permet pas de prendre en compte la structure de l'instance considérée : limiter s_{max} à 32 sur des instances de problèmes ayant des séparateurs de grande taille conduira à fusionner tous les clusters. De plus, avec ce critère, on pourra fusionner deux clusters de taille 100 partageant 32 variables mais pas ceux de taille 32 partageant 31 variables. Pour remédier à ces inconvénients nous proposons une fusion basée sur la notion *d'absorption*.

3.2 Fusion de clusters basée sur l'absorption

Afin de tenir compte des tailles respectives des clusters et de leurs séparateurs, dans le mécanisme de fusion, nous introduisons un second critère appelé *absorption*.

Définition 5 *Soit (C_T, T) une décomposition arborescente. Soit C_i et C_j deux clusters de C_T. L'absorption de C_i par C_j, notée $Ab(C_i, C_j)$, est définie par :*

$$Ab(C_i, C_j) = \max\left(\frac{|Sep(C_i, C_j)|}{|C_i|}, \frac{|Sep(C_i, C_j)|}{|C_j|}\right)$$

Soit Ab_{max} le taux d'absorption maximal autorisé, et min_{size} la taille minimale autorisée pour un cluster. La fusion s'effectue sur la décomposition obtenue par `MCS`. En partant des feuilles de cette décomposition, tous les clusters $< C_i, \texttt{parent}(C_i) >$ ayant un taux d'absorption supérieur à Ab_{max} ou tels que $|C_i| < min_{size}$ sont fusionnés.

Pour nos expérimentations (cf. Section 5), la taille minimale d'un cluster (min_{size}) a été fixée à la valeur 5 et Ab_{max} à 70%. Ainsi, les clusters partageant un peu plus de 2/3 de leurs variables seront fusionnés.

3.3 Comparaison des deux méthodes de fusion

Cette section compare, de manière qualitative, les décompositions obtenues par les deux méthodes de fusion sur une instance du problème `tagSNP` et sur une instance du problème `SPOT5` (cf. Section 4).

Les résultats de ces comparaisons sont présentés aux Figures 1 et 2. Pour chaque figure, les 3 premiers graphiques correspondent à la fusion s_{max} et le dernier correspond à la fusion par absorption. Chaque graphique indique, pour une valeur fixée de s_{max}, (i) la répartition des clusters de la décomposition initiale obtenue par `MCS`, et (ii) la répartition des clusters obtenus par fusion, en fonction de leur taille et de leur pourcentage de variables propres. Les points rouges représentent les clusters de la décomposition initiale, les points verts les clusters obtenus après fusion, et les points noirs les clusters communs. Chaque point vert indique aussi le nombre de clusters qui ont été fusionnés pour son obtention.

Considérons les résultats obtenus pour l'instance `tagSNP` #4449 (cf. Fig. 1). Pour (s_{max}=32), 24 clusters ont été fusionnés en 4 nouveaux clusters, dont un partageant plus de 90% de ses variables et un autre contenant plus de 120 variables. Mais, un grand nombre de clusters ayant 100% de leurs variables partagées n'ont pas été fusionnés. (s_{max}=24) permet de réduire quelque peu cette redondance. Par contre, la fusion par absorption permet de supprimer la totalité des clusters redondants et de générer une décomposition contenant des clusters ayant au moins 50% de variables propres. Par ailleurs, sur cette instance, $s_{max} = 14$. On peut dresser les mêmes constats sur la très grande majorité des autres instances `tagSNP`.

Considérons les résultats obtenus pour l'instance `SPOT5` #507 (cf. Fig. 2). Pour (s_{max}>12), les décompositions produites sont très similaires à la décomposition initiale. Celles-ci sont caractérisées par des clusters qui se recouvrent très fortement. Pour ($s_{max}\leq$12), les décompositions obtenues ont des largeurs arborescentes plus élevées, mais les clusters contiennent plus de variables propres. À l'inverse, la fusion par absorption produit une décomposition constituée de beaucoup plus de clusters qui se chevauchent moins fortement (i.e., $s_{max} = 10$). Par ailleurs, la valeur de la largeur de décomposition est deux fois moins élevée comparée à celle obtenue pour (s_{max}=4). On peut dresser les mêmes constats sur une grande majorité des autres instances `SPOT5`.

Les résultats obtenus montrent clairement l'apport de la fusion par absorption. En effet, pour des valeurs de s_{max} élevées, la fusion modifie peu la décomposition initiale. En revanche, pour des valeurs de s_{max} faibles, les décompositions produites ont peu de clusters et possèdent des largeurs arborescentes très élevées. De plus, la plupart des clusters partageant 100% de leurs variables sont conservés. À l'inverse, la fusion par absorption permet de produire des décompositions de bien meilleure qualité : des clusters de taille raisonnable ayant une forte proportion de variables propres. On observe toutefois une augmentation relative de la largeur de décomposition sur certaines instances.

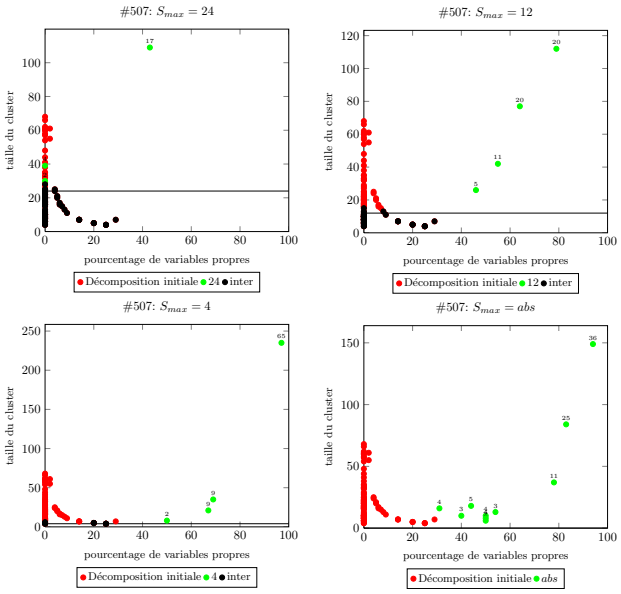


FIGURE 2 – Comparaison des décompositions obtenues par les deux méthodes de fusion sur une instance `SPOT5`.

4 Jeux de test

Les expérimentations ont été menées sur différentes instances de quatre problèmes différents.

- Instances `RLFAP` : Le `CELAR` (Centre d'électronique de l'Armement) a mis à disposition un ensemble d'instances du problème d'affectation de fréquences radio (`RLFAP`) [4]. L'objectif est d'assigner un nombre limité de fréquences à un ensemble de liens radios entre des couples de sites, afin de minimiser les interférences dues à la réutilisation des fréquences. Nous reportons les résultats sur les instances les plus difficiles : Scen06, Scen07 et Scen08.

- Instances `GRAPH` : Le générateur `GRAPH` (Generating Radio link frequency Assignment Problems Heuristically) a été développé par le projet CALMA [28] afin de proposer des instances aléatoires ayant une structure proche des instances `RLFAP`.

- Instances `SPOT5` : La planification quotidienne d'un satellite d'observation de la terre (`SPOT5`) consiste à sélectionner les prises de vue à effectuer dans la journée en prenant en compte les limites matérielles du

Inst.	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	Taille C_i moy.	max.
Scen06	4	50/50	368	3,389	45	4.1	20
$e = 1222$	$+\infty$	**50/50**	**112**	**3,389**	55	4.9	12
$S^* = 3,389$							
Scen07	4	5/50	1,908	344,215	76	4.1	52
$n = 200$	8	8/50	2,379	343,803	96	4.64	39
$e = 2,665$	12	7/50	2,400	343,601	100	4.85	35
$S^* = 343,592$	16	5/50	2,305	343,795	105	5.31	26
	24	5/50	2,272	348,640	109	5.72	23
	$+\infty$	**40/50**	**317**	**345,614**	110	5.72	23
Scen08	4	-	-	314	180	3.98	66
$n = 458$	8	-	-	314	228	4.5	65
$e = 5,286$	12	-	-	313	243	4.88	43
$S^* = 262$	16	-	-	312	251	5.17	31
	24	-	-	312	258	5.49	27
	$+\infty$	**3/50**	**1,811**	**275**	259	5.51	27
Graph06	4	50/50	287	4,123	9	24.22	193
$n = 200$	8	50/50	256	4,123	36	11.47	164
$e = 1,970$	12	50/50	262	4,123	41	11.32	160
$S^* =4,123$	16	50/50	277	4,123	43	11.4	158
	24	50/50	263	4,123	49	12.43	151
	32	**50/50**	**230**	**4,123**	50	12.74	150
	$+\infty$	50/50	367	4,123	123	36.07	63
Graph11	4	-	-	31,286	8	45.5	331
$n = 340$	8	-	-	24,685	45	13.6	292
$e = 3417$	12	-	-	23,453	60	12.73	277
$S^* = 3,080$	16	-	-	23,353	63	12.75	274
	24	-	-	23,238	71	13.48	265
	32	-	-	23,144	73	13.9	263
	$+\infty$	**8/50**	**3,046**	**4,234**	191	65.04	118
Graph13	4	-	-	19,104	7	67.0	453
$n = 458$	8	3/50	2,428	12,010	62	13.32	397
$e = 4,915$	12	2/50	2,893	12,053	74	12.77	385
$S^* = 10,110$	16	2/50	3,129	12,192	75	12.8	384
	24	1/50	3,473	11,999	83	13.42	376
	32	**7/50**	**3,247**	**11,766**	85	13.76	374
	$+\infty$	-	-	22,489	262	83.98	155

TABLE 1 – Comparaison de DGVNS pour différentes valeurs de s_{max} sur les instances RLFAP et GRAPH.

Inst.	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	Taille C_i moy.	max.
#408	4	49/50	361	6,228	17	13.88	127
$n = 200$	**8**	**49/50**	**78**	**6,228**	34	10.03	89
$e = 2,232$	12	44/50	461	6,228	48	10.17	76
$S^* = 62,28$	16	44/50	845	6,228	56	10.79	67
	24	43/50	460	6,228	62	11.52	65
	32	42/50	298	6,228	71	14.0	45
	$+\infty$	49/50	117	6,228	72	14.46	43
#412	**4**	**48/50**	**136**	**32,381**	16	21.0	228
$n = 300$	8	48/50	241	32,381	34	13.18	132
$e = 4,348$	12	47/50	211	32,381	45	12.42	132
$S^* = 32,381$	16	43/50	365	32,381	51	12.65	101
	24	44/50	512	32,381	75	14.95	61
	32	42/50	619	32,381	89	17.02	58
	$+\infty$	36/50	84	32,381	96	19.04	45
#414	4	38/50	935	38,478	19	21.05	289
$n = 364$	**8**	**45/50**	**532**	**38,478**	38	13.68	192
$e = 2,242$	12	45/50	721	38,478	49	12.88	192
$S^* = 21,253$	16	36/50	654	38,478	55	13.04	161
	24	35/50	594	38,478	73	14.53	159
	32	41/50	826	38,478	77	15.31	158
	$+\infty$	38/50	554	38,478	99	26.68	111
#505	4	50/50	99	21,253	27	11.15	174
$n = 240$	8	50/50	212	21,253	50	9.0	71
$e = 2,242$	12	50/50	116	21,253	65	9.34	53
$S^* = 21,253$	16	49/50	218	21,253	82	10.38	42
	24	49/50	232	21,253	91	11.4	33
	$+\infty$	**50/50**	**63**	**21,253**	93	11.69	31
#507	4	32/50	565	27,390	20	17.55	235
$n = 311$	8	35/50	544	27,390	39	12.05	145
$e = 5,732$	**12**	**38/50**	**663**	**27,390**	49	11.71	112
$S^* = 27,390$	16	28/50	542	27,390	60	12.32	111
	24	29/50	1,039	27,390	81	14.28	109
	32	30/50	1,248	27,390	86	15.21	108
	$+\infty$	33/50	71	27,390	101	20.48	68
#509	4	43/50	1,036	36,446	19	20.21	273
$n = 348$	**8**	**47/50**	**438**	**36,446**	38	13.24	176
$e = 8,624$	12	45/50	814	36,446	50	12.52	145
$S^* = 36,446$	16	38/50	794	36,446	57	12.82	143
	24	29/50	1,039	27,390	73	14.23	143
	32	47/50	681	36,446	77	14.96	142
	$+\infty$	40/50	265	36,446	94	29	100

TABLE 2 – Comparaison de DGVNS pour différentes valeurs de s_{max} sur les instances SPOT5.

satellite, tout en maximisant l'importance des photographies sélectionnées [3]. Nous reportons les résultats sur six instances sans contraintes dures de capacité.

- Instances tagSNP : Un SNP (Single Nucleotide Polymorphism) -ou polymorphisme nucléotidique- est la variation d'une seule paire de nucléotides dans l'ADN de deux individus d'une même espèce ou dans une paire de chromosomes d'un même individu. Les SNP sont des marqueurs biologiques qui peuvent être utilisés pour la prédiction des risques de développement de certaines maladies [8, 11]. Le problème de sélection des tagSNP consiste à choisir un sous-ensemble de SNP, appelé tagSNP, qui permet de capturer le maximum d'information génétique. Ce problème est considéré comme très difficile du fait de sa proximité avec le problème de *set covering* (NP-difficile) [25]. Nous rapportons les résultats des expérimentations menées sur 10 instances issues des données du chromosome1 humain [4] avec r_0=0.5 (instances modélisées sous forme de CFN binaires [25] ayant jusqu'à $n = 1550$ variables, avec des domaines de taille d allant de 30 à 266 valeurs et jusqu'à $e = 250,000$ fonctions de coût). 8 instances sont de taille moyenne et 2 de grande taille.

5 Expérimentations

5.1 Protocole expérimental

Chaque méthode a été appliquée sur chaque instance, avec une *discrepancy* de 3 pour LDS, ce qui correspond à la meilleure valeur trouvée sur les instances RLFAP [20]. Les valeurs de k_{min} et k_{max} ont été respectivement fixées à 4 et n (nombre total de variables). Le $TimeOut$ à été fixé à 3 600 secondes pour les instances RLFAP et SPOT5. Pour les instances tagSNP de taille moyenne (resp. de grande taille), le $TimeOut$ a été fixé à 2 heures (resp. 4 heures). Nous avons considéré différentes valeurs de s_{max} : $4, 8, 12, 16, 24$ et 32. Un ensemble de 50 essais par instance a été réalisé sur un AMD opteron 2.1 GHz et 256 Go de RAM. Toutes les méthodes ont été implantées en C++ en utilisant la librairie toulbar2 [5].

Pour chaque instance et chaque méthode, nous reportons (i) le nombre de succès (l'optimum est atteint), (ii) le temps de calcul moyen pour atteindre l'optimum, (iii) le coût moyen des solutions trouvées sur les 50 essais, (iv) le nombre de clusters de la décomposition obtenue, ainsi que (v) la valeur moyenne et la valeur maximale de la taille des clusters.

Dans cette section, nous évaluons successivement les apports de la fusion basée sur la taille maximale des séparateurs (cf. Section 5.2) et les apports de la fusion basée sur le taux d'absorption (cf. Section 5.3). Puis, nous dressons une étude comparative des deux méthodes de fusion de clusters (cf. Section 5.4).

4. http://www.costfunction.org/benchmark

5. http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP

Inst.	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	Taille C_i moy.	max.
#3792	4	50/50	2,381	6,359,805	11	49.36	283
$n = 528$	8	50/50	1,498	6,359,805	20	29.8	255
$d = 59$	12	50/50	1,391	6,359,805	29	23.68	157
$e = 12,084$	16	50/50	959	6,359,805	37	21.64	139
$S^* = 5,094,256$	24	50/50	1,028	6,359,805	42	21.42	133
	32	**50/50**	**832**	**6,359,805**	**49**	**22.18**	**121**
	$+\infty$	50/50	954	6,359,805	70	30.37	95
#4449	4	50/50	2,399	5,094,256	7	67.85	305
$n = 464$	8	50/50	1,396	5,094,256	16	33	195
$d = 64$	12	50/50	1,216	5,094,256	17	31.70	195
$e = 12,540$	16	50/50	996	5,094,256	19	29.73	159
$S^* = 5,094,256$	24	50/50	663	5,094,256	26	27.07	121
	32	**50/50**	**587**	**5,094,256**	**49**	**22.18**	**121**
	$+\infty$	50/50	665	5,094,256	56	36.71	101
#9319	4	50/50	1,027	6,477,229	14	41.85	197
$n = 562$	8	50/50	853	6,477,229	22	28.72	173
$d = 58$	12	50/50	897	6,477,229	30	23.93	171
$e = 14,811$	16	50/50	698	6,477,229	35	22.48	115
$S^* = 6,477,229$	24	50/50	656	6,477,229	40	22	115
	32	**50/50**	**555**	**6,477,229**	**44**	**22.5**	**115**
	$+\infty$	50/50	788	6,477,229	62	34.41	89
#16421	4	50/50	2,170	3,436,849	5	82.2	207
$n = 404$	8	50/50	2,273	3,436,849	7	60.14	201
$d = 75$	12	50/50	2,209	3,436,849	11	42.27	201
$e = 12,138$	16	50/50	2,369	3,436,849	14	36.28	133
$S^* = 3,436,849$	24	50/50	1,253	3,436,849	17	33.47	133
	32	**50/50**	**1,038**	**3,436,849**	**20**	**32.8**	**133**
	$+\infty$	50/50	2,673	3,436,849	35	42.54	113
#16706	4	50/50	131	2,632,310	11	41.36	129
$n = 438$	8	50/50	132	2,632,310	19	26.68	125
$d = 30$	12	50/50	105	2,632,310	33	19.66	55
$e = 6,321$	**16**	**50/50**	**99**	**2,632,310**	**37**	**19**	**51**
$S^* = 2,632,310$	24	50/50	124	2,632,310	46	18.91	49
	32	50/50	137	2,632,310	49	19.45	49
	$+\infty$	49/50	153	2,632,310	49	19.45	49

TABLE 3 – Comparaison de DGVNS pour différentes valeurs de s_{max} sur les instances tagSNP.

Inst.	s_{max}	Succ.	Temps	Moy.	$\|C_T\|$	Taille C_i moy.	max.
#6835	4	46/50	10,781	4,571,140	3	167	469
$n = 496$	8	46/50	9,925	4,571,122	3	167	469
$d = 90$	12	50/50	8,075	4,571,108	7	76.71	443
$e = 18,003$	16	50/50	7,966	4,571,108	7	76.71	443
$S^* = 4571108$	24	50/50	3,775	4,571,108	16	44.62	235
	32	50/50	3,321	4,571,108	23	39.26	229
	$+\infty$	**50/50**	**2,409**	**4,571,108**	**56**	**53.5**	**109**
#8956	4	42/50	9,037	6,660,310	9	56.11	215
$n = 486$	8	46/50	7,109	6,660,309	14	38.42	215
$d = 106$	12	44/50	7,592	6,660,310	15	36.6	215
$e = 20,832$	16	42/50	7,580	6,660,310	16	35.12	215
$S^* = 6,660,308$	24	41/50	7,015	6,660,311	26	29.53	215
	32	46/50	7,192	6,660,309	30	29.33	215
	$+\infty$	**50/50**	**4,911**	**6,660,308**	**54**	**52.03**	**185**
#14226	4	26/50	11,973	26,131,481	20	54.7	237
$n = 1,058$	8	13/50	11,737	26,387,778	31	37.19	185
$d = 95$	12	28/50	11,657	25,758,768	38	32.21	185
$e = 36,801$	16	35/50	11,333	25,828,540	42	30.52	179
$S^* = 25,665,437$	24	46/50	10,930	25,712,052	58	27.48	179
	32	22/50	10,434	26,108,198	66	27.54	179
	$+\infty$	**46/50**	**7,606**	**25,688,751**	**94**	**38.19**	**159**
#15757	4	50/50	195	2,278,611	13	27.61	215
$n = 342$	8	50/50	82	2,278,611	19	21	101
$d = 47$	**12**	**50/50**	**54**	**2,278,611**	**25**	**18.44**	**101**
$e = 5,091$	16	50/50	64	2,278,611	29	17.82	73
$S^* = 2,278,611$	24	50/50	72	2,278,611	41	18.31	73
	32	50/50	100	2,278,611	43	18.62	71
	$+\infty$	50/50	60	2,278,611	45	20.24	67
#17034	4	5/50	13,096	39,635,091	21	56.23	445
$n = 1,142$	8	2/50	11,979	39,665,758	30	41.26	445
$d = 123$	12	5/50	12,419	39,573,906	42	32.42	233
$e = 47,967$	16	14/50	12,149	39,359,507	55	27.98	231
$S^* = 38,318,224$	24	24/50	11,407	39,114,487	63	27.19	231
	32	32/50	11,058	38,838,844	79	27.12	229
	$+\infty$	**41/50**	**8,900**	**38,563,232**	**139**	**52.95**	**189**

TABLE 4 – Comparaison de DGVNS pour différentes valeurs de s_{max} sur les instances tagSNP.

5.2 Apports de la fusion basée sur s_{max}

Dans cette section, nous considérons différentes valeurs de s_{max} et nous comparons les performances respectives des DGVNS-s_{max} sur les instances RLFAP, SPOT5, GRAPH et tagSNP que nous avons retenues. s_{max}=$+\infty$ désigne le cas où aucune fusion n'est effectuée sur la décomposition initiale obtenue par MCS.

Instances RLFAP. Sur ces trois instances (cf. Table 1), la fusion dégrade considérablement les performances de DGVNS en termes de taux de succès et de temps de calcul, comparé à $s_{max} = +\infty$. Pour l'instance Scen06, toutes les décompositions donnent 100% de succès mais c'est $s_{max} = +\infty$ qui obtient les meilleurs temps de calcul. Notons que, sur cette instance, la taille du plus grand séparateur de la décomposition initiale est de 8. Par conséquent, toutes les décompositions obtenues avec $s_{max} \geq 8$ sont identiques. Seuls les résultats de DGVNS-4 sont reportés. Pour l'instance Scen07, DGVNS-8 obtient cinq fois moins de succès et le temps de calcul moyen est multipliée par 7. Enfin, pour Scen08, DGVNSs_{max} est incapable de trouver l'optimum. La meilleure solution trouvée est de coût 309.

Instances GRAPH. Les résultats varient selon les instances (cf. Table 1). Pour Graph11, la fusion dégrade considérablement les performances de DGVNS. En revanche, pour les deux autres instances, la fusion procure des gains substantiels. En effet, pour Graph06, DGVNS-4 améliore le temps de calcul de DGVNS d'environ 21% comparé à s_{max}=$+\infty$. Cette amélioration atteint 37% avec s_{max}=32 (meilleur valeur de s_{max}). Pour Graph13, DGVNS-s_{max} atteint l'optimum pour la première fois (excepté pour s_{max}=4). Les meilleurs résultats sont obtenus avec s_{max}=32 : 7/50 essais avec succès et une déviation moyenne de l'optimum d'environ 16%.

Instances SPOT5. Sur les instances SPOT5 (cf. Table 2), la fusion permet d'améliorer très sensiblement les performances de DGVNS à la fois en termes de taux de succès et en temps de calcul. Pour la plupart de ces instances (exceptée #507), s_{max}=8 donne les meilleurs résultats. En effet, cette valeur présente un meilleur compromis entre le nombre de clusters, la taille moyenne des clusters et la taille du plus grand cluster (cf. les colonnes 6-8, Table 2). Pour les instances #412 (resp. #509), DGVNS-8 améliore le taux de succès de DGVNS de 24% (resp. 14%).

Instances tagSNP. Sur les cinq instances de tagSNP décrites à la Table 4, la fusion dégrade considérablement les performances de DGVNS aussi bien en termes de taux de succès qu'en temps de calcul. Cette tendance s'amplifie pour les petites valeurs de s_{max} (≤ 12). Pour l'instance #8956, te taux de succès diminue de 8% (resp. 18%) pour s_{max}=32 (resp. s_{max}=24), et le temps de calcul augmente de 42% (resp. 46%) pour s_{max}=24 (resp. s_{max}=32). Pour $s_{max} \in \{12, 16\}$, le temps de calcul est multiplié par 2. On retrouve des comportements similaires sur les instances #14226 et

Inst.	Méthode	Succ.	Temps	Moy.	$\|C_T\|$	Taille C_i moy.	max.
Scen06	DGVNS	50/50	112	3,389	55	4.9	12
S^* =3,389	DGVNS-4	50/50	368	3,389	45	4.1	20
	DGVNS-abs	**50/50**	**103**	**3,389**	**11**	**11.82**	**26**
Scen07 2	DGVNS	40/50	317	345,614	110	5.72	23
S^* =343,59	DGVNS-8	8/50	2,379	343,803	106	4.64	39
	DGVNS-abs	**50/50**	**514**	**343,592**	**17**	**14.82**	**49**
Scen08	DGVNS	3/50	1,811	275	276	5.51	22
S^* =262	DGVNS-16	-	-	312	251	5.17	31
	DGVNS-abs	**5/50**	**726**	**274**	**47**	**12.02**	**63**
Graph06	DGVNS	50/50	367	4,123	119	36.07	61
S^* =4,123	DGVNS-32	**50/50**	**230**	**4,123**	**50**	**12.74**	**150**
	DGVNS-abs	50/50	260	4,123	4	69.5	134
Graph11	DGVNS	8/50	3,046	4,234	194	65.04	121
S^* =3,080	DGVNS-32	-	-	23,144	73	13.9	263
	DGVNS-abs	**25/50**	**2,967**	**3,789**	**3**	**122.0**	**320**
Graph13	DGVNS	-	-	22,489	265	83.98	147
S^* =10,110	DGVNS-32	**7/50**	**3,247**	**11,766**	85	13.76	374
	DGVNS-abs	**1/50**	**3,520**	**16,756**	**5**	**134.2**	**292**

TABLE 5 – Comparaison entre DGVNS, DGVNS-abs et DGVNS-s_{max} sur les instances RLFAP et GRAPH.

Inst.	Méthode	Succ.	Temps	Moy.	$\|C_T\|$	Taille C_i moy.	max.
#408	DGVNS	49/50	117	6,228	72	14.46	43
S^* =6,228	DGVNS-8	49/50	78	6,228	34	10.03	89
	DGVNS-abs	**50/50**	**71**	**6,228**	**9**	**25.56**	**84**
#412	DGVNS	36/50	84	32,381	96	19.04	45
S^* =32,381	DGVNS-4	48/50	136	32,381	16	21.0	228
	DGVNS-abs	**50/50**	**84**	**32,381**	**8**	**41.25**	**137**
#414	DGVNS	38/50	554	38,478	99	26.68	111
S^* =38,478	DGVNS-8	45/50	532	38,478	38	13.68	192
	DGVNS-abs	**48/50**	**635**	**38,478**	**8**	**49.25**	**200**
#505	DGVNS	**50/50**	**63**	**21,253**	**93**	**11.69**	**31**
S^* =21,253	DGVNS-4	50/50	99	21,253	27	11.15	174
	DGVNS-abs	50/50	76	21,253	15	21.33	71
#507	DGVNS	33/50	71	27,390	101	20.48	68
S^* =27,390	DGVNS-12	38/50	663	27,390	49	11.71	112
	DGVNS-abs	**49/50**	**665**	**27,390**	**10**	**35.1**	**149**
#509	DGVNS	40/50	265	36,446	94	29	100
S^* =36,446	DGVNS-8	47/50	438	36,446	38	13.24	176
	DGVNS-abs	**50/50**	**437**	**36,446**	**9**	**42.67**	**184**

TABLE 6 – Comparaison entre DGVNS, DGVNS-abs et DGVNS-s_{max} sur les instances SPOT5.

#17034, pour lesquelles DGVNS-1 obtient les plus mauvais résultats, plus particulièrement pour des petites valeurs de s_{max}.

Les contre-performances de DGVNS-1 sur ces instances (cf. Table 4) peuvent s'expliquer par le fait que la fusion basée sur s_{max} tend à augmenter les largeurs (w^-) des décompositions obtenues, en particulier pour des petites valeurs de s_{max}. Ainsi, l'impact de la stratégie de diversification de DGVNS s'en trouve fortement affaiblie : les voisinages deviennent trop grands et leur nombre trop petit. Pour l'instance #6835, avec s_{max}=4, w^- passe de 108 à 468 et le nombre de clusters diminue de 56 à 3.

A l'inverse, pour les cinq instances de la Table 3, la fusion permet d'améliorer les performances de DGVNS. Pour la plupart de ces instances (excepté l'instance #16706), s_{max}=32 donne les meilleurs résultats. Pour l'instance #16421, avec s_{max}=32, le temps de calcul moyen est divisé par 2 comparé à s_{max}=$+\infty$, tandis que pour l'instance #9315, le gain en temps de calcul est d'environ 29%. Pour les autres instances, ce gain est d'environ 12%.

Sur ces instance, la fusion permet de supprimer les clusters qui se recouvrent fortement. Les voisinages obtenus sont plus pertinents (clusters faiblement connectés et de tailles raisonnables), car la plupart des clusters fusionnés possèdent peu de variables propres. De plus, pour ces instances, la largeur de décomposition ainsi que le nombre de clusters varient peu (cf. colonnes 6-8 de la Table 3).

Cette section a montré que fusionner, selon le critère s_{max}, les clusters redondants (contenant peu de variables propres) permet de renforcer la pertinence des décompositions obtenues et d'améliorer les performance de DGVNS sur les instances SPOT5 et sur certaines instances tagSNP. Au contraire, sur les autres instances, l'amélioration n'est pas systématique.

5.3 Apports de la fusion basée sur l'absorption

Dans cette section, nous comparons les performances de DGVNS et DGVNS-*abs* sur les instances RLFAP, SPOT5, GRAPH et tagSNP. Pour toutes nos expérimentations, Ab_{max} a été fixé à 70%. Ainsi, les clusters partageant un peu plus de 2/3 de leurs variables seront fusionnés.

Instances RLFAP. Sur ces instances, DGVNS-*abs* surclasse DGVNS (cf. Table 5). Pour Scen06, les deux méthodes obtiennent 100% de succès ; toutefois, DGVNS-*abs* est plus rapide. Pour Scen07, DGVNS-*abs* améliore le taux de succès de DGVNS de 20%. Pour Scen08, DGVNS-*abs* obtient deux succès de plus que DGVNS et le temps de calcul est divisé par 2.

Instances GRAPH. Sur ces instances, l'apport de la fusion par absorption est très important (cf. Table 5), particulièrement sur les instances Graph11 et Graph13 réputées difficiles, pour lesquelles on observe de fortes améliorations par rapport à DGVNS. Pour Graph11, DGVNS-*abs* améliore le taux de succès de DGVNS de 34% (de 16% à 50%) et réduit la déviation moyenne de l'optimum de (37.4% à 23%). Pour Graph13, DGVNS-*abs* obtient un succès (DGVNS n'atteint jamais l'optimum) et réduit la déviation moyenne de l'optimum de moitié (de 122% à 65%). Pour Graph06, les deux méthodes obtiennent 100% de succès, toutefois, DGVNS-*abs* est plus rapide (gain en temps de calcul de 29% par rapport à DGVNS).

Instances SPOT5. DGVNS-*abs* se montre aussi très efficace sur les instances SPOT5 (cf. Table 6), où il obtient 100% de succès sur quatre des six instances. Sur les instances de grande taille (#412, #414, #507 et #509), DGVNS-*abs* améliore le taux moyen de succès de DGVNS de 25%. Sur les autres instances, les deux méthodes obtiennent les mêmes taux de succès, mais DGVNS-*abs* est plus rapide.

Instances tagSNP. Une fois de plus, les résultats sont en faveur de DGVNS-*abs* (cf. Table 7). Pour les instances de taille moyenne (excepté #6835 et #8956), DGVNS-*abs* obtient de meilleurs temps de calcul comparé à DGVNS. Pour les instances #16421 et #16706, DGVNS-*abs* est 2 à 3 fois plus rapide. Pour les instances de grande taille, DGVNS-*abs* améliore le taux moyen de succès de 10%. Pour les instances de grande taille, c'est encore DGVNS-*abs* qui obtient les meilleures performances.

5.4 Comparaison des deux méthodes de fusion

Cette section compare DGVNS-*abs* et DGVNS-s_{max} pour la valeur de s_{max} ayant obtenu les meilleurs résultats.

Instances RLFAP. Les résultats sont en faveur de DGVNS-*abs* (cf. Table 5). Pour Scen06, DGVNS-*abs* est trois fois plus rapide. Pour Scen07, DGVNS-*abs* obtient 100% de succès contre 22% pour DGVNS-8 et divise le temps de calcul par 4. Pour Scen08, DGVNS-abs obtient 5 succès et une déviation moyenne à l'optimum de 4% contre 19% pour DGVNS-16.

Les bonnes performances de DGVNS-*abs* sur ces instances peuvent s'expliquer par la qualité des décompositions produites par la fusion basée sur l'absorption. En effet, ces dernières présentent des clusters ayant une forte proportion de variables propres et contenant en moyenne 13 variables. Ceci renforce la pertinence du mécanisme de diversification de DGVNS car les clusters sont beaucoup moins redondants. Ce n'est pas le cas pour les décompositions fournies par la fusion basée sur s_{max}, où beaucoup de clusters partageant 100% de leur variables sont encore conservés (cf. Section 3.3). Par ailleurs, en raison du nombre peu élevé de clusters obtenus, différents régions de l'espace de recherche sont couvertes beaucoup plus rapidement, permettant de converger vers la région contenant l'optimum global.

Instances SPOT5. La tendance se confirme sur les instances SPOT5 (cf. Table 6), pour lesquelles DGVNS-*abs* obtient de légères améliorations aussi bien en taux de succès qu'en temps de calcul. Notons, toutefois, que pour l'instance #507, l'amélioration est beaucoup plus significative, où DGVNS-*abs* obtient 98% de succès contre 76% pour DGVNS-12 (gain de 22%). Sur ces instances, les mêmes constats, que ceux dressés sur les instances du RLFAP, peuvent être effectués concernant la pertinence des décompositions produites par la fusion par absorption. De plus, les largeurs des décompositions (w^-) obtenues restent comparables, voire nettement inférieures à celles des décompositions produites par la fusion basée sur s_{max}. Par exemple, pour l'instance #412, w^- passe de 227 (avec $s_{max} = 4$) à 136 (avec l'absorption).

Inst.	Méthode	Succ.	Temps	Moy.	$\|C_T\|$	Taille C_i moy.	max.
#3792	DGVNS	50/50	954	6,359,805	70	30.37	95
S^* =6,359,805	DGVNS-32	50/50	832	6,359,805	49	22.18	121
	DGVNS-abs	**50/50**	**602**	**6,359,805**	**12**	**50.17**	**140**
#4449	DGVNS	50/50	665	5,094,256	56	36.71	101
S^* =5,094,256	DGVNS-32	50/50	587	5,094,256	49	22.18	**121**
	DGVNS-abs	**50/50**	**578**	**5,094,256**	**8**	**64.5**	**164**
#6835	DGVNS	**50/50**	**2,409**	**4,571,108**	**56**	**53.5**	**109**
S^* =4,571,108	DGVNS-32	50/50	3,321	4,571,108	23	39.26	229
	DGVNS-abs	50/50	2,947	4,571,108	4	140.0	262
#8956	DGVNS	**50/50**	**4,911**	**6,660,308**	**54**	**52.03**	**185**
S^* =6,660,308	DGVNS-32	46/50	7,192	6,660,309	30	29.33	215
	DGVNS-abs	50/50	6,555	6,660,308	6	88.17	216
#9319	DGVNS	50/50	788	6,477,229	62	34.41	89
S^* =6,477,229	DGVNS-**32**	**50/50**	**555**	**6,477,229**	**44**	**22.5**	**115**
	DGVNS-abs	50/50	738	6,477,229	10	59.9	181
#15757	DGVNS	50/50	60	2,278,611	45	20.24	67
S^* =2,278,611	DGVNS-**12**	**50/50**	**54**	**2,278,611**	**25**	**18.44**	**101**
	DGVNS-abs	**50/50**	**58**	**2,278,611**	**10**	**40.4**	**92**
#16421	DGVNS	50/50	2,673	3,436,849	35	42.54	113
S^* =3,436,849	DGVNS-32	**50/50**	**1,038**	**3,436,849**	**20**	**32.8**	**133**
	DGVNS-abs	**50/50**	**1,114**	**3,436,849**	**6**	**74**	**134**
#16706	DGVNS	49/50	153	2,632,310	49	19.45	49
S^* =2,632,310	DGVNS-16	50/50	99	2,632,310	37	19	51
	DGVNS-abs	**50/50**	**53**	**2,632,310**	**12**	**41.33**	**94**
#17034	DGVNS	41/50	8,900	38,563,232	139	52.95	189
S^* =38,318,224	DGVNS-32	32/50	11,058	38,838,844	79	27.12	229
	DGVNS-abs	**47/50**	**13,108**	**38,318,226**	**15**	**80.73**	**237**
#14226	DGVNS	46/50	7,606	25,688,751	94	38.19	159
S^* =25,665,437	DGVNS-24	46/50	10,930	25,712,052	58	27.48	179
	DGVNS-abs	**50/50**	**10,417**	**25,665,437**	**15**	**76.27**	**184**

TABLE 7 – Comparaison entre DGVNS, DGVNS-abs et DGVNS-s_{max} sur les instances tagSNP.

Instances GRAPH. Sur ces instances (cf. Table 5), aucune des deux méthodes ne domine clairement l'autre. En effet, DGVNS-*abs* surclasse DGVNS-32 sur Graph11 (50% de succès), alors que sur Graph13, c'est DGVNS-32 qui obtient les meilleurs résultats en termes de taux de succès (gain de 12%) et en qualité des solutions obtenues (réduction de la déviation moyenne de l'optimum de 65.73% à 16.37%). Notons, que sur cette instance difficile, la fusion permet à DGVNS d'atteindre l'optimum pour la première fois, avec un taux de succès de 14%. La contre-performance de DGVNS-*abs* sur Graph13 peut probablement s'expliquer par la taille moyenne des clusters qui est trop importante (134 variables par cluster) et par un fort chevauchement entre clusters (s_{max}=153). Pour comparaison, les clusters issus de la fusion basée sur s_{max} contiennent en moyenne 14 variables et s_{max}=29.

Instances tagSNP. Pour les instances de taille moyenne (cf. Table 7), DGVNS-*abs* obtient de meilleurs résultats sur 5 instances parmi 8 (meilleur taux de succès sur l'instance #8956 et meilleurs temps de calcul sur quatre instances). Pour les trois autres instances (exceptée pour #15757 où les résultats sont très similaires), c'est DGVNS-32 qui obtient les meilleurs temps de calcul. Pour les instances de grande taille (#14226 et #17034), DGVNS-*abs* surclasse nettement DGVNS-32, avec un gain en taux de succès moyen de 12.5%.

Bilan expérimental. Sur l'ensemble des instances, la fusion de clusters procure des gains substantiels comparés aux résultats obtenus par DGVNS sur la décomposition initiale de MCS. Ces résultats confirment l'intérêt de réduire la redondance entre clusters pour identifier des structures de voisinages plus pertinentes et ainsi renforcer l'effort de diversification de DGVNS.

Par ailleurs, la fusion par absorption permet de produire des décompositions avec des clusters de taille moyenne et ayant une forte proportion de variables propres. En effet, pour la plupart des instances (cf. les colonnes 6-8 des différentes figures), la taille des clusters est en moyenne 2 fois plus grande comparée à celle de la décomposition initiale. De plus, le nombre de clusters reste relativement petit. Sur l'instance #17034, le nombre de clusters diminue de 139 à 15, la taille moyenne augmente de 52.95 à 80.73 et w^- passe de 189 à 237.

6 Conclusions et perspectives

Nous avons proposé deux schémas de raffinement de la décomposition arborescente permettant de limiter la redondance entre clusters, dans le but de guider DGVNS vers des structures de voisinages plus pertinentes. Les expérimentations menées sur plusieurs instances difficiles montrent la pertinence et l'intérêt de la fusion par absorption pour renforcer la qualité des voisinages. Comme travail futur, il serait intéressant d'envisager des critères de fusion dynamiques lors de la recherche, selon les valeurs des variables instanciées.

Acknowledgements. Ce travail à été soutenu par l'Agence Nationale de la Recherche, référence projet FiCOLOFO ANR-10-BLA-0214.

Références

[1] E. Amir and S. Mcllraith. Solving satisfiability using decomposition and the most constrained subproblem. *Electronic Notes in Discrete Mathematics*, 9 :329–343, 2001.

[2] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM. J. on Algebraic and Discrete Methods*, 8 :277–284, 1987.

[3] E. Bensana, M. Lemaître, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4(3) :293–299, 1999.

[4] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4(1) :79–89, 1999.

[5] S. de Givry, T. Schiex, and G. Verfaillie. Exploiting tree decomposition and soft local consistency in weighted CSP. In *AAAI*, pages 22–27. AAAI Press, 2006.

[6] R. Dechter and Y. El Fattah. Topological parameters for time-space tradeoff. *Artif. Intell.*, 125(1-2) :93–118, 2001.

[7] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artif. Intell.*, 38(3) :353–366, 1989.

[8] C.S. Carlson et al. Selecting a maximally informative set of single-nucleotide polymorphisms for association analyses using linkage disequilibrium. *Am. J. of Hum. Genetics*, 74(1) :106–120, 2004.

[9] M. Fontaine, S. Loudni, and P. Boizumault. Guiding VNS with tree decomposition. In *ICTAI*, pages 505–512, 2011.

[10] M. Fontaine, S. Loudni, and P. Boizumault. Exploiting tree decomposition for guiding neighborhoods exploration for VNS. *RAIRO Operations Research*, 47(2) :91–123, 2013.

[11] S. Gopalakrishnan and Z. S. Qin. Tagsnp selection based on pairwise ld criteria and power analysis in association studies. In *Pacific Symposium on Biocomputing*, pages 511–522, 2006.

[12] G. Gottlob, S. T. Lee, and G. Valiant. Size and treewidth bounds for conjunctive queries. In Jan Paredaens and Jianwen Su, editors, *PODS*, pages 45–54. ACM, 2009.

[13] G. Gottlob, R. Pichler, and F. Wei. Tractable database design through bounded treewidth. In Stijn Vansummeren, editor, *PODS*, pages 124–133. ACM, 2006.

[14] D. Habet, L. Paris, and C. Terrioux. A tree decomposition based approach to solve structured SAT instances. In *ICTAI*, pages 115–122, 2009.

[15] P. Hansen, N. Mladenovic, and D. Perez-Brito. Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4) :335–350, 2001.

[16] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995.

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

[18] Ph. Jégou, S. Ndiaye, and C. Terrioux. Strategies and Heuristics for Exploiting Tree-decompositions of Constraint Networks. In *WIGSK'06*, pages 13–18, 2006.

[19] M. Kitching and F. Bacchus. Exploiting decomposition on constraint problems with high tree-width. In *IJCAI*, pages 525–531, 2009.

[20] S. Loudni and P. Boizumault. Combining VNS with constraint programming for solving anytime optimization problems. *EJOR*, 191(3) :705–735, 2008.

[21] R. Marinescu and R. Dechter. AND/OR branch-and-bound search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17) :1457–1491, 2009.

[22] B. Neveu, G. Trombettoni, and F. Glover. ID Walk : a candidate list strategy with a simple diversification device. In *CP*, pages 423–437, 2004.

[23] I. Rish and R. Dechter. Resolution versus search : Two strategies for SAT. *J. Autom. Reasoning*, 24(1/2) :225–275, 2000.

[24] N. Robertson and P.D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3) :309–322, 1986.

[25] M. Sánchez, D. Allouche, S. de Givry, and T. Schiex. Russian doll search with tree decomposition. In *IJCAI*, pages 603–608, 2009.

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

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

[28] H. van Benthem. GRAPH : Generating radiolink frequency assignment problems heuristically, 1995.

STR et compression de contraintes tables

Nebras Gharbi **Fred Hemery** **Christophe Lecoutre** **Olivier Roussel**

CRIL - CNRS UMR 8188,
Université Lille Nord de France, Artois,
rue de l'université, 62307 Lens cedex, France
{gharbi,hemery,lecoutre,roussel}@cril.fr

Résumé

Les contraintes tables sont très étudiées par la communauté de programmation par contraintes. De nombreux algorithmes de filtrage pour ce type de contraintes (et aussi leurs combinaisons, par exemple, voir [6]) ont été proposés ces dernières années. L'un des meilleurs algorithmes, appelé STR2, qui est basé sur la technique dite de réduction tabulaire simple (STR pour Simple Tabular Reduction), maintient dynamiquement la liste des tuples supports de chaque contrainte au cours des processus d'inférence et de recherche. Sur certains problèmes, cet algorithme est toutefois supplanté par une approche consistant à représenter de manière compacte l'ensemble des tuples à l'aide d'un diagramme multivalué (MDD). Dans cet article, nous étudions la possibilité de combiner l'approche STR avec une forme compressée basée sur l'identification de motifs récurrents dans l'ensemble des tuples de chaque contrainte table.

Abstract

Over the recent years, many filtering algorithms have been developed for table constraints. STR2, one of the most efficient algorithms, is based on the technique of simple tabular reduction, meaning that it maintains dynamically the list of supports in each constraint table during inference and search. However, for some specific problems, the approach that consists in representing in a compact way tables by means of multi-valued decision diagrams (MDD) overcomes STR2. In this paper, we study the possibility of combining simple tabular reduction with a compression form of tables based on the detection of recurrent patterns in tuples.

1 Introduction

L'intérêt pour les contraintes tables, qui sont des contraintes définies en extension en listant les tuples autorisés (ou ceux qui sont interdits), est important en programmation par contraintes. En effet, nombre de problèmes impliquent l'utilisation de ce type de contraintes. Dans certains cas, représenter celles-ci n'est malheureusement pas possible car l'espace mémoire requis est trop important. Rappelons que la complexité spatiale pour représenter tous les tuples croît de manière exponentielle avec l'arité des contraintes. En vue de réduire la complexité spatiale, différentes approches ont été proposées dans la littérature. Certaines introduisent des structures de données compactes telles que les tries (arbres de préfixes) [2], les MDDs [1], les tables compressées [3] ou encore les automates déterministes finis (DFA) [7]).

À ce jour, les algorithmes de filtrage les plus efficaces pour les contraintes tables sont ceux basés sur l'utilisation des MDDs, et surtout ceux basés sur la technique STR (Simple Tabular Reduction). En particulier, les variantes STR2 [4] et STR3 [5] de l'algorithme STR1 original [9] ont été montrées particulièrement compétitives sur de nombreuses classes de problèmes. De manière schématique, hormis les problèmes pour lesquels le taux de compression possible avec l'utilisation de MDDs est très élevé, l'approche STR est celle qui est la plus efficace.

Pour étendre la gamme de problèmes pour lesquels STR est appropriée, nous proposons de combiner cette approche avec une technique simple de compression de tuples (différente de celle proposée dans [3] qui remplace des ensembles de tuples par un produit cartésien de sous-domaines). Le principe est d'identifier les motifs (sous-tuples) récurrents au sein des tuples de chaque contrainte, et d'intégrer une indirection au niveau des tuples vers ces motifs. L'algorithme de filtrage STR doit être alors modifié pour prendre en compte ces indirections, un système d'horodatage permettant d'éviter d'effectuer plusieurs fois les tests de validité pour chaque motif.

Dans cet article, nous détaillons notre approche en décrivant l'utilisation d'arbres de préfixes à la lecture des contraintes tables, puis celle de structures annexes pour les motifs et l'horodatage. Nous présentons quelques résultats expérimentaux avant de conclure.

2 Méthode de compression

Une contrainte table est une contrainte qui est définie en extension par un ensemble de tuples. Un tuple représente une combinaison de valeurs pour les variables de la contrainte, autorisée en cas de contrainte positive et réfutée en cas de contrainte négative. Un tuple est dit valide lorsque les valeurs de celui-ci sont présentes dans les domaines courants des variables correspondantes.

Un motif μ est une séquence de valeurs consécutives dans un tuple τ d'une table. On note $|\mu|$ la longueur d'un motif μ, et $\text{nbOcc}(\mu)$ le nombre d'occurrences du motif repérées dans l'ensemble des tuples d'une contrainte donnée.

Afin de réduire la complexité spatiale de la représentation des tuples de chaque contrainte, nous allons repérer les motifs les plus fréquents et remplacer chaque occurrence de motif par un symbole unique. De ce fait, la taille utilisée pour représenter la table sera d'autant plus faible que la longueur des motifs sera grande et que leur nombre d'occurrences sera important.

Il est important de noter que nous considérons que les motifs extraits dans le cadre de notre approche sont indépendants de leur position initiale dans le tuple. En conséquence, un motif ne correspond pas obligatoirement à l'affectation des mêmes valeurs aux mêmes variables mais plutôt la même suite d'affectation à une séquence de variables consécutives. Ce choix a été fait dans l'espoir d'obtenir des motifs les plus fréquents possibles, et donc une meilleure compression.

Pour identifier les motifs pertinents, nous allons dans un premier temps créer une forêt d'arbres de préfixes à partir des différents tuples d'une contrainte table donnée. Un arbre enregistre toutes les séquences existantes de valeurs de longueur donnée et leur nombre d'occurrences. Pour garantir un certain niveau d'efficacité de compression, la longueur minimale des séquences est fixée dans notre approche à 3, et la longueur maximale à l'arité de la contrainte moins 1.

Dans un second temps, il nous faut identifier les motifs les plus efficaces pour le processus de compression. Pour cela nous allons introduire la notion de **score** d'un motif μ comme suit :

$$\text{score}(\mu) = |\mu| \times \text{nbOcc}(\mu)$$

Un seuil de sélection est fixé, seuls les motifs dont le score est supérieur au seuil de sélection sont retenus dans l'algorithme de compression et stockés dans la table des motifs. Pour des raisons d'efficacité, le nombre total de motifs retenus est borné par un second paramètre afin de contrôler le temps de compression.

Le processus de compression utilise donc les motifs dont le score est supérieur au seuil de sélection. Un parcours de la table est effectué pour détecter la présence des motifs retenus et établir une référence vers la table de motifs. Si dans un tuple, plusieurs motifs se recouvrent, l'algorithme de compression choisit en priorité le motif ayant le meilleur score. Après compression, la table contient des tuples de longueurs différentes composés de valeurs et de références vers la table des motifs.

Nous donnons maintenant une illustration de ce processus. La table 1 représente une contrainte table positive d'arité 5, portant sur les variables x_1, x_2, …, x_5. Nous pouvons remarquer que plusieurs motifs se répètent au sein des tuples parmi lesquels nous pouvons citer *cbc*, *aab* et *abb* comme des motifs de longueur 3 et *aabb*, *acbc* et *cbca* comme des motifs de longueur 4. À la lecture de notre contrainte table nous pouvons construire les arbres de préfixes schématisés à la figure 1 correspondant aux motifs possibles de tailles 3 (minimum) et 4 (arité de la contrainte moins 1). Chaque feuille identifie un chemin μ (allant de la racine à la feuille) auquel il est possible d'associer (au niveau de la feuille) le compteur $\text{nbOcc}(\mu)$ (ceci n'est pas montré sur la figure).

	x_1	x_2	x_3	x_4	x_5
τ_1	(c,	b,	c,	a,	c)
τ_2	(a,	a,	b,	b,	a)
τ_3	(a,	c,	b,	c,	a)
τ_4	(b,	a,	c,	b,	c)
τ_5	(b,	a,	a,	b,	b)
τ_6	(a,	c,	b,	c,	b)
τ_7	(a,	c,	a,	c,	a)

TABLE 1 – Contrainte en extension C_{x_1,x_2,x_3,x_4,x_5}

L'application de l'algorithme de compression nous permet d'avoir une version compressée de la contrainte table ; sa représentation logique est donnée par la figure 2(a) tandis que la table des motifs est donnée par la figure 2(b). Les tuples τ_1, τ_3, τ_4, τ_5 de la table compressée font référence au motif μ_3 aux positions respectives 1, 2, 3, et 2, et les tuples τ_2, τ_6 font référence au motif μ_2 aux positions respectives 1 et 2. Ce mécanisme de compression peut entraîner une réduction importante de l'espace mémoire occupé par la contrainte table de départ. Ceci est illustré par la vue physique donnée par la figure 2(c).

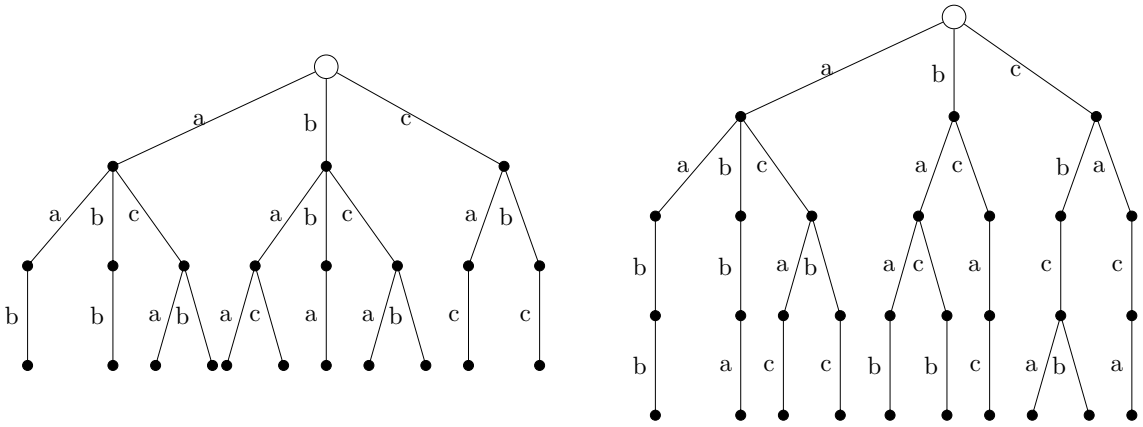


FIGURE 1 – Arbres de préfixes de longueur 3 et 4 construits à partir de la contrainte de la table 1

	x_1	x_2	x_3	x_4	x_5
τ_1	μ_3			a	c
τ_2	μ_2			b	a
τ_3	a	μ_3			a
τ_4	b	a	μ_3		
τ_5	b	μ_2			b
τ_6	a	μ_3			b
τ_7	a	c	a	c	a

(a) Table compressée (vue logique)

μ_1	...
μ_2	a,a,b
μ_3	c,b,c
...	...

(b) Table des motifs

τ_1	μ_3	a	c		
τ_2	μ_2	b	a		
τ_3	a	μ_3	a		
τ_4	b	a	μ_3		
τ_5	b	μ_2	b		
τ_6	a	μ_3	b		
τ_7	a	c	a	c	a

(c) Table compressée (vue physique)

FIGURE 2 – Compression de la table

3 Algorithme de filtrage

Lors du filtrage d'une contrainte table, il est nécessaire de vérifier la validité des tuples, ce qui implique de vérifier la validité des motifs. Quand un motif apparaît plusieurs fois dans la table à partir de la même position, nous souhaitons n'effectuer le test de validité du motif qu'une seule fois, ce qui permet de traduire la compression spatiale en une réduction du temps de filtrage.

Pour ce faire, nous utilisons un compteur *time* qui est incrémenté à chaque filtrage de la contrainte table et un tableau $stamps[\mu_i, j]$ associé à la contrainte. Pour un motif μ_i qui s'applique à partir d'une position j, $stamps[\mu_i, j]$ donne le résultat du dernier test de validité de (μ_i, j) (champ $stamps[\mu_i, j].valid$) ainsi que la valeur $stamps[\mu_i, j].time$ du compteur *time* lors de ce test. Chaque fois que la validité de (μ_i, j) doit être testée, nous vérifions d'abord si $stamps[\mu_i, j].time$ est égal à la valeur courante de *time*. Si c'est le cas, la validité a déjà été testée dans l'opération de filtrage courante et donc $stamps[\mu_i, j].valid$ fournit directement la réponse, ce qui évite des calculs inutiles. Sinon, il faut effectivement tester la validité de (μ_i, j) et sauvegarder le résultat dans $stamps[\mu_i, j]$.

	position		
	1	2	3
μ_2	$0, ?$	$0, ?$	$0, ?$
μ_3	$0, ?$	$0, ?$	$0, ?$

(a) Initialisation des champs $(time, valid)$

	position		
	1	2	3
μ_2	$1, F$	$1, V$	$0, ?$
μ_3	$1, F$	$1, V$	$1, F$

(b) À la fin de $time = 1$

FIGURE 3 – Évolution de la structure *stamps*

La figure 3 présente un exemple d'évolution de la structure *stamps*. Au départ, tous les éléments sont initialisés à $time = 0$ et le champ *valid* reste non assigné (voir figure 3(a)). Lors du premier filtrage de la table, le compteur global *time* passe à 1. Pour déterminer si le tuple τ_3 (par exemple) est valide, il faut s'assurer que μ_3 à la position $j = 2$ est valide. Comme $stamps[\mu_3, 2].time$ n'est pas égal à la valeur courante de *time*, nous savons que ce test n'a pas déjà été fait. Nous vérifions donc si c, b et c sont toujours présents dans les domaines de x_2, x_3 et x_4 respectivement. Nous supposons ici que le résultat est positif. Nous stockons donc $(1, Vrai)$ dans $stamps[\mu_3, 2]$. Lorsque nous testons ensuite la validité du tuple τ_5, on s'aperçoit immédiatement que la validité de μ_3 a été testée lors du filtrage courant et il suffit de prendre le résultat sauvegardé dans $stamps[\mu_3, 2].valid$. En supposant que μ_2 et μ_3 sont tous les deux invalides en position 1, valides en position 2 et que μ_3 est non valide en position 3,

nous obtenons à la fin du filtrage le résultat présenté en figure 3(b).

Pour une contrainte d'arité r et un motif μ, il n'y a que $r - |\mu| + 1$ couples (μ, j) possibles (car $1 \leq j \leq r - |\mu| + 1$). La structure *stamps* a donc une taille en $O(m.r)$ où m est le nombre de motifs identifiés.

4 Résultats expérimentaux

Pour montrer le potentiel de notre approche (STR^c), nous avons comparé le comportement des algorithmes STR1, STR2, STR3 et STR^c lorsqu'ils sont intégrés à l'algorithme de recherche MAC (qui maintient la propriété de cohérence d'arc généralisée lors d'une recherche arborescente). Nous avons effectué quelques tests sur des instances de deux problèmes distincts : le premier correspond aux séries mdd introduites dans [1] et le second à la construction de nonogrammes [8]. Les résultats figurent en table 2 ; l'heuristique dom/ddeg est utilisée pour garantir le même parcours d'arbre (dom/wdeg est plus versatile).

L'algorithme STR^c permet une économie spatiale d'au moins 50% par rapport à STR1 et STR2, et jusqu'à un facteur 4 par rapport à STR3. Les temps de résolution sont donnés sous la forme build+search où build indique le temps pour construire l'instance et search le temps nécessaire pour trouver une solution. Lorsqu'on considère le temps de recherche, il apparaît que STR^c rivalise avec STR1, mais reste toutefois supplanté par STR2. Il est vraisemblable qu'un meilleur réglage des paramètres utilisés pour STR^c (dans notre expérimentation, 500 motifs autorisés avec un seuil de sélection égal à 50) permettrait de mieux contrôler le temps nécessaire à la construction.

Instance		STR1	STR2	STR3	STR^c
mdd-25-7-23	mem	147M	147M	384M	102M
	CPU	2.3+26.3	2.3+13.7	2.7+50.0	11.4+30.6
mdd-23-15-1	mem	223M	238M	597M	127M
	CPU	4.0+31.1	3.9+10.8	4.8+220	37.6+33.7
non-gp-65	mem	33M	33M	76M	21M
	CPU	0.5+41.1	0.6+12.0	0.7+9.1	7.3+37.5
non-gp-130	mem	221M	230M	663M	129M
	CPU	4.1+0.5	4.1+0.3	5.3+0.6	120+0.5

TABLE 2 – Espace mémoire et temps CPU pour résoudre quelques instances avec MAC.

5 Conclusion

Dans cet article, nous avons cherché à combiner l'approche STR avec une compression originale des contraintes tables. En identifiant les motifs récurrents dans l'ensemble de tuples présents dans les différentes tables, il est possible d'économiser l'espace mémoire (à l'aide d'un système d'indirection) et également le temps CPU en évitant les tests de validité redondants sur les motifs enregistrés. L'algorithme STR^c que nous proposons semble, d'après nos tests préliminaires, rivaliser avec STR1 (au niveau de la recherche) mais supplanté par STR2. Nous envisageons d'étudier diverses optimisations de ce nouvel algorithme, ainsi qu'un réglage plus fin des paramètres nécessaires à l'identification des motifs. Nous projetons également, dans un futur proche, d'étudier d'autres modèles de compression, toujours en conjonction avec l'approche STR.

Remerciements

Ce travail bénéficie du soutien du CNRS et d'OSEO dans le cadre du projet ISI Pajero.

Références

[1] 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.

[2] 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.

[3] G. Katsirelos and T. Walsh. A compression algorithm for large arity extensional constraints. In *Proceedings of CP'07*, pages 379–393, 2007.

[4] C. Lecoutre. STR2 : Optimized simple tabular reduction for table constraint. *Constraints*, 16(4) :341–371, 2011.

[5] C. Lecoutre, C. Likitvivatanavong, and R. Yap. A path-optimal GAC algorithm for table constraints. In *Proceedings of ECAI'12*, pages 510–515, 2012.

[6] A. Paparrizou and K. Stergiou. An efficient higher-order consistency algorithm for table constraints. In *Proceedings of AAAI'12*, pages 335–541, 2012.

[7] G. Pesant. A regular language membership constraint for finite sequences of variables. In *Proceedings of CP'04*, pages 482–495, 2004.

[8] G. Pesant, C.-G. Quimper, and A. Zanarini. Counting-based search : Branching heuristics for constraint satisfaction problems. *Journal of Artificial Intelligence Research*, 43 :173–210, 2012.

[9] J.R. Ullmann. Partition search for non-binary constraint satisfaction. *Information Science*, 177 :3639–3678, 2007.

Stratégies d'élimination des clauses apprises dans les solveurs SAT modernes

Long Guo Said Jabbour Lakhdar Sais

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

{guo,jabbour,sais}@cril.fr

Résumé

L'apprentissage de clauses est un composant important des solveurs SAT modernes. Toutefois le nombre de clauses apprises peut être exponentiel dans le pire des cas. Pour gérer ces clauses, plusieurs politiques ont été proposées pour maintenir une base de clauses pertinentes de taille polynomiale. Dans ce papier, nous donnons, tout d'abord, une comparaison des différentes stratégies proposées dans la littérature avec une stratégie simple basée sur un choix aléatoire des clauses à supprimer. Nous proposons ensuite de nouveaux critères permettant de juger de la pertinence d'une clause par rapport à un état de la recherche. Nous proposons également différentes fréquences de réduction de la base des clauses apprises. Dans la partie expérimentale, nous évaluons d'abord les performances des stratégies proposées et nous sélectionnons les stratégies de réductions complémentaires pour une intégration dans un solveur parallèle de type portfolio.

Abstract

Conflict based clause learning is known to be an important component in Modern SAT solving. Because of the exponential blow up of the size of learnt clauses database, maintaining a relevant and polynomially bounded set of learnt clauses is crucial for the efficiency of clause learning SAT solvers. In this paper, we first compare several criteria for selecting the most relevant learnt clauses with a simple random selection strategy. We then propose new criteria allowing us to select relevent clauses w.r.t. a given search state. Several application frequency of such reduction strategies are then proposed. The experiments give us an exhaustive evaluation of the proposed strategies. The complementary reduction strategies are then integrated in a parallel portfolio based solver.

1 Introduction

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

Au cours de ces deux dernières décennies le problème SAT, avec l'arrivée d'une nouvelle génération de solveurs, a énormément gagné en intérêt. Ces solveurs, appelés CDCL[15, 11], (Conflict Driven, Clause Learning) ont pour la première fois été capables de résoudre des instances conséquentes codant des problèmes de la vie réelle. Leur architecture est basée sur les notions suivantes : l'heuristique de choix de variables VSIDS (Variable State Independant, Decading Sum)[15], l'utilisation de redémarrage[14, 12] et d'une structure de données efficace (ex. Watched literals) ainsi que l'apprentissage de clauses[17, 15, 18]. Cette dernière est une composante très importante pour ce type de solveur. En effet d'un point de vue théorique, K. Pipatsrisawat et A. Darwiche [16] ont récemment prouvé que l'apprentissage de clauses tel qu'il est fait dans les démonstrateurs SAT modernes permet de simuler la résolution générale. Ce résultat montre que l'apprentissage rend les démonstrateurs de type DPLL[8] aussi puissants que la résolution générale. Mais en pratique, cet ajout de clauses peut poser problème. En effet, dans le pire des cas, la taille de la base de clauses apprises peut croître de manière expo-

nentielle. Pour maintenir une base des clauses apprises de taille polynomiale - permettant aussi un coût raisonnable pour la propagation unitaire - un ensemble de stratégies a été proposé pour supprimer les clauses supposées inutiles pour la suite de la recherche. Parmi ces stratégies, deux ont montré des résultats pratiques intéressants. La première nommée CSIDS [15](*Clause State Independent, Decaying Sum*), la plus populaire, est basée sur la notion de *first fail*. Cette stratégie considère qu'une clause apparaissant peu dans l'analyse des conflits récents est inutile et qu'elle doit être supprimé. La seconde est basée sur une mesure nommée LBD (*Literal Block Distance*)[6]. Cette mesure correspondant au nombre de niveaux différents intervenant dans la génération de la clause apprise. Cette mesure appelée LBD dans [6] est introduite pour la première fois dans [3, 2] pour montrer que le schéma first UIP est optimal pour cette mesure. Dans [6], les auteurs considèrent alors que les clauses ayant un LBD élevés sont inutiles pour la suite de la recherche et qu'elle peuvent donc être supprimées. Plus récemment, les auteurs de [4] ont proposé une heuristique dynamique de gestion de la base de clauses apprises, ils ont utilisé un critère nommé PSM (Progress Saving based Measure), les clauses apprises avec une petite valeur de PSM ont de fortes chances d'être utilisées pour la propagation unitaire ou d'être falsifiées. Au contraire, une clause avec un grand PSM a plus de chance d'être satisfaite par plus d'un littéral et donc être inutile pour la suite de recherche.

Dans ce papier nous donnons, après quelques définitions préliminaires, une comparaison des différentes stratégies proposées dans la littérature avec une stratégie simple basée sur un choix aléatoire des clauses à supprimer. Nous proposons ensuite de nouveaux critères permettant de juger de la pertinence d'une clause par rapport à un état de la recherche. Ces critères seront utilisés ensuite avec des différentes fréquences de réduction pour définir les différentes façons de gérer la base des clauses apprises. Avant de conclure et donner quelques perspectives, nous donnons les résultats expérimentaux obtenus sur les instances de la dernière compétition SAT 2011 et nous proposons également une intégration des stratégies complémentaires dans un solveur parallèle de type portfolio.

2 Définitions, notations et notions préliminaires

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

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

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

Algorithm 1: Solveur CDCL

```
   Données: une formule CNF Σ
   Résultat: SAT ou UNSAT
1  Δ = ∅ ;   /* la base de clauses apprises */
2  tant que (true) faire
3      si (!propagate()) alors
4          si ((c == analyseConflit()) == ∅) alors
5              return UNSAT;
6          Δ = Δ ∪ c;
7          si (timeToRestart()) alors
8              backtrack au niveau 0;
9          sinon
10             backtrack au niveau d'assertion de c;
11     si ((l == decide()) == null) alors
12         return SAT;
13     affecter l à un niveau de décision;
14     si (timeToReduce()) alors
15         clean(Δ);
```

L'algorithme 1 donne le schéma général d'un solveur de type CDCL. Typiquement, un tel solveur peut être assimilé à une séquence de décisions et de propagations des littéraux unitaires, jusqu'à l'obtention d'un conflit. Chaque littéral choisi comme littéral de décision (ligne 12) est affecté suivant une certaine polarité à un niveau

de décision donné (ligne 13). Si tous les littéraux sont affectés *(decide() == null)*, alors $\mathcal{I}$ est un modèle de Σ (lignes 12). À chaque fois qu'un conflit est atteint par propagation (ligne 3), une clause(*no good*) c est calculée en utilisant une méthode d'analyse de conflits donnée, le plus souvent selon le schéma « First UIP » (*« Unique Implication Point »* [18]), et un niveau de *backjump* est calculé en fonction de la clause c. À ce moment, il est possible de prouver l'inconsistance (c est la clause vide) de la formule (ligne 4). Si ce n'est pas le cas, un saut arrière est effectué et le niveau de décision devient égal au niveau de *backjump* (ligne 9). Ensuite, certains solveur CDCL forcent le redémarrage (diverse stratégies sont possibles[14]) et dans ce cas, on remonte en haut de l'arbre de recherche (ligne 7). Finalement, la base de clauses apprises peut être réduite lorsque celle-ci est considérée comme étant trop volumineuse (ligne 14).

Cette dernière composante, bien que souvent omise de la description des démonstrateurs CDCL, est pourtant cruciale pour leurs performances. En effet, conserver un grand nombre de clauses peut avoir des conséquences néfastes sur l'éfficacité de la propagation unitaire, tandis que supprimer trop de clauses peut faire perdre le bénéfice de l'apprentissage.

Algorithm 2: Stratégie de réduction

Données: Δ L'ensemble de clauses apprises de taille n
Résultat: Δ L'ensemble de clauses apprises de taille $n/2$

```
1  si faireReduction() alors
2      TrierLesClausesApprises() ;  /* Selon les
       critères définies */
3      limit = n/2;
4      ind = 0;
5      tant que ind < limit faire
6          clause = Δ[ind] ;
7          si clause.size() > 2 alors
8              EliminerLaCLause() ;
9          sinon
10             GarderLaClause() ;
11         ind++;
12 retourner Δ ;
```

L'algorithme 2 donne le schéma d'une stratégie de réduction.

Généralement, une stratégies de réduction de la base de clauses apprises contient les deux parties suivantes :

2.1 Les clauses à conserver/enlever (lesquelles ?)

De nombreux critères ont été proposés pour estimer la qualité des clauses apprises. La question de déterminer si une clause est utile est liée à la question difficile de trouver une preuve minimale. La plupart des approches conservent systématiquement les clauses de taille deux et les clauses considérées comme raison d'un littéral propagé au moment de l'appel à la fonction réduction. Pour ce qui est des clauses binaires, en plus de pouvoir être gérées facilement, leurs maintient permet de contraindre fortement l'espace de recherche et donc d'améliorer la propagation unitaire. En ce qui concerne les clauses raisons ce n'est pas par soucis d'efficacité mais plus par nécessité qu'elles sont conservées. En effet, supprimer une clause apprise intervenant dans le processus de propagation unitaire ne permet plus d'assurer la construction du graphe d'implications et par conséquent l'analyse de conflits.

2.2 La fréquence de la réduction (Quand ?)

Un autre point important concerne la fréquence avec laquelle la base de clauses apprises est nettoyée. Cette fréquence, calculée de manière heuristique (fonction `faireReduction`), et si elle est mal gérée, peut conduire à rendre le solveur incomplet. En effet, contrairement à la méthode DPLL, l'approche CDCL a besoin des clauses apprises afin de se souvenir de l'espace de recherche qu'elle a déjà exploré. Ceci implique que supprimer une clause peut amener le solveur à parcourir plusieurs fois le même espace de recherche et donc à visiter encore et encore le même conflit sans jamais s'arrêter. Afin de pallier ce problème et de garantir la complétude de la méthode, il est nécessaire que l'intervalle de temps atteigne une taille telle que l'ensemble des clauses apprises pouvant être généré dans cette intervalle permet d'assurer la terminaison de l'algorithme quelles que soient les clauses supprimées précédemment. À l'heure actuelle deux approches sont principalement utilisées afin de gérer cette intervalle. La première approche, implémentée dans MINISAT 2.2 [10], consiste à appeler la fonction de réduction de la base de clauses apprises une fois que la taille de celle-ci a atteint un certain seuil. Ce seuil initialisé à 1/3 de la taille de la base de clauses initiale est augmenté à chaque $100, 150, \ldots, 100 * 1.5^n$ conflits de 10%. La deuxième approche est la stratégie implantée dans le solveur GLUCOSE 2.0 [6]. Elle consiste à réduire la base de clauses apprises lorsque le nombre de conflits obtenu depuis le dernier nettoyage est supérieur à $4000 + 300 \times x$ où x représente le nombre d'appels à la fonction de réduction.

Dans la suite de ce papier, pour avoir un point de vue global sur les différents critères utilisés dans les stratégies de gestion de la base de clauses apprises et leurs efficacités, nous menons une première expérimentation sur les instances de la dernière compétition internationale SAT2011. Nous comparons les deux

critères de suppression de clauses apprises les plus connues CSIDS et LBD avec un critère simple RANDOM qui donne une activité aléatoire aux clauses. Les trois critères sont intégrées dans le solveur MINISAT 2.2 et comme les approches de réduction classique, l'ensemble des clauses apprises est tout d'abord trié par ordre croissant (*r*esp. décroissant) selon leurs activités (*r*esp. LBD, PSM). Une fois les clauses sont triées, la base de clauses apprises sera réduite de moitié. Toutes nos expérimentations (pour celles dans la suite aussi) ont été menées sur un Quad-Core Intel XEON X5550 avec 32Gb de mémoire. 300 instances ont été utilisées, elles sont issues de la compétition SAT2011, catégorie industrielle. Pour l'ensemble des expérimentations le temps CPU est limité à 1 heure pour chaque instance.

Pour chaque solveur, nous indiquons le nombre d'instances résolues (#Résolues) ainsi que le nombre d'instances résolues satisfiables (#SAT) et insatisfiables(#UNSAT). Nous donnons aussi la moyenne du temps nécessaire pour résoudre une instance (tps moy). Le temps moyen est calculé sur l'ensemble de 300 instances, 3600 secondes est utilisé pour les instances non résolues.

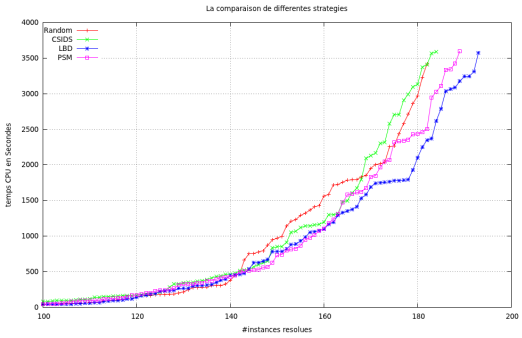


FIGURE 1 – Nombre d'instances résolues en fonction du temps pour le solveur Minisat2.2 intégrant les différentes stratégies de l'état-de-l'art et la stratégie RANDOM.

	Minisat2.2	
Stratégies	#Résolues (#SAT-#UNSAT)	tps moy
RANDOM	182 (**92**-90)	1670.12s
CSIDS	184 (88-96)	1673.83s
LBD	193 (**92**-101)	**1596.66s**
PSM	189 (88-101)	1628.84s

TABLE 1 – La comparaison des stratégies de état-de-l'art avec une stratégie RANDOM intégrée dans MiniSAT2.2

La Figure 1 nous montre les résultats obtenus dans le temps limite.

Le Tableau 1 détaille les résultats obtenus. On voit bien que la stratégie LBD est pour l'instant le meilleure critère dans la littérature. La surprise vient du résultat de la stratégie simple RANDOM. En effet, contrairement à notre prédiction, la stratégie RANDOM nous apparaît compétitive en résolvant seulement 2 instance de moins que CSIDS et 7 instances de moins que PSM. De plus, cette stratégie simple a résolu le plus d'instances SAT (92). Cette observation soulève la question de l'éfficacité et de pertinence des stratégies proposées dans la littérature. Ceci nous conforte dans notre objectif de chercher d'autres critères plus pertinents.

Dans la section suivante, nous présentons nos nouveaux critères qui nous permettent d'évaluer la pertinence des clauses apprises à partir d'un état de la recherche.

3 Nouveaux critères d'identification de l'importance d'une clause apprise

Comme précisé précédemment, les solveurs SAT de type CDCL peuvent être formulés comme un système de preuve par résolution[16, 7]. Le principal inconvénient des méthodes basées sur la résolution générale est lié à leurs complexité en espace. D'un certain point de vue, les solveurs SAT modernes peuvent être vus comme une méthode de preuve par résolution avec une stratégie de suppression de clauses. Par conséquent, la complétude des solveur SAT modernes est fortement liée à la politique de suppression de clauses et aux restarts. Par exemple, supposons que la stratégie de nettoyage de la base de clauses apprises soit très agressive, dans ce cas il nous serait impossible de garantir la complétude du solveur. Définir quelle clause serait utile pour la preuve est une question importante pour l'efficacité des solveurs. Répondre à une telle question est cependant difficile d'un point de vue calculatoire et est très proche du problème qui consiste à trouver une preuve de taille minimale. Pour apporter une solution à ce problème nous définissons des critères permettant d'évaluer la pertinence d'une clause apprise et nous évaluons expérimentalement leurs efficacités.

3.1 Représentation d'une clause apprise

Une clause apprise peut être représentée comme suite : $c = x_1^{i_1} \vee x_2^{i_2} \vee x_3^{i_3} \vee \ldots x_n^{i_n}$ où $x_j^{i_j}$ signifie que le litéral x_j est affecté au niveau i_j. Il faut noter que les niveaux des littéraux d'une même clause varient au cours de la recherche. En utilisant cette information dynamique, nous allons définir deux critères, basés les

niveaux des littéraux dans une clause apprise, pour estimer la pertinence de cette clause.

3.2 L'activité basée sur le niveau du saut arrière

Chaque fois qu'une clause apprise est générée, elle permet au solveur d'effectuer un retour en arrière à un certain niveau de l'arbre de recherche. Dans notre approche, le niveau de saut arrière : BTL (BackTrack Level) est utilisé pour estimer l'importance d'une clause apprise. Une clause est considérée pertinente si elle permet au solveur d'effectuer un saut en arrière à un niveau le plus haut possible (le plus proche de la racine). L'intuition est basée sur les faits suivants. D'abord, les littéraux de décision choisis en haut de l'arbre de recherche correspondent aux variables les plus pondérés selon VSIDS et donc les plus importantes. Deuxièmement, chaque clause apprise est considérée pertinente si elle nous permet de couper des branches en haut de l'arbre de recherche. En effet, plus haut est le niveau de backtrack, plus la clause apprise partage des variables avec la partie la plus haute de la branche, et donc plus elle sera utilisée pour couper l'espace.

Notre critère est défini comme suit : pour chaque clause apprise c, au moment de sa génération, nous lui associons une valeur $BTL(c)$, le niveau du saut arrière. Pour toutes les clauses apprises, cette valeur de BTL peut être mise-à-jour comme suit : si une clause apprise c est utilisé pour propager un littéral au niveau i, et $i < BTL(c)$, alors nous adaptons $BTL(c) = i$. De cette façon, nous gardons son niveau le plus haut, où la clause a été la raison d'un littéral propagé.

Pour analyser et valider cette hypothèse, un ensemble d'expérimentations ont été mené. La Figure 2 reporte, pour quelques instances représentatives, le nombre de fois où une clause avec une certaine valeur de BTL a été utilisée durant le processus de propagation unitaire. Dans cette expérimentation, lorsqu'une clause c provenant de la base de clauses apprises est utilisée dans le processus de propagation unitaire nous incrémentons $\alpha(BTL(c))$ (qui correspond au nombre de fois où une clause avec une telle valeur de BTL est utilisée pour propager un littéral).

D'après la Figure 2, les clauses apprises possédant une petite valeur de BTL (axe des x) sont plus souvent utilisées dans le processus de propagation (axe des y) que les clauses possédant une grande valeur de BTL. Si nous regardons plus en détail, nous pouvons voir que les clauses les plus utilisées ont une valeur de BTL inférieur à 10. Nous pouvons donc affirmer, puisque l'expérimentation a été menée sur un très large panel d'instances, que pour la majorité des instances considérées la distribution de la valeur de BTL ressemble à celle des deux courbes en haut de la Figure 2.

3.3 La distance basée sur les niveaux

Pour une clause apprise $c = x_1^{i_1} \vee x_2^{i_2} \vee x_3^{i_3} \vee \ldots x_n^{i_n}$, elle a deux littéraux $x_j^{i_j}$ et $x_k^{i_k}$ qui est de niveau plus haut(respectivement plus bas) dans l'arbre de recherche, telle que $\forall x_m^{i_m} \in c, i_m \geq i_j$ et $\forall x_m^{i_m} \in c, i_m \leq i_k$. Nous définissons la distance d'une clause apprise comme suit : $Dist(c) = i_k - i_j$; d'où i_k et i_j sont respectivement les niveaux plus bas et plus haut d'un littéral dans la clause c. Ce critère est intéressant, supposons qu'on a une clause $c = x_1^{45} \vee x_2^{45} \vee x_3^{45}$, $BTL(c) = 45$ et $Dist(c) = 0$. Selon le critère BTL, cette clause est considéré peu importante puisqu'elle a une valeur BTL grande, cependant si on affectait $\neg x_1$ en haut de l'arbre de recherche par exemple au niveau 1, cette clause c a de fortes chances d'être utilisée pour propager un littéral au niveau 1, donc elle est importante dans ce cas là. Cette différence entre le niveaux max et min d'une clause, évaluent l'étendu des litéraux de la clauses sur la branche courante.

On peut également constater que pour chaque clause apprise c on a toujours $LBD(c) \leq Dist(c) + 1$.

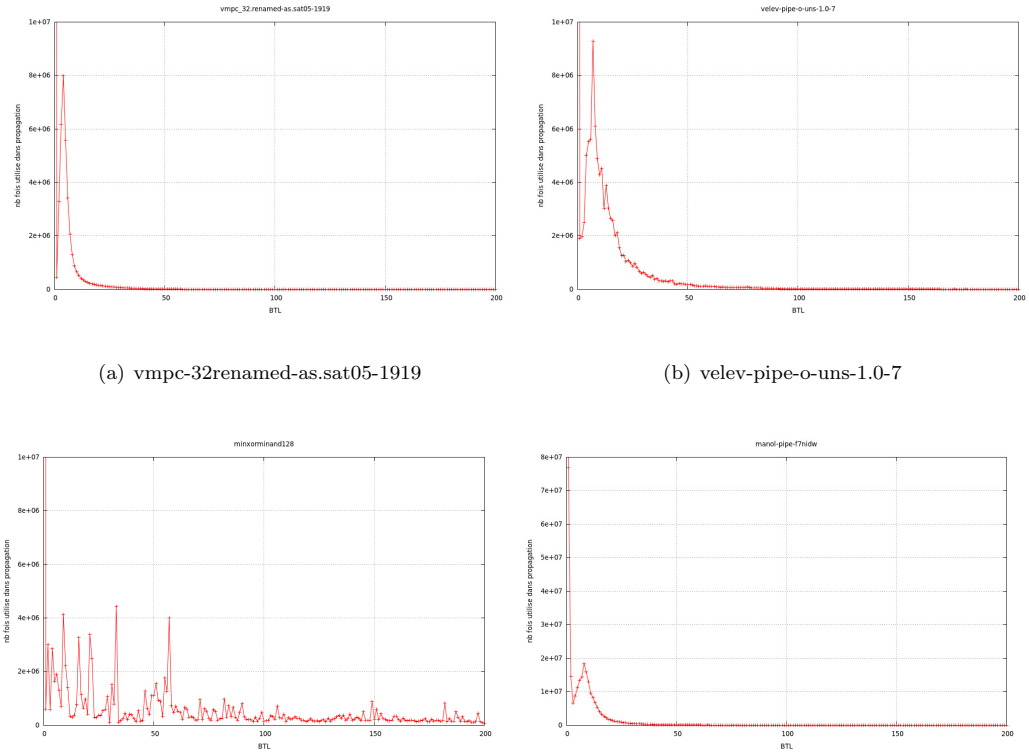
4 Fréquences d'élimination

Comme expliqué précédement dans la section 2.1, une stratégie d'élimination doit répondre aux deux questions suivantes :

- Quelle type de clauses supprimer ou conserver ?
- Avec quelle fréquence doit-on réduire la base de clauses apprises ?

Nous avons déjà défini deux nouveaux critères pour identifier les clauses apprises pertinentes, nous allons maintenant analyser l'impact de la fréquence de réduction de la base de clauses apprises sur les performances des solveurs SAT.

La fréquence de réduction est importante, puisqu'elle a une influence directe sur la durée de vie d'une clause apprise. On peut noter que l'importance d'une clause peut évoluer au cours de la recherche. Une clause apprise est par exemple "1- empowerment" au moment de sa génération, alors qu'elle peut ne plus l'être après la génération d'autres clauses apprises [16]. Une clause apprise est 1-empowerment par rapport à l'asserting litéral si ce litéral ne peut pas être impliqué par unit propagation au niveau du saut sans ajout de la clause apprise. Une clause actuellement considérée comme peu importante pourrait donc être nécessaire par la suite ou inversement. En général, une réduction agressive permet de garder une base de clauses apprises plus petite et maintenir une bonne vitesse du propagation unitaire, mais elle risque aussi d'enle-



(a) vmpc-32renamed-as.sat05-1919

(b) velev-pipe-o-uns-1.0-7

(c) minxorminand128

(d) manol-pipe-f7nidw

FIGURE 2 – Nombre de fois qu'une clause est utilisée par PU en fonction de BTL

ver des clauses apprises nécessaires. En revanche, une réduction prudente ne réduit pas souvent la base de clauses apprises afin d'éviter de supprimer des clauses pertinentes. Cependant, une grande base de clauses apprises a pour effet de ralentir le processus de propagation unitaire et donc de dégrader les performances du solveur.

Dans cette section, nous allons présenter trois fréquences différentes afin de gérer l'intervalle entre deux réductions.

4.1 Fréquence de Minisat - FreqMiniSAT

La première approche que nous avons choisi est celle implémentée dans MINISAT 2.2 [10] qui est le solveur le plus utilisé actuellement. Sa fonction de réduction de la base de clauses apprises est appelée une fois que la taille de celle-ci dépasse un certain un seuil, ce seuil est initialisé à 1/3 de la taille de la base de clauses initiale. Ce seuil est augmenté de 10% lorsque le nombre de clauses apprises est supérieur à 100×1.5^n où n représente le nombre d'augmentations. Les differentes constantes sont fixées expérimentalement.

4.2 Fréquence de Glucose – FreqGlucose

La deuxième fréquence que nous avons choisi est la stratégie implantée dans le solveur GLUCOSE 2.0 [5]. Elle consiste à appeler la fonction de réduction de la base de clauses apprises lorsque le nombre de conflits obtenu depuis le dernier appel est supérieur à $4000 + 300 \times n$, n représente le nombre d'appels à la fonction de réduction.

Avec cette fréquence, la fonction de réduction de la base de clauses apprises sera appelé plus souvent qu'avec avec FreqMiniSAT.

4.3 Une fréquence prudente – FreqPrudente

Nous avons présenté une fréquence classique (la fréquence de Minisat2.2) et une fréquence qui réduit la base de clauses apprises plus souvent (la fréquence de Glucose2.0). Maintenant, nous allons définir une fréquence prudente . Nous voulons qu'avec cette fréquence, la fonction de réduction de la base de clauses

apprises soit appelée moins souvent, les clauses apprises seront gardées plus longtemps qu'auparavant. Donc, une suite géométrique est définie. La fonction de réduction est appelé lorsque le nombre de nouveaux conflits est supérieur à 4000×1.2^n, n représente le nombre de réductions.

Avec cette fréquence de réduction, la durée de vie d'une clause apprise est plus longue.

Dans la section suivante, une comparaison des performances de ces trois fréquences de réduction sera donnée.

5 Expérimentations

Dans cette section, nous allons d'abord comparer les performances de nos nouveaux critères avec celles de l'état de l'art en utilisant les différentes fréquences de réduction présentées précédemment. Nous sélectionons ensuite quatre stratégies complémentaires, que nous intégrerons dans MANYSAT [13] (un solveur parallèle de type portfolio).

Notons que pour toutes les stratégies de réduction implémentées, la base de clauses apprises est réduite de moitiée.

L'ensemble des résultats expérimentaux reportés dans cette section ont été obtenus sur un Quad-core Intel XEON X5550 avec 32Gb de mémoire. Le temps CPU est limité à 3600 secondes. Pour ces expérimentations nous avons utilisé les 300 instances de la catégorie Application de la dernière compétition SAT [1]. Toutes les instances sont pré-traitées à l'aide de SATElite [9].

5.1 Intégration dans minisat 2.2

5.1.1 Comparaison avec la fréquence de minisat 2.2

Nous comparons d'abord nos critères BTL et DISTANCE et les trois critères de l'état de l'art implémentés dans MINISAT 2.2 avec sa propre fréquence de réduction.

	Minisat2.2-F1	
Stratégies	#Résolues (#SAT-#UNSAT)	tps moy
CSIDS	184 (88-96)	1673.83s
LBD	193 (92-101)	1596.66s
PSM	189 (88-101)	1628.84s
BTL	196 (90- **106**)	1566.56s
DISTANCE	**197** (**93**-104)	**1561.94s**

TABLE 2 – Comparaison avec les critères de état-de-l'art intégrés dans MiniSAT2.2 avec FreqMiniSATt

Le tableau 8 nous montre les résultats obtenus dans le temps limite. Pour chaque critère, nous indiquons le nombre d'instances résolues (#Résolues) ainsi que le nombre d'instances résolues satisfiables (#SAT) et insatisfiables (#UNSAT). Nous donnons aussi la moyenne du temps nécessaire pour résoudre une instance (tps moy). Le temps moyen est calculé sur l'ensemble de 300 instances, 3600 secondes est utilisé si l'instance n'est pas résolue en 1h.

Il est claire que nos nouveaux critères BTL et DISTANCE sont nettement meilleurs que les trois critère de l'état de l'art. Avec le critère DISTANCE, le solveur MINISAT 2.2 résout 197 instances (93 SAT et 104 UNSAT), ce qui est meilleur que les autres critères, il est aussi le plus rapide des cinq. L'autre critère BTL résout seulement une instance de moins, est classé deuxième et il a résolu le plus d'instance UNSAT.

Le tableau 3 liste les instances pour les cinq critères, les instances résolues par un seul critère en moins d'une heure. Nous pouvons constater que généralement nos deux critères sont meilleurs sur le plupart des instances, mais il n'y a pas de critère qui domine tous les autres sur l'ensemble des instances. Cette constatation nous inspire l'idée de différencier les bases de clauses apprises dans un solveur parallèle de type portfolio.

5.1.2 Comparaison avec la fréquence de glucose 2.0

Pareillement, une comparaison avec FreqGlucose est donnée dans cette section.

	Minisat2.2-F2	
Stratégies	#Résolues (#SAT-#UNSAT)	tps moy
CSIDS	189 (**96**-93)	1605.96s
LBD	189 (92-97)	1588.82s
PSM	188 (89-99)	1594.60s
BTL	189 (91-**98**)	1584.00s
DISTANCE	190 (92-**98**)	1550.03s

TABLE 4 – Comparaison avec les critères de état-de-l'art intégrés dans MiniSAT2.2 avec FreqGlucose

Selon le tableau 4, on peut voir que les cinq critères, avec une fréquence assez agressive, ne présentent pas beaucoup de différence ni en nombre d'instances résolues, ni en temps moyen de calcul. Notre critère DISTANCE gagne légèrement.

Un tableau d'instances résolues par un seul critère en moins d'une heure est également présenté dans le tableau 5.

5.1.3 Comparaison avec la fréquence prudente

Enfin, nous comparons les cinq critères avec FreqPrudente que nous avons défini, une fréquence géométrique qui garde les clauses apprises avec une durée plus longue.

Cette fois, notre critère DISTANCE est encore le meilleur, il résout 198 instances au total (93 SAT et

Instance	Statues	CSIDS	LBD	PSM	BTL	Distance
AProVE07-01	UNSAT	-	-	-	**2825.70s**	-
gss-27-s100	SAT	-	-	-	-	**1192.10s**
myciel6-tr.used-as.sat04-320	UNSAT	-	-	-	**2087.45s**	-
slp-synthesis-aes-bottom14	UNSAT	-	-	**2434.34s**	-	-
smtlib-qfbv-aigs-VS3-benchmark-S2-tseitin	UNSAT	-	-	-	**3002.81s**	-
vmpc_34.renamed-as.sat05-1926	SAT	-	**92.57s**	-	-	-

TABLE 3 – 2011 SAT Compétition, Industriel : temps (s) résultats 5 critères avec FreqMiniSAT

Instance	Statut	CSIDS	LBD	PSM	BTL	Distance
aes-32-5-keyfind-1	SAT	**1129.55**	-	-	-	-
dated-10-17-u	UNSAT	-	-	**1825.41**	-	-
dated-5-13-u.cnf	UNSAT	-	-	**2332.84**	-	-
homer16.shuffled	UNSAT	-	-	-	-	**2260.10**
k2fix-gr-rcs-w8.shuffled	UNSAT	**3180.65**	-	-	-	-
ndhf-xits-19-UNKNOWN	SAT	**6.43**	-	-	-	-
slp-synthesis-aes-bottom14	UNSAT	-	-	**2920.47**	-	-
total-10-17-u.cnf	UNSAT	-	-	**2338.94**	-	-

TABLE 5 – 2011 SAT Compétition, Industriel : temps (s), résultats des 5 critères avec FreqGlucose

	Minisat2.2-F3	
Stratégies	#Résolues (#SAT-#UNSAT)	tps moy
CSIDS	192 (91-101)	1611.71s
LBD	197 (**95**-102)	1556.56s
PSM	191 (92-99)	1571.07s
BTL	196 (94-102)	1577.00s
DISTANCE	**198** (93-**105**)	**1514.18s**

TABLE 6 – Comparaison avec les critères de état-de-l'art intégrés dans MiniSAT2.2 avec FreqPrudente

105 UNSAT) (voir le tableau 6). Il est aussi le plus rapide. Cependant, le critère LBD a aussi une bonne performance, il a résolu seulement une instance de moins et plus d'instances SAT (95).

5.2 Intégration dans manysat

Notons que l'idée d'un solveur parallèle de type portfolio est de laisser des solveurs séquentiels ayant des stratégies complémentaires à travailler en mode compétition/coopération afin d'être le premier à résoudre l'instance. Alors la sélection des stratégies complémentaires est très importante pour les performances d'un solveur parallèle de type portfolio.

En analysant les performances des stratégies que nous avons présenté précédemment, nous avons choisi les 4 stratégies : CSIDS, LBD, BTL et DISTANCE avec la fréquence FreqPrudente qui nous apparaît plus performante. Nous allons maintenant intégrer ces quatre stratégies dans MANYSAT (chaque cœur utilise une stratégie différente).

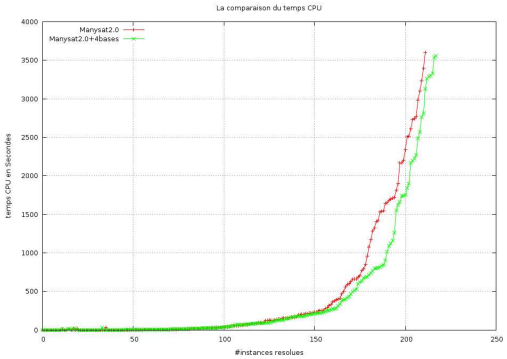


FIGURE 3 – Comparaison de Manysat2.0 et Manysat2.0 avec des strétagies de réduction différentes

Stratégies	#Résolues (#SAT-#UNSAT)	tps moy
Manysat2.0	213 (99-114)	1339.63s
Manysat2.0+4bases	**219 (103-116)**	**1267.07s**

TABLE 8 – Comparaison de Manysat2.0 et Manysat2.0 avec des strétagies de réduction différentes

La figure 3 nous montre qu'en diversifiant les stratégies de nettoyage de la base de clauses apprises, la performance du solveur est bien améliorée.

6 Conclusion et Perspectives

Dans ce papier, nous avons étudié une composante très importante du solveur SAT moderne, la stratégie

Instance	Statut	CSIDS	LBD	PSM	BTL	Distance
aaai10-planning-ipc5-pathways-17-step21	SAT	-	**756.03**	-	-	-
aes-32-3-keyfind-1	SAT	-	-	-	-	**591.76**
comb1.shuffled	UNSAT	-	-	-	**2954.96**	-
E00N23	UNSAT	**3310.27**	-	-	-	-
sokoban-sequential-p145-microban-sequential.050-NOTKNOWN	SAT	-	-	-	**3484.32**	-
sokoban-sequential-p145-microban-sequential.070-NOTKNOWN	SAT	-	-	**982.25**	-	-
vmpc-34.renamed-as.sat05-1926	SAT	-	-	-	**1492.39**	-
vmpc-35.renamed-as.sat05-1921	SAT	-	**2532.44**	-	-	-

TABLE 7 – 2011 SAT Compétition, Industriel : temps (s), résultats 5 critères avec FreqPrudente

de réduction de la base de clauses apprises. Motivé par les résultats compétitifs d'une comparaison entre un critère simple (aléatoire) et ceux utilisés par les solveurs état de l'art, nous avons introduit deux nouveaux critères permettant d'identifier les clauses pertinentes pour la suite de la recherche.

Ces deux critères (plutôt statiques même ils peuvent être ajusté dynamiquement) visent à garder les clauses qui permettent d'élaguer l'espace de recherche. Nos critères sont basés sur les niveaux d'affectations des littéraux de la clause apprise. Cela nous permet d'identifier les clauses apprises qui pourront être utilisées dans le processus de propagation en haut de l'arbre de recherche.

Afin de justifier la pertinence de nos critères, nous avons évalué les stratégies de réduction avec différentes fréquences, et démontré expérimentalement que nos deux nouveaux critères sont meilleurs que ceux de l'état de l'art.

Nous avons montré aussi qu'en diversifiant les stratégies de nettoyage de la base de clauses apprises dans un solveur parallèle de type portfolio, la performance du solveur est bien améliorée.

En perspective, nous envisageons de mener une étude pour determiner le nombre de clauses apprises à éliminer à chaque étape. Dans le cadre parallèle, les clauses apprises sont échangées en fonction de la taille, trouver d'autres critères pour déterminer les clauses les plus pertinentes à échanger est une perspective prometteuse.

Références

[1] International sat 20011 competition - 13th international conference on theory and applications of satisfiability testing. Juin 2011.

[2] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Sais. A generalized framework for conflict analysis. In *SAT*, pages 21–27, 2008.

[3] Gilles Audemard, Youssef Hamadi, Said Jabbour, and Lakhdar Sais. A generalized framework for conflict analysis. Technical report TR-2008-34, Microsoft Research, 16 Pages, ftp ://ftp.research.microsoft.com/pub/tr/TR-2008-34.pdf, 2008.

[4] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. On freezing and reactivating learnt clauses. In *SAT*, pages 188–200, 2011.

[5] Gilles Audemard and Laurent Simon. Glucose : a solver that predicts learnt clauses quality. Technical report, 2009. SAT 2009 Competition Event Booklet.

[6] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, pages 399–404, 2009.

[7] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Intell. Res. (JAIR)*, 22 :319–351, 2004.

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

[9] Niklas Eén and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *SAT*, pages 61–75, 2005.

[10] 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.

[11] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, pages 502–518, 2003.

[12] Carla P. Gomes, Bart Selman, and A. Kautz. Boosting combinatorial search through randomization. In *AAAI/IAAI*, pages 431–437, 1998.

[13] Y. Hamadi, S. Jabbour, and L. Sais. Manysat : a parallel sat solver. *JSAT*, 6(4) :245–262, 2009.

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

[15] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *DAC*, pages 530–535, 2001.

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

[17] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

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

Approche unifiée pour la Décomposition et la Résolution de PCSP: étude expérimentale sur FAP.

Zineb Habbas [1] **Sébastien Martin** [1] **Lamia Sadeg-Belkacem** [123] **Daniel Singer** [1]

[1] LCOMS, Université de Lorraine, Ile du Saulcy, 57045 Metz cedex, France
[2] Laboratoire Mathématiques Appliquées, EMP, Alger, Algérie
[3] Ecole Nationale Supérieure d'Informatique, ESI, Algérie
{zineb.habbas, sebastien.martin, daniel.singer}@univ-lorraine.fr, l_sadeg@esi.dz

Résumé

Cet article présente de façon formelle et unifiée différentes méthodes de décompositions basées sur la multicoupe pour résoudre les PCSP (Partial Constraint Satisfaction Problems). Les PSCP constituent une classe générale de problèmes combinatoires incluant par exemple le problème MaxSAT, le problème d'affectation quadratique, le problème de coloration de graphe .... Afin d'illustrer notre démarche, nous accordons dans ce papier une attention particulière au problème d'affectation de fréquence (Frequency Assignment Problem : FAP). Les méthodes exploitant les décompositions arborescentes sont connues pour être les meilleures en terme de complexité temporelle théorique. Cependant, ces méthodes ne sont pas suffisamment efficaces face aux problèmes du monde réel comme FAP. Dans cette étude, nous proposons une approche de décomposition visant à résoudre rapidement les grandes instances et de manière heuristique. Le principe de notre méthode est de générer k sous-problèmes ou k clusters inter-connectés, pouvant être résolus par des algorithmes séquentiels ou parallèle, de façon exacte ou heuristique. Afin d'explorer largement cette approche, nous considérons plusieurs variantes selon différents critères qui pourraient affecter la performance de l'algorithme de résolution et la qualité de la solution globale. Nous considérons la décomposition Mincut ou multicoupe de taille minimale (déjà explorée dans la littérature) et la décomposition Maxcut ou multicoupe de taille maximale ainsi que les algorithmes de résolution correspondants. Toutes les décompositions sont modélisées selon le paradigme de la programmation mathématique et mises en oeuvre dans l'environnement Cplex. Afin de valider notre méthodologie nous avons expérimenté certaines variantes de décomposition sur des benchmarks du problème FAP, connus dans la littérature. Nous présenterons des premiers résultats encourageants obtenus en utilisant un algorithme génétique pour résoudre les sous problèmes ainsi qu'une méthode de recherche locale.

1 Introduction

Les PCSP (Partial Constraint Satisfaction Problems) sont des problèmes de satisfaction de contraintes auxquels des pénalités sont affectées aux contraintes non satisfaites. Dans cet article, nous nous intéressons aux PCSP binaires où toutes les contraintes sont composées deux variables. La résolution d'un PCSP consiste à affecter des valeurs à toutes les variables tout en minimisant la somme des pénalités induites par les contraintes violées. Les PCSP forment une classe générale de problèmes incluant de façon non exhaustive le problème MaxSAT, le problème d'affectation quadratique ou le problème de coloration de graphe. Le problème d'affectation de fréquence ou FAP, un des problèmes combinatoires les plus connus dans la littérature est considéré dans ce papier comme une application afin de valider notre approche. Il existe, comme pour les CSP, des solveurs génériques basés sur des méthodes exactes et/ou des métaheuristiques pour rechercher la solution globale d'un PCSP. Ces derniers peuvent surprendre par leur efficacité dans certains cas. Cependant, dans la plupart des cas leur efficacité est rstreinte face à des problèmes difficiles de grande taille. Pour faire face à cette situation, plusieurs approches récentes proposent d'explorer les propriétes structurelles du graphe de contraintes associé au problème. En particulier, les méthodes exploitant les décompositions arborescentes [14] sont connues pour être parmi les meilleures en terme de complexité temporelle

théorique. Leur efficacité pratique n'a cependant pas encore été prouvée sur des problèmes de grande taille. Dans ce papier, notre objectif principal est de proposer des algorithmes visant à résoudre rapidement, mais pas nécessairement de façon exacte des problèmes de grande taille. Notre approche utilise des techniques de décomposition exploitant la multicoupe, techniques consistant à diviser le graphe de contraintes associé à un PCSP en k sous-graphes ou k clusters interconnectés par un ensemble de contraintes qui constitue la multicoupe. Cette étape de décomposition utilisée comme une étape de prétraitement induira bien sûr une ou plusieurs techniques de résolution différentes. Nous allons présenter ici des décompositions guidées par deux algorithmes de résolution possibles. La première stratégie dite "orientée sous-graphes" résout le sous-problème associé à chaque sous-graphe avant de vérifier la multicoupe pour obtenir une solution globale quasi-optimale du problème initial. Dans ce cas, une décomposition dite "équilibrée" formée de k clusters regroupant quasiment le même nombre de variables du problèmes, serait apriori plus appropriée pour la résolution efficace du problème. La deuxième stratégie appelée " Orientée inter-graphes" résout d'abord le problème induit par la multicoupe puis les sous problèmes induits par les k clusters. Dans ce cas, maximiser la multicoupe devrait être une stratégie plus appropriée. Une des contributions de ce papier est de présenter de façon unifiée, en utilisant la programmaton mathématique comme cadre formel, une méthodologie générénérale de résolution des PCSP. Afin de valider notre méthodologie, nous avons expérimenté certaines variantes de décomposition associées à des méthodes de résolution possibles sur des benchmarks FAP réels. Les premiers résultats expérimentaux présentés dans ce papier sont obtenus en utilisant un algorithme génétique ad-hoc que nous avons développé ainsi que 1-opt comme heuristique de réparation locale.

Le reste du papier est organisé de la manière suivante. La section 2 donne une définition formelle du PCSP et de FAP ainsi qu'un état de l'art succinct des méthodes de résolution. La section 3 propose une approche unifiée pour étudier à la fois les stratégies de résolution "orientée sous-graphes" et "orientée inter-graphes" et exploitant les décompositions basées sur la multi-coupe en terme de graphe. La section 4 est consacrée à la présentation de la programmation linéaire en nombres entiers(Integer Linear Program : ILP) de ces stratégies de décomposition ainsi que leurs modèles de résolution associés. La section 5 introduit trois algorithmes de résolution génériques utilisés pour l'expérimentation ainsi que les premiers résultats expérimentaux obtenus. La section 6 permet de conclure ce papier par quelques observations et perspectives.

2 Résolution des PCSP et FAP : Formalisme et état de l'art

2.1 Définition formelle d'un PCSP

Selon Koster et al. [11], un PCSP (Partial Constraint Satisfaction Problem) est défini par un quintuplet $< X, D, C, P, Q >$ où $< X, D, C >$ est un CSP classique consistant en un ensemble de variables $X = \{x_1, x_2, \ldots, x_n\}$, un ensemble de domaines contenant les valeurs possibles de chaque variable $D = \{D_1, D_2, \ldots, D_n\}$ et un ensemble C de contraintes binaires. P est l'ensemble des fonctions de pénalités sur les contraintes, associant une valeur réelle à chaque couple d'éléments dans $D_i \times D_j$ tel que $\{x_i, x_j\} \in C$ et Q est l'ensemble des fonctions de pénalités sur les variables, fonction associant une valeur réelle à chaque instanciation de variable de X. Si le problème $\mathcal{P}$ n'a pas de pénalité alors $\mathcal{P}$ est un CSP classique. Par ailleurs quelques contraintes dites "hard" ou dures ne peuvent pas être violées par opposition à d'autres contraintes dites "soft" ou souples qui peuvent être violées modulo une pénalité un coût ou pénalité. C_h est l'ensemble des contraintes de $\mathcal{P}$ et une contrainte entre deux variables est notée $(x_i, x_j) \in C$. De ce point de vue un PCSP peut être considéré comme *un graphe de contraintes valué*.

2.2 Définition formelle de MI-FAP

Dû au développement de plus en plus croissant des réseaux sans fils tels que les réseaux GSM, le problème d'affectation de fréquences (FAP pour Frequency Assignment Problem) occupe une place de plus en plus importante au sein de la communauté scientifique qui a fait de lui un des problèmes combinatoires les plus étudiés. Les applications de FAP dans le monde réel sont nombreuses et incluent entre autres la téléphonie mobile, la communications radio, ... Le problème FAP est un problème NP-difficile [8] apparu en 1960's [16], voir aussi [1] pour un excellent état de l'art sur FAP. Dans ce papier, nous nous intéressons à la variante MI-FAP (*Minimum Interference-FAP*) dont l'objectif est d'allouer un nombre restreint de fréquences à un nombre très important d'antennes (émetteurs/récepteurs), tout en minimisant l'interférence globale dans le réseau induite par les réutilisation des fréquences disponibles. Formellement, MI-FAP est un PCSP $< X, D, C, P, Q >$ où X correspond à un ensemble fini d'antennes, D un ensemble de domaines où chaque D_i contient les fréquences possibles pou-

vant être affectées à l'antenne t_i, C est l'ensemble de contraintes binaires reliant deux émetteurs.

2.3 Résolution : travaux antérieurs

Cette section présente quelques travaux ayant fait l'objet d'étude des PCSP et/ou de FAP via les méthodes de décomposition. Dans [5] les auteurs ont proposé une approche de décomposition pour résoudre les instances difficiles du problème MI-FAP. Cette méthode qui peut être appliquée à n'importe quel PCSP, décompose le problème en plusieurs sous-problèmes, puis applique un algorithme résolvant tous les sous-problèmes de façon récursive en vue d'améliorer l'interférence globale. Cet algorithme est générique car les sous problèmes peuvent être résolus indifféremment par une méthode exacte ou une métaheuristique. Plusieurs stratégies de décomposition ont été testées dans le but de comparer leurs performances. Des résultats experimentaux ont montré l'utilité des décompositions dans la résolution des problèmes de grande taille telles que les instances COST259 [6]. Dans [2] le logiciel de partitionement de graphes METIS est utilisé pour décomposer le PCSP initial en plusieurs sous-problèmes qui sont ensuite organisés de façon arborescente autour d'une structure de type DFS (Depth First Search). Le PCSP arborescent résultant est résolu par un système multi-agent distribué. Cette approche n'a pas été appliquée aux PCSP. Quelques chercheurs ont plutôt exploité les propriétés structurelles du problème. Cette méthode est très adaptée au problème FAP. En effet, les instances FAP sont géographiquement structurées. Chaque antenne est localisée dans un espace à deux dimensions. Cette répartition géographique a un impact sur le risque d'interférence entre les antennes. Deux antennes éloignées géographiquement n'engendrent pas d'interférence. Dans [11, 14, 3] les auteurs utilisent une décomposition arborescente pour décomposer le problème et des algorithmes exacts pour le résoudre. Cette approche a amélioré plusieurs bornes inférieures de plusieurs instances du projet CALMA. Certaines instances ont été résolues de façon optimales mais en des temps d'exécution très importants. Pour pallier au problème du temps, C. Terrioux et al. ont proposé dans [9] la méthode BTD (Backtracking bounded by Tree Decomposition). Plus récemment, dans [17] cette méthode a été étendue aux CSP avec des contraintes dites souples et des contraintes de préférence. Bien que l'approche BTD s'est avérée plus performante que la décomposition arborescente, aucun résultat significatif n'a été obtenu sur FAP, à notre connaissance.

3 Partitionnement de Graphe pour Décomposer et Résoudre les PCSP

Cette section présente une formalisation à base de graphes qui permet de présenter de manière unifiée, sous certaines conditions, à la fois différentes méthodes de décomposition et la résolution des PCSP. Après l'introduction des notations utiles, deux méthodes importantes de décomposition de PCSP seront définies comme des problèmes de partitionnement de graphe. La première est ascendante pour une résolution "orientée sous-graphes" et la seconde est descendante pour une résolution "orientée inter-graphes". Ensuite nous soulignerons le fait que la résolution du PCSP elle-même peut être formulée comme un problème de partitionnement de graphe.

Soit $\mathcal{P} = < X, D, C, P, Q >$ un PCSP. On definit $G = (V, E)$ le graphe valué (non orienté) associé à $\mathcal{P}$ de la manière suivante : pour chaque variable $x \in X$ on associe un sommet $v_x \in V$ et pour chaque contrainte (binaire) $(x_1, x_2) \in C$ on definit une arête $(v_{x_1} v_{x_2}) \in E$, un poids $w(v_{x_1} v_{x_2})$ et une pénalité definie dans P associés. Si deux variables $x_1, x_2 \in X$ n'apaprtiennet pas à une même contrainte alors on pourra associer à l'arête $(v_{x_1} v_{x_2})$ un poids $w(v_{x_1} v_{x_2}) = 0$.

Une Réduction de Graphe utile : il est particulièrement intéressant de contracter les sommets qui appartiennent à des *contraintes hard* et qui forment des cliques de G. Cela signifie que, du point de vue de la décomposition, les variables apparaissant dans des contraintes hard ne pourront pas être dans des parties différentes. Plus précisément, pour toute clique $K \subseteq C_h$ tous les sommets v de K sont contractés en un seul sommet $v_K \in V$. Dans la suite de cet article, nous ne considèrerons que les cliques de taille 2 qui correspondent aux contraintes hard du problème. De plus, pour tout $K \in C_h$ et pour tout $x \in X \backslash K$ tq. $\exists x_i \in K$ et $(x, x_i) \in C$ on définit une arête $(v_x v_K) \in E$ et un poids associé $w(v_x v_K)$ égal à $\sum_{x_i \in K} w(x_i, x)$ et pour toute paire $K, K' \in C_h$ tq. $K' \neq K$ on définit une arête $(v_K v_{K'}) \in E$ et un poids associé $w(v_K v_{K'})$ égal à $\sum_{v \in K} \sum_{v' \in K'} w(v, v')$.

Dans la suite de cet article, $G = (V, E)$ dénotera le graphe obtenu après cette réduction sauf mention du contraire.

Nous allons maintenant présenter différentes variantes de décomposition des PCSP selon leur technique de résolution. Grâce à la formulation en termes de graphes, toutes ces variantes pourront être définies comme des problèmes de Partitionnement de Graphes, à savoir : trouver une partition $\mathbb{P}$ de G en k parties $P_1 \cup P_2 \cup ... \cup P_k = V$ sous certaines conditions.

De plus, nous définissons un autre paramètre q qui représentera le déséquilibre maximum accepté entre deux différentes parties.

$[P_i, P_j]$ dénotera l'ensemble des arêtes entre les deux parties $P_i, P_j \subseteq V$.

3.1 Partitionnement de Graphe pour une Résolution Ascendante orientée sous-graphes

Une résolution ascendante consiste à résoudre dans un premier temps, les sous-problèmes associés aux parties $P_1, P_2, ..., P_k$ de la partition $\mathbb{P}$ puis de vérifier dans un deuxième temps, les contraintes souples appartenant à la multicoupe. Cette dernière étape peut d'ailleurs ne pas améliorer la qualité de la solution tout en augmentant son coût. De manière évidente, plus le poids de la coupe de la coupe meilleure devrait être la qualité de la solution globale. En conséquence, la méthode de partitionnement du graphe adéquate dans ce cas est définie de la manière suivante :

(condition d'équilibrage)

$$\min_{i,j \in \{1,...,k\}} |[P_i, P_j]|$$

$$\text{tq.} \quad ||P_i| - |P_j|| \leq q \quad \forall i, j \in \{1, ..., k\}.$$

Cette condition doit être considérée pour éliminer la partition triviale et inutile qui a $k-1$ parties vides et la k^{eme} qui contient tous les sommets de V. Noter que cette méthode de partitionnement sera particulièrement intéressante dans le cas d'une résolution parallèle. Mais, dans le cas d'une résolution séquentielle, si nous ajoutons la condition de niveaux suivante, alors nous pouvons définir pour $k > 2$ un algorithme de résolution qui résoudra d'abord les sous-problèmes associés aux sous-graphes $P_1, P_2, ..., P_k$ avant de de vérifier les contraintes de la multicoupe entre des variables des différentes parties.

(condition par niveau) :

$$|[P_i, P_j]| = 0 \quad \forall i, j \in \{1, ..., k\} \,\text{tq.}\, i \neq j{+}1 \,\text{or}\, j \neq i{+}1$$

La Figure 1(a) respecte la "condition par niveaux" et nous appellerons *Décomposition Niveaux* une telle décomposition. La Figure 1(b) ne respecte pas la "condition par niveaux" et nous appellerons simplement *Graphe Décomposition* une telle décomposition.

3.2 Partitionnement de Graphe pour une Résolution Descendante orientée inter-graphes

L'idée principale derrière la résolution descendante est de résoudre en premier le sous-problème induit par les contraintes de la multicoupe et de résoudre

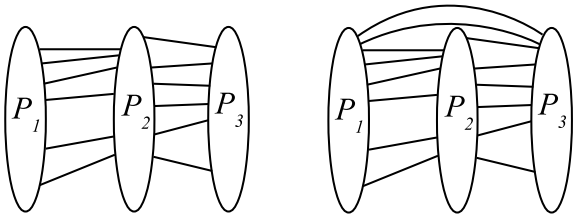


FIGURE 1 – (a) Décomposition Niveaux (b)Graphe Décomposition

ensuite les sous-problèmes associés aux différents éléments de la partition. Ceci conduit naturellement à une méthode, opposée à celle exposée précédemment, qui consiste à rechercher une partition en k petits sous-problèmes reliés entre eux par une multicoupe de poids élevé. Une fois que la meilleure solution est trouvée pour la multicoupe alors le problème restant devient beaucoup plus facile. Le problème de partitionnement de graphe associé à cette méthode peut être défini de la manière suivante

$$\max_{i,j \in \{1,...,k\}} |[P_i, P_j]|.$$

Noter que, dans ce cas, la *condition d'équilibrage* n'est plus nécessaire du faite que tous les éléments de la partition seront non vides, car on a un objectif Max. Par contre, comme dans la première méthode, on peut rajouter la *"condition par niveau"* sur la partition. De plus, une nouvelle condition de voisinage interdisant à un sommet d'une partition d'avoir deux voisins dans deux autres partitions pourrait être utile.

(condition de voisinage) :

$$(v, w) \notin [P_j, P_l] \quad \forall i, j, l \quad \forall u \in P_i \, \forall v \in P_j \, \forall w \in P_l$$

$$\text{tq.}\, (u, v) \in [P_i, P_j]$$

.

La Figure 2(a) viole la *"condition de voisinage"* alors que la Figure 2(b) la vérifie. Lorsqu'une décomposition respecte une telle condition on parlera de *Décomposition voisinage*.

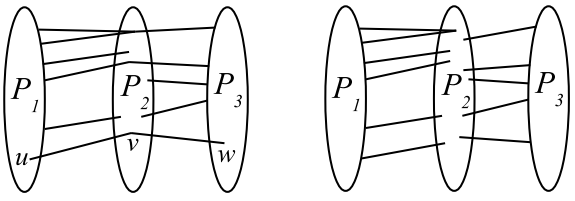


FIGURE 2 – (a) (b) Décomposition voisinage

3.3 Résolution d'un PCSP

Résoudre un PCSP consiste à chercher une répartition des valeurs à l'ensemble des variables problème. Si k représente le nombre de valeurs possibles que peuvent prendre les variables, alors résoudre un PCSP revient à partitionner l'ensemble des variables en k parties tout en respectant les contraintes. Ainsi, de cette manière, résoudre un PCSP est réduit à trouver une partition en k sous-graphes. De manière analogue à la tâche de décomposition, on peut améliorer la formalisation de la résolution sous la forme d'un problème de partitionnement de graphe en réduisant tous les sommets appartenant à une même *contrainte hard* en un seul sommet dont le domaine consistera dans l'ensemble de tous les couples autorisés de valeurs (dans le cas de contraintes binaires).

4 Formulation par la programmation mathématique

Dans cette section, nous proposons de formuler les problèmes de décompositions et de résolution des PCSP à l'aide de programmes quadratiques en nombres entiers. Nous donnons, par la suite, une procédure de linéarisation permettant de transformer nos formulations quadratiques en programmes linéaires en nombres entiers. Ensuite, nous définissons des familles d'inégalités correspondant aux différentes variantes de décompositions présentées dans la section précédente. Enfin, nous donnons une nouvelle famille d'inégalités valides permettant d'améliorer la vitesse de résolution de nos décompositions.

4.1 Programmes quadratiques en nombres entiers

Dans cette section, nous donnons une formulation quadratique en nombres entiers correspondant aux problèmes de décomposition de graphes. Nous définissons, pour chaque sommet $v \in V$ et pour chaque sous ensemble P_i, $i \in \{1, ..., k\}$, une variable binaire x_{vi} égale à 1 si $v \in P_i$ et 0 sinon.

Nous en déduisons que les inégalités linéaires suivantes, sont nécessaires et valides, pour les problèmes de décomposition de graphes.

$$\sum_{i \in \{1,...,k\}} x_{vi} = 1 \ \ \forall v \in V, \tag{1}$$

$$x_{vi} \in \{0, 1\}, \qquad \forall v \in V, \, i \in \{1, ..., k\}, \tag{2}$$

Pour les problèmes de Mincut, la fonction objectif associée est quadratique et s'écrit de la manière suivante

$$\min \sum_{uv \in E} \sum_{i \in \{1,...,k\}} \sum_{j \in \{1,...,k\} \setminus \{i\}} (w_{uv} \times x_{ui} \times x_{vj}) \tag{3}$$

alors que pour les problèmes de Maxcut, la fonction objectif associée est elle aussi quadratique et s'écrit de la manière suivante

$$\max \sum_{uv \in E} \sum_{i \in \{1,...,k\}} \sum_{j \in \{1,...,k\} \setminus \{i\}} (w_{uv} \times x_{ui} \times x_{vj}) \tag{4}$$

sous les contraintes (1-2).

Nous allons désormais introduire les inégalités associées à chaque variante.

- **La condition d'équilibrage** : $||P_i| - |P_j|| \leq q$ $\forall i, j \in \{1, ..., q\}$. Nous permet de déduire que les contraintes suivantes sont valides.

$$\sum_{v \in V} x_{vi} - \sum_{v \in V} x_{vj} \leq q \ \ i, j \in \{1, ..., k\}. \tag{5}$$

  Notons que la valeur absolue a pu être supprimée. En effet, si $|P_i| - |P_j| \leq q$ et $|P_j| - |P_i| \leq q$ alors cela implique que $|(|P_j| - |P_i|)| \leq q$. Comme l'inégalité (5) est valide pour tout i, j ainsi que pour tout j, i, où $i, j \in \{1, ..., k\}$, nous en déduisons que la contrainte est nécessaire et suffisante pour répondre à la condition d'équilibrage.
- **La condition par niveau** : $|[P_i, P_j]| = 0$ si $i \neq j + 1$ ou $j \neq i + 1$, peut-être formulée par les inégalités suivantes :

$$x_{ui} + x_{vj} \leq 1, \quad \forall uv \in E, \, i \in \{1, ..., k\} \quad j \in \{1, ..., k\} \setminus \{i - 1, i, i + 1\}. \tag{6}$$

  Ces contraintes nous garantissent qu'il n'existera pas d'arête entre deux ensembles non consécutifs de la partition.
- **La condition de voisinage :** Elle peut être formulée par les inégalités suivantes

$$x_{ui} + x_{vj} + x_{wl} \leq 2, \quad \forall uv, vw \in E \quad i, j, l \in \{1, ..., k\}, i \neq j \neq l. \tag{7}$$

  En effet, s'il existe deux arêtes $uv, vw \in E$ connectées par le sommet v alors, afin de respecter la condition de voisinage, les arêtes uv et vw peuvent apparaître dans la coupe si et seulement si uv et vw connectent les mêmes sous-ensembles, c.à.d. qu'elles sont entres les deux mêmes ensembles de la partition.

4.2 Programmes linéaires en nombres entiers

Les programmes quadratiques en nombres entiers sont difficiles à résoudre et demande souvent énormément de temps. C'est pour cela que nous proposons dans la suite une procédure de linéarisation afin d'utiliser les outils de la programmation linéaire en nombres entiers.

Notre procédure de linéarisation nécessite l'ajout de nouvelles variables associées aux arêtes. Nous notons par $\mathcal{E}_\mathcal{P}$ l'ensemble des arêtes appartenant à la coupe, c.à.d. les arêtes reliant deux sous-ensembles $\cup_{i\in\{1,...,k\}} \cup_{j\in\{1,...,k\}\setminus\{i\}} [P_i, P_j]$ définies par la partition $\mathbb{P}$. Nous associons, pour chaque arête $e \in E$, une variable binaire y_e égale à 1 si $e \in \mathcal{E}_\mathcal{P}$ et 0 sinon.

Afin de s'assurer que le problème linéarisé et le problème quadratique possèdent des solutions équivalentes pour les deux objectifs Mincut et Maxcut, nous proposons d'ajouter les contraintes suivantes :

$$x_{ui} + x_{vj} - 1 \leq y_{uv}, \quad \forall uv \in E,\, i \in \{1,...,k\},\, j \in \{1,...,k\} \setminus \{i\}, \quad (8)$$

$$2 - (x_{ui} + x_{vi}) \geq y_{uv}, \, \forall uv \in E,\, i \in \{1,...,k\}. \quad (9)$$

Les contraintes (8) impliquent que y_{uv} est égale à 1 si $x_{ui} = x_{vj} = 1$ et sinon y_{uv} est égale à 0 ou 1, pour tout $uv \in E$. Les contraintes (9) impliquent que y_{uv} est égale à 0 si $x_{ui} = x_{vi} = 1$ et sinon y_{uv} est égale à 0 ou 1, pour tout $uv \in E$. Les contraintes (8) sont uniquement nécessaires dans le cas des problèmes de Mincut alors que les contraintes (9) sont uniquement nécessaires dans le cas des problèmes de Maxcut.

La fonction objectif linéarisée correspondant aux problèmes du Mincut est donnée par

$$\min \sum_{e\in E} w_e y_e \quad (10)$$

alors que pour les problèmes du Maxcut la fonction objectif linéarisée est donnée par

$$\max \sum_{e\in E} w_e y_e. \quad (11)$$

Le programme linéaire en nombres entiers (P) correspondant aux problèmes de décomposition est donné par

$$\min / \max \sum_{e\in E} w_e y_e \quad (12)$$

$$\sum_{i\in\{1,...,k\}} x_{vi} = 1 \qquad \forall v \in V\ , \quad (13)$$

$$x_{ui} + x_{vj} - 1 \leq y_{uv} \quad \forall uv \in E,\, i \in \{1,...,k\},\, j \in \{1,...,k\} \setminus \{i\}, \quad (14)$$

$$2 - (x_{ui} + x_{vi}) \geq y_{uv} \quad \forall uv \in E,\, i \in \{1,...,k\}, \quad (15)$$

$$x_{ui} \in \{0,1\} \qquad \forall u \in V,\, i \in \{1,...,k\}, \quad (16)$$

$$y_e \in \{0,1\} \qquad \forall e \in E. \quad (17)$$

Remarquons que toutes les inégalités associées aux différentes variantes restent valides pour ce programme linéaire en nombres entiers.

4.3 Inégalités valides

Le point le plus intéressant de cette formulation est la possibilité de renforcer, à l'aide d'inégalités valides, aussi bien la formulation générale (P), que chaque variante. De plus, la plupart des inégalités valides pour la formulation générale ou pour l'une des variantes peuvent souvent s'étendre à l'ensemble des variantes. En effet, si nous trouvons une inégalité valide permettant de renforcer la formulation associée à l'une des variantes alors nous pouvons espérer qu'elle renforcera, aussi, les formulations associées aux autres variantes.

Nous allons illustrer cette idée en proposant une famille d'inégalités valides pour (P) permettant d'accélérer la résolution du programme linéaire en nombres entiers, nous déduirons que cette famille de contraintes reste valide pour toutes les autres variantes.

Pour $k > 2$ les inégalités (14) indiquent, que si deux sommets u et v d'une arête uv appartiennent à deux ensembles P_i et P_j, alors y_{uv} doit être égale à 1. Il est possible d'améliorer cette inégalité par

$$x_{ui} - \sum_{j\in\{1,...,k\}\setminus\{i\}} x_{vj} - 1 \leq y_{uv}, \quad \forall uv \in E \quad \forall i \in \{1,...,k\}. \quad (18)$$

En effet, pour $uv \in E$, si le sommet u est dans P_i alors y_{uv} doit être égale à 1 si v appartient à un autre ensemble de la partition. Nous avons amélioré la famille d'inégalités (14) pour $k > 2$. En effet, les contraintes (18) dominent les contraintes (14). Il est facile de vérifier que tous les vecteurs d'incidences valides pour (18) sont aussi valides pour (14) alors que l'inverse n'est pas vrai.

Théorème. *Les contraintes* (18) *sont valides pour* (P).

Preuve. Pour tout $k > 2$, $uv \in E$ et un entier $i \in \{1, ..., k\}$, les contraintes

$$x_{ui} + x_{vj} - 1 \leq y_{uv}, \ \forall j \in \{1, ..., k\} \setminus \{i\} \quad (19)$$

sont valides. En additionnant ces contraintes, nous obtenons la contrainte suivante :

$$(k-1)x_{ui} + \sum_{j \in \{1,...,k\} \setminus \{i\}} x_{vj} - (k-1) \leq (k-1)y_{uv}, \quad (20)$$

désormais en additionnant l'inégalité (20), $(k-2)$ fois la contrainte $\sum_{j \in \{1,...,k\}} x_{vj} \leq 1$ (cette contrainte correspond à la contrainte (1) relaxée), l'inégalité $-x_{ui} \leq 0$ et en divisant la contrainte obtenue par $(k-1)$, et en arrondissant, à l'entier inférieur, le membre de droite, nous obtenons la contrainte (18).□

La famille de contraintes (18) reste valide pour l'ensemble des variantes mais améliore les temps de résolution, uniquement, pour les problèmes de Mincut.

5 Résultats expérimentaux

Cette section présente les premiers résultats expérimentaux effectués sur des benchmarks FAP issus du projet CALMA [4]. L'ensemble d'instances considéré est constitué de deux parties. Le premier groupe consitue la famille de problèmes CELAR, regroupant des instances du monde réel issues d'application militaires, quant au deuxième groupe nommé GRAPH correspond à un ensemble d'instances générées aléatoirement respectant toutefois les caractéristiques des instances réelles.

5.1 Méthodes de décomposition

Les méthodes de décomposition présentées formellement dans la section 4 ont été implémentées dans l'environnement Cplex. La table 1 regroupe les résultats obtenus sur toutes les instances MI-FAP de CALMA en utilisant la variante Mincut sur les décompositions dites "Graphe -Décomposition" et "Décomposition Niveau". Nous représentons par Pb le nom de l'instance, k le nombre de partitions et t(s) le temps CPU en secondes.

La constante optim est positionnée à "yes" si les inégalités (18) sont considérées et à "no" sinon. La mention $1h^*$ précise que le solveur Cplex a dû être arrêté au bout d'une heure d'exécution sans trouver la solution optimale dans ce cas. Ce premier test montre de façon générale, que seul l'ajout d'une famille d'inégalités, améliore de façon considérable le temps CPU. De plus toutes les instances pour lesquelles la solution n'a pu être obtenue, ont été améliorées par l'utilisation de la famille d'inégalité (18). Nous n'avons pas présenté ici les résultats relatifs à la variante Maxcut car notre formulation actuelle est peu efficace et nécessite des améliorations. Nous noterons toutefois que ces premiers résultats sont plutôt encourageants et nous laissent espérer que renforcer la formulation mathématique par de nouvelles contraintes permettrait de décomposer de façon optimale toutes les instances en quelques secondes.

Pb	k	t(s)	optim	Pb	k	t(s)	opt
CELAR06	2	1	no	CELAR07	2	1	no
	2	1	yes		2	1	yes
	3	2	no		3	15	no
	3	1	yes		3	3	yes
	4	83	no		4	329	no
	4	2	yes		4	10	yes
GRAPH05	2	1	no	GRAPH06	2	1	no
	2	1	yes		2	1	yes
	3	25	no		3	$1h^*$	no
	3	2	yes		3	19	yes
	4	$1h^*$	no		4	$1h^*$	no
	4	6	yes		4	$1h^*$	yes
GRAPH11	2	37	no	GRAPH13	2	$1h^*$	no
	2	37	yes		2	$1h^*$	yes
	3	$1h^*$	no		3	$1h^*$	no
	3	$1h^*$	yes		3	$1h^*$	yes
	4	$1h^*$	no		4	$1h^*$	no
	4	$1h^*$	yes		4	$1h^*$	yes

TABLE 1 – Variante Mincut

5.2 Algorithmes de résolution

Dans ce papier nous proposons trois algorithmes génériques différents. Nous appelerons par abus de langage *clusters* les sous problèmes P_i induits par les partitions $\mathbb{P}$ ainsi que le problème induit par la multicoupe.

Le premier algorithme dit Bottom-up (Algorithm 1) résout en premier lieu les problèmes associés aux clusters. Cette première étape donne lieu à une première solution globale Sol associée à un coût $cost1$. la deuxième étape de cet algorithme consiste à vérifier parmi les contraintes de la multicoupe celles qui sont satisfaites. Un second coût $cost2$ relatif aux contraintes violées de la multicoupe, devra être cumulé au coût $cost1$. Enfin, la troisième étape optionnelle de cet algorithme consistera à améliorer la qualité de la solution obtenue si c'est nécessaire. Notons que la procédure *Solve* utilisée dans tous nos algorithmes peut correspondre à une méthode de résolution exacte ou à une méthode basée sur des heuristiques et/ou des métaheuristiques.

L'algorithme Top-down (Algorithm 2), contrairement au premier algorithme résout dabord le problème associé à la multicoupe. Soit $sol1$ la solution partielle ainsi obtenueet $cost1$ le coût qui lui est associé. Une

Algorithm 1 Algorithme Bottom-up

1: **Input** : $G =< V, E >$
2: **ouptut** : A global solution Sol , $Cost$ the global cost of Sol.
3: **Decompose**(G, C1, C2, ... Ck, Multi-Cut)
4: **for** i = 1 to k [1] **do**
5: **Solve** (Ci, s_i, $cost_i$)
6: **end for**
7: $Sol \leftarrow s_1 \odot s_2, \ldots \odot s_k$
8: $Cost_1 \leftarrow \sum_{i=1}^{k} cost_i$
9: **Check** (Multi-Cut, Sol, cost2)
10: $Cost \leftarrow cost1 + cost2$
11: **Improve** (Sol, Cost)

deuxième étape résout de façon séquentielle ou parallèle les sous problèmes associés aux k clusters. La troisième étape de cet algorithme est identique à celle de l'algorithme précédent.

Algorithm 2 Algorithme Top-down

1: **Input** : $G =< V, E >$
2: **ouptut** : A global solution Sol , $Cost$ the global cost of Sol
3: **Decompose**(G, C1, C2, ... Ck), Multi-Cut)
4: **Solve**(Multi-Cut, Sol1, cost1)
5: **for** i = 1 to k [1] **do**
6: **Solve**(Ci, s_i, $cost_i$, Sol1)
7: **end for**
8: $Sol \leftarrow s_1 \odot s_2, \ldots \odot s_k$
9: $Cost_2 \leftarrow \sum_{i=1}^{k} cost_i$
10: $Cost \leftarrow cost1 + cost2$
11: **Improve** (Sol, Cost)

Le troisième algorithme nommé "Center-cluster" (Algorithm 3) est seulement utilisable dans le cas où le nombre de clusters k est de 3. La première étape de cet algorithme résout le cluster central (cluster $C2$), étape donnant lieu à une première solution partielle s_2 associée à un coût $cost_2$. Les étapes suivantes étendent cette solution à une solution globale en résolvant les clusters $C1$ et $C3$. La dernière étape qui induira un coût supplémentaire $cost_2$ consistera à vérifier la multicoupe. Le coût global de la solution est la somme des 4 coûts.

Algorithm 3 Algorithme Center-cluster

1: **Input** : $G =< V, E >$
2: **ouptut** : A global solution Sol , $Cost$ the global cost of Sol
3: **Decompose**(G, C1, C2, C3, Multi-Cut)
4: **Solve** (C2, s_2, $cost_2$)
5: **Solve** (C1, s_2, s_1, $cost_1$)
6: **Solve** (C3, s_2, s_3, $cost_3$)
7: $Sol \leftarrow s_1 \odot s_2, \ldots \odot s_3$
8: **Check** (Multi-Cut, Sol, $cost_4$)
9: $Cost \leftarrow cost_1 + cost_2 + cost_3 + cost_4$
10: **Improve** (Sol, Cost)

1. **Cette étape peut s'effectuer en séquentiel ou en parallèle**

5.3 Résultats expérimentaux

Les trois algorithmes présentés ici sont génériques dans le sens où la procédure **Solve** peut être implémentée en utilisant indifféremment une méthode exacte ou une métaheuristique, de même que la procédure **Improve** peut être implémentée de différentes façons (réparation locale par exemple, métaheuristique, ...). Nous présentons ici les premiers résulats obtenus en utilisant les décompositions " Graph Décomposition" et "Décomposition Niveaux " pour la variante Mincut. Nous avons développé un algorithme génétique spécifique [18] pour implémenter la procédure **Solve** et pour l'implémentation de la procédure **Improve** nous avons utilisé la méthode de recherche locale 1-opt. Rappelons que le premier objectif de ce papier n'est pas de tester l'efficacité de toutes les décompositions proposées mais de démontrer la faisabilité d'une telle approche qui servira éventuelement de phase d'apprentissage et de prétraitement.

La table 2 illustre la méthode " Graphe Décomposition" en utilisant l'algorithme 1 avec la variante Mincut. Des résultats comparables en termes de coût, pour $k = 2$ et $k = 3$ sont obtenus très rapidement. Ce premier résultat encourageant permet d'envisager la possibilité d'explorer dans nos perspectives ce gain de temps, au profit d'une solution de meilleure qualité. Nous constatons aussi que l'équilibre des clusters en termes de nombre de variables n'est pas suffisant. En effet, même si nous n'avons pas reporté les résultats d'exécution détaillés dans nos tables, les temps d'exécution de clusters de même taille peuvent être totalement différents. Nous avons pu constater que dans certains cas, toute la difficulté du problème initiale peut être ramenée à celle d'un seul sous problème cluster de très petite taille, partie la plus dure du problème. Il serait intéressant de développer un algorithme de résolution qui accorderait une attention particulière a ce cluster, en le résolvant en premier, par exemple par un algorithme dédié.

La table 3 présente les résultats de comparaison des approches "Graphe Décomposition" et "Décomposition Niveaux" en utilisant l'algorithme 1 et la variante Mincut. Seuls les résultats pour 3 clusters sont pris en compte, car dans le cas de deux clusters, les approches sont identiques. Pour 4 clusters, nous avons constaté une dégradation rapide des performances. La table 3 met légèrement en avant l'approche "Décomposition Niveaux" en terme de temps CPU. Ceci est dû au fait que la taille de la multicoupe est plus importante et par conséquent le temps de résolution des clusters est plus réduit. Cependant la qualité de la solution globale est moins bonne, ce qui nous amène à conjecturer que l'algorithme 1 n'est pas adapté à l'approche "Décom-

Pb	k	Mincut	Cost	t(s)	Bestcost
CELAR06	1	-	3411	78	3389
	2	3694	3409	29	
	3	6129	3521	17	
	4	8315	4942	11	
CELAR07	1	-	343714	410	343792
	2	1016187	373697	149	
	3	2616269	374102	92	
	4	3208718	383909	53	
GRAPH05	1	-	221	53	221
	2	10437	257	22	
	3	14222	261	10	
	4	18298	301	4	
GRAPH06	1	-	4157	270	4125
	2	25547	4224	145	
	3	35771	4791	58	
	4	43516	4760	41	
GRAPH11	1	-	3248	800	3088
	2	9216	4969	383	
	3	55194	5488	274	
	4	63041	6775	151	
GRAPH13	1	-	12581	1800	10110
	2	60260	12459	761	
	3	95366	18086	523	
	4	129649	17557	383	

TABLE 2 – "Graphe Décomposition", Algorithme 1

position Niveaux".

Pb	Graph / Level decompositions		
	Mincut	Cost	t(s)
CELAR06	6129 / 7929	3521 / 6131	17 / 13
CELAR07	2616269 / 3138002	374102 / 384304	92 / 87
GRAPH05	14222 / 33632	261 / 516	10 / 10
GRAPH06	35771 / 83439	4791 / 4982	58 / 53
GRAPH11	55194 / 174702	5488 / 7203	274 / 202
GRAPH13	95366 / 305780	18086 / 19908	525 / 400

TABLE 3 – "Graphe Décomposition " et "Décomposition Niveaux", Algorithm 1, $k = 3$

Dans la table 4 nous présentons les résultats obtenus en utilisant l'algorithme 2 et la variante Mincut pour $k = 2$ et $k = 3$. Nous constatons que les qualités des solutions obtenues avec 3 clusters sont souvent meilleures pour les instances les plus diificiles (GRAPH11 et GRAPH13), alors que les temps d'exécution sont naturellement plus importants. Ceci est dû au fait qu'avec 3 clusters la multicoupe recouvre un nombre plus important de noeuds et de ce faite, l'algorithme génétique résout la plus grande partie du problème. Pour cette même raison, nous observons aussi que la qualité de la solution augmente de façon proportionnelle au nombre de clusters.

6 Conclusion

La principale contribution de cet article est la proposition d'une méthode générale de résolution des PCSP baséee sur des techniques de décomposition multicoupe. Ces différentes techniques de décomposition sont définies comme des problèmes de partitionnement

Pb	k	t(s)	Cost	Best-Sol
CELAR06	2	14	4016	3389
	3	14	**3523**	
CELAR07	2	90	**354531**	343592
	3	142	464508	
GRAPH05	2	17	**234**	21
	3	89	348	
GRAPH06	2	131	4625	4125
	3	280	**4593**	
GRAPH11	2	523	10773	3088
	3	1374	**7767**	
GRAPH13	2	1117	27143	10110
	3	1923	**19847**	

TABLE 4 – "Décomposition Niveau", Algorithm 2

de graphe sous certaines conditions. Chacune de ces méthodes est naturellement associée à un algorithme de résolution approprié. Une autre contribution importante de cette étude est la formalisation de ces techniques de décomposition s'appuyant sur la programmation mathématique comme paradigme. Nous montrons que ce modèle formel peut être adapté à la modélisation de la résolution d'un PCSP lui même sans décomposition préalable. L'objectif de notre approche est assez différent de celui mené par Koster et al. En effet dans son étude, Koster a utilisé une décomposition arborescente, méthode qui s'est avérée inefficace pour résoudre des grandes instances de PCSP. C'est pourquoi dans notre approche nous favorisons pour le moment la résolution rapide des problèmes sans s'intéresser à l'optimum. Dans un second temps nous avons cherché à améliorer cette solution. Afin de valider notre méthode, nous avons mené une étude expérimentale sur des benchmarks réels (projet CALMA) en utilisant des variantes associées au problème Mincut. En ce qui concerne la résolution des sous problèmes, nous avons développé un algorithme génétique ad-hoc dont le résultat est amélioré par la méthode de recherche locale 1-opt. Les premiers résultats obtenus sont encourageants. En effet, la plupart des problèmes peuvent être rapidement décomposés et ce de façon quasi optimale, grâce à l'utilisation de nouvelles inégalités. Toutefois des améliorations de notre approche sont en cours pour des problèmes plus difficiles. Les résultats obtenus avec la méthode "Graphe-décomposition" utilisant l'algorithme "botton-up" sont appréciables en terme de temps d'exécution pour un nombre de clusters variant de 2 à 4. En comparant pour 3 clusters, la méthode "Graphe-decomposition" et 'Decomposition Niveau", toujours en utilisant l'algorithme "bottom-up", nous avons été forcés de constater que la méthode " Decomposition Niveau" est inappropriée. En effet, les solutions obtenues sont de qualité inférieure bien que les temps d'exécution soient plus rapides. Nous pouvons conclure que notre approche est intéressante et mérite d'être approfondie. La loca-

lisation de la zone la plus difficile d'un problème va nous permettre de l' isoler dans un cluster et d'envisager dans le futur un algorithme de résolution spécifique, par exemple. Ces premiers résultats sont encourageants et montrent l'intérêt des méthodes de décomposition.

Références

[1] K. Aardal , S Van Hoessel, A. Koster, C. Mannino, A. Sassano , Models and solution techniques for frequency assignment problems, 4OR, Quaterly Journal of the Belgian, French and Italian Operations Research Sciences, Vol.1 , pp. 261–317, 2003.

[2] M. Abril, M. A. Salido, F. Barber, Distributed search in railway scheduling problem, Engineering Applications of Artificial Intelligence, Vol. 21/5, pp. 744–755, 2008.

[3] D. Allouche, S. Givry, T. Schiex, Towards Parallel Non Serial Dynamic Programming for Solving Hard Weighted CSP,Proceeding of CP'2010, pp. 53–60, 2010.

[4] Calma-website, Euclid Calma project, ftp ://ftp.win.tue.nl/pub/techreports/CALMA/, 1995.

[5] G. Colombo, S. M. Allen, Problem decomposition for Minimum Interference Frequency Assignment, Proc. of the IEEE Congress in and Evolutionary Computation, Singapor, 2007.

[6] A. Eisenblatter, Frequency Assignment in GSM Networks : Models, Heuristics, and Lower Bounds, Technische Universitat Berlin, Berlin, Germany, April 2001.

[7] E. Freuder, R. Wallace, Partial Constraint Satisfaction, Artificial Intelligence Vol. 58, pp. 21–70, Dec. 1992.

[8] W. K Hale, Frequency assignment : Theory and applications, Proceedings of The IEEE-PIEEE, vol. 68, no. 12, pp. 1497–1514, 1980.

[9] P. Jégou, C. Terrioux, Hybrid backtracking bounded by tree-decomposition of constraint networks, Artificial Intelligence, Vol 146, pp. 43–75, 2003.

[10] A. Kolen, A genetic algorithm for the partial binary constraint satisfaction problem : an application to a frequency assignment problem, Report research RM/06/045, University of Maastricht, 2006.

[11] A. Koster, C. Van Hoessel, A. Kolen, The partial constraint satisfcation problems : Factes and lifting theorems. Operations Research Letters, Vol. 23(3-5), pp. 89–97, 1998.

[12] A. Koster, C. Van Hoesel, A. Kolen, Solving Frequency Assignment Problems Via Tree-Decomposition, RM 99/011, Maastricht University, 1999. http ://www.zib.de/koster/research.html

[13] A. Koster, Frequency Assignment - Models and Algorithms, *PhD.Thesis*, Maastricht University,1999, http ://www.zib.de/koster/thesis.html

[14] A. Koster, C. van Hoesel, A. Kolen, Solving partial constraint satisfaction problems with tree decomposition, Network Journal, vol.40/3, pp. 170–180, 2002.

[15] C. Mannino, A. Sassano, An enumerative algorithm for the frequency assignment problem, Discrete Applied Mathematic, Vol 129.1, pp. 155–169, 2003.

[16] B. H. Metzger, Spectrum management technique, 38th National ORSA Meeting, Detroit, pp. 1497–1514, 1970.

[17] S. N. Ndiaye, P. JÃ©gou, C. Terrioux, Extending to Soft and Preference Constraints a Framework for Solving Efficiently Structured Problems, Proceedings of the 20th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2008), pp. 299–306, 2008.

[18] L. Sadeg-Belkacem, Z. Habbas, H. Drias, Decomposition methods for solving Partial Constraint Satisfaction Problem, Case study : Minimum Interference Frequency Assignment Problem, Accepted to ICAART'2013.

[19] M. Stoer, F. Wagner, A simple Min-cut algorithm, Journal of the ACM, vol. 44/4, pp. 585–591, 1997.

[20] C. Voudouris, E. Tsang, Partial Constraint Satisfaction Problems and Guided Local Search, Department of Computer Science, University of Essex,Technical Report CSM-25,1995.

[21] Q. Yang , P. W. L. Fong, Solving partial constraint satisfaction problems using local search and abstraction, Technical report, 1992.

La programmation par contraintes pour l'assistance à l'agencement d'environnements virtuels

Marouene KEFI[1] **Vincent BARICHARD**[2] **Paul RICHARD**[1]

[1] Laboratoire d'Ingénierie des Systèmes Automatisés, 62 Avenue ND du Lac, Angers, France
[2] Laboratoire d'Etude et de Recherche en Informatique d'Angers, 2 Bd Lavoisier, Angers, France
{marouene.kefi,vincent.barichard,paul.richard}@univ-angers.fr

Résumé

Dans cet article, nous proposons d'utiliser la programmation par contraintes (PPC) pour assister l'utilisateur durant l'agencement d'environnements virtuels (EVs). Pour ce faire, noufs avons intégré un solveur de contraintes dans un outil de conception et de développement d'applications 3D temps réel. Le solveur est utilisé pour formaliser et développer différents types de contraintes définissant un problème d'agencement tridimensionnel, et pour trouver une solution à celui-ci. Un module de communication a été mis en place pour transmettre les solutions trouvées à l'EV qui se ré-agence automatiquement. La modélisation du problème d'agencement, ainsi que la recherche de solutions sont complètement transparentes vis à vis de l'utilisateur puisqu'il interagit seulement avec l'EV sans se soucier des mécanismes de résolution. En plus du placement automatique que le système propose, celui-ci peut être utilisé pour assister l'utilisateur à travers des retours visuels indiquant les zones 3D dans lesquelles un objet donné ne peut pas être placé. Une étude expérimentale a été réalisée pour étudier l'effet de l'assistance proposée, basée sur un mécanisme d'anticipation (*look ahead*), sur les performances des utilisateurs durant des tâches d'agencement 3D.

Abstract

We propose to use constraint programming (CP) to assist the user in 3D layout of restricted virtual environments (VEs). With this aim in mind, we integrated a constraint solver, into a professional authoring tool allowing the development of 3D real-time applications. The solver is used to formalize and develop different types of layout constraints and to find a solution to a user-specified layout problem. A communication module was developed to transmit the solution to the VE which automatically re-arranges itself. Modeling the problem and finding solutions are completely transparent for the user since he/she interacts with the VE regardless of the resolution mechanisms. In addition to the automatic objects placement, the proposed system can be used to assist the user during a manual layout by providing visual informations about the areas in which a given object cannot be placed. An experimental study has been carried out to investigate the effect of the provided *look ahead*-based assistance on user performance in 3D layout tasks.

1 Introduction

Résoudre un problème d'agencement défini par un ensemble de contraintes consiste à chercher un positionnement de composants (objets, matériels, meubles, etc.) à l'intérieur d'un contenant (bureaux, véhicules, etc.). Dans de tels problèmes, les contraintes présentent les relations fonctionnelles et géométriques liant d'une part les composants entre eux et d'autre part les composants et le contenant. Ce type de problème présente des applications dans plusieurs domaines industriels comme par exemple l'aménagement d'intérieurs, de bâtiments, de véhicules, etc. Souvent résolus manuellement à partir du savoir faire et de l'expérience des concepteurs, le développement de méthodes de résolution représente un réel défi à l'heure où les systèmes deviennent de plus en plus complexes. Ceci s'explique par la difficulté à modéliser ces problèmes et d'identifier une stratégie de résolution efficace appropriée.

La réalité virtuelle (RV) est un domaine rassemblant l'ensemble des techniques qui procurent à l'utilisateur un sentiment de présence dans des Environnements Virtuels (EVs), avec lesquels il pourra communiquer et interagir via des techniques et protocoles basés sur ses facultés naturelles [4]. Ces techniques permettent à l'utilisateur d'agir dans les EVs par des moyens d'action intuitifs (mouvements du corps, gestes, voix, etc.)[1]. La RV a fait ses preuves dans plusieurs domaines mais son utilisation

comme un simple outil de visualisation et de validation du prototype final ne répond pas efficacement à la problématique de l'agencement 3D vu l'absence d'un module intelligent capable de modéliser et satisfaire les contraintes.

Dans cet article, nous proposons d'utiliser la Programmation Par Contraintes (PPC) pour résoudre un problème d'agencement 3D et assister l'utilisateur. Nous visons spécifiquement le développement d'un système d'aide à la décision proposant plusieurs solutions (quand cela est possible) à un problème d'agencement préalablement défini par l'utilisateur. Dans la section suivante, nous présentons quelques travaux relatifs à l'état de l'art. Dans la section 3, nous proposons une formulation d'un problème d'agencement 3D spécifique. La section 4 est consacrée à la description du système développé, les contraintes supportées et les mécanismes d'assistance fournies durant une tâche d'agencement 3D. Dans la section 5, nous discutons les résultats d'une étude expérimentale menée pour évaluer le système proposé. Enfin, dans la section 6, nous concluons en donnant quelques pistes pour les travaux futurs.

2 Travaux antérieurs

L'agencement 3D a fait l'objet de plusieurs travaux dans la littérature. Par exemple, Fernando et al, ont travaillé sur la conception d'un environnement virtuel (EV) supportant l'intégration de techniques de programmation par contraintes [10]. Le travail réalisé porte sur la conception d'EVs pouvant supporter des contraintes sans pour autant traiter un problème spécifique.

Des travaux plus ciblés ont traité le problème d'agencement dans les EVs. En effet, Xu et .al, ont combiné des contraintes physiques et géométriques dans le but de faciliter la disposition ou l'aménagement d'une scène 3D [20]. Ils ont montré que l'agencement de la scène pourrait être accéléré avec un moteur pseudo-physique et quelques informations sémantiques. Dans le même contexte, Sanchez et al., ont présenté un nouveau concept pour concevoir un système dédié à la disposition des scènes 3D. Ce travail est basé sur l'utilisation d'un algorithme génétique [17]. Le système peut traiter un ensemble complexe de contraintes, y compris des contraintes géométriques et pseudo-physiques. Calderon et al., ont élaboré un travail original pour l'usage des EVs dans la résolution des problèmes d'agencement interactifs [5]. Ce travail étend les EVs afin qu'ils servent d'interface réactive permettant l'exploration de l'espace de solutions. Le système proposé est basé sur une communication entièrement interactive où la visualisation et la génération d'une nouvelle solution sont sous le contrôle de l'utilisateur.

Fages et al., ont implémenté une interface graphique générique (CLPGUI) pour visualiser et commander des programmes contraints [9]. L'architecture proposée se base sur une communication entre deux parties : un processus de résolution et une interface graphique. Le CLPGUI a été testé sur un problème d'agencement 3D impliquant quelques objets à placer dans un espace restreint. Plus récemment, Tim et al. [19], ont introduit une nouvelle approche d'agencement afin d'accélérer les méthodes de conception des pièces d'un jeu 3D. Leur approche de résolution peut être utilisée pour la génération procédurale, en fournissant au solveur utilisé un plan défini par l'utilisateur.

Excepté les approches proposées par Calderon et al. [5] et Fages et al. [9] qui permettent à l'utilisateur un certain degré d'interactivité avec l'EV, la plupart des travaux antérieurs ([17, 20, 10]) n'ont pas privilégié l'interactivité et l'assistance que peut fournir un solveur de contraintes.

Notre approche, basée sur l'intégration d'un solveur de contraintes Gecode (dans lequel nous disposons de connaissances approfondies) dans un EV, étend les travaux antérieurs dans différentes directions. Par exemple, en termes de mécanismes d'interaction, nous offrons une plus grande intéractivité pour rendre la manipulation d'objets plus efficace et plus intuitive [3, 14, 18]. En effet, au cours d'une tâche d'agencement 3D, l'utilisateur est en mesure de : (1) changer différentes vues de la scène 3D, (2) ajouter des objets, préciser leurs dimensions et sélectionner les contraintes à appliquer à tout moment, et (3) exprimer ses préférences en plaçant manuellement des objets, le placement des autres objets est laissé à la charge du solveur. En plus de l'intéractivité, le système peut proposer une assistance à l'utilisateur en : (1) offrant plus d'une solution (quand cela est possible) et lui permettre de naviguer dans l'espace des solutions pour sélectionner celle qu'il préfère, (2) fournissant un retour visuel illustrant les "mauvaises" zones de placement pour aider l'utilisateur à choisir correctement les positions d'objets (voir la sous-section 4.1), et (3) annulant toute manipulation d'objets 3D conduisant à une violation de contraintes.

3 Formulation d'un problème d'agencement 3D

Nous traitons dans cet article un problème d'agencement d'intérieur défini par les éléments suivants :

- *Un conteneur* : un EV simulant l'intérieur à agencer.
- *Des composants* : un ensemble de modèles 3D représentant des meubles (chaise, lampe, canapé, etc.) à placer dans le conteneur.
- *Des contraintes* : des relations reliant les meubles entre eux et avec le conteneur.

La formulation de ce genre de problèmes nécessite un formalisme efficace et bien structuré, permettant de représenter tous les paramètres (inconnues et contraintes) du problème. Nous avons choisi d'utiliser la PPC, à travers le formalisme CSP. Notre choix est motivé par les deux raisons suivantes :

- Comme montré par Mizoguchi [13] et Pfefferkorn [16], la PPC est particulièrement appropriée pour résoudre des problèmes liés à l'agencement spatial,
- Bien que la plupart des contraintes géométriques de placement (non chevauchement, objet-sur-objet, etc) peuvent être satisfaites manuellement sans faire appel à des systèmes de résolution, la satisfaction des contraintes physiques (voir Section 4.3) est très difficile, voir impossible sans l'aide d'un solveur.

Le formalisme CSP (Constraint Satisfaction Problem) constitue un cadre simple et formel, offrant des mécanismes de représentation et de résolution des problèmes de satisfaction de contraintes [11]. Dans notre contexte, les inconnues du problème sont les positions des objets 3D, les contraintes d'agencement sont les relations ou les restrictions reliant ces objets entre eux et avec le conteneur.

Pour donner une formulation de notre problème sous forme de CSP nous supposons que la scène 3D de dimensions (**w**,**h**,**d**) est composée de n objets de formes rectangulaires (hypothèse industrielle) liés par m contraintes, X est l'ensemble des inconnues du problème (positions 3D des objets), $\mathscr{D}$ est une fonction qui associe à chaque variable un domaine (valeurs autorisées) et C l'ensemble des contraintes d'agencement. Ainsi, le problème peut être défini par le triplet ($\boldsymbol{X}$,$\boldsymbol{D}$,$\boldsymbol{C}$) :

- $\boldsymbol{X} = \{x_1, y_1, z_1, ..., x_n, y_n, z_n \in \mathbf{R}\} / (x_i, y_i, z_i) / i \in [1,n])$ est la position du centre de l'$objet_i$
- $\mathscr{D}(x_i) = [w_i/2, w - w_i/2]$, w_i /i$\in$ $[1,n]$ est la largeur de l'$objet_i$
  $\mathscr{D}(y_i) = [h_i/2, h - h_i/2]$, h_i /i$\in$ $[1,n]$ est la hauteur de l'$objet_i$
  $\mathscr{D}(z_i) = [d_i/2, d - d_i/2]$, d_i /i$\in$ $[1,n]$ est la profondeur de l'$objet_i$
- $\boldsymbol{C} = \{c_1^{i,j}, c_2^{i,j}, ..., c_m^{i,j}\}$, $c_k^{i,j} / (k \in [1,m]$ et $i, j \in [1,n])$ est la contrainte numéro k entre l'$objet_i$ et l'$objet_j$.

4 Description du système

Le système proposé est un environnement 3D temps-réel développé avec Unity3D (version 3.2.0f4), couramment utilisé pour la création d'EVs et de jeux 3D. Unity3D propose un éditeur universel de scripts permettant de coder des routines en C#, JavaScript et Boo [8].

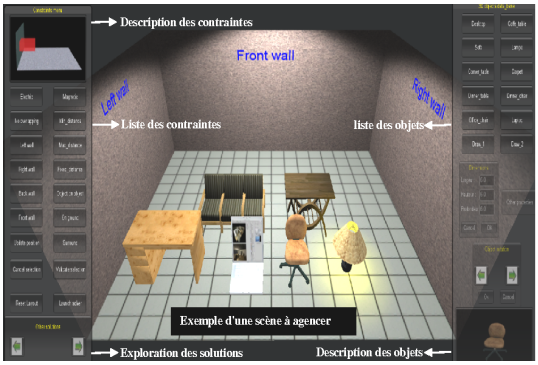


FIGURE 1 – *L'interface graphique du système.*

A travers l'interface développée sous Unity 3D (Fig. 1), l'utilisateur utilise un menu pour sélectionner les objets 3D (meubles, matériels, etc) qu'il veut agencer. Un autre menu lui permet de poster des contraintes géométriques et/ou physiques à appliquer sur les objets sélectionnés. Afin d'aider l'utilisateur à comprendre l'intérêt de chaque contrainte, une description schématique simplifiée est affichée sur le haut du menu de contraintes (Fig. 1). Chaque sélection effectuée par l'utilisateur est associée à un retour visuel pour confirmer la validation de l'action. A tout moment, l'utilisateur peut déclencher la résolution, ce qui conduit à la création d'un CSP dans lequel les variables sont les positions 3D des centres des objets. La résolution de ce CSP est laissée à la charge du solveur qui cherche des configurations ou placements possibles des objets. Les données relatives aux solutions trouvées sont transmises à l'EV qui se met automatiquement à jour selon les nouvelles positions calculées par le solveur. L'utilisateur est également en mesure de naviguer dans l'espace des solutions afin de choisir celle qui lui convient le plus. En effet, nous nous sommes limités dans un premier temps au calcul des dix premières solutions. Celles-ci sont stockées dans un tableau et sont accessibles à l'utilisateur. Deux cas d'interaction utilisateur-système sont possibles :

- L'utilisateur peut augmenter la taille du problème en ajoutant de nouveaux objets et/ou contraintes. Dans ce cas, le système définit un nouveau CSP en ajoutant les nouvelles variables (positions d'objets) ou contraintes. Le solveur se chargera de résoudre le nouveau CSP et de transmettre les solutions.
- L'utilisateur peut se contenter de la configuration courante et modifier, selon ses préférences, l'emplacement de certains objets. Contrairement au premier cas, le solveur ajoute alors des contraintes aux objets déplacés et met à jour le CSP courant.

Afin de conserver les CSPs déjà créés et une partie du travail effectué par le solveur, une pile (*Stack_spaces*)

initialement vide est utilisée pour stocker chaque CSP créé et valide (tous les domaines des variables sont non vides). Le caractère évolutif de notre système (ajout dynamique d'objets et/ou contraintes) justifie l'utilisation d'une suite de CSPs liés. En effet, sont stockés dans la pile *Stack_spaces* une séquence de CSPs telle que chaque CSP_i est une restriction de CSP_{i-1}. Même si nous avons opter pour le mécanisme *CSP_backtrack*, le paradigme des CSP dynamiques [7] semble approprié pour résoudre à cette problématique. Cette piste sera privilégiée dans nos futurs travaux.

Avant de commencer la description des algorithmes, il est important de comprendre pourquoi nous avons utilisé le mécanisme de *CSP_backtrack*. L'exemple simple donné à la Figure 2 montre qu'une partie du travail du solveur a été conservée et utilisée pour résoudre le problème et réduire le temps de calcul. Nous supposons que la forme CSP du problème est donnée par :

- $X = \{x_1, x_2\}$, $\mathscr{D}(x_1) = \mathscr{D}(x_2) = [0,5]$ and $C = \{c_1, c_2\}$ $(c_1 = \{x_1 > 2\}, c_2 = \{x_1 = x_2\})$

Comme décrit précédemment, l'utilisateur peut appliquer des contraintes sur un objet obj_i et le positionner manuellement s'il le souhaite. Dans les deux cas, le solveur met à jour le CSP courant en modifiant les domaines des variables x_i, y_i et z_i (Algorithme 1). Par propagation de contraintes, le solveur vérifie la validité du CSP modifié. Dans le cas où le CSP est non valide (*failed*), il explore la pile des CSPs *"Stack_spaces"* (en utilisant le mécanisme *CSP_backtrack*) pour en trouver un qui "accepte" la nouvelle contrainte. Sinon (CSP *not failed*), il retourne la solution trouvée et insère le CSP dans la pile. La Figure 2 illustre un exemple dans lequel l'ajout de la contrainte C2 rend invalide le CSP courant, ce qui amène à l'exploration de la pile *"Stack_spaces"* afin de revenir en arrière et tenter de résoudre le problème.

Contrairement à l'ajout de contraintes, le solveur crée un nouveau CSP lorsque l'utilisateur insère un nouveau objet dans la scène 3D (Algorithme 2). Les dimensions de l'objet inséré, spécifiées par l'utilisateur, serviront à créer de nouvelles variables dans le nouveau CSP. Par conséquent, le système définit un nouveau problème à résoudre.

4.1 Assistance à l'agencement 3D

Malgré l'intérêt des approches antérieures proposées pour la résolution des problèmes d'agencement, l'assistance de l'utilisateur lors de son interaction avec la scène 3D à agencer n'a pas été privilégiée ([2, 9, 17, 20, 10]). En effet, dans la plupart de ces approches, l'assistance proposée survient après le placement effectif des objets. Comme décrit précédemment, notre système permet la résolution d'un problème d'agencement en proposant des

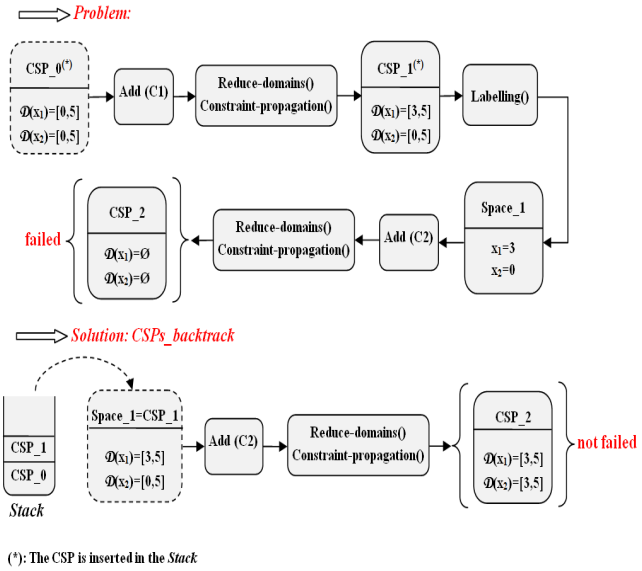


FIGURE 2 – *Exemple du mécanisme du CSP_backtrack.*

configurations avec lesquelles l'utilisateur peut interagir. Le système proposé est capable d'annuler ou rejeter un placement manuel d'objets s'il conduit à une violation de contraintes. Dans l'objectif de concevoir et présenter un système plus efficace et générique, nous avons décidé de fournir une assistance "prédictive", visuelle et temps-réel avant même de placer un objet donné. A travers ce type d'assistance, l'utilisateur est en mesure d'identifier les zones 3D (appelées *zones impossibles*) où le placement de l'objet provoque une violation de contraintes. Le solveur identifie ces zones et transmet les informations nécessaires à l'EV afin de les visualiser sous forme de parallélépipèdes 3D marqués par une couleur spécifique.

Cette approche est basée sur le mécanisme d'anticipation (***look ahead***) [6]. Le terme ***look ahead*** est un terme générique de toute méthode consistant à anticiper l'effet de l'affectation d'une variable (affectation partielle) sur les domaines des variables qui ne sont pas encore affectées. Après propagation, si le domaine d'une variable non affectée devient vide et donc il ne contient plus de valeur pouvant être localement consistante avec l'affectation partielle, alors il est inutile de continuer à explorer cette branche, et l'on peut retourner en arrière pour explorer d'autres possibilités. Un tel mécanisme permet donc d'anticiper les échecs dans un arbre de recherche et de prévoir une possible inconsistance.

Pour rechercher les zones impossibles pour un objet obj_i, une décomposition de l'espace de travail (la scène à agencer) en plusieurs zones 3D doit être effectuée. Chaque zone obtenue est définie par six paramètres :

Algorithm 1 Ajout d'une nouvelle contrainte pour l'objet_i

```
1: Ajout_nouvelle_contrainte(i)
2: CurrentSpace.Contraction_des_domaines(x_i, y_i, z_i)
3: CurrentSpace.Propagation_par_contraintes()

4: Solution ← true
5: X'← CurrentSpace.X
6: C'← CurrentSpace.C
7: S'← Stack_Spaces
8: for all var ∈ X' do
9:    if D(var)= ∅ then
10:      if S'≠ ∅ then
11:         CurrentSpace ← S'.pop()
12:         CurrentSpace.C←C'
13:         CurrentSpace.Contraction_des_domaines()
14:         CurrentSpace.Propagation_par_contraintes()
15:         X'← CurrentSpace.X
16:      else
17:         X'← ∅
18:         Solution ← false
19:      end if
20:   else
21:      X'← X'/{var}
22:   end if
23: end for

24: if Solution = true then
25:    Envoi_Event(Solution_Found)
26:    Stack_Spaces.push_back(CurrentSpace)
27: else
28:    CurrentSpace ← Stack_Spaces.pop()
29:    Envoi_Event(No_Solution_Found)
30: end if
```

Algorithm 2 Ajout d'un nouveau objet à la scène 3D

```
1: Ajout_nouveau_objet(dim_x,dim_y,dim_z)
2: Créer_variables(x_{n+1},y_{n+1},z_{n+1})

3: CSP1 ← new_CSP(CurrentCSP)
4: CSP1.X=CSP1.X⋃{x_{n+1},y_{n+1},z_{n+1}}
5: %(w,h,d) are the dimensions of the container%
6: CSP1.D(x_{n+1}) ← [dim_x,w-dim_x]
7: CSP1.D(y_{n+1}) ← [dim_y,h-dim_y]
8: CSP1.D(z_{n+1}) ← [dim_z,d-dim_z]
9: X'← CurrentSpace.X
10: S'← Stack_Spaces
11: CurrentSpace.Contraction_des_domaines()
12: CurrentSpace.Propagation_par_contraintes()

13: if Est_Failed(CurrentSpace) then
14:    CurrentSpace ← Stack_Spaces.pop()
15:    Envoi_Event(No_Solution_Found)
16: else
17:    Envoi_Event(Solution_Found)
18:    Stack_Spaces.push(CurrentSpace)
19: end if
```

- $[Z_{xmin}, Z_{xmax}] \Longrightarrow$ *délimitation de la zone sur l'axe X*,
- $[Z_{ymin}, Z_{ymax}] \Longrightarrow$ *délimitation de la zone sur l'axe Y*,
- $[Z_{zmin}, Z_{zmax}] \Longrightarrow$ *délimitation de la zone sur l'axe Z.*

Pour chacune des zones créées, le solveur crée un nouveau CSP (à partir du CSP courant) dans lequel les variables relatives à obj_i (x_i, y_i et z_i) ont chacune un nouveau domaine :

- $\mathscr{D}(x_i)$=[Z_{xmin},Z_{xmax}],
- $\mathscr{D}(y_i)$=[Z_{ymin},Z_{ymax}],
- $\mathscr{D}(z_i)$=[Z_{zmin},Z_{zmax}].

Ensuite, un ***look ahead*** agissant sur trois variables non instanciées (x_i, y_i et z_i) ainsi que sur celles qui leurs sont liées, est utilisé pour "forcer" la consistance locale en supprimant les valeurs dont le choix conduit à une inconsistance. Le solveur vérifie alors si le CSP créé est localement consistant ou non (*zone impossible*). A travers ce type d'assistance, l'utilisateur est guidé par une information visuelle afin de mieux placer (manuellement) un objet donné. L'approche proposée est illustrée par l'Algorithme 3.

Il est à noter que le temps requis par le solveur pour la recherche des zones impossibles dépend fortement de la décomposition effectuée sur la scène 3D. En effet, plus le nombre de zones à tester est grand, plus le temps de calcul est long. A titre d'exemple, le solveur Gecode nécessite moins d'une seconde pour tester 1000 zones ($10 \times 10 \times 10$). Le même principe est valable pour la précision ; une décomposition précise de la scène 3D (un grand nombre de zones) assure une meilleure identification des zones impossibles. L'identification des zones impossibles pour un objet obj_i ne signifie pas que toutes les autres zones sont des placements valides pour obj_i ; ceci dépend principalement de la précision de la décomposition.

Pour accélérer l'identification des zones impossibles, une recherche dichotomique est utilisée pour ne pas tester certaines zones et donc accélérer les temps de calculs. Par exemple si un objet doit être placé sur le sol, le solveur élimine toutes les zones dont le paramètre Z_{ymin} dépasse la hauteur de l'objet en question.

La Figure 3 montre un exemple simple dans lequel la chaise du bureau doit être placée sur le sol et à droite du bureau. Le système propose les zones rouges comme celles impossibles pour le placement de la chaise.

4.2 Module d'interaction avec le solveur

En général, le processus de résolution d'un CSP est déclenché par la modification des valeurs des variables ou des contraintes. Dans notre cas, l'interaction de l'utilisateur avec les objets virtuels se traduit par des commandes (ou requêtes) envoyées au solveur, celui-ci agit alors sur les variables et modifie leurs valeurs (Fig. 4). Par exemple, l'utilisateur interagit avec une configuration proposée par le solveur en déplaçant manuellement certains objets. Cette

Algorithm 3 Zones impossibles pour l'objet obj_i

1: **Affichage_zones_impossible**(obj_i)
2: **Décompostion_Scene_3D**()*%décomposer la scène en k zones%*
3: $Zi_{xmin}, Zi_{xmax}, Zi_{ymin}, Zi_{ymax}, Zi_{zmin}, Zi_{zmax}$ / i in $\in [1..n]$ *sont les paramètres de la zone* Z_i
4: **for** all $j \in [1..n]$ **do**
5: CSP_temp $\leftarrow$ CurrentCSP
6: CSP_temp.$\mathscr{D}(x_i) \leftarrow [Zj_{xmin}, Zj_{xmax}]$
7: CSP_temp.$\mathscr{D}(y_i) \leftarrow [Zj_{ymin}, Zj_{ymax}]$
8: CSP_temp.$\mathscr{D}(z_i) \leftarrow [Zj_{zmin}, Zj_{zmax}]$
9: CSP_temp.**Contraction_domaines**(x_i, y_i, z_i)
10: CSP_temp.**Propagation_de_contraintes**()
11: **if Est_Failed**(CSP_temp) **then**
12: L'objet obj_i ne peut être placé quelque part dans la zone Zj
13: **Affichage_zone_impossible_dans_EV**(Zj)
14: **else**
15: L'objet obj_i peut être placé quelque part dans la zone Zj
16: **end if**
17: **end for**

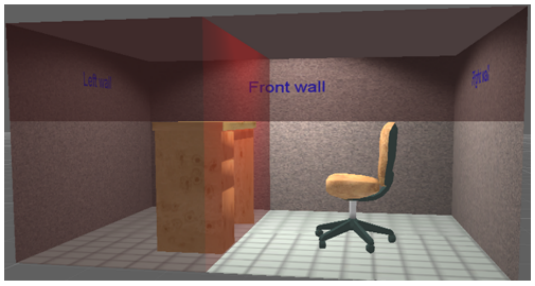


FIGURE 3 – *Visualisation des zones impossibles pour la chaise du bureau.*

interaction modifie les domaines des variables et déclenche le processus de propagation des contraintes (*étape 1*-traits pleins). Le solveur est donc "réveillé" pour (i) mettre à jour le CSP, (ii) vérifier la validité de celui-ci et finalement (iii) accepter ou décliner la solution proposée par l'utilisateur (*étape 2*-traits pointillés).

4.3 Implémentation des contraintes

Pour illustrer l'approche proposée, nous avons choisi de développer des contraintes spécifiques liées à un problème d'agencement proposé par un partenaire industriel. Les *contraintes géométriques* reflètent les exigences naturelles qui peuvent être définies pour un agencement concret. Quant aux *contraintes physiques*, elles ont été développées pour tenir compte de l'effet que peuvent avoir certains objets sur d'autres. La contrainte globale Geost [15] portant sur le placement géométrique des objets à n-dimensions aurait pu être utilisée pour traiter de manière globale le problème d'agencement 3D. Cependant, nous avons privilégié un premier modèle attestant de la faisabilité en utilisant des contraintes non globales. L'utilisation de contraintes globales sera étudiée pour des travaux futurs. Les contraintes implémentées via le solveur sont classifiées et décrites ci-après :

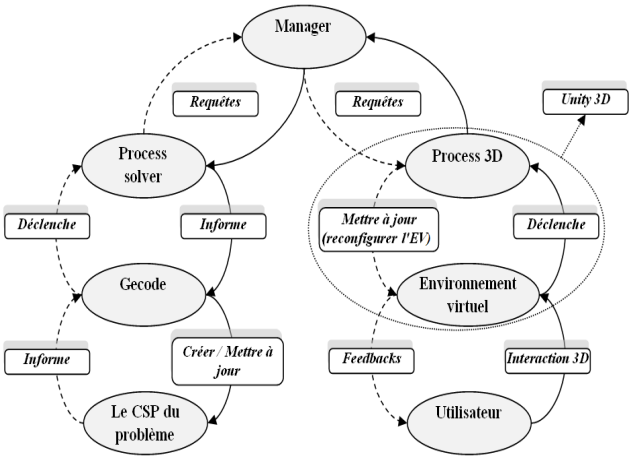


FIGURE 4 – *Module d'interaction avec le solveur.*

Contraintes géométriques

- *Contrainte "Distance_minimale"* Cette contrainte peut être appliquée sur au moins deux objets présélectionnés. Le solveur fera en sorte que les objets sélectionnés soient séparés par une distance minimale (*dmin*) qui peut être fixée par l'utilisateur. Le même principe est utilisé pour les contraintes *"Distance_maximale"* et *"Distance_fixe"*.
- *Contrainte "Non_chevauchement"* Considérée comme un cas particulier de la contrainte *"Distance_minimale"*, cette contrainte assure la non superposition de deux objets. Dans tous les cas, cette contrainte est postée par défaut.
- *Contrainte "A_droite"* La contrainte "A_droite" place un premier objet sélectionné à droite d'un deuxième objet. Les deux objets ne seront pas forcément collés. Cette contrainte peut être utilisée pour placer un objet contre le mur droit puisque ce dernier est considéré comme un objet. Le même principe est utilisé pour les contraintes *"A_gauche"*, *"Devant"* et *"Derrière"*.
- *Contrainte "Objet_sur_objet"* Elle peut être appliquée sur deux objets ; celui sélectionné en deuxième sera placé sur le premier. Placer un objet sur le sol constitue un cas particulier de cette contrainte.

Les contraintes physiques Comme leur nom l'indique, ces contraintes font appel aux lois de la physique et prennent en entrée les valeurs des paramètres définissant chaque loi. Dans certains cas la durée de vie d'un objet ou matériel peut être diminuée en présence de fortes valeurs des champs électriques ou magnétiques par exemple. Ces contraintes ont été développées afin de "minimiser" l'effet de ces champs sur les objets qui leurs sont sensibles. Pour ce faire, et lors de l'ajout d'un objet dans la scène à agencer, l'utilisateur doit spécifier si ce dernier est susceptible d'émettre un champ particulier en saisissant les valeurs définissant le champ en question. Par exemple pour le champ magnétique, il faut saisir la valeur de la charge électrique $\boldsymbol{q}$ ainsi que sa vitesse $\boldsymbol{v}$.

- *Contrainte "Électrique"* :
  L'utilisateur doit sélectionner l'objet qu'il veut protéger. Il doit ensuite saisir la valeur maximale du champ électrique que l'objet peut supporter. Étant donné que le solveur est déjà informé de l'existence d'objets émetteurs de ce type de champ et en se basant sur la formule permettant de calculer le module du champ électrique en fonction de la distance par rapport à une charge $\boldsymbol{q}$ (Equation 1), il place l'objet en question à une distance minimale afin de le protéger.

$$\| E \| = \frac{q}{4.\Pi.\varepsilon_0.d_E^2} \tag{1}$$

- *Contrainte "Magnétique"* :
  Dans ce cas les calculs du solveur sont basés sur la formule générale permettant de calculer le module du champ magnétique en fonction de la distance par rapport à une charge $\boldsymbol{q}$ (Equation 2). Comme pour la contrainte *Électrique*, le solveur positionne l'objet suffisamment loin afin de le protéger.

$$\| B \| = \frac{10^{-7}.q.v}{d_M^2} \tag{2}$$

5 Etude expérimentale

Une étude expérimentale a été mise en place afin d'évaluer l'effet de l'assistance fournie sur des facteurs de performances tels que le temps d'accomplissement d'une tâche d'agencement, les erreurs d'agencement et le comportement de l'utilisateur durant la tâche. Une comparaison de deux approches différentes (avec/sans assistance) a été réalisée. Contrairement à la tâche assistée, l'affichage des zones impossibles est désactivé dans la tâche assistée. Afin d'obtenir des informations spécifiques sur les difficultés ressenties par les sujets durant les tâches, une mesure subjective de la charge de travail a été réalisée à travers un questionnaire basé sur la NASA-TLX (Task Load Index, développée à la NASA) [12]. D'autres données subjectives ont été également visées par ce questionnaire, portant sur le ressenti des sujets par rapport à l'assistance fournie et leur satisfaction par rapport l'agencement réalisé.

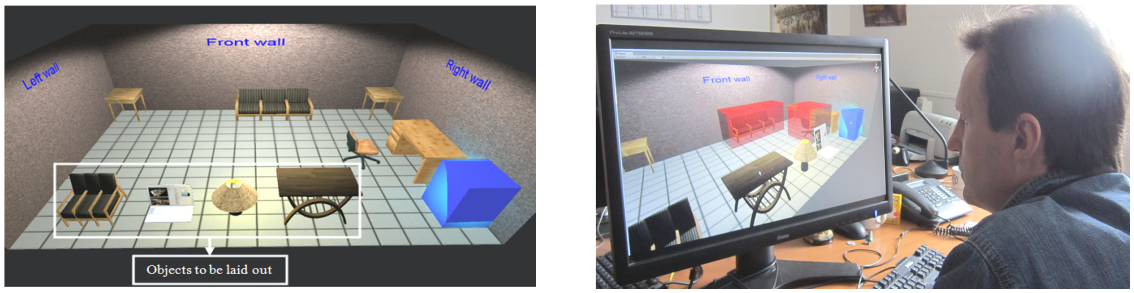
5.1 Protocole expérimental

La tâche consistait à placer manuellement quatre objets 3D (un canapé, un ordinateur portable, une lampe et une table de salon) en tenant compte des objets déjà placés (un bureau, un canapé, 2 tables de coin, une chaise de bureau et un objet générique de couleur bleu émetteur du champ électrique), tout en respectant un ensemble de contraintes.

Douze sujets volontaires, âgés de 22 à 34 ans, ont participé à l'expérimentation. Ils avaient le même niveau d'expertise dans les jeux vidéo et les applications de RV. Une souris a été utilisée comme périphérique d'entrée pour sélectionner, désélectionner et déplacer les objets dans l'espace 3D affiché sur un écran de moyenne dimension (Fig 5). Les sujets pouvaient changer leur point de vue dans la scène 3D en utilisant les touches de direction du clavier. L'objectif principal de l'étude a été expliqué aux sujets. Une brève description de deux conditions expérimentales (sans/avec assistance) a été également présentée. Avant le début de la tâche, les sujets étaient informés de l'ensemble des contraintes exigées pour l'agencement, de la valeur de la charge électrique $\boldsymbol{q}$ de l'objet bleu et de la valeur maximale du champ électrique supportée par l'ordinateur portable. En effet, et afin d'illustrer les contraintes physiques, nous avons supposé que l'objet bleu émettait un champ électrique et que l'ordinateur portable était sensible à un tel champ et ne pouvait supporter qu'un seuil bien déterminé.

Une session d'essai a été proposée à chaque sujet afin de se familiariser avec l'interface du système, le contrôle de la caméra virtuelle, la manipulation des objets et la sélection des contraintes. L'ordre de passage des conditions a été contrebalancé afin d'éviter un biais dû au transfert d'apprentissage.

Les quatre objets à agencer et ceux déjà placés ont été disposés comme le montre la Figure 5. Au cours de la tâche, les sujets devaient déplacer un objet donné en utilisant le bouton gauche de la souris. Chaque sujet devait : (1) placer l'ordinateur portable sur le bureau, (2) protéger l'ordinateur portable du champ électrique émis par l'objet bleu, (3) mettre la lampe sur la table$_1$ du coin gauche, (4) placer le canapé$_2$ contre le mur avant, (5) séparer le canapé$_2$ et le canapé$_1$ par une distance minimale ***d_min***, (6) séparer le canapé$_2$ et la table$_1$ du coin par une distance minimale ***d_min***, (7) séparer la table de salon et le canapé$_1$ par une distance fixe ***dist***, (8) séparer la table



(a) (b)

FIGURE 5 – *Copie d'écran de la condition initiale de la tâche (a) et un sujet agence la scène 3D à l'aide la visualization des zones impossibles (b).*

de salon et la chaise du bureau par une distance minimale ***d_min***. Les graduations sur le sol de la scène 3D (Fig. 5) ont permis aux sujets d'estimer les distances entre les objets.

Agencement non assisté
En utilisant la souris et en changeant la position de la caméra virtuelle, les sujets devaient placer manuellement les quatre objets tout en respectant les contraintes. Aucune information ou indication n'a été donnée pour les aider durant l'agencement.

Agencement assisté
Via le menu des contraintes, les sujets devaient sélectionner les contraintes définies pour les quatre objets à agencer. Cette opération est primordiale afin d'"informer" le solveur quant aux contraintes requises pour l'agencement. En utilisant le même menu, les sujets pouvaient saisir les valeurs correspondant à la charge électrique $\boldsymbol{q}$ de l'objet bleu et le champ électrique maximal supporté par l'ordinateur portable. Ces valeurs sont transmises au solveur afin d'identifier les zones dans lesquelles la valeur du champ électrique dépasse la valeur maximale supportée par l'ordinateur portable (voir Section 4.3). Avant de placer un objet donné, l'utilisateur peut activer/désactiver la visualisation des zones impossibles.

5.2 Données recueillies

Données objectives

1. *Temps d'agencement*
2. *Erreurs d'agencement* En se basant sur les positions finales de chaque objet, une vérification est faite pour la satisfaction de chaque contrainte. En effet, une erreur d'agencement survient quand une contrainte est violée. Nous avons compté le nombre total des erreurs dans chaque condition expérimentale.
3. *Nombre d'actions de l'utilisateur* Pour obtenir des informations sur le taux d'interaction des sujets, nous avons décidé de compter chaque sélection/manipulation d'objet effectuée par les sujets.

Données subjectives

1. *Charge de travail* Après la fin de chaque condition expérimentale, les sujets devaient répondre à un questionnaire basé sur la NASA-TLX afin d'évaluer le niveau de la charge de travail causé par la tâche.
2. *Satisfaction des sujets et assistance fournie*
   – *Satisfaction* En utilisant une échelle de Likert à sept points (1 pour "très faible" à 7 pour "très élevé"), les sujets devaient évaluer leur satisfaction concernant l'agencement réalisé. Leur satisfaction dépendait fortement de leur ressenti à propos du respect des contraintes.
   – *Le besoin d'assistance* En utilisant la même échelle, les sujets devaient exprimer s'ils avaient besoin d'une assistance durant les tâches. Chaque sujet a évalué le niveau de ce besoin.
   – *Effets de l'assistance* Deux questions supplémentaires ont été posées après la fin des tâches assistées. Les sujets devaient exprimer leur ressenti par rapport à l'assistance fournie par le solveur. Le but de la première question était de savoir jusqu'à quel niveau la visualisation des zones a facilité ou non l'agencement (*facilité de l'agencement*). Les réponses de la deuxième question ont permis de savoir si la visualisation des zones a aidé les sujets à se souvenir des contraintes au cours de la tâche (*souvenir des contraintes*).

6 Résultats et discussion

6.1 Résultats objectifs

Une analyse de variance (ANOVA) intra-groupes (*within subjects*) à un facteur de variabilité a été réalisée

pour évaluer l'effet de la variable indépendante (assistance) sur le temps d'agencement, les erreurs d'agencement et le nombre des actions des sujets durant les tâches. Le Tableau 1 illustre l'effet de ***A*** (assistance) sur chacun des facteurs mesurés. Comme indiqué dans le tableau, ***A*** affecte d'une manière significative les erreurs d'agencement et le nombre des actions des sujets durant les tâches. Quant au temps d'agencement, les résultats ont montré qu'il n'y pas de différence significative entre les temps d'accomplissement des tâches non assistées et celles assistées.

Facteur	Temps	Erreurs	Actions
A	$p = 0.3259$ $F_{1,11} = 1.31$	$p < 0.05$ $F_{1,11} = 50$	$p < 0.05$ $F_{1,11} = 77.12$

TABLE 1 – *Effet de l'assistance (A) sur les données objectives.*

1. *Erreur d'agencement* L'effet significatif de **A** sur les erreurs d'agencement est expliqué par le fait que durant les tâches non assistées, aucune information ou indication n'est fournie aux sujets concernant les zones non susceptibles de contenir un objet donné. De ce fait, les sujets plaçaient aléatoirement les objets selon leurs propres estimations, ce qui génére souvent des erreurs de placement. Une difficulté particulière a été ressenti par la plupart des sujets quand ils voulaient placer l'ordinateur portable en le protégeant du champ électrique émis par l'objet bleu, bien qu'ils étaient informés des valeurs des paramètres requis (la charge ***q*** et le champ électrique maximum supporté). Ceci est confirmé par le nombre moyen d'erreurs d'agencement pour chaque type de tâche. En effet, les sujets ont commis moins d'une erreur en moyenne (0.77 erreurs) au cours des tâches assistées contre 4 erreurs au cours des tâches non assistées (Tab. 2).
2. *Nombre d'actions* Comme illustré dans le Tableau 2, les sujets ont interagi avec la scène 3D moins souvent quand la visualisation des zones impossibles était activée. Ils sélectionnaient et manipulaient les objets 39 fois en moyenne durant les tâches non assistées, contre 19 fois en moyenne durant les tâches assistées. Ces résultats peuvent être expliqués par le fait que les sujets ont eu beaucoup de doutes concernant les placements d'objets effectués puisque il n'y a pas d'indication sur la validité de leurs choix. Pour cette raison, la plupart d'entre eux changeaient fréquemment la position des objets dans l'espoir de satisfaire le maximum des contraintes. Au cours des tâches assistées, leur fréquence d'actions est plus faible parce qu'ils avaient une meilleure vision des impossibilités de placement de chaque objet.
3. *Temps d'agencement* Aucune différence significative ($p = 0.3259$, $F_{1,11} = 1.31$) n'a été observée pour les deux niveaux de **A** (sans/avec visualisation des zones). Même si les utilisateurs ont commis beaucoup plus d'erreurs durant, ceci n'a pas d'incidence sur le temps puisqu'ils ne se rendaient pas compte de la non-validité des placements proposés. Interagir deux fois plus avec l'EV durant les tâches non-assistées a certes augmenter le temps de réalisation de la tâche mais le temps nécessaire à l'activation/désactivation des zones durant les tâches assistées vient équilibrer le temps passé dans les deux types de tâches.

	Assistance(A)	Moy	E-type
Temps(sec.)	*Sans assistance*	126.76	33.01
	Avec assistance	129.31	32.09
Erreurs	*Sans assistance*	4	1.22
	Avec assistance	0.77	0.83
Actions des sujets	*Sans assistance*	39.44	5.81
	Avec assistance	19.22	3.73

TABLE 2 – *Moyennes et écart-types des données objectives dans chaque condition expérimentale.*

6.2 Aspects subjectifs

1. *Charge de travail* A travers une échelle de Likert, les sujets devaient évaluer le niveau de la charge de travail ressenti durant les sous-tâches. Les résultats révèlent que l'agencement non assisté a causé un niveau élevé de charge mentale, charge physique, charge temporelle, frustration et a nécessité plus d'effort par rapport à l'agencement assisté.
2. *Satisfaction* 11 sujets ont exprimé une grande satisfaction concernant l'agencement final réalisé durant les tâches assistées. Un seul sujet a donné une réponse neutre à cette question. Pour les tâches non assistées, 4 sujets ont exprimé une satisfaction moyenne, 3 ont donné une réponse neutre et 5 ont affirmé leurs insatisfaction à cause des doutes qu'ils ont eu concernant les placements effectués. Ces résultats confirme l'effet de la visualisation des zones sur la satisfaction ressentie par les sujets. En effet, quand l'assistance était activée, les sujets ont eu l'impression de respecter les contraintes en évitant simplement de placer les objets dans les zones impossibles.
3. *Besoin d'assistance* Une grande majorité des sujets ont affirmé qu'il était primordial d'avoir une assistance afin de perfectionner l'agencement réalisé. Tous

les sujets ont ressenti un besoin d'assistance élevé durant les tâches non assistées. Quant aux tâches assistées, 8 sujets ont rapporté que la visualisation des zones était largement suffisante pour respecter les contraintes. 4 sujets ont exprimé un besoin d'avoir un autre type d'aide durant ces tâches.
Ces résultats révèlent que la visualisation des zones impossibles était, selon les sujets, une assistance suffisante et efficace et à travers laquelle ils sont parvenus à satisfaire le maximum de contraintes.

4. *Facilité d'agencement et souvenir des contraintes* Comme dit précédemment, deux questions supplémentaires ont été posées pour évaluer l'assistance fournie durant les tâches assistées. les résultats montrent que 11 sujets ont trouvé que la tâche est devenue plus facile en étant assisté par le solveur. Un seul sujet a rapporté qu'il n'a pas senti de différence entre la difficulté rencontrée durant les deux types de tâches (avis neutre). 50% des sujets ont affirmé que l'assistance les a beaucoup aidé pour se souvenir des contraintes durant la tâche et par conséquent satisfaire le maximum de celles-ci. Un seul sujet a donné une opinion neutre.

7 Conclusion

Nous avons proposé d'utiliser la PPC afin d'assister l'utilisateur durant l'agencement d'EVs. Un solveur de contraintes efficace (Gecode) a été intégré dans un outil professionnel couramment utilisé pour la création d'EVs et jeux 3D (Unity3D). Notre approche est basée sur un couplage fort entre le solveur et la visualisation 3D des calculs de celui-ci. Une étude expérimentale a été réalisée afin d'étudier l'effet de l'assistance proposée par le solveur (visualisation des zones impossibles) sur des données objectives et subjectives. Cette étude est basée sur une comparaison entre deux types de tâches d'agencement 3D (sans et avec assistance). Les résultats obtenus indiquent que l'assistance visuelle proposée réduit considérablement les erreurs d'agencement et le taux d'interaction avec l'EV. La majorité des sujets ont exprimé une satisfaction élevée vis à vis de l'agencement réalisé et l'assistance fournie. Nous travaillons actuellement sur l'enrichissement de notre système à travers la conception et le développement de nouvelles formes d'assistance.

Références

[1] S. Aukstakalnis, D. Blatner, and F. Roth Stephen. *Silicon Mirage - The Art and Science of Virtual Reality*. Peachpit Press, 1992.

[2] J. Bénabés. Optimisation integrée et interactive pour l'agencement d'espace. *Thése de Doctorat, Ecole Centrale de Nantes*, 2011.

[3] W.-R. Bukowski and H.-C. Séquin. Object associations. In *ACM Symp. On Interactive 3D Graphics, Monterey, CA, USA*, 1995.

[4] G. Burdea and P. Coiffet. La Réalité Virtuelle, Hermes Sciences Pub., 1993.

[5] C. Calderon, M. Cavazza, and D. Diaz. A new approach to the interactive resolution of configuration problems in virtual environments. *Lecture notes in computer science*, 2733 :112 – 122, 2003.

[6] F. Daniel and D.Rina. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, (IJCAI)*, pages 572–578, 1995.

[7] R. Dechter and A. Dechter. Belief maintenance in dynamic constraint networks. *7th Conference On Arti ?cial Intelligence*, 1988.

[8] Unity 3D Game Engine. http ://unity3d.com. 2012.

[9] F. Fages, S. Soliman, and R. Coolen. Clpgui : A generic graphical user interface for constraint logic programming. *Constraints*, 9 :241 – 262, 2004.

[10] T. Fernando, N. Murray, K. Tan, and P. Wimalaratne. Software architecture for a constraint-based virtual environment. *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 147 – 154, 1999.

[11] T. Frühwirth and S. Abdennadher. Essentials of constraint programming,. *Springer Verlag*, 2003.

[12] S. Hart and L. Staveland. Development of nasa-tlx (task load index) : Results of empirical and theoretical research. In *Human mental workload.*, pages 139–183, 1988.

[13] K. Honda and F. Mizoguchi. Constraint-based approach for automatic spatial layout planning. In *Conference on Artificial Intelligence for Applications*. IEEE Press, 1995.

[14] M. Kallman and D. Thalman. Direct 3d interaction with smart objects. In *ACM International Symposium on Virtual Reality Software and Technology, London. UK*, 1999.

[15] E. Poder R. Sadek N. Beldiceanu, M. Carlsson and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic *k*-dimensional objects. In *CP*, pages 180–194, 2007.

[16] C. Pfefferkorn. A heuristic problem solving design system for equipment or furniture layouts. *Communications of the ACM*, 18(5), 1975.

[17] S. Sanchez, O. Le Roux, F. Inglese, H. Luga, and V. Gaildart. Constraint-based 3d-object layout using a genetic algorithm. 2002.

[18] W. Stuerzlinger and G. Smith. Efficient manipulation of object groups in virtual environments. 2000.

[19] T. Tim, B. Rafael, M. S. Ruben, and J. K. Klaas. Rule-based layout solving and its application to procedural interior generation. In *Proceedings of the CASA workshop on 3D advanced media in gaming and simulation (3AMIGAS)*, pages 212–227, 2009.

[20] K. Xu, J. Stewart, and E. Fiume. Constraint-based automatic placement for scene composition. In *Graphics Interface Proceedings, University of Calgary*, 2002.

cptest4choco : test et mise au point des modèles à contraintes

Nadjib Lazaar[1] **Arnaud Gotlieb** [2] **Yahia Lebbah** [3]

[1] LIRMM- Université de Montpellier 2, France
[2] Certus Software V&V Center, Simula Research Laboratory, Oslo, Norway
[3] Laboratoire LITIO, Université d'Oran, ,B.P. 1524 EL-M'Naouar, 31000 Oran, Algérie
lazaar@lirmm.fr arnaud@simula.no ylebbah@gmail.com

Résumé

Les programmes à contraintes écrits dans un langage de haut-niveau (e.g., OPL, COMET, ZINC ou CHOCO) sont de plus en plus utilisés dans des domaines critiques qui nécessitent une phase de test et de validation. Dans ce papier, nous décrivons un environnement logiciel de test et de mise-au-point, nommé *cptest4choco*, dédié aux programmes écrits en CHOCO. Cet environnement implémente des approches de détection, de localisation et de correction automatique de fautes présentes dans les modèles CHOCO. Les résultats obtenus montrent l'efficacité des approches de test pour la mise-au-point des modèles à contraintes écrits en CHOCO.

Abstract

Constraint programs, such as those written in high-level constraint modelling languages (e.g., OPL, COMET, ZINC or CHOCO), are more and more used in critical programs which require to be thoroughly tested and corrected to prevent catastrophic outcomes. This paper presents a software environment for constraint programs written in CHOCO, called *cptest4choco*, and different approaches for an automatic CP tuning (including fault detection, localization and correction). This environment takes into account the specificities of the software development process of CP programs as well as their typical faults. We have obtained experimental results on well-known constraint programs written in CHOCO, showing that their tuning is effective and illustrate the efficiency of *cptest4choco* functionalities.

1 Introduction

Ces dernières années, les langages de modélisation en programmation par contraintes (PPC) [14] ont connu un réel développement. Ils ont été conçus avec un haut niveau d'abstraction permettant une grande expressivité. Des langages comme OPL (Optimization Programming Language)[16],COMET [17], ZINC [12] ainsi que CHOCO [1] proposent des solutions robustes à des problèmes réels. De plus, ces langages commencent à être utilisés dans des applications critiques comme la gestion et le contrôle du trafic aérien [3, 7], le e-commerce [6] et le développement de programmes critiques [1, 5].

Comme tout processus de développement logiciel effectué dans un cadre industriel, les développements de ces programmes à contraintes doivent désormais inclure une phase de test, de vérification formelle et/ou de validation. Ceci ouvre la voie à des recherches orientées vers les aspects génie logiciel dédiés à la PPC, et en particulier le test des programmes à contraintes. A notre connaissance, il n'existe pas d'outils et d'environnements dédiés au test de ces langages de haut niveau. Par ailleurs, les théories de test des programmes conventionnels existantes semblent être des schémas inopérants pour capter les spécificités du test des programmes à contraintes pour différentes raisons :

- En PPC, un problème est modélisé avec un ensemble de contraintes non ordonné. Il n'y a donc pas de notion de séquence, comme on en trouve dans les programmes conventionnels.
- Le flot de contrôle dans un programme conventionnel est guidé par les données alors qu'en PPC il est guidé par les contraintes.
- L'optimisation et le raffinement du code est une phase susceptible d'introduire des fautes dans n'importe quel type de programmation. Les techniques appliquées sur des programmes conventionnels sont complètement différentes de celles des programmes à

1. www.emn.fr/z-info/choco-solver/

contraintes.

Dans un processus de développement des programmes à contraintes, tel qu'il semble être pratiqué dans les milieux industriels, il est usuel de démarrer à partir d'un modèle simple et très déclaratif, une traduction fidèle de la spécification du problème, sans accorder un intérêt à ses performances. Par la suite, ce modèle est raffiné par l'introduction de contraintes redondantes ou reformulées, l'utilisation de structures de données optimisées, de contraintes globales, de contraintes qui cassent les symétries, etc. Nous pensons que l'essentiel des fautes introduites est compris dans ce processus de raffinement. Le test et la révision de contraintes dans un but de correction sont des tâches manuelles qui nécessitent une grande expertise.

Dans cet article, nous présentons le développement d'une bibliothèque de test et de mise-au-point basé sur un cadre formel de test des programmes à contraintes [9, 11]. Cette bibliothèque, nommée *cptest4choco*, permet d'effectuer de la détection, de la localisation et de la correction automatique des fautes des programmes à contraintes écrit en CHOCO. Étant donnée une spécification d'un problème, la traduction de cette spécification en un premier modèle à contraintes, noté MO pour *Modèle-Oracle*, représente notre référence de test qui permettra de détecter des fautes introduites dans un *programme à contraintes sous test* (i.e., noté CPUT pour *Constraint Program Under Test*). A l'instar du MO, le CPUT est dédié à résoudre des instances difficiles du problème. La distance et la divergence (i.e., ensemble de non-conformités) entre le MO et le CPUT sont dues aux fautes introduites durant le processus de raffinement.

Dans [8], nous avons augmenté le cadre de test avec une approche de localisation de faute qui retourne un ensemble de contraintes susceptibles de contenir des fautes. Dans [10], nous avons proposé une approche qui permet de calculer automatiquement des corrections possibles.

La bibliothèque *cptest4choco* nous a permis de faire une étude expérimentale sur des problèmes académiques : règles de golomb et les n-reines). L'objectif de cette étude est de montrer qu'il est possible de tester et/ou corriger automatiquement des programmes à contraintes.

Le reste de cet article est organisé comme suit. La section 2 présente le contexte et illustre nos approches sur l'exemple des règles de Golomb. La section 3 passe en revue la partie théorique du cadre de test et de mise-au-point en PPC. La section 4 présente la bibliothèque de test *cptest4choco*. En section 5, nous présentons quelques résultats expérimentaux, enfin, la section 6 conclut l'article.

2 Exemple Illustratif

Dans cette section, nous illustrons nos approches de test et de mise-au-point en PPC sur les règles de Golomb. Ces règles trouvent leur terrain d'application dans des domaines variés tels les communications radio, rayons X en cristallographie, les codes convolutionnels doublement orthogonaux, tableaux des antennes linéaires, communications PPM (Pulse Phase Modulation) [15, 2]. La formulation de ce problème est assez simple, en revanche, la résolution est très couteuse sur des instances qui semblent être accessibles (e.g., règles de 15 marques) [13]. Une règle de Golomb est définie comme un ensemble de m entiers $0 = x_1 < x_2 < ... < x_m$ tel que les $m(m-1)/2$ distances $\{x_j - x_i | 1 \leq i < j \leq m\}$, sont différentes. Une telle règle d'ordre m est dite de longueur x_m. L'objectif est de trouver une règle de longueur minimale.

```
/*******************
 *   Model-Oracle  *
 *******************/

int m = size;
MO = new CPModel();
// variables
IntegerVariable[] marks = Choco.makeIntVarArray("mark", m, 0, m*m);
IntegerVariable length = Choco.makeIntVar("length", 1, m*m, Options.V_OBJECTIVE);
// Constraints
MO.addConstraint(Choco.eq(length, marks[m-1]));                          //c1

for (int i = 0; i < m-1; i++)
    MO.addConstraint(Choco.lt(marks[i], marks[i+1]));                    //c2


for (int i = 0; i < m-1; i++)                                            //c3
    for (int j = i+1; j < m; j++)
        for (int k = 0; k < m-1; k++)
            for (int l = k+1; l < m; l++)
                if(i!=k || j!=l)
    MO.addConstraint(Choco.neq(Choco.minus(marks[j],marks[i]),
                     Choco.minus(marks[l],marks[k])));
```

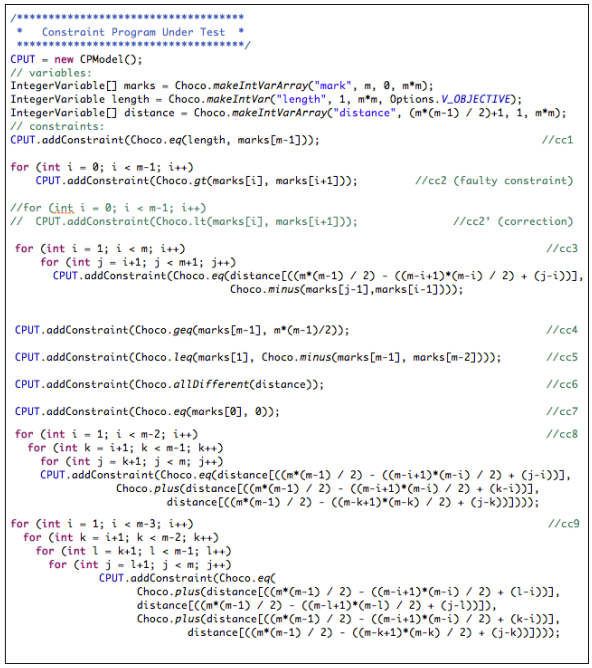
FIGURE 1 – MO for Golomb rulers written in CHOCO.

Un premier Modèle déclaratif du problème en CHOCO est donné dans la Figure 1. Dans notre cadre de test, ce modèle représente le MO. Nous obtenons par la suite un modèle optimisé suite à un processus de raffinement incluant l'ajout de contraintes globales, reformulation de contraintes, cassure de symétrie, ajout de contraintes redondantes, etc. Ce modèle représente le CPUT. Un des CPUT possibles, écrit en CHOCO, est donné dans la Figure 2. Il est incontestable que le modèle de la Figure 1 résolve bien le problème des règles de Golomb. Ce qui n'est pas aussi évident pour le modèle de la Figure 2.

Prenons l'instance $m = 6$. *cptest4choco* retourne l'état *unsat* pour le CPUT de la Figure 2. Ce qui révèle la présence d'une faute dans le CPUT en question. En phase de localisation, *cptest4choco* retourne `cc2` comme étant une contrainte suspecte. En effet, une mauvaise formulation de cette contrainte réduit l'ensemble des solutions du CPUT à vide. *cptest4choco* propose une correction qui consiste à remplacer `cc2` par `cc2'`.

3 Contexte

Comme nous l'avons mentionné, nous considérons le premier modèle déclaratif comme oracle de test (i.e., MO). Le MO représente l'ensemble des solutions (au moins une) du problème et est strictement conforme à la spécification de départ.



```
/************************************
 *   Constraint Program Under Test  *
 ************************************/
CPUT = new CPModel();
// variables:
IntegerVariable[] marks = Choco.makeIntVarArray("mark", m, 0, m*m);
IntegerVariable length = Choco.makeIntVar("length", 1, m*m, Options.V_OBJECTIVE);
IntegerVariable[] distance = Choco.makeIntVarArray("distance", (m*(m-1) / 2)+1, 1, m*m);
// constraints:
CPUT.addConstraint(Choco.eq(length, marks[m-1]));                                    //cc1

for (int i = 0; i < m-1; i++)
    CPUT.addConstraint(Choco.gt(marks[i], marks[i+1]));          //cc2 (faulty constraint)

//for (int i = 0; i < m-1; i++)
//  CPUT.addConstraint(Choco.lt(marks[i], marks[i+1]));              //cc2' (correction)

 for (int i = 1; i < m; i++)                                                         //cc3
     for (int j = i+1; j < m+1; j++)
       CPUT.addConstraint(Choco.eq(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (j-i))],
                                   Choco.minus(marks[j-1],marks[i-1])));


 CPUT.addConstraint(Choco.geq(marks[m-1], m*(m-1)/2));                               //cc4

 CPUT.addConstraint(Choco.leq(marks[1], Choco.minus(marks[m-1], marks[m-2])));       //cc5

 CPUT.addConstraint(Choco.allDifferent(distance));                                   //cc6

 CPUT.addConstraint(Choco.eq(marks[0], 0));                                          //cc7

 for (int i = 1; i < m-2; i++)                                                       //cc8
   for (int k = i+1; k < m-1; k++)
     for (int j = k+1; j < m; j++)
     CPUT.addConstraint(Choco.eq(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (j-i))],
                 Choco.plus(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (k-i))],
                         distance[((m*(m-1) / 2) - ((m-k+1)*(m-k) / 2) + (j-k))])));

for (int i = 1; i < m-3; i++)                                                        //cc9
  for (int k = i+1; k < m-2; k++)
    for (int l = k+1; l < m-1; l++)
      for (int j = l+1; j < m; j++)
             CPUT.addConstraint(Choco.eq(
                   Choco.plus(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (l-i))],
                   distance[((m*(m-1) / 2) - ((m-l+1)*(m-l) / 2) + (j-l))]),
                   Choco.plus(distance[((m*(m-1) / 2) - ((m-i+1)*(m-i) / 2) + (k-i))],
                           distance[((m*(m-1) / 2) - ((m-k+1)*(m-k) / 2) + (j-k))])));
```

FIGURE 2 – CPUT pour le problème des règles de Golomb écrit en CHOCO.

Le programme à contraintes sous test, le CPUT, est sensé être conforme aux solutions du MO. La notation $sol(\text{MO})$ (respectivement $sol(\text{CPUT})$) représente l'ensemble des solutions du MO (respectivement CPUT).

Definition 1 (***relation de conformité***)

$$\text{CPUT } conf \text{ MO} \Leftrightarrow sol(\text{CPUT}) \neq \emptyset \wedge sol(\text{CPUT}) \subseteq sol(\text{MO})$$

3.1 Modèle de faute

Dans notre cadre de test, nous nous intéressons aux fautes qui correspondent à des états de non-conformité. Les autres fautes de type syntaxique sur la formulation des contraintes et/ou les fautes présentent au niveau du solveur de contraintes, ne sont pas prises en compte par notre approche.

Notre modèle de fautes est défini à l'aide de trois types de fautes :

Definition 2 (Faute positive ϕ^+) *Une faute positive sur un* CPUT *par rapport à son Modèle-Oracle* MO*, notée ϕ^+, est une faute qui ajoute des solutions au* CPUT *:*

$$\phi^+ \triangleq sol(\text{CPUT}) \backslash sol(\text{MO}) \neq \emptyset$$

Definition 3 (Faute négative ϕ^-) *Une faute négative sur un* CPUT *par rapport à son Modèle-Oracle* MO*, notée ϕ^-, est une faute qui supprime des solutions du* CPUT *:*

$$\phi^- \triangleq sol(\text{MO}) \backslash sol(\text{CPUT}) \neq \emptyset$$

Definition 4 (faute zéro ϕ^*) *Une faute zéro sur un* CPUT *par rapport à son Modèle-Oracle* MO*, notée ϕ^*, est une faute qui réduit l'ensemble des solutions du* CPUT *à vide :*

$$\phi^* \triangleq sol(\text{CPUT}) = \emptyset$$

En utilisant ces trois définitions, une non-conformité entre le CPUT et son MO est tirée d'une faute positive, négative ou de type zéro.

La définition 1 est une définition générique de la relation de conformité. Il est à noter que cette notion de conformité entre ces deux entités peut être plus spécifique en prenant en compte la classe des problèmes abordés : les problèmes de satisfaction de contraintes (recherche d'une ou de toutes les solutions) et les problèmes d'optimisation (recherche d'une solution faisable dans un intervalle donné ou d'une solution optimale).

En conséquence, une faute négative (ϕ^-) n'affecte pas la conformité si nous abordons la classe de problèmes qui cherche une solution, puisque l'objectif est d'atteindre au moins une solution acceptable via le Modèle-Oracle. Ainsi, tout raffinement qui peut supprimer des solutions (mais non pas toutes), comme la cassure de symétrie, reste acceptable.

3.2 Détection de faute

La preuve de conformité sur l'ensemble des instances du problème est un problème indécidable dans le cas général. Ainsi, nous avons proposé dans [9] un processus de test qui permet de détecter les non-conformités. Etant donnée une instance, une non-conformité est soit une solution du CPUT qui n'est pas une solution du MO, soit un état *unsat* du CPUT (i.e., $sol(\text{CPUT}) = \emptyset$). La détection systématique des non-conformités peut être atteinte en combinant la négation des contraintes du MO avec les contraintes du CPUT (i.e., $\text{CPUT} \wedge \neg C_i$ avec $C_i \in$ MO). Une mauvaise formulation d'une contrainte du CPUT peut ajouter/supprimer des solutions. Dans [9], nous avons proposé un algorithme, nommé $one_negated$ donné ci-dessous, qui retourne des non-conformités si ces dernières existent. Il est à noter qu'une non-conformité entre un CPUT et son MO est une solution x retournée par un processus de test (du fait d'une faute ϕ^+ ou ϕ^-)), ou un état $unsat$ (du fait d'une faute ϕ^*).

Algorithm 1: one_negated(CPUT, MO)

```
foreach C_i ∈ MO do
    x ← solve(CPUT ∧ ¬C_i)
    if x then return x
return ∅
```

3.3 Localisation de faute

Dans [8], nous avons proposé une approche qui permet d'expliquer une faute dans un CPUT. Cette approche de localisation retourne une explication formée de contraintes suspectes. L'algorithme $locate$, donné dans cette section, permet de calculer cet ensemble de contraintes suspectes. L'approche est basée sur la définition suivante. Etant donné un CPUT $= \{C_1, C_2...C_n\}$ qui n'est pas conforme à son MO avec la présence d'une faute ϕ, un *ensemble suspect* est un sous ensemble de contraintes suspectes qui explique la faute ϕ :

Definition 5 (contrainte suspecte) $C_i \in$ CPUT *est suspecte ssi :*
$\phi^+, \phi^* : sol(\text{MO}) \cap sol(\text{CPUT} \backslash C_i) \neq \emptyset$
$\phi^- \quad : sol(\text{MO} \wedge \neg C_i) \neq \emptyset$

Algorithm 2: locate(CPUT, MO, nc)

```
1  SuspiciousSet ← ∅
2  if (φ+ ∨ φ*) then
3      foreach Ci ∈ CPUT do
4          if sol(MO ∧ CPUT\Ci) ≠ ∅ then
5              SuspiciousSet ← SuspiciousSet ∪ {Ci}
6  else
7      foreach Ci ∈ CPUT do
8          if sol(MO ∧ ¬Ci) ≠ ∅ then
9              SuspiciousSet ← {Ci}
10             break
11 if CPUT ≡ SuspisiousSet then SuspisiousSet ← ∅
12 return SuspiciousSet
```

L'algorithme 2 prend en entrée le Modèle-Oracle MO, le CPUT et la non-conformité nc. L'idée de base développée dans cet algorithme est de localiser les contraintes fautives en parcourant l'ensemble des contraintes du CPUT et en retenant que celles qui répondent à la définition 5.

3.4 Correction automatique des fautes

Une fois une faute localisée dans un CPUT, notre approche de correction tente de remplacer la ou les contraintes fautives par une correction possible afin de rétablir un état de conformité avec le Modèle-Oracle. L'approche proposée est basé sur le calcul d'un sous-ensemble de contraintes à partir de MO qui remplace la partie fautive des contraintes. L'algorithme 3 permet de calculer l'ensemble des contraintes $CorrectionSet$ qui corrige le CPUT. Corriger le CPUT consiste à enlever les solutions du CPUT qui ne sont pas des solutions de MO. En d'autres termes, réduire l'ensemble des solutions de MO $\wedge \neg$CPUT à vide ($sol(\text{MO} \wedge \neg\text{CPUT})$). L'algorithme 3 prend en entrée le $SuspiciousSet$ retourné par $locate$. Si cet ensemble est vide (i.e., CPUT sous-contraint), $correction$ calcule l'ensemble des contraintes du Modèle-Oracle à ajouter au CPUT. Autrement, l'algorithme calcule une correction possible qui doit remplacer le $SuspiciousSet$ pour rétablir la conformité. En d'autres termes, l'ensemble des contraintes correctrices C_i représentent des contraintes du MO qui satisfont : $sol((\text{CPUT} \backslash suspiciousSet) \wedge \neg C_i) \neq \emptyset$ où il existe des solutions de $(\text{CPUT} \backslash suspiciousSet)$ qui ne satisfont pas C_i.

Algorithm 3: correction(MO, CPUT, $SuspiciousSet$)

```
1  correctionSet ← ∅
2  P ← CPUT\SuspiciousSet
3  foreach Ci ∈ MO do
4      if sol(P ∧ ¬Ci) ≠ ∅ then
5          correctionSet ← correctionSet ∪ {Ci}
6  return correctionSet
```

4 *cptest4choco* library

Dans cette section, nous donnons un survol de notre cadre de test *cptest4choco* pour tester les programmes à contraintes écrits en CHOCO. *cptest4choco* est une librairie de test écrite en Java pour l'environnement CHOCO, en produisant en sortie des programme CHOCO dédiés. Ces programmes à contraintes en sortie sont résolus pour détecter, localiser et corriger les fautes. *cptest4choco* implémente les algorithmes 1,2 et 3 donnés dans les sections précédentes. La spécification de la librairie *cptest4choco* et tous les modèles utilisés pour l'expérimenter, sont accessibles en ligne [2]. La distribution courante contient quatre paquetages, `de détection`, `de localisation`, `de correction` et `de négation`, décrits ci-dessous.

4.1 *cptest4choco* : le paquetage de `détection`

En premier, ce paquetage contient une implémentation des modèles de faute et les relations de non-conformité (i.e., ϕ^+, ϕ^- et ϕ^*). Deuxièmement, il implémente les relations de conformité décrites dans [11]. Ce dernier travail (i.e., [11]) contient seulement les définitions de base ((e.g., définition 1). Ce paquetage contient aussi une implémentation complète de l'Algorithme 1 utilisé dans le processus de détection des fautes. Cet algorithm prend entrée deux modèles CHOCO et retourne, si possible, un(des) cas de non-conformité, c'est-à-dire une instanciation des variables satisfaisant un modèle et non-satisfaisant le deuxième. D'un

2. www.lirmm.fr/∼lazaar/cptest4choco

point de vue utilisateur, plusieurs fonctionnalités de *cptest4choco* sont disponibles en invoquant cette bibliothèque dans un projet Choco.

4.2 *cptest4choco* : le paquetage de `localisation`

Ce paquetage implémente notre approche de localisation des fautes que nous avons introduite dans la section 3.3. Le processus de localisation prend en entrée le cas de non-conformité retourné par le paquetage de `detection`, et retourne l'ensemble des contraintes *suspectes*. Cet ensemble permet de repérer la contrainte à l'origine de l'erreur dans le modèle à contrainte.

4.3 *cptest4choco* : le paquetage de `correction`

Ce paquetage implémente l'algorithme 3. Le processus de correction automatique prend en entrée l'ensemble des contraintes suspectes, et propose en sortie un ensemble de corrections possibles pour rétablir la conformité entre CPUT et son MO. Nous verrons dans la partie expérimentale que ce paquetage est le plus élaboré dans notre environnement *cptest4choco*, et nécessite un effort considérable de compréhension de la part de l'utilisateur.

4.4 *cptest4choco* : négation des contraintes

Notre approche de détection des fautes et leur correction automatique est basée sur la réfutation des contraintes, et l'exploitation d'une négation automatique des contraintes. La librairie *cptest4choco* dispose de la négation de plusieurs contraintes globales sur les domaines discrets, soit en les réécrivant automatiquement en utilisant les opérateurs arithmétiques et relationnels, soit en faisant appel à des contraintes globales. Par exemple, la négation de la contrainte globale *atLeast* peut être obtenue en utilisant la contrainte globale *atMost*. Cependant, les contraintes globales telles que la *cumulative* ou la contrainte *circuit*, nécessitent, pour leur négation, de lui développer une contrainte dédiée. La négation des contraintes globales complexes fait partie de nos travaux futures.

5 Validation expérimentale

Dans cette section, nous présentons les premiers résultats expérimentaux de *cptest4choco* sur des problèmes académiques. Nous avons sélectionné les règles de golomb ainsi que les n-reines. Les résultats que nous donnons sur la détection des fautes ainsi que la mise-au-point des programmes à contraintes ont été produits sur une machine Intel Core i7 CPU 2.9Ghz Mac OS, 8 GO de RAMM DDR3.

5.1 Procédure d'expérimentation

L'objectif de notre étude expérimentale est multiple : valider nos approches de test, de localisation des fautes et de correction automatique ; valider la capacité de *cptest4choco* à détecter des fautes dans des modèles CHOCO ; montrer que le test d'un programme à contraintes est moins coûteux que le résoudre ; étudier le temps nécessaire à la localisation et la correction automatique des fautes.

Étant donné un problème, la procédure que nous avons adoptée est la suivante :

- Traduire, en premier lieu, la spécification du problème en un premier modèle à contraintes MO en CHOCO.
- Appliquer les différents raffinements possibles pour obtenir un programme à contraintes $\mathcal{P}$ dédié à résoudre des instances difficiles du problème.
- Injecter manuellement des fautes significatives dans $\mathcal{P}$ et obtenir des CPUT incluant ces fautes.
- Soumettre chaque CPUT à *cptest4choco* pour une phase de test.
- Dans le cas où une faute ϕ est détectée, lancer *cptest4choco* pour une phase de localisation.
- Lancer la correction automatique de *cptest4choco*.

Les fautes significatives ici représentent une relaxation et/ou un renforcement de contraintes, une mauvaise connexion d'une variable auxiliaire, l'ajout d'une contrainte de symétrie et sa négation, mauvaise formulation de contrainte redondante, utilisation inadaptée d'une contrainte globale, etc.

Le tableau 1 présente les résultats de *cptest4choco* sur les règles de Golomb et les n-reines. Le tableau est constitué de quatre colonnes : CPUTs : cette colonne décrit les différents CPUT et l'injection de fautes qui indique dans quelle contrainte du CPUT la faute ϕ a été introduite ; Test : Cette colonne est dédiée à la phase de test de *cptest4choco* avec des temps en seconde ; Localisation : Cette partie du tableau présente les $SuspiciousSet$ retournés par $locate$; Correction : Les résultats de la phase de correction de *cptest4choco* avec la taille des $CorrectionSet$ et des temps en secondes.

5.2 Règles de Golomb

Comme nous l'avons présenté dans la section 2, le problème des règles de Golomb consiste à trouver des règles où les distances entre les paires de marques sont de valeurs différentes. C'est aussi un problème d'optimisation où on cherche une règle de longueur minimale. Pour nos expérimentations, nous avons choisi l'instance $m = 8$ qui couvre l'ensemble des contraintes. En alternant la suite de raffinements, nous avons introduit des fautes ϕ, ce qui nous a permis de produire sept CPUT différents. Par exemple, la faute injectée dans CPUT5 consiste à remplacer la contrainte globale `allDifferent` par :

TABLE 1 – Détection des fautes, localisation et correction sur des problèmes classiques

	Mutants	Fault injected	Detection		Localization		Correction		
			non-conformity	time	susp. cstr	time	removed cstr	added cstr	time
Golomb rulers (n=8)	Mut1	cc2	ϕ^+:[0 1 2 3 4 5 6 7]	0.87	∅	2.48	∅	368 EC	42.13
	Mut2	cc3	ϕ^+:[0 1 2 3 4 5 6 7]	0.11	∅	1.69	∅	431 EC	43.53
	Mut3	cc3	ϕ^-:[0 7 15 17 18 31 37 58]	0.08	∅	1.72	∅	466 EC	46.59
	Mut4	cc5	ϕ^+:[0 4 8 12 16 20 24 28]	0.16	∅	1.46	∅	340 EC	14.25
	Mut5	cc1	ϕ^*:sol(Mut5)=∅	0.35	cc1	315.85	cc1	763 EC	30.49
	Mut6	cc7	ϕ^*:sol(Mut6)= ∅	0.37	cc7	137.07	cc7	763 EC	34.81
	Mut7	cc9	ϕ^*:sol(Mut7)= ∅	35.10	cc9	13.50	cc9	∅	108.45
n-queens (n=8)	Mut1	cc12	ϕ^*:sol(Mut1)=∅	0.34	cc12	0.87	cc12	∅	0.17
	Mut2	cc11	ϕ^*:sol(Mut2)=∅	0.39	cc11	0.99	cc11	∅	0.44
	Mut3	cc12	ϕ^*:sol(Mut3)=∅	0.20	cc12	0.40	cc12	∅	0.36
	Mut4	cc11	ϕ^*:sol(Mut2)=∅	0.01	cc11	0.18s	cc11	∅	0.30
	Mut5	cc6	ϕ^+:[8 7 6 5 4 3 2 1]	0.52	cc6	0.23	cc6	28 EC	0.67
	Mut6	cc7	ϕ^-[1 5 8 6 3 7 2 4]	0.24	∅	0.31	∅	28 EC	0.44

susp. cstr : suspicious constraints EC : elementary constraint

$$\forall i, j, k \quad s.t., i \neq j :$$
$$mark_{i+1} = mark_i + k \Rightarrow mark_{j+1} = mark_j + k$$

Prenons le CPUT2, CPUT3 et CPUT5 de la parte règles de Golomb du tableau 1. *cptest4choco* retourne une non-conformité pour le CPUT2 (i.e., x=[0 1 2 3 4 5 6 7] où x n'est pas une règle de Golomb avec 1−0=2−1). Cette non-conformité représente une solution du CPUT2 qui n'est pas une solution du Modèle-Oracle MO (i.e., faute de type ϕ^+). La phase de localisation de *cptest4choco* retourne l'état *sous-contraint* avec un ensemble vide pour dire que le CPUT en question contient une contrainte relâchée et ne prend pas en compte la totalité de la spécification du problème. Comme phase de correction, *cptest4choco* nous retourne un ensemble de 431 contraintes élémentaire à ajouter au CPUT2 pour corriger la faute et rétablir la conformité avec le MO.

En ce qui concerne le CPUT3, *cptest4choco* détecte une faute de type ϕ^-. En effet, la solution retournée x=[0 7 15 17 18 31 37 58] représente une non-conformité où x ne peut être retournée par le CPUT3. En phase de localisation, *cptest4choco* pointe la contrainte cc3 du CPUT3 comme contrainte suspecte. En effet, la faute a été introduite dans la contrainte cc7 du CPUT3. La phase de correction retourne, en moins de 1min, 466 contraintes élémentaires qui peuvent remplacer cc3 dans l'instance $m = 8$ du CPUT3 pour rétablir la conformité avec le Modèle-Oracle.

cptest4choco arrive à détecter également les fautes de type ϕ^*. Prenons maintenant le CPUT5 dont la faute introduite dans cc1 le rend insatisfiable (i.e., $\phi^* \Rightarrow sol(Mut5) = \emptyset$). La phase de localisation retourne toutes les contraintes de type cc1 comme étant des contraintes suspectes. *cptest4choco* propose par la suite de remplacer les contraintes de type cc1 par un ensemble de 763 contraintes élémentaires.

```
/*******************
 *   Model-Oracle   *
 *******************/

int n = size;
int sum= n*(n+1)/2;
MO = new CPModel();
// variables
IntegerVariable[] queens = Choco.makeIntVarArray("queen", n, 1, n);
// constraints
for (int i = 0; i < n-1; i++)                                    //c1
    for(int j=i+1; j<n; j++)
        MO.addConstraint(Choco.neq(queens[i], queens[j]));

for (int i = 0; i < n-1; i++)                                    //c2
    for(int j=i+1; j<n; j++)
        MO.addConstraint(Choco.neq(Choco.plus(queens[i], i),
                Choco.plus(queens[j], j)));


for (int i = 0; i < n-1; i++)                                    //c3
    for(int j=i+1; j<n; j++)
        MO.addConstraint(Choco.neq(Choco.minus(queens[i], i),
                Choco.minus(queens[j], j)));
```

FIGURE 3 – MO for n-queens written in CHOCO.

5.3 N-reines

Le problème des n-reines est un problème classique en PPC qui consiste à placer n reines d'un jeu d'échecs sur un échiquier de $n \times n$ cases sans que les reines ne puissent se menacer mutuellement. Par conséquent, deux dames ne devraient jamais partager la même ligne, colonne, ou diagonale. La Figure 3 présente le Modèle-Oracle des n-reines. Nous avons appliqué une suite de raffinements pour avoir un modèle optimisé avec de nouvelles structures de données qui représentent les diagonales descendantes et ascendantes, des contraintes qui cassent la symétrie, des contraintes redondantes, des contraintes globales, etc. Durant cette suite de raffinements, nous avons introduit des fautes ϕ afin d'obtenir sept CPUT contenant des fautes. Prenons le CPUT3 et CPUT5 du tableau 1 :

Le CPUT3 est constitué de 12 ensembles différents de contraintes, une instance de 8-reines génère 48 variables et 236 contraintes. La faute ϕ injectée dans CPUT3 repré-

sente une mauvaise formulation de la contrainte `cc12` qui réduit son ensemble de solutions à vide. Soumettre CPUT3 à *cptest4choco* pour une phase de test permet de détecter une faute de type ϕ^* en moins de 1s. La phase de localisation de *cptest4choco* retourne `cc12` comme contrainte suspecte pour CPUT3. En effet, la faute ϕ^* a été introduite dans `cc12`. Pour corriger cette faute et étant donnée la contrainte `cc12` une contrainte redondante, *cptest4choco* propose d'enlever cette contrainte de CPUT3 pour rétablir la conformité avec le Modèle-Oracle.

Pour le CPUT5, nous avons introduit une faute ϕ dans la formulation de la contrainte `cc6`. CPTEST détecte une faute ϕ^+ avec la solution `q1=[8 7 6 5 4 3 2 1]`. Cette solution n'est pas une solution valide des 8-reines où les reines sont positionnées sur une même diagonale descendante. Cette non-conformité affirme la présence des faute de type ϕ^+ dans CPUT5. La phase de localisation retourne également la contrainte fautive comme suspecte `cc6`. Ici, *cptest4choco* propose de remplacer la `cc4` par 28 contraintes élémentaires pour corriger l'instance $n = 8$ du CPUT5. Les 28 contraintes sont un sous-ensemble des contraintes élémentaires qu'encapsule `c2` dans le Modèle-Oracle. Par conséquent, remplacer `cc6` par c2 corrige toute instance du CPUT5.

cptest4choco retourne une non-conformité pour le CPUT6. Cette non-conformité est une solution du Modèle-Oracle qui n'est pas une solution du CPUT6 (i.e., `q2=[1 5 8 6 3 7 2 4]`). Ainsi, `q2` révèle la présence d'une faute de type ϕ^-. Cette faute a été injectée dans `cc7`. La phase de localisation retourne toutes les contraintes de type `cc7` comme suspectes dans CPUT6. Une des corrections possibles est celle proposée par *cptest4choco* qui propose un ajout de 28 contraintes élémentaires pour rétablir une conformité entre CPUT6 et le MO.

5.4 Discussion

Une des reproches que nous pouvons émettre sur la validité de notre approche concerne les problèmes que nous avons utilisés dans nos expérimentations. Nous avons choisi les règles de Golomb et les n-reines comme problèmes bien connus dans la communauté PPC. Simples mais non triviaux, ces problèmes servent souvent comme des exemples pour illustrer des techniques en PPC. Ceci nous a permis de valider notre approche sur des données empiriques pertinentes. Cependant, ces problèmes viennent du milieu académique et peuvent ne pas refléter pleinement l'utilisation industrielle de la programmation par contraintes dans des applications critiques. En outre, nous avons injecté manuellement les fautes dans nos propres modèles à contraintes pour des phase de test et de mise-au-point. Bien que ces fautes ont été bien choisies pour montrer les capacités des approches de test et de mise-au-point, elles ne sont pas faciles à repérer et à réparer.

```
/**********************************
 *   Constraint Program Under Test *
 **********************************/
int n = size;
int sum= n*(n+1)/2;
CPUT = new CPModel();
// variables:
IntegerVariable[] queens = Choco.makeIntVarArray("queen", n, 1, n);
IntegerVariable[] rows = Choco.makeIntVarArray("row", n, 1, n);
IntegerVariable[] diag1 = Choco.makeIntVarArray("diag1", n);
IntegerVariable[] diag2 = Choco.makeIntVarArray("diag2", n);
IntegerVariable[] diag3 = Choco.makeIntVarArray("diag3", n);
IntegerVariable[] diag4 = Choco.makeIntVarArray("diag4", n);
// constraints:
CPUT.addConstraint(Choco.allDifferent(queens));                  //cc1


for(int i=0; i<n; i++)                                           //cc2
    CPUT.addConstraint(Choco.eq(diag1[i], Choco.plus(queens[i], i)));

for(int i=0; i<n; i++)                                           //cc3
    CPUT.addConstraint(Choco.eq(diag2[i], Choco.minus(queens[i], i)));

for(int i=0; i<n; i++)                                           //cc4
    CPUT.addConstraint(Choco.eq(diag3[i], Choco.plus(rows[i], i)));

for(int i=0; i<n; i++)                                           //cc5
    CPUT.addConstraint(Choco.eq(diag4[i], Choco.minus(rows[i], i)));

CPUT.addConstraint(Choco.allDifferent(diag1));                   //cc6
CPUT.addConstraint(Choco.allDifferent(diag2));                   //cc7
CPUT.addConstraint(Choco.allDifferent(diag3));                   //cc8
CPUT.addConstraint(Choco.allDifferent(diag4));                   //cc9

CPUT.addConstraint(Choco.inverseChanneling(queens, rows));       //cc10

CPUT.addConstraint(Choco.eq(Choco.sum(queens),sum));             //cc11
CPUT.addConstraint(Choco.eq(Choco.sum(rows),sum));               //cc12
```

FIGURE 4 – CPUT for n-queens written in CHOCO.

Nous ne savons pas si elles sont réalistes ou non. Contrairement à d'autres langages où plusieurs programmes et bugs sont disponibles (par exemple, Java, C et C ++), les programmes écrits dans des langages PPC ne sont pas disponibles à partir des référentiels sur le Web.

Etant donné que CHOCO est une bibliothèque Java, nous pouvons utiliser la notation JML pour exprimer des post-conditions du modèle en question. Des environnements tels que [1] permettent de trouver des contre-exemples où certaines données de test satisfont le programme Java et ne répondent pas à la post-condition écrite en JML. Ces environnements sont dédiés à la vérification sémantique des programmes et ils sont souvent confrontés à des difficultés pour faire face à la nature exponentielle de l'espace de recherche des variables Java. En outre, ces environnements ne sont pas capables de capturer la sémantique des contraintes et de remédier aux mauvais usages des contraintes. Dans les approches que nous avons proposées, nous travaillons à un niveau d'abstraction élevé où l'espace de recherche est beaucoup plus raisonnable et est défini uniquement par les variables de décision. Ici, la post-condition est exprimée comme un modèle CHOCO, ce qui est plus concis et plus compact qu'une post-condition JML écrite en termes de variables Java. En fait, notre approche est étroitement dédié aux développeurs CHOCO, en facilitant la validation d'un processus de raffinement et d'opti-

misation d'un modèle CHOCO.

Au final, nos approches ont été bâties sur l'hypothèse d'avoir une référence de test (modèle-oracle), un modèle de contraintes initial extrait de la spécification du problème. Cependant, cette hypothèse pourrait être difficile à satisfaire. Les premiers modèles ne sont pas nécessairement conservés pour analyse dans le processus de développement industriel.

6 Conclusion

Dans cette article, nous avons présenté une nouvelle plateforme de test et de mise-au-point dédiée aux modèles écrits en CHOCO, nommée *cptest4choco*. Cette plateforme est présentée sous forme d'une bibliothèque Java avec une implémentation des fonctionnalités de test, de localisation de faute et de correction automatique. Ces fonctionnalités et ces approches sont basées sur la notion de Modèle-Oracle de test représenté par un premier modèle déclaratif, simple et fidèle à la spécification de départ du problème en question. La réalisation de *cptest4choco* et son expérimentation nous indiquent que l'approche qui consiste à valider automatiquement des programmes à contraintes en CHOCO, lorsqu'un Modèle-Oracle est disponible, est viable et pourrait être généralisée à d'autres langages PPC.

Références

[1] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Cpbpv : a constraint-programming framework for bounded program verification. *Constraints*, 15(2) :238–264, 2010.

[2] Carlos Cotta, Iván Dotú, Antonio J. Fernández, and Pascal Hentenryck. Local search-based hybrid algorithms for finding golomb rulers. *Constraints Journal*, 12 :263–291, Septembre 2007.

[3] Pierre Flener, Justin Pearson, and Marc Bourgois. Constraint programming for air traffic management : preface. *Knowledge Eng. Review*, 27(3) :287–289, 2012.

[4] Pierre Flener, Justin Pearson, Luis G. Reyna, and Olof Sivertsson. Design of financial cdo squared transactions using constraint programming. *Constraints*, 12(2) :179–205, 2007.

[5] Arnaud Gotlieb. Tcas software verification using constraint programming. *Knowledge Eng. Review*, 27(3) :343–360, 2012.

[6] Alan Holland and Barry O'Sullivan. Robust solutions for combinatorial auctions. In *ACM Conference on Electronic Commerce (EC-2005)*, pages 183–192, 2005.

[7] U. Junker and D. Vidal. Air traffic flow management with ilog cp optimizer. In *International Workshop on Constraint Programming for Air Traffic Control and Management*, 2008. 7th EuroControl Innovative Research Workshop and Exhibition (INO'08).

[8] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. Fault localization in constraint programs. In *ICTAI (1)*, pages 61–67, 2010.

[9] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. On testing constraint programs. In *CP*, pages 330–344, 2010.

[10] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A framework for the automatic correction of constraint programs. In *ICST*, pages 319–326, 2011.

[11] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A cp framework for testing cp. *Constraints*, 17(2) :123–147, 2012.

[12] K. Marriott, N. Nethercote, R. Rafeh, P. J. Stuckey, M. Garcia De La Banda, and M. Wallace. The design of the zinc modelling language. *Constraints*, 13(3) :229–267, 2008.

[13] W. T. Rankin. Optimal golomb rulers : An exhaustive parallel search implementation. Master's thesis, Duke University, Durham, 1993.

[14] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, USA, 2006.

[15] Barbara M. Smith, Kostas Stergiou, and Toby Walsh. Modelling the golomb ruler problem. 1999.

[16] P. Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

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

Réutiliser ou adapter les prouveurs SAT pour l'optimisation booléenne

Daniel Le Berre **Emmanuel Lonca** *

Université Lille Nord de France, F-59000 Lille, France
Université d'Artois, CRIL, F-62300 Lens, France
CNRS, UMR8188, F-62300 Lens, France
{leberre,lonca}@cril.fr

Abstract

D'années en années, le succès des approches basées sur SAT dans de nombreux domaines a contribué à la création de solveurs SAT robustes, efficaces, et pensés pour être réutilisables dans des applications tierces. Dans de nombreux cas, ces applications utilisent l'interface incrémentale des prouveurs SAT proposée par Minisat, qui leur permet d'utiliser les solveurs SAT comme des boites noires, c'est à dire sans se préoccuper de leur fonctionnement interne. Si cette approche semble limitée à la résolution de problèmes de décision, elle s'avère aussi utilisée avec succès dans le cadre de problèmes d'optimisation en variables booléennes, tels que l'optimisation pseudo-booléenne ou MAXSAT. Il parait cependant plus intéressant dans ce contexte de permettre au solveur de poursuivre sa recherche quand il trouve une solution, plutôt que de le laisser s'arrêter et commencer la résolution d'un nouveau problème de décision. Cela nécessite au niveau du prouveur SAT de pouvoir recevoir de nouvelles contraintes durant la recherche. C'est ce qui se passe par exemple dans le cadre des prouveurs SMT, ou lors de la génération paresseuse de contraintes en CSP : un prouveur SAT est adapté pour recevoir des clauses pendant la recherche. Dans le cadre des problèmes d'optimisation qui nous intéressent, on souhaite pouvoir créer de manière paresseuse des contraintes de borne, c'est à dire ajouter des contraintes de cardinalité ou des contraintes pseudo-booléennes pendant la recherche. Nous avons intégré cette fonctionnalité sur la plate-forme d'optimisation booléenne libre Sat4j, et intégré un algorithme d'optimisation par contraintes de bornes paresseuses. Nous comparons l'approche boite noire et l'approche paresseuse sur l'ensemble des problèmes d'optimisation issus des compétitions PB10 et MAXSAT10. On n'observe pas de différences flagrantes de performance (en terme de nombre d'instances résolues et de temps d'exécution) entre les deux approches, malgré des comportements différents. Nous montrons que l'approche par ajout de contraintes à la volée peut cependant s'avérer intéressante, par exemple dans le cadre de l'énumération de solutions.

*Ce travail est financé en partie par le conseil régional Nord-Pas de Calais et le programme FEDER.

1 Introduction

D'années en années, le succès des approches basées sur SAT dans de nombreux domaines comme la vérification de matériel [6], la vérification de logiciels [13], la planification [21] ont contribué à la création de solveurs SAT robustes, efficaces, et pensés pour être réutilisables dans des applications tierces. Dans de nombreux cas, ces applications utilisent l'interface incrémentale des prouveurs SAT proposée par Minisat [8, 9], qui leur permet d'utiliser les solveurs SAT comme des boites noires, c'est à dire sans se préoccuper de leur fonctionnement interne (bien que l'état interne de ces solveurs puisse évoluer au fur et à mesure de leur utilisation, notamment en ce qui concerne la base des clauses apprises, mais sans le contrôle de l'utilisateur). Si cette approche semble limitée à la résolution de problèmes de décision, elle s'avère aussi utilisée avec succès dans le cadre de problèmes d'optimisation en variables booléennes, tels que l'optimisation pseudo-booléenne [22] ou MAXSAT[15]. Il parait cependant plus intéressant dans ce contexte de permettre au solveur de poursuivre sa recherche quand il trouve une solution, plutôt que de le laisser s'arrêter et commencer la résolution d'un nouveau problème de décision. Cela nécessite l'introduction au niveau du prouveur SAT d'un mécanisme permettant de pouvoir recevoir de nouvelles contraintes durant la recherche. C'est

ce qui se passe par exemple dans le cadre des prouveurs SMT [4], ou lors de la génération paresseuse de contraintes en CSP [20] : un prouveur SAT est adapté de manière à pouvoir recevoir des clauses pendant la recherche. Dans le cadre des problèmes d'optimisation qui nous intéressent, on souhaite pouvoir créer de manière paresseuse des contraintes de borne, c'est-à-dire ajouter des contraintes de cardinalité ou des contraintes pseudo-booléennes pendant la recherche. L'idée n'est elle-même pas nouvelle, puisque dès l'apparition des prouveurs SAT dirigés par les conflits, des solveurs basés sur ce principe ont vu le jour (comme par exemple BSOLO [16]). Cependant, les prouveurs « spécialisés » conçus il y a une dizaine d'année ont tendance à disparaître au profit de ceux qui réutilisent sans modifications les solveurs SAT (c'est particulièrement évident ces dernières années pour la résolution du problème MAXSAT [10, 18, 2]). Ce qui nous intéresse ici est d'évaluer l'impact (positif ou négatif) d'une communication privilégiée avec le solveur SAT par rapport à une approche boite noire sur une plateforme commune, et de définir précisément quelles sont les conditions nécessaires pour permettre cette communication sans remettre en cause le fonctionnement du solveur.

Nous avons intégré l'ajout de contraintes à la volée sur la plate-forme d'optimisation booléenne libre Sat4j, et intégré un algorithme d'optimisation par contraintes de bornes paresseuses. Nous comparons l'approche boite noire utilisée jusqu'à présent dans Sat4j et l'approche paresseuse sur l'ensemble des problèmes d'optimisation issus des compétitions PB10 et MAXSAT10. On n'observe pas de différence flagrante de performance (en terme de nombre d'instances résolues et de temps d'exécution) entre les deux approches, ce qui s'explique à la fois par les particularités des benchmarks de ces compétitions et par la spécificité du processus d'optimisation. On observe cependant des comportements particuliers, notamment en terme de nombre de solutions intermédiaires rencontrées lors du calcul de la valeur optimale de la fonction objectif sur certaines instances.

Nous présentons tout d'abord l'observation qui a motivé cette étude : l'amélioration significative des performances de l'énumération de solutions basée sur la production de clauses paresseuses. Nous présentons ensuite le cadre de notre travail : l'optimisation en variables booléennes et les approches par résolution successive de problèmes de satisfaction de contraintes booléennes. Nous détaillons ensuite les conditions nécessaires pour l'apprentissage de contraintes dynamiques. Nous présentons ensuite les résultats expérimentaux et concluons par quelques perspectives.

2 Motivation

Nous utilisons les prouveurs Abscon[17] et Sat4j[5] pour illustrer les forces et les faiblesses des approches SAT et CSP pour la résolution de divers problèmes académiques à nos étudiants de Master. Sat4j-CSP traduit les problèmes CSP en SAT par l'encodage support pour les contraintes binaires [11] et direct [27] pour les contraintes n-aires . Afin de fournir des fonctionnalités similaires à Abscon, il était nécessaire d'ajouter l'énumération de solutions à Sat4j-CSP. Sat4j permet d'énumérer les solutions d'une CNF en « bloquant » chaque solution trouvée à l'aide d'une clause. Cette approche n'est généralement pas envisageable sur des instances issues de problèmes réels, car il faut ajouter autant de clauses que de modèles, et chaque clause contient l'ensemble des variables du problème (voir [19] par exemple pour plus de détails). Cependant, cette approche fonctionne relativement bien sur les problèmes jouets que nous utilisons en enseignement.

L'architecture de Sat4j a évolué récemment pour lui permettre à terme d'intégrer un prouveur SMT. Parmi les évolutions, on retrouve la possibilité d'ajouter une clause conflictuelle à certains moments clés de la recherche (lorsqu'une solution a été trouvée par exemple). Afin de tester cette fonctionnalité, nous avons décidé de réaliser un énumérateur de solutions basé sur cette production de clause à la volée. Cette nouvelle approche permet d'énumérer les 14200 solutions d'un problème de 12 reines en 113 secondes sur un MacBook 2009 contre 593 secondes pour l'ancienne approche.

Nous avons évalué les deux approches sur les instances SAT utilisées évaluer une nouvelle technique d'énumération dans [19]. La principale difficulté était de trouver des instances satisfiables avec un nombre raisonnable de solutions (de quelques centaines à quelques centaines de millier). CBS correspond à des instances dont la taille du backbone est fixe pour des instances 3-SAT à 100 variables booléennes [25] (nous avons utilisé les plus grandes instances contenant 449 clauses). UF correspond aux instances satisfiables 3-SAT aléatoires de SATLIB (de 20 à 250 variables booléennes). FLAT correspond à des problèmes de coloriage de graphes 3-coloriables (de 20 à 600 variables booléennes). Le tableau 1 résume les résultats de cette expérimentation, effectuée sur des machines à base de processeurs Intel Quad-core XEON X5550 2,66 GHz équipées de 32 Go de mémoire avec un temps limite de 2 min. Pour chaque approche, on comptabilise comme résolue une instance pour laquelle le solveur a pu énumérer toutes les solutions de la CNF dans le temps limite.

Dans les trois cas, l'approche par ajout de clauses paresseuses permet d'obtenir de bien meilleurs résultats que l'approche par clause bloquante. Si on considère les temps de calcul sur les instances résolues par les deux approches (voir le tableau 2), on se rend compte qu'en moyenne l'approche « à la volée » énumère les solutions environ dix fois plus rapidement. Les médianes relativement proches nous apprennent que plus les instances sont difficiles, plus le gain est important, ce qui semble naturel. Par ailleurs, nous avons lancé les mêmes expérimentations, avec un temps limite de 20 minutes : bien que les deux solveurs profitent de l'augmentation du temps qui leur est imparti (voir le tableau 4), la différence d'efficacité entre les deux approches est encore plus flagrante, comme le montre le tableau 4 ; ce qui confirme que plus le nombre de modèles est important, plus le gain de l'approche « à la volée » est important dans ce contexte.

Pour conclure, lors de nos expérimentations, l'approche « à la volée » a été capable d'énumérer jusqu'à 735847 solutions en deux minutes, quand l'autre approche n'a pas pu dépasser 67637 solutions.

catégorie	#inst.	boite noire	à la volée
CBS	5000	3622	**4724**
uf	3700	2991	**3300**
flat	1700	1058	**1232**

TABLE 1 – Comparaison boite noire vs ajout de clauses à la volée pour l'énumération de solutions en nombre d'instances dont les solutions ont toutes été énumérées — temps limite de 2 minutes

catégorie	boite noire	à la volée
CBS	10373 (770)	**1029** (**647**)
uf	3653 (**94**)	**423** (103)
flat	13168 (2009)	**1319** (**1010**)

TABLE 2 – Comparaison des temps de calcul moyens (et médian) par séries, en ms, pour l'énumération de solutions — temps limite de 2 minutes, cas des instances résolues par les deux solveurs

catégorie	#inst.	boite noire	à la volée
CBS	5000	4076	**4881**
uf	3700	3110	**3408**
flat	1700	1151	**1675**

TABLE 3 – Comparaison boite noire vs ajout de clauses à la volée pour l'énumération de solutions en nombre d'instances dont les solutions ont toutes été énumérées — temps limite de 20 minutes

catégorie	boite noire	à la volée
CBS	53239 (1045)	**1584** (**694**)
uf	18264 (**101**)	**667** (121)
flat	39730 (2447)	**1672** (**1013**)

TABLE 4 – Comparaison des temps de calcul moyens (et médian) par séries, en ms, pour l'énumération de solutions — temps limite de 20 minutes, cas des instances résolues par les deux solveurs

Ces résultats encourageants nous ont conduit à généraliser ce principe au cadre des contraintes de cardinalité et des contraintes pseudo-booléennes, dans le but d'améliorer les performances des prouveurs pseudo-booléens et MAXSAT de Sat4j.

Nous présentons tout d'abord l'approche dite « boite noire » utilisée par de nombreux prouveurs (dont Sat4j) pour résoudre des problèmes d'optimisation en variables booléennes. Nous présentons ensuite l'approche « boite grise », qui permet une plus forte communication avec le prouveur SAT, pour permettre la génération de contraintes à la volée.

3 Optimisation par boite noire

Soit V un ensemble de variables booléennes. Un littéral l représente une variable booléenne $v \in V$ ou sa négation $\neg v$. On note L l'ensemble des littéraux construits à partir de V. On note $\bar{l}$ l'opposé d'un littéral, c'est à dire $\bar{l} = v$ si $l = \neg v$ et $\bar{l} = \neg v$ si $l = v$. Une clause est une disjonction de littéraux, $\bigvee l_i$. Une contrainte de cardinalité est une fonction booléenne de la forme $\sum l_i \bowtie k$ où les l_i sont des littéraux, $\bowtie \in \{<, \leq, =, \geq, >\}$ et k, le degré de la contrainte, est un entier naturel. Les variables booléennes sont interprétées comme des variables entières dans $\{0, 1\}$ avec $\bar{l} = 1 - l$. Une clause est une contrainte de cardinalité de degré 1, i.e. $a \vee \neg b \vee c \equiv a + \neg b + c \geq 1$. Toute contrainte de cardinalité peut être représentée sous la forme $\sum l_i \geq k$: $a + \neg b + c < 2 \equiv a + \neg b + c \leq 1 \equiv (1 - \neg a) + (1 - b) + (1 - \neg c) \leq 1 \equiv -\neg a - b - \neg c \leq -2 \equiv \neg a + b + \neg c \geq 2$. Une contrainte pseudo-booléenne est une fonction booléenne de la forme $\sum w_i \times l_i \bowtie k$ où les l_i sont des littéraux, $\bowtie \in \{<, \leq, =, \geq, >\}$ et les w_i, les coefficients des littéraux, et k, le degré de la contrainte, sont des entiers naturels. Une contrainte de cardinalité est une contrainte pseudo-booléenne particulière dont les coefficients sont tous égaux à 1. Toute contrainte pseudo-booléenne peut être représentée sous la forme $\sum w_i \times l_i \geq k$. Voir [22] pour plus de détails.

Soit ϕ une conjonction de contraintes booléennes (clauses, contraintes de cardinalité, contraintes pseudo-booléennes) sur l'ensemble de variables V et f

Algorithme 1 : Optimisation par renforcement (recherche linéaire)

entrée : Un ensemble de clauses, de contraintes de cardinalité et de contraintes pseudo-booléennes ϕ, et une fonction d'objectif à minimiser f

sortie : un modèle de ϕ, ou UNSAT si le problème est insatisfiable.

```
⟨réponse,certificat⟩ ← estSatisfiable (φ);
si réponse est UNSAT alors
    retourner UNSAT
fin
répéter
    m ← certificat;
    ⟨réponse,certificat⟩ ← estSatisfiable (φ ∪ {f < f (m)});
jusqu'à (réponse est UNSAT);
retourner m ;
```

une fonction linéaire de L dans $\mathbb{N}$. Il est possible de calculer un modèle de ϕ minimisant f, en utilisant l'algorithme 1. Le problème d'optimisation pseudo-booléen est alors remplacé par un nombre fini de problèmes de décision pseudo-booléens. On distingue deux approches principales pour l'implémentation de cet algorithme : celle qui réutilise des solveurs capables de raisonner avec des contraintes pseudo-booléennes nativement (des prouveurs pseudo-booléens) et celle qui réutilise des prouveurs SAT classiques et traduit la contrainte pseudo-booléenne $f < f(m)$ en CNF.

La première approche est utilisée dans Sat4j. Son principal intérêt est de représenter de manière compacte ces contraintes. Sat4j traduit les problèmes MaxSat en problèmes d'optimisation pseudo-booléens. Les résultats de Sat4j lors de la compétition MaxSat 2009 [1] le situent dans l'état de l'art, malgré des performances sur la résolution de contraintes clausales (compétition SAT [2]) bien en deça des autres prouveurs (dues principalement au choix du langage utilisé et au support de contraintes génériques).

La seconde approche est utilisée par QMaxSat[12], l'un des meilleurs prouveurs MaxSat lors des récentes compétitions MaxSat (2010-2012), en traduisant les contraintes de cardinalité (il ne gère pas les poids) sous forme de CNF [3]. QMaxSat utilise les prouveurs SAT comme des boites noires : en 2012, la version utilisant Glucose 2 s'avère plus efficace que la version utilisant Minisat sur les instances dites « industrielles » avec contraintes dures (*Industrial Partial Max Sat*), alors que la version utilisant Minisat s'avère plus efficace sur les instances dites « fabriquées » avec contraintes dures (*Crafted Partial Max Sat*). Cela montre l'intérêt de l'approche boite noire : il suffit de choisir parmi les divers solveurs disponibles celui qui fonctionne le mieux sur le type de problème à résoudre.

Une autre approche permettant la réutilisation des solveurs SAT pour résoudre des problèmes d'optimisation existe, basée elle aussi sur la traduction de contraintes pseudo-booléennes en CNF : l'optimisation basée sur la détection de noyau incohérent (*unsat core guided MaxSat solvers*)[10, 18, 2]. Cette approche est spécifique à la résolution de problèmes MaxSat, mais permet de résoudre d'autres problèmes d'optimisation par traduction vers MaxSat.

Les approches basées sur la réutilisation de prouveurs SAT ne sont pas toujours les plus adaptées. Les approches de type *Branch & Bound* fonctionnent généralement très bien sur les instances MaxSat aléatoires ou fabriquées : akmaxsat[14] était le meilleur solveur MaxSat dans ces catégories en 2011 et 2012.

4 Optimisation par contraintes de bornes dynamiques

Afin de comprendre les enjeux de l'ajout de contraintes à la volée dans un prouveur CDCL, nous devons tout d'abord présenter brièvement et informelement le fonctionnement de ce type de solveur. Nous renvoyons le lecteur à [23] pour de plus amples informations sur ce sujet.

4.1 Fonctionnement des solveurs CDCL

L'algorithme 2 représente de manière abstraite un prouveur SAT de type « Conflict Driven Clause Learning » (CDCL). Les composants principaux sont la procédure de propagation unitaire (`propager`), la procédure d'analyse de conflits (`analyser`) et l'heuristique de choix de variable (`décider`). Lors de l'apparition d'un conflit (clause falsifiée), le mécanisme de retour arrière est nécessaire pour permettre au solveur de défaire le dernier choix de l'heuristique. En effet, pour chaque conflit rencontré, une nouvelle clause c, falsifiée dans le contexte courant, est retournée par la fonction `analyser` et ajoutée au problème. Plus précisément, cette nouvelle clause est construite de telle sorte qu'elle propage une valeur de vérité quand la dernière décision est annulée car elle ne contient qu'un seul littéral affecté au dernier niveau de décision (Unique Implication Point)[24]. Ainsi, l'apprentissage de la clause devenue unitaire va permettre à la procédure `propager`

1. `http://maxsat.ia.udl.cat:81/09/`
2. `http://www.satcompetition.org/2009/`
3. De nombreux travaux concernent la traduction des contraintes de cardinalité et des contraintes pseudo-booléennes en CNF (voir par exemple [3, 1] pour un aperçu des travaux récents).

de propager une nouvelle valeur de vérité. En pratique, on ne défait pas uniquement la dernière décision, mais toutes les décisions qui n'affectent pas la clause apprise (*backjump*). On peut voir cette action comme le fait de remettre le prouveur SAT dans un état compatible avec celui d'un prouveur qui aurait eu cette clause dès le départ : la clause unitaire est propagée dès qu'elle est détectée.

Algorithme 2 : Algorithme CDCL

entrée : Un ensemble de clauses, de contraintes de cardinalité et de contraintes pseudo-booléennes ϕ
sortie : un modèle de ϕ, ou UNSAT si le problème est insatisfiable.

```
1  répéter
2      conflit ← propager() ;
3      % invariant : les conflits sont traités dès qu'ils sont détectés ;
4      si conflit ≠ NIL alors
5          si pileDecisionsVide() alors
6              retourner UNSAT;
7          fin
8          clause ← analyser (conflit);
9          défaireChoixPourPropager (clause);
10         apprendre (clause);
11     sinon
12         si décider() = NIL alors
13             retourner SAT;
14         fin
15     fin
16 jusqu'à temps limite atteint;
17 retourner TIMEOUT;
```

4.2 Ajouter des contraintes à la volée

Lorsque l'on souhaite ajouter des contraintes de borne à la volée dans le solveur, on se trouve dans une situation différente de celle de l'apprentissage de clauses : la contrainte ajoutée est falsifiée, mais la décision ayant amené cette satisfaction n'est pas nécessairement la dernière à avoir été prise [4]. Il est donc nécessaire de dépiler toutes les propagations et décisions jusqu'au premier niveau où la contrainte est falsifiée de manière à ce que l'état du solveur soit cohérent avec l'approche CDCL : un conflit doit être traité dès qu'il est détecté. L'algorithme 3 représente un solveur de type CDCL capable de travailler avec différents type de contraintes, avec ajout de contraintes à la volée lorsqu'une solution est trouvée. On note ligne 11 une fonction `defaireJusqueFalsif` qui a pour but de dépiler les décisions jusqu'au niveau de celle provoquant la falsification de la contrainte. Cette fonction agit de manière spécifique pour chaque type de contraintes :

clause une clause est falsifiée ssi tous ses littéraux sont falsifiés, il suffit donc de dépiler toutes les décisions jusqu'au dernier niveau de décision de la clause. On se retrouve ici dans un cas similaire à un retour arrière lors de l'apprentissage d'une clause après analyse de conflit.

cardinalité une contrainte de cardinalité de n littéraux et de degré k est falsifiée ssi $n-k+1$ littéraux sont falsifiés. Il faut donc détecter les $n-k+1$ premières falsifications de littéraux.

pseudo-booléenne une contrainte pseudo-boléenne de n littéraux et de degré k est falsifiée ssi $\sum_{i=1}^{n} w_i \times l_i \equiv \sum w_i - \sum_{l_j=0} w_j < k$. Il faut donc détecter le premier niveau de décision pour lequel la somme des poids des littéraux falsifiés dépasse $\sum w_i - k$.

On peut remarquer que dans le cas des contraintes de cardinalité et des contraintes pseudo-booléennes, l'ordre dans lequel les littéraux sont falsifiés est important, ce qui n'est pas le cas pour les clauses.

Notons enfin que les solveurs CDCL modernes effacent les clauses apprises durant la recherche. Il est donc nécessaire que les contraintes de borne soient gérées spécifiquement, pour éviter qu'elles ne soient effacées. On notera de plus qu'il n'est pas nécessaire de garder plus d'une contrainte de borne, car ces contraintes sont de plus en fortes (c'est aussi le cas dans l'approche par boite noire [5]).

Si maintenir l'invariant que chaque conflit est analysé dès qu'il est détecté est obligatoire pour garantir le bon fonctionnement d'un prouveur CDCL lors de l'ajout de contraintes à la volée, nous allons aussi voir qu'en fonction du type de contrainte ajouté certaines vérifications doivent être faîtes afin de garantir l'efficacité du solveur.

4.3 Propager les contraintes ajoutées à la volée

Il est possible que la contrainte ajoutée puisse propager des valeurs de vérité lorsque l'on défait des décisions après l'analyse de conflits.

Considérons par exemple la contrainte $l_1 + l_2 + ... + l_n \leq 0$ produite comme contrainte de borne pour un modèle qui satisfait un unique littéral (l_j) de la fonction objectif. Après ajout de cette contrainte, tous les littéraux qu'elle contient doivent être propagés à faux au niveau de décision 0, puisque qu'un modèle doit dorénavant contenir $\neg l_1$, $\neg l_2$, ..., et $\neg l_n$. En revanche, si on considère uniquement la clause issue de l'analyse de conflits, nous ne pouvons propager que

4. Ce qui était le cas dans notre étude préliminaire d'énumération de modèles, car la clause contenait toutes les variables.

Algorithme 3 : Optimisation par ajout de contraintes de bornes dynamique

entrée : Un ensemble de clauses, de contraintes de cardinalité et de contraintes pseudo-booléennes ϕ, et une fonction d'objectif à minimiser f

sortie : un modèle de ϕ minimisant f, ou UNSAT si le problème est insatisfiable.

```
1  satisfiable ← FALSE;
2  dynConflit ← NIL;
3  répéter
4  |  conflit ← propager ();
5  |  si conflit = NIL alors
6  |  |  si décider() = NIL alors
7  |  |  |  dynConflit ← genContrainteBorne() ;
8  |  |  |  si dynConflit = NIL alors
9  |  |  |  |  retourner SAT;
10 |  |  |  fin
11 |  |  |  défaireJusqueFalsif (dynConflit);
12 |  |  |  satisfiable ← TRUE;
13 |  |  |  conflit ← dynConflit ;
14 |  |  fin
15 |  fin
16 |  si conflit ≠ NIL alors
17 |  |  si pileDecisionsVide() alors
18 |  |  |  si satisfiable alors
19 |  |  |  |  retourner OPT;
20 |  |  |  sinon
21 |  |  |  |  retourner UNSAT;
22 |  |  |  fin
23 |  |  fin
24 |  |  clause ← analyser (conflit);
25 |  |  défaireChoixPourPropager (clause);
26 |  |  si conflit = dynConflit alors
27 |  |  |  propager (dynConflit);
28 |  |  |  dynConflit ← NIL;
29 |  |  fin
30 |  |  apprendre (clause);
31 |  fin
32 jusqu'à temps limite atteint;
33 retourner TIMEOUT ;
```

le littéral $\neg l_j$, et non les littéraux l_i tels que $i \neq j$ ($i \in \{1, 2, ..., n\}$). Il est donc nécessaire de permettre à la nouvelle contrainte de propager des valeurs de vérité. C'est le but de la fonction `propager` ligne 27 dans l'algorithme 3.

Il existe de plus un problème spécifique aux contraintes pseudo-booléennes : celles-ci ont la particularité de pouvoir propager des valeurs à divers niveaux de décision, alors qu'une clause ne propage qu'une valeur à un unique niveau de décision, et qu'une contrainte de cardinalité peut propager plusieurs valeurs de vérité là aussi à un unique niveau de décision. L'ajout de contraintes pseudo-booléennes implique donc de remonter au premier niveau de propagation, qui peut être le niveau de propagation 0 si le poids d'un littéral est nécessaire à la satisfaction de la contrainte : $5x_1 + 3x_2 + x_3 \geq 6$ par exemple implique x_1 au niveau de décision 0. Cette contrainte peut être falsifiée soit en falsifiant x_1, soit en falsifiant x_2 et x_3. Dans le premier cas, la propagation se fera effectivement au niveau de décision 0. Dans le second, x_1 ne sera pas propagé. Nous n'avons pas trouvé de solution élégante pour régler ce problème à l'heure actuelle.

Dans le cas où la contrainte est une simple clause, une seule valeur de vérité peut être propagée (le seul littéral non falsifié), et cette valeur sera propagée par la clause issue de l'analyse de conflits. En effet, si une clause apprise propage une valeur de vérité quand les décisions sont dépilées, c'est qu'il s'agit d'une clause contenant un seul littéral au niveau de décision courant (*First UIP*). Donc la procédure d'analyse de conflits va retourner exactement cette clause, ou une clause simplifiée [26] dont le littéral propagé sera exactement celui qui serait propagé par la clause ajoutée à la volée. Il n'y a donc pas de propagation supplémentaire possible lors de la génération de clauses conflictuelles à la volée.

5 Expérimentations et résultats

Nous avons intégré notre algorithme d'optimisation par contrainte de bornes dynamiques à la plateforme Sat4j, dans les solveurs d'optimisation pseudo-booléenne et MaxSat. Il est à noter qu'aucun de ces solveurs n'effectue de pré-traitement. Nos expérimentations ont été menées sur des machines équipées de processeurs Intel XEON 3,0 GHz associées à 2 Go de mémoire.

En plus des résultats de performance globale, nous souhaitons vérifier les hypothèses concernant les cas ou une approche par contraintes de borne dynamique peut s'avérer plus intéressante que lors d'une approche boite noire.

– La preuve unsat finale devrait être plus courte,

puisque l'on bénéficie de l'analyse de conflits
– Mais dans certains cas, on risque d'énumérer énormément de solutions

Voyons si ces hypothèses se vérifient en pratique.

5.1 Optimisation pseudo-booléenne

Nous avons utilisé des benchmarks issus de la compétition PB10 [5] afin de comparer les résultats de nos deux approches, en laissant un temps de trente minutes à nos solveurs pour trouver une solution optimale (soit le même temps limite que celui de la compétition). Le tableau 5 présente le nombre d'instances résolues par nos deux algorithmes (le nombre d'instances prouvées insatisfiables est indiqué entre parenthèses).

catégorie	#inst.	b. noire	b. grise
BIGINT-LIN	532	**125** (57)	115 (57)
SMALLINT-LIN	699	**270** (33)	266 (33)
SMALLINT-NLC	409	273 (0)	**275** (0)

TABLE 5 – Instances PB10 résolues (et UNSAT)

Concernant ces benchmarks, on remarque un léger avantage à l'utilisation de l'algorithme utilisant le solveur SAT en tant que boîte noire. Cependant, ces chiffres sont à nuancer légèrement car ces résultats globaux cachent le fait que chaque approche est capable de résoudre des problèmes que l'autre approche ne peut pas résoudre : 12 instances sont uniquement résolues par l'approche externe, et 6 instances sont uniquement résolues par l'approche interne. Le cas pathologique de l'approche à base de contraintes de bornes dynamiques est d'énumérer un nombre de solutions beaucoup plus important que l'approche boite noire, ce qui fait perdre du temps, mais aussi beaucoup de mémoire. C'est par exemple le cas pour l'instance `normalized-mps-v2-20-10-glass4.opb`, pour laquelle le solveur « boîte noire » trouve la solution optimale au bout de 50 secondes en ayant énuméré 16 solutions intermédiaires, alors que le solveur utilisant l'autre approche meurt car il nécessite trop de mémoire, après avoir calculé 8715 solutions intermédiaires. Ce cas pathologique n'est présent que rarement dans nos expérimentations, bien que cette dernière approche a tendance à énumérer plus de solutions avant de trouver la valeur optimale, comme le montre la figure 1. En termes quantitatifs, l'algorithme d'optimisation classique énumère en moyenne 5 solutions quand l'algorithme utilisant des contraintes de borne dynamiques énumère en moyenne 17 solutions. Le nombre médian de solutions énumérées est lui de 2 dans les deux cas, ce qui montre que la différence entre les nombres de solutions intermédiaires parcourue par nos algorithmes est d'autant plus importante que le nombre de solutions énumérées est important.

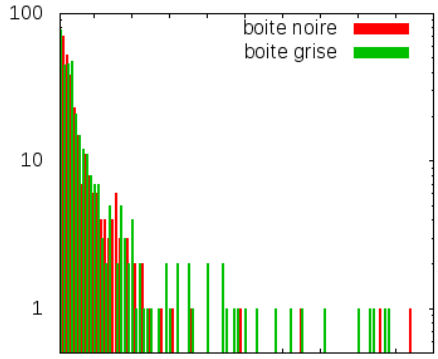


FIGURE 1 – Nombre de solutions énumérées pour PB (graphe partiel jusqu'à 100 solutions) — nombre de solutions énumérées par instances en abscisse, nombre d'instances en ordonnée

La figure 2 représente la distribution des temps des solveurs sous forme de « cactus », c'est à dire en triant les temps par ordre croissant. Les prouveurs peuvent avoir des temps différents sur des instances individuelles, mais globalement, la distribution des temps est quasi-identique dans les deux cas. La figure 3 représente de la même manière la distribution du temps nécessaire à la preuve de l'optimalité, c'est à dire à la résolution du dernier problème de décision, non satisfiable. La encore, si les temps individuels par instance varient, les distributions de temps ne sont pas dissociables.

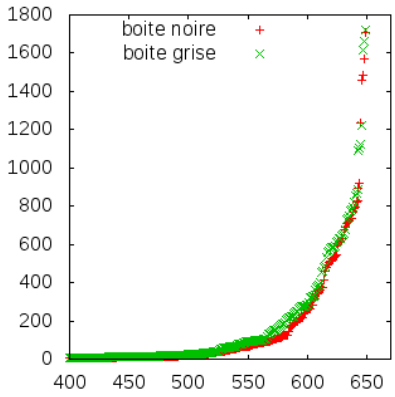


FIGURE 2 – Temps de résolution des instances PB

5. `http://www.cril.univ-artois.fr/PB10/`

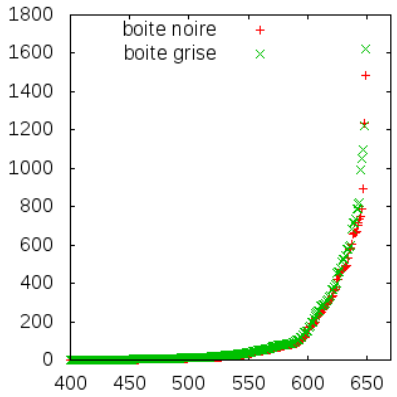


FIGURE 3 – Temps nécessaire à la preuve de l'optimalité de la dernière solution énumérée — cas de PB

5.2 Optimisation MaxSat

Nous avons comparé ces deux approches dans le cadre de l'optimisation MaxSat sur les instances de la compétition MaxSat10, avec un temps limite de vingt minutes (soit le même que lors de la compétition). Les résultats sont présentés dans le tableau 6.

catégorie	#inst.	b. noire	b. grise
ms_crafted	167	**2**	**2**
ms_industrial	77	**8**	5
ms_random	300	**0**	**0**
pms_crafted	385	**190**	187
pms_industrial	497	**270**	258
pms_random	240	**26**	**26**
wms_crafted	149	**43**	**43**
wms_random	200	**16**	**16**
wpms_crafted	378	**146**	142
wpms_industrial	132	**36**	35
wpms_random	150	**29**	**29**

TABLE 6 – Instances MaxSat10 résolues

Ici encore, les résultats sont très proches, avec un très léger avantage pour l'approches « boîte noire ». Encore une fois, cette légère différence est en partie imputable à une énumération d'un trop grand nombre de solutions concernant l'autre algorithme pour certaines instances. Dans ces jeux de test, 31 instances sont résolues par l'approche externe et non par l'interne, et 8 autres sont dans le cas inverse. De la même manière que l'algorithme d'optimisation par contraintes de bornes dynamiques énumère de manière générale plus de solutions dans le cadre de l'optimisation pseudo-booléenne, on observe un comportement similaire pour MaxSat (voir la figure 4). En termes de statistiques, l'approche « boîte noire » énumère en moyenne 7 solutions quand l'approche « boîte grise » en énumère 13. Encore une fois, une étude du nombre médian de modèles découverts lors de la recherche (4 pour la première, 5 pour la deuxième) nous montre que plus le nombre de solutions énumérées est important, plus la différence entre les deux approches sur ce point l'est aussi.

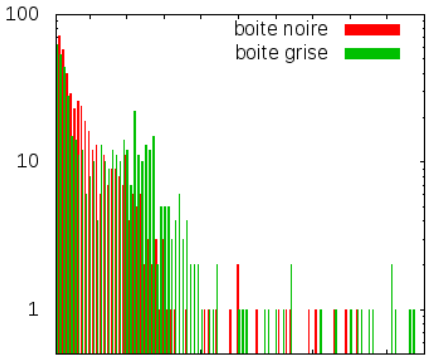


FIGURE 4 – Nombre de solutions énumérées pour MaxSat (graphe partiel jusqu'à 100 solutions) — nombre de solutions énumérées par instances en abscisse, nombre d'instances en ordonnée

Une fois encore, comme le montrent les figures 5 et 6, on ne peut départager de manière globale nos deux approches ni sur la distribution des temps de calcul des solutions optimales, ni sur la distribution des temps de la preuve d'optimalité.

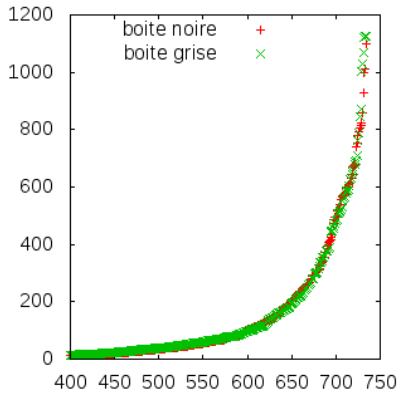


FIGURE 5 – Temps de résolution des instances MaxSat

6 Analyse des résultats

A la lumière de ces expérimentations, l'approche par apprentissage de bornes à la volée ne s'avère pas plus efficace que l'approche incrémentale ou boite noire.

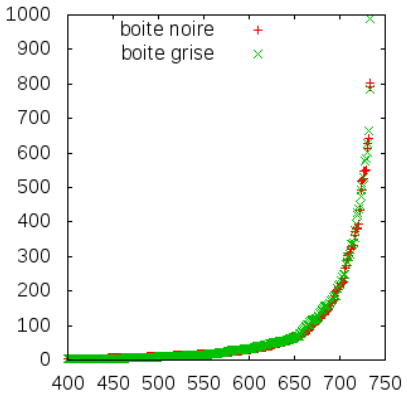


FIGURE 6 – Temps nécessaire à la preuve de l'optimalité de la dernière solution énumérée — cas de MaxSat

Ces résultats sont décevants compte tenu des résultats initiaux obtenus sur l'énumération de solutions. Il nous semble important de mettre en avant quelques faits qui peuvent expliquer ces résultats.

Tout d'abord, dans nos deux problèmes d'optimisation, les prouveurs trouvent une solution très proche de l'optimale dans plus de la moitié des cas (nombre de solutions intermédaires médian variant de 2 à 5 selon les problèmes et les approches). Il faut mettre cela en balance avec les milliers de solutions générées dans le cadre de l'énumération.

Les cas (peu nombreux) pour lesquels une solution n'est pas trouvée sont souvent liés à une grande différence en terme de nombre de solutions/contraintes générées. En énumération de solutions, les deux approches doivent générer exactement le même nombre de solutions. Il n'est cependant pas clair à l'heure actuelle si cela est dû à l'approche d'ajout de contraintes à la volée, qui force le solveur à continuer de chercher des solutions dans l'espace de recherche courant ou s'il s'agit d'illustrations de la faiblesse de propagation mentionnée dans la section précédente.

Enfin, nous avons essayé différentes heuristiques, stratégies de redémarrages, stratégies de minimisation de clauses : les résultats individuels des solveurs changent légèrement, mais les tendances restent identiques.

7 Conclusions et perspectives

Nous avons présenté dans le papier les résultats de notre étude sur l'utilisation d'une communication privilégiée avec un prouveur SAT générique permettant l'ajout de contraintes à la volée pour résoudre des problèmes d'optimisation en variables booléennes. Nous avons montré que si l'ajout dynamique de clauses se fait sans difficulté, l'ajout de contraintes de cardinalité ou de contraintes pseudo-booléennes nécessite quelques précautions. Les résultats expérimentaux sont assez décevants : on ne remarque pas de différence flagrante d'efficacité, comme on avait pu le constater sur le cas de l'énumération de solutions qui avait motivé cette étude. Ces résultats peuvent s'expliquer par la nature des benchmarks utilisés : nombre d'entre eux sont résolus après production de quelques solutions intermédiaires. Les solveurs SAT utilisant des stratégies de redémarrage rapide, et bénéficiant d'une interface incrémentale, il semble que le gain d'une analyse de conflit sur les contraintes ajoutées à la volée soit quasi nul. Nous avons identifié une faiblesse dans notre approche lors de l'apprentissage de contraintes pseudo-booléennes, qui pourrait manquer certaines propagations. Nous allons vérifier si cela se produit effectivement en pratique et chercher un moyen d'éviter ce problème.

Références

[1] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at bdds for pseudo-boolean constraints. *J. Artif. Intell. Res. (JAIR)*, 45 :443–480, 2012.

[2] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. Sat-based maxsat algorithms. *Artif. Intell.*, 196 :77–105, 2013.

[3] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks : a theoretical and empirical study. *Constraints*, 16(2) :195–221, 2011.

[4] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Biere et al. [7], pages 825–885.

[5] Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3) :59–6, 2010.

[6] Armin Biere. Bounded model checking. In Biere et al. [7], pages 457–481.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[8] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[9] Niklas Eén and Niklas Sörensson. Temporal induction by incremental sat solving. *Electr. Notes Theor. Comput. Sci.*, 89(4) :543–560, 2003.

[10] Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. In Armin Biere and Carla P. Gomes, editors, *SAT*, volume 4121 of *Lecture Notes in Computer Science*, pages 252–265. Springer, 2006.

[11] Ian P. Gent. Arc consistency in sat. In Frank van Harmelen, editor, *ECAI*, pages 121–125. IOS Press, 2002.

[12] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. Qmaxsat : A partial max-sat solver. *JSAT*, 8(1/2) :95–100, 2012.

[13] Daniel Kroening. Software verification. In Biere et al. [7], pages 505–532.

[14] Adrian Kuegel. Improved exact solver for the weighted max-sat problem. In *Workshop Pragmatics of SAT*, 2010.

[15] Chu Min Li and Felip Manyà. Maxsat, hard and soft constraints. In Biere et al. [7], pages 613–631.

[16] Vasco M. Manquinho and João P. Marques Silva. On solving boolean optimization with satisfiability-based algorithms. In *AMAI*, 2000.

[17] Sylvain Merchez, Christophe Lecoutre, and Frédéric Boussemart. Abscon : A prototype to solve csps with abstraction. In Toby Walsh, editor, *CP*, volume 2239 of *Lecture Notes in Computer Science*, pages 730–744. Springer, 2001.

[18] António Morgado, Federico Heras, and João Marques-Silva. Improvements to core-guided binary search for maxsat. In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 284–297. Springer, 2012.

[19] António Morgado and João P. Marques Silva. Good learning and implicit model enumeration. In *ICTAI*, pages 131–136. IEEE Computer Society, 2005.

[20] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3) :357–391, 2009.

[21] Jussi Rintanen. Planning and sat. In Biere et al. [7], pages 483–504.

[22] Olivier Roussel and Vasco M. Manquinho. Pseudo-boolean and cardinality constraints. In Biere et al. [7], pages 695–733.

[23] João P. Marques Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning sat solvers. In Biere et al. [7], pages 131–153.

[24] João P. Marques Silva and Karem A. Sakallah. Grasp : A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5) :506–521, 1999.

[25] Josh Singer, Ian P. Gent, and Alan Smaill. Backbone fragility and the local search cost peak. *J. Artif. Intell. Res. (JAIR)*, 12 :235–270, 2000.

[26] Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009.

[27] Toby Walsh. Sat v csp. In Rina Dechter, editor, *CP*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.

Résolution du problème WCSP par extraction de noyaux insatisfiables minimaux

Christophe Lecoutre **Nicolas Paris** **Olivier Roussel** **Sébastien Tabary**

CRIL - CNRS UMR 8188,
Université Lille Nord de France, Artois,
rue de l'universite,
62307 Lens cedex, France
{lecoutre,paris,roussel,tabary}@cril.fr

Résumé

Les méthodes usuelles de résolution pour le cadre WCSP utilisent des techniques de transferts de coûts couplées à une recherche arborescente de type séparation et évaluation. Dans cet article, nous nous intéressons à une approche intégrant l'extraction et le relâchement de noyaux insatisfiables minimaux afin de résoudre ce problème, soit de manière incomplète (gloutonne), soit de manière complète.

1 Introduction

Le problème de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem) consiste à déterminer si un réseau de contraintes donné (CN pour Constraint Network), ou instance CSP, est satisfiable ou non. Il s'agit d'un problème de décision, où on recherche une instantiation complète des variables du CN satisfaisant toutes les contraintes de celui-ci. Lorsque des préférences doivent être représentées, il est possible d'exprimer celles-ci à l'aide de contraintes souples. Dans le cadre WCSP (Weighted CSP), chaque contrainte souple associe en fait un coût aux différentes instanciations des variables sur lesquelles elle porte, permettant ainsi de modéliser différents degrés de violation. L'objectif est de trouver une instanciation qui minimise le coût cumulé des différentes contraintes souples.

La plupart des méthodes actuelles de résolution de réseaux de contraintes pondérées (WCN pour Weighted Constraint Network), ou instances WCSP, se fondent sur une recherche par séparation et évaluation (Branch and Bound) et utilisent diverses cohérences locales pour estimer le coût minimum du réseau simplifié au cours de la recherche. Ces cohérences locales (AC* [10, 11], FDAC [3], EDAC [6], VAC [4], OSAC [5]) sont fondées sur des méthodes de transferts de coût qui préservent l'équivalence. Parce qu'ils doivent prendre en compte plus d'informations que dans le cadre CSP, les algorithmes établissant les cohérences locales souples sont souvent plus complexes que leurs équivalents CSP.

Dans cet article, nous proposons une approche originale pour le cadre WCSP, consistant pour un WCN donné à résoudre une séquence de CNs générés itérativement à partir de ce WCN, et permettant ainsi la réutilisation de solveurs de contraintes éprouvés. Ce type d'approche a déjà été utilisée avec succès pour établir la cohérence souple VAC [4] : l'établissement de la cohérence d'arc sur un CN généré à partir d'un WCN a pour objectif (contrairement à notre approche) l'identification de transferts de coûts.

Pour résoudre un WCN, nous durcissons les contraintes en ne retenant que les tuples ayant un coût donné afin d'obtenir des contraintes dures (CNs). Le principe de la méthode est alors d'énumérer ces CNs par ordre croissant de coût (au niveau du WCN). Lorsqu'un CN est insatisfiable, un noyau minimalement inconsistant (MUC pour Minimal Unsatisfiable Core) est identifié pour déterminer les contraintes souples pour lesquelles il faut accepter un coût plus important afin d'obtenir une solution. Cette approche est déclinée en un algorithme complet et un algorithme glouton (donc incomplet). Notons que cette approche a été utilisée avec succès pour le cadre SAT [1, 8, 13].

Cet article est construit comme suit. Après les définitions d'usage, nous présentons le fonctionnement gé-

néral des algorithmes et les structures de données utilisées. Nous détaillons alors la version complète ainsi que la version gloutonne de notre approche et terminons par la présentation de quelques résultats expérimentaux.

2 Préliminaires

2.1 Généralités

Un *réseau de contraintes* (CN) P est constitué d'un ensemble fini de n variables noté $vars(P)$ et d'un ensemble fini de e contraintes, noté $cons(P)$. Chaque variable x a un domaine associé noté $dom(x)$, qui contient l'ensemble fini des valeurs pouvant être affectées à x ; le domaine initial de x est noté $dom^{init}(x)$. Chaque contrainte dure c porte sur un ensemble ordonné de variables, noté $scp(c)$ et appelé *portée* de c. Elle est définie par une relation contenant l'ensemble des tuples autorisés pour les variables de $scp(c)$. L'arité d'une contrainte c est le cardinal de $scp(c)$. Une *instanciation* I d'un ensemble $X = \{x_1, \ldots, x_p\}$ de variables est un ensemble $\{(x_1, a_1), \ldots, (x_p, a_p)\}$ tel que $\forall i \in 1..p, a_i \in dom^{init}(x_i)$. I est *valide* sur P ssi $\forall (x, a) \in I, a \in dom(x)$. Une solution est une instanciation complète de $vars(P)$ (i.e., l'affectation d'une valeur à chaque variable) qui satisfait toutes les contraintes. P est satisfiable ssi il admet au moins une solution. Pour plus d'information sur les réseaux de contraintes, voir [7, 12, 15].

Un noyau insatisfiable de P est un sous-ensemble insatisfiable de contraintes de P. Un noyau est un MUC (Minimal Unsatisfiable Core) si et seulement si tout sous-ensemble strict du MUC est satisfiable. Une approche composée de deux étapes pour extraire les MUCs de réseaux de contraintes est présentée dans [9]. Montrée comme la plus efficace parmi celles présentées dans l'article précédemment cité, nous avons opté dans notre implantation pour la version *dichotomique*, nommée *dcMUC*.

Un *réseau de contraintes pondérées* (WCN) W est constitué d'un ensemble fini de n variables noté $vars(W)$, un ensemble fini de e contraintes souples, noté $cons(W)$, et d'une valeur k qui est soit un entier naturel strictement positif soit $+\infty$. Chaque contrainte souple $w \in cons(W)$ possède une portée $scp(w)$ et est définie par une fonction de coût de $l(scp(w))$ vers $\{0, \ldots, k\}$, où $l(scp(w))$ est le produit cartésien des domaines des variables sur lesquelles porte w ; pour toute instanciation $I \in l(scp(w))$, on notera le coût de I dans w par $w(I)$. Une instanciation de coût k (noté aussi $\top$) est interdite. Autrement, elle est autorisée avec le coût correspondant (0, noté aussi $\bot$, est complètement satisfaisant). Les coûts sont combinés par la somme bornée $\oplus$ définie par :

$$\forall a, b \in \{0, \ldots, k\}, a \oplus b = min(k, a + b)$$

L'objectif du problème de satisfaction de contraintes pondéré (WCSP) est, pour un WCN donné, de trouver une instanciation complète de coût minimal. Pour plus d'information sur les contraintes pondérées, voir [2, 14].

Différentes variantes de la cohérence d'arc souple pour le cadre WCSP ont étés proposées durant ces dix dernières années. Il s'agit de AC* [10, 11], la cohérence d'arc directionnelle complète (FDAC) [3], la cohérence d'arc directionnelle existentielle (EDAC) [6], la cohérence d'arc virtuelle (VAC) [4] et la cohérence d'arc souple optimale (OSAC) [5]. Tous les algorithmes proposés pour atteindre ces différents niveaux de cohérence utilisent des opérations de transfert de coûts (ou transformations préservant l'équivalence) telles que la projection unaire, la projection et l'extension.

2.2 Strates et fronts

Dans ce papier, nous considérons des contraintes souples données en extension. Il s'agit donc de contraintes tables souples, où certains tuples sont listés explicitement, accompagnés de leur coût, tandis qu'un coût par défaut indique le coût des tuples implicites (non représentés). Nous introduisons ci-dessous les notions de strate et front.

Les tuples d'une contrainte souple ayant le même coût peuvent être regroupés dans un même sous-ensemble S nommé *strate*. Le coût des tuples d'une strate S est donné par $cost(S)$. Une strate particulière correspond au coût par défaut : cette strate ne contient aucun tuple explicite, mais représente implicitement tous les tuples qui n'apparaissent pas explicitement dans une strate. Le nombre de strates d'une contrainte w sera désigné par $nbLevels(w)$. Au sein d'une contrainte, nous considérons les strates ordonnées par coût strictement croissant. Par souci de simplicité, nous utiliserons des indices (de 0 à $nbLevels(w) - 1$) pour identifier les strates d'une contrainte, et quand le contexte est suffisamment clair confondrons les indices avec les strates qu'ils représentent.

Un *front* est une fonction qui à chaque contrainte d'un WCN associe l'une de ses strates. Ce front représente une frontière entre des strates qui seront autorisées à un instant donné de notre approche, et celles qui seront interdites. Nous utiliserons une notation de tableau pour les fronts. Ainsi, $f[w]$ représentera la strate associée à la contrainte w dans un front f. $cost(f)$ est la fonction qui, à un front f, associe la somme des coûts des strates désignées par ce front :

$$cost(f) = \sum_{w \in cons(W)} cost(f[w]).$$

Un front f' est le successeur direct d'un front f s'il existe une contrainte w_j telle que $f'[w_j] = f[w_j] + 1$ et $\forall i \neq j, f'[w_i] = f[w_i]$. On choisit de noter $f \rightarrow f'$ si f' est un successeur direct de f et $f \rightarrow_* f'$ s'il existe une suite de relations $f \rightarrow f_1, f_1 \rightarrow f_2, \ldots f_n \rightarrow f'$. Comme pour une contrainte donnée, les strates ont des coûts strictement croissants, on en déduit que :

$f \rightarrow_* f' \Rightarrow cost(f) < cost(f')$.

On notera que les fronts avec la relation successeur direct ont une structure de treillis, où le plus petit front (noté $\bot$) désigne pour toute contrainte sa strate de plus faible coût et le plus grand front (noté $\top$) désigne pour toute contrainte sa strate de plus grand coût.

3 Fonctionnement général

La figure 1 décrit le fonctionnement général de l'approche que nous proposons dans cet article pour la résolution de réseaux de contraintes pondérées. Pour commencer, un premier réseau de contraintes P est construit (via l'appel à la fonction `toCN`) à partir du réseau de contraintes pondérées W à résoudre : pour chaque contrainte de W, les tuples apparaissant dans la strate de coût minimum sont considérés comme autorisés par P tandis que tous les autres tuples sont considérés comme interdits. Cette méthode de construction est similaire à celle proposée dans [4]. Le réseau de contraintes P ainsi construit est alors résolu par un solveur de contraintes (via l'appel à la fonction `solveCN`). Si P n'est pas satisfiable, l'algorithme extrait un MUC (via l'appel à la fonction `extractMUC`) à partir de P.

Le MUC extrait de cette façon peut être utilisé soit dans une version gloutonne, soit dans une version complète.

Dans la version gloutonne, certaines contraintes du MUC vont être successivement relâchées jusqu'à restaurer la satisfiabilité (via l'appel à la fonction `relax`). Plus précisément un sous-ensemble des strates de certaines contraintes du MUC vont basculer du statut "interdit" au statut "autorisé". Lorsque la satisfiabilité du MUC est restaurée, les relaxations sont retranscrites dans le réseau de contraintes initial et le processus se répète jusqu'à établir la satisfiabilité globale du CN. Cela signifie donc que les MUCs sont identifiés et traités de manière itérative par la méthode jusqu'à l'obtention d'un CN satisfiable. Dans ce cas, une solution peut être retournée.

Dans la version complète, une fois le MUC identifié, on insère dans une file de priorité une représentation d'un ensemble de réseaux de contraintes à résoudre correspondant aux différentes manières de relâcher le MUC précédemment identifié. Les CNs seront ainsi résolus successivement jusqu'à l'identification d'un réseau de contraintes satisfiable. Dans ce cas, une solution optimale est retournée.

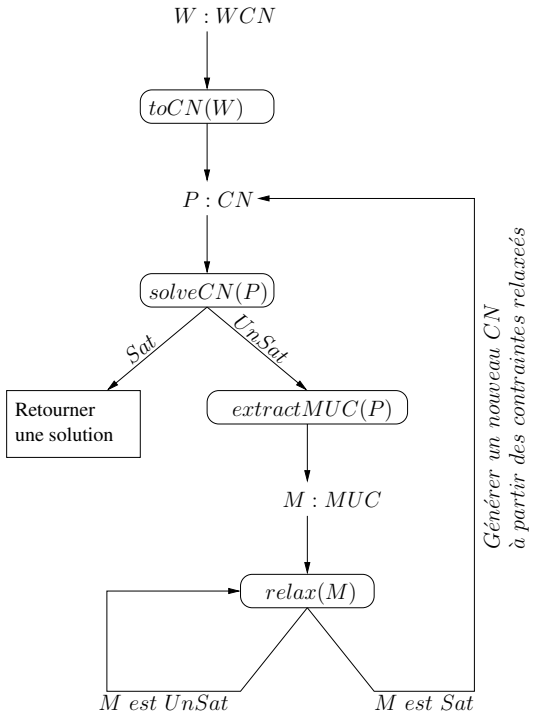


FIGURE 1 – Principe de fonctionnement du Relâchement itératif de MUC.

Il est important de noter que le fonctionnement que nous avons décrit ci-dessus peut être généralisé afin de prendre en compte tout type de contraintes : les contraintes dures du cadre CSP, les contraintes violables du cadre Weighted Max-CSP et les contraintes souples du cadre WCSP. Il suffit en effet pour chaque type de contraintes de spécifier la manière d'effectuer la traduction lorsque la fonction `toCN` est appelée. Pour une contrainte dure, aucune transformation n'est nécessaire (une seule strate de coût 0 est gérée), et pour une contrainte violable, deux strates sont gérées : la strate de coût 0 correspondant à la satisfaction de la contrainte et la strate de coût égal au poids associé à la contrainte lorque celle-ci n'est pas satisfaite. En d'autres termes, la méthode que nous présentons permet une intégration simple et naturelle de ces différents cadres.

4 Structures de données et fonctions

Le tableau $front$ est utilisé par l'opération `toCN` qui construit un CN à partir d'un WCN et des strates sélectionnées. En fait, il existe deux versions de cette opération. La première version notée $\texttt{toCN}_{=}(W, front)$

construit un CN en ne retenant comme tuples autorisés pour une contrainte w du WCN W que les tuples de la strate $front[w]$; cette strate est dite autorisée. La seconde version notée $\texttt{toCN}_{\leq}(W, front)$ construit un CN en ne retenant comme tuples autorisés pour une contrainte w du WCN W que les tuples des strates d'indice inférieur ou égal à $front[w]$ (donc de coût inférieur ou égal) ; ces strates sont dites autorisées. Autrement dit, pour toute contrainte dure c construite à partir d'une contrainte souple w de W, c n'autorise que les tuples dont le coût est égal (respectivement, inférieur ou égal) au coût de la strate $front[w]$.

En pratique, il y a deux manières de générer une contrainte dure à partir d'une contrainte souple. D'une part, quand les (ou la) strates autorisées ne contiennent pas la strate correspondant au coût par défaut, `toCN` se contente de créer une contrainte dure positive qui liste les tuples de ces strates autorisées comme étant des tuples autorisés. D'autre part, quand les (ou la) strates autorisées incluent la strate correspondant au coût par défaut, `toCN` crée une contrainte dure négative qui interdit tous les tuples des strates non autorisées, et ceci afin d'éviter de devoir énumérer tous les tuples ayant le coût par défaut.

Nous considérons maintenant chaque contrainte souple w comme un objet que nous décrivons. Tout d'abord, $w.defaultCost$ désigne le coût par défaut associé aux tuples implicites de la contrainte. Ensuite, les strates de la contrainte w sont stockées dans le tableau $w.levels$. Pour une strate d'indice i donnée, on accède d'une part au coût des tuples par $w.levels[i].cost$ et d'autre part à la liste des tuples par $w.levels[i].tuples$. Ce tableau de strates est trié par ordre de coûts strictement croissants. Le nombre de strates pour w est obtenu simplement par l'expression $nbLevels(w)$. Pour finir, le tableau $front$, déjà évoqué précédemment, fait correspondre à chaque contrainte w l'indice de la strate $front[w]$ qui, selon l'approche employée, sera soit la seule autorisée, soit la plus grande strate autorisée. Ces structures sont notamment utilisées dans les fonctions `toCN` et `cost`.

La figure 2 décrit pour deux contraintes w_0 et w_1 les structures de données présentées ci-dessus. La contrainte w_0 possède un coût par défaut $w_0.defaultCost$. Le tableau $w_0.levels$ permet d'accéder aux différentes strates dans lesquelles sont stockés les tuples de même coût. La contrainte w_0 possède un ensemble de strates numérotées à partir de 0. Les strates étant triées par ordre de coût strictement croissant, la strate $w_0.levels[0]$ contient l'ensemble des tuples de coût minimum (dont le coût est accessible par $w_0.levels[0].cost$). La liste des tuples de la deuxième strate, à savoir (τ_2, τ_3), est accessible par $w_0.levels[1].tuples$. Dans cet exemple $front[w_0]$ est

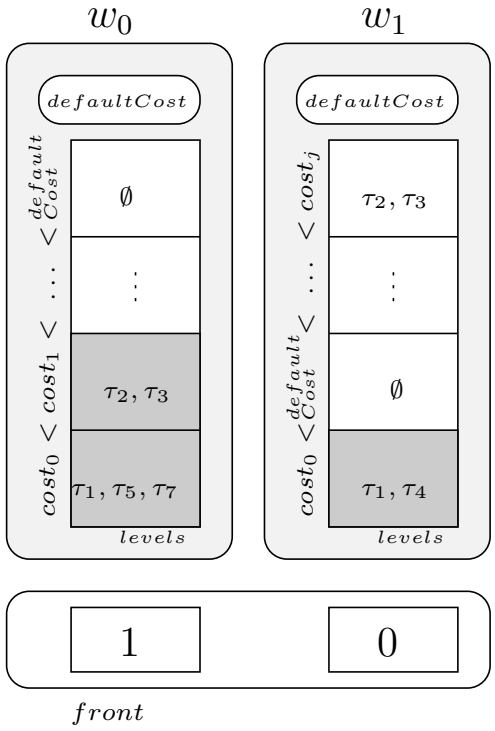


FIGURE 2 – Description des structures de données.

égal à 1, ce qui signifie que, en fonction de l'opération *toCN* utilisée, 1) soit sont autorisées les strates dont l'indice est ≤ 1 (c'est à dire l'ensemble des tuples de la première et de la seconde strate soit τ_1, τ_5, τ_7, τ_2 et τ_3), 2) soit sont autorisées les strates dont l'indice est égal à 1 (c'est à dire l'ensemble des tuples de la seconde strate soit τ_2 et τ_3). Pour la contrainte w_1, nous avons fait apparaître le coût par défaut au niveau de la deuxième strate (indice 1). Rappelons que cette strate est ajoutée de manière à conserver la structure *levels* triée par coût strictement croissant, afin de simplifier les traitements algorithmiques. Comme déjà indiqué, les tuples implicites associés à ce type de strates ne sont pas représentés, et ceci est symbolisé dans la figure par le symbole $\emptyset$.

Nous terminons cette section en décrivant les fonctions utilisées par nos algorithmes, mais non détaillées dans le papier. La fonction `solveCN` résout un CN donné en paramètre et retourne soit une solution, soit $\perp$ si le CN est insatisfiable. La fonction `extractMUC` retourne un MUC (sous-ensemble de contraintes minimalement insatisfiable) à partir d'un CN P passé en paramètre (celui-ci doit être insatisfiable). La fonction $\texttt{cons}(W, M)$ retourne les contraintes du WCN W qui correspondent à celles du MUC M (toute contrainte

dure est associée à une contrainte souple au moment de la génération). La fonction `restrict`(W, M) retourne un WCN qui contient uniquement les contraintes du WCN W qui correspondent à des contraintes présentes dans le MUC M.

5 Algorithmes

5.1 Version complète préliminaire

Pour faciliter la compréhension de notre approche, nous présentons un premier algorithme complet qui n'exploite pas les MUC et n'est donc pas efficace. La fonction completeSearchNoMUC commence avec un premier front qui ne retient que la strate 0 de chaque contrainte (ligne 1 à 3). Ensuite, tant qu'il reste un front à étudier, on extrait d'une file de priorité celui de plus faible coût. Notons que le coût d'un front peut être facilement calculé en sommant les coûts associés aux strates identifiées au niveau de chaque contrainte souple. On peut alors construire avec l'opérateur $\texttt{toCN}_=$ un CN que l'on cherche à résoudre avec `solveCN`. Si une solution est obtenue, elle est optimale, et l'algorithme peut s'arrêter. En effet, l'algorithme garantit une énumération des fronts par ordre de coût croissant (voir proposition 1) et assure que tous les fronts précédents étaient insatisfiables. Si le CN s'est avéré insatisfiable, on énumère les successeurs directs du front courant (ligne 10 à 15) et on les insère dans la file de priorité. Bien évidemment, il faut vérifier que l'on ne dépasse pas la strate maximale d'une contrainte donnée. Par ailleurs, les fronts qui auraient un coût égal à k doivent être ignorés.

Function completeSearchNoMUC(W : WCN)

```
1  foreach w ∈ cons(W) do
2      front[w] ← 0
3  Q ← {front}
4  while Q ≠ ∅ do
5      f ← pick and delete least cost front in Q
6      P ← toCN_=(W, f)
7      sol ← solveCN(P)
8      if sol ≠ ⊥ then
9          return
10     else
11         foreach w ∈ cons(W) do
12             if f[w] ≠ nbLevels(w) − 1 then
13                 f' ← f
14                 f'[w] ← f'[w] + 1
15                 if cost(f') ≠ k then
16                     Q ← Q ∪ f'
```

La file de priorité Q utilisée dans cet algorithme est une version particulière car elle doit non seulement garantir que les fronts sont extraits par ordre croissant de coût, mais également garantir qu'un front donné n'est pas inséré deux fois dans la file. Par exemple, si l'on considère un réseau de 2 contraintes ayant chacune au moins 2 strates, le premier front sera le tableau $< 0, 0 >$. Lors de la première itération, on insérera dans la file les voisins $< 1, 0 >$ et $< 0, 1 >$. Lorsque ces fronts seront à leur tour extraits de la file, ils généreront tous deux le même voisin $< 1, 1 >$ qui ne doit être inséré qu'une seule fois dans la file pour éviter des calculs redondants. Cette file particulière s'obtient en modifiant très légèrement une implantation classique d'une file de priorité par tas binomial.

On peut vérifier aisément que l'algorithme completeSearchNoMUC énumère tous les fronts possibles quand `solveCN` ne trouve jamais de solution. En effet, en faisant abstraction des coûts, il s'agit d'une exploration en largeur d'abord d'un arbre énumérant les fronts. La proposition 1 garantit en plus que cet algorithme ne peut pas boucler et que les fronts sont énumérés par ordre de coût croissant.

Proposition 1 *Les deux propriétés suivantes sont vérifiées par l'algorithme completeSearchNoMUC :*

A) les fronts sont énumérés par ordre croissant de coût ;

B) une fois qu'un front f est extrait de la file de priorité, il ne peut plus jamais y être inséré.

Preuve Soit f_1 un premier front extrait de la file et f_2, un front extrait de la file après l'extraction de f_1.

A) Deux cas sont possibles. Soit (A.1) f_2 se trouvait déjà dans la file quand f_1 en a été extrait, soit (A.2) f_2 a été placé dans la file après l'extraction de f_1. Dans le cas (A.1), la file de priorité garantit que $cost(f_1) \leq cost(f_2)$. Dans le cas (A.2), f_2 est issu d'un front f tel que $f \rightarrow_* f_2$ et tel que soit $f = f_1$, soit f était présent dans la file quand f_1 a été extrait. Dans les deux cas, $cost(f_1) \leq cost(f)$. Comme $f \rightarrow_* f_2 \Rightarrow cost(f) < cost(f_2)$, on en conclut que $cost(f_1) < cost(f_2)$. On obtient donc dans tous les cas que $cost(f_1) \leq cost(f_2)$.

B) Supposons que $f_1 = f_2$. Comme la file garantit qu'elle ne contient jamais deux fronts identiques à un instant donné, on en conclut que l'on se situe nécessairement dans le cas (A.2) précédent. Or, on a prouvé que dans ce cas, $cost(f_1) < cost(f_2)$ ce qui contredit $f_1 = f_2$. □

Il est à noter que dans notre approche, les contraintes unaires sont considérées comme des contraintes tout à fait ordinaires. Par ailleurs, dans la mesure où une seule strate d'une contrainte est retenue

à un instant donné, les CNs générés sont de taille beaucoup plus réduite que le WCN de départ. On peut espérer que le test de satisfiabilité soit assez simple (donc rapide) à réaliser dans la plupart des cas. Une dernière observation est que cet algorithme exploite une forme d'abstraction en considérant que, du point de vue du coût global, il n'y a pas lieu de distinguer entre eux les tuples de même coût d'une même contrainte.

Pour finir, l'inconvénient de cette première version est qu'elle énumère toutes les combinaisons possibles de strates. Dans le pire des cas, elle énumère un nombre de combinaisons égal au produit du nombre de strates de chaque contrainte, soit $\prod_{w \in cons(W)} nbLevels(w)$. Selon le nombre de variables et de contraintes, cette complexité peut s'avérer nettement plus grande que dans une approche Branch and Bound classique (i.e. quand $\prod_{w \in cons(W)} nbLevels(w) \ggg \prod_i |dom(x_i)|$). Cela s'explique en particulier par le fait qu'une même instanciation peut être explorée plusieurs fois dans les différents tests de satisfiabilité. Néanmoins, même dans ce cas, cette approche peut se révéler payante si le coût de l'optimum est faible.

5.2 Exploitation des MUC

La complexité de l'algorithme précédent peut être fortement réduite en pratique en remarquant qu'il est vain de passer à la strate supérieure pour une contrainte qui ne participe pas à l'insatisfiabilité du CN. L'idée est donc d'identifier un MUC et de ne passer à la strate supérieure que pour les contraintes faisant partie de ce MUC.

Dans le pire des cas, le MUC retourné peut contenir toutes les contraintes du problème et donc la complexité dans le pire des cas n'est pas changée. En pratique cependant, sur des instances concrètes, les MUC sont souvent de petite taille. De ce fait, le voisinage généré est bien plus petit et la complexité pratique est fortement réduite.

5.2.1 Approche complète

La fonction completeSearch (inspirée de [1, 8, 13]) présente l'algorithme modifié pour prendre en compte les MUC dans une version complète. Les premières étapes de l'algorithme sont les mêmes que pour completeSearchNoMUC. Quand le CN courant est insatisfiable un MUC est identifié, et on va progressivement autoriser des strates plus élevées dans les contraintes du MUC. Plus précisément, l'algorithme énumère toutes les manières possibles de relâcher ce MUC. Pour chaque contrainte du MUC (obtenue par l'appel $\texttt{cons}(W, M)$), l'algorithme génère un successeur f' du front courant qui ne diffère de ce dernier que par l'incrémentation de la strate autorisée de cette contrainte. Ce nouveau front est alors inséré dans la file Q à une position dépendante de la valeur de $cost(f')$.

Function completeSearch(W : WCN)

```
1  foreach w ∈ cons(W) do
2      front[w] ← 0
3  Q ← {front}
4  while Q ≠ ∅ do
5      f ← pick and delete least cost front in Q
6      P ← toCN_=(W, f)
7      sol ← solveCN(P)
8      if sol ≠ ⊥ then
9          return sol
10     else
11         M ← extractMUC(P)
12         foreach w ∈ cons(W, M) do
13             if f[w] ≠ nbLevels(w) − 1 then
14                 f' ← f
15                 f'[w] ← f'[w] + 1
16                 if cost(f') ≠ k then
17                     Q ← Q ∪ {f'}
```

Proposition 2 *L'algorithme completeSearch est complet.*

Preuve Par abus de langage, on utilisera le terme front pour désigner implicitement le CN associé à ce front. Soit f_u le front qui est identifié comme insatisfiable et M le MUC extrait de ce front. La seule différence entre completeSearch et completeSearchNoMUC est que lorsqu'un MUC est identifié, on restreint la recherche en s'imposant de relâcher d'abord ce MUC.

Si le WCN n'admet pas de solution, restreindre ainsi la recherche ne fait pas perdre de solution. On peut donc se restreindre au cas où le WCN admet au moins une solution S. Soit f_S le front correspondant à une quelconque solution S. L'algorithme completeSearchNoMUC garantit qu'il existe toujours un front f dans la file de priorité tel que $f \rightarrow_* f_S$.

Ce point est facile à montrer par récurrence : le premier front $\perp$ placé dans la file est un ancêtre de tous les fronts ($\perp \rightarrow_* f_s$). Quand cette propriété est vérifiée pour un front f extrait de la file ($f \rightarrow_* f_s$), elle est vérifiée pour au moins l'un de ses successeurs directs (par définition de $\rightarrow_*$). Donc, par récurrence, cette propriété est toujours garantie.

Si $f_u \not\rightarrow_* f_s$, la restriction proposée ne peut pas nous faire perdre la solution S. On peut donc faire l'hypothèse supplémentaire que $f_u \rightarrow_* f_s$. De ce fait, $\forall w \in cons(W), f_s[w] \geq f_u[w]$. Par ailleurs, il existe

nécessairement une contrainte $w_i \in cons(W, M)$ t.q. $f_s[w_i] > f_u[w_i]$, faute de quoi f_S serait toujours insatisfiable. Soit f' le front défini par $f'[w_i] = f_u[w_i] + 1$ et $\forall w \in cons(W)$ t.q. $w \neq w_i$, $f'[w] = f_u[w]$. f' est un front qui est généré par l'algorithme et par construction $f' \rightarrow_* f_s$. □

La figure 3 présente un exemple où on ne peut pas se contenter de relâcher un MUC d'une seule manière, fût-ce le relâchement de coût optimal. Dans cet exemple, un WCN est composé de trois contraintes w_x, w_{xy} et w_y. Le premier front sélectionne les strates de coût 0, ce qui ne conserve comme tuples autorisés que $\{(x, a)\}$, $\{(x, a), (y, b)\}$ et $\{(y, a)\}$. Bien évidemment, les contraintes w_{xy} et w_y forment dans ce cas un MUC. Il y a deux manières de le relâcher : la première consiste à passer à la strate suivante de w_{xy} (c'est localement le meilleur choix), et la seconde à la strate suivante de w_y. Dans le premier cas, le nouveau front obtenu conserve comme tuples autorisés $\{(x, a)\}$, $\{(x, c), (y, a)\}$ et $\{(y, a)\}$. Cette fois, le MUC porte sur w_x et w_{xy}. On peut soit relâcher w_{xy} pour aboutir à une solution ayant un coût de 100, soit relâcher w_x pour aboutir à un autre MUC que l'on peut relâcher de deux manières pour aboutir soit à une solution de coût 105, ou à une solution de coût 110.

Si dans le premier MUC on avait relâché w_y, on aurait obtenu directement une solution de coût 10 qui est l'optimum.

coût	tuple
100	{(x,c)}
10	{(x,b)}
0	{(x,a)}

(a) w_x

coût	tuple
100	default
5	{(x,c),(y,a)}
0	{(x,a),(y,b)}

(b) w_{xy}

coût	tuple
100	{(y,c)}
10	{(y,b)}
0	{(y,a)}

(c) w_y

FIGURE 3 – De la nécessité d'énumérer tous les relâchements d'un MUC.

Function incompleteSearch(W : WCN)

```
1  foreach w ∈ cons(W) do
2      front[w] ← 0
3  repeat
4      P ← toCN≤(W, front)
5      sol ← solveCN(P)
6      if sol ≠ ⊥ then
7          return sol
8      else
9          M ← extractMUC(P)
10         W' ← restrict(W, M)
11         front ← relax(W', front)
12 until sol ≠ ⊥
```

Function relax(W : WCN, front : array of indexes of levels) : array of indexes of levels

```
1  Q_loc ← {front}
2  while Q_loc ≠ ∅ do
3      f ← pick and delete least cost front in Q_loc
4      P ← toCN≤(W, f)
5      if solveCN(P) ≠ ⊥ then
6          return f
7      else
8          M ← extractMUC(P)
9          foreach w ∈ cons(W, M) do
10             if f[w] ≠ nbLevels(w) − 1 then
11                 f' ← f
12                 f'[w] ← f'[w] + 1
13                 if cost(f') ≠ k then
14                     Q_loc ← Q_loc ∪ {f'}
```

5.2.2 Approche gloutonne

Nous présentons maintenant une version incomplète de notre approche utilisant l'extraction et le relâchement de MUC pour la résolution de WCNs.

À partir d'un WCN W, la fonction incompleteSearch retourne une solution d'un CN dérivé de W. La structure $front$ est tout d'abord initialisée à 0, ce qui correspond à la première strate de chaque contrainte du WCN.

À partir du WCN W et de la structure $front$ on extrait un CN et ce réseau de contraintes est ensuite résolu. Si une solution est trouvée alors cette dernière est renvoyée par la fonction incompleteSearch (ligne 6). Si le réseau de contraintes est prouvé insatisfiable, il est alors nécessaire de relâcher des contraintes du WCN. Pour cela, l'algorithme extrait un WCN W' à partir d'un MUC calculé (lignes 9 et 10). La structure $front$ est alors mise à jour par la fonction $relax$ (ligne 11). Cette procédure relâchera une (ou plusieurs contraintes) de W' afin de rendre satisfiable le MUC associé à W'. Ce processus boucle tant qu'une solution n'est pas trouvée au réseau de contraintes associé à W.

La fonction $relax(W, front)$ modifie le tableau $front$ pour autoriser de nouvelles strates. Il faut noter que le corps de cette fonction est similaire à celui de la fonction *completeSearch*, mais que le résultat est de nature différente et la méthode $\texttt{toCN}_{\leq}$ appelée à la place de $\texttt{toCN}_{=}$. Le principe général est d'incrémenter $front[w]$ pour au moins une contrainte w, de manière à rendre le MUC satisfiable. Autrement dit, pour au moins l'une des contraintes, on s'autorise une augmentation du coût maximal de cette contrainte. Dans

l'approche gloutonne adoptée ici, on cherche juste à modifier le tableau $front$ de manière à ce que le MUC soit cassé.

L'algorithme relax utilise une file de priorité locale. La file Q_{loc} est tout d'abord initialisée avec la structure $front$. Tant que Q_{loc} n'est pas vide (ligne 2), on extrait de la liste le front f ayant le coût $cost(f)$ le plus faible. La fonction $\texttt{toCN}_{\leq}$ construit un réseau de contraintes à partir du WCN et du front f précédemment sélectionné (ligne 4). Si ce réseau de contraintes est satisfiable, la structure $front$ passée initialement en paramètre est mise à jour et retournée par l'algorithme relax. Si le réseau de contraintes n'est pas satisfiable, le relâchement n'est pas suffisant et l'algorithme met à jour sa liste de fronts en générant tous les voisins de f. Pour chaque contrainte apparaissant dans le WCN (issu du MUC), on génère une nouvelle structure $front$ différente de la structure initiale par l'incrémentation d'une strate d'une unique contrainte.

On remarquera que, dans les appels à la procédure relax, on peut se restreindre à un sous-ensemble de la structure $front$. En effet les contraintes susceptibles d'être relâchées sont celles faisant partie du WCN W associé au MUC. Pour économiser de l'espace, on peut donc se restreindre en pratique à une structure $front$ ne contenant pas l'intégralité des contraintes initiales du WCN (comme présentée dans les algorithmes) mais contenant uniquement les contraintes du WCN apparaissant dans W.

Dans la version incomplète, la fonction $\texttt{toCN}_{\leq}$ permettant l'extraction d'un réseau de contraintes à partir d'un front est différente de la fonction $\texttt{toCN}_{=}$ utilisée dans la version complète. En effet, dans la version complète, tous les relâchements possibles d'un MUC sont envisagés et les fronts sont énumérés par ordre de coût croissant, aussi lorsqu'on teste la satisfiabilité d'un CN associé à un front f, on sait que tous les CNs associés à des fronts f' tel que $f' \rightarrow_* f$ ont déjà été prouvés insatisfiables. Dans la version complète, la fonction `toCN` extrait donc un CN constitué des strates courantes d'un front. Dans la version incomplète, tous les relâchements possibles d'un MUC ne sont pas envisagés. On ne peut donc pas garantir que lorsqu'on teste la satisfiabilité d'un CN associé à un front f, tous les CNs associés à des fronts f' tel que $f' \rightarrow_* f$ ont déjà été prouvés insatisfiables. La fonction `toCN` extrait donc un CN à partir des strates courantes et inférieures d'un front.

Cette approche ne garantit pas l'identification de la solution optimale. D'une part, l'ensemble des relâchements issus des MUCs ne sont pas considérés contrairement à la version complète (voir exemple 3). En effet on se contente d'identifier le premier relâchement permettant de restaurer la satisfiabilité des MUCs. D'autre part, considérer uniquement le relâchement localement optimal du MUC ne garantit pas d'obtenir la solution optimale du WCN.

6 Résultats expérimentaux

Les expérimentations ont été menées sur un ordinateur équipé de processeurs Intel(R) Core(TM) i7-2820QM CPU 2.30GHz. Nous avons choisi de comparer notre approche gloutonne, notée `GMR`, à deux approches complètes avec algorithme de transferts de coûts maintenant EDAC proposées par notre solveur AbsCon et par le solveur Toulbar2. À ce stade de notre étude, la version complète présentée dans ce papier est en cours d'implantation dans notre solveur. Le temps limite alloué pour résoudre chaque instance est de 600 secondes. Le temps CPU total pour résoudre chaque instance est donné ainsi que la borne trouvée (UB). Si l'exécution de l'algorithme n'est pas terminée avant le temps limite (CPU supérieur à 600 secondes), la borne affichée correspond à la borne trouvée au bout des 600 secondes. Le tableau 1 représente les résultats obtenus pour la série d'instances *spot*5 constituées de contraintes d'arité au maximum égale à 3.

Nous observons que notre approche donne de meilleures bornes que celles retournées par les deux approches complètes excepté pour les instances *spot*5-42, *spot*5-404, *spot*5-503 et *spot*5-1502. Au delà du fait que notre approche ne soit pas complète, ceci s'explique par le fait qu'il s'agisse d'instances relativement faciles (moins de 1400 contraintes) et donc les algorithmes à transferts de coûts parviennent à fournir une meilleure borne dans le temps imparti. Cependant, il est intéressant de noter que la borne trouvée par notre approche n'est relativement pas si éloignée de celles trouvées par les approches complètes, et ceci dans un temps très faible. Concernant les autres instances, considérées comme plus difficiles (notamment plus de 13000 contraintes pour les instances *spot*5-1403, *spot*5-1407 et *spot*5-1506), nous pouvons observer que notre approche gloutonne renvoie une borne meilleure que celles des approches complètes et dans un temps bien plus faible que le temps limite.

7 Conclusion

Dans ce papier, nous proposons une approche originale permettant la résolution d'un réseau de contraintes pondéré (WCN) grâce à la résolution successive de réseaux de contraintes (CNs) dérivés du WCN initial. Plus précisément, les réseaux de contraintes sont énumérés en relâchant un sous-ensemble des contraintes du WCN initial tant que les CNs sont prouvés insatisfiables. Afin d'améliorer la

Instances		AbsCon GMR	AbsCon EDAC	Toulbar2 EDAC
*spot*5-42	CPU	5.65	> 600	> 600
	UB	162050	**161050**	**161050**
*spot*5-404	CPU	3.17	> 600	217
	UB	118	**114**	**114**
*spot*5-408	CPU	7	> 600	> 600
	UB	**6235**	8238	6240
*spot*5-412	CPU	11.4	> 600	> 600
	UB	**33403**	43390	37399
*spot*5-414	CPU	23.5	> 600	> 600
	UB	**40500**	56492	52492
*spot*5-503	CPU	3.67	> 600	> 600
	UB	12125	13119	**12117**
*spot*5-505	CPU	7.61	> 600	> 600
	UB	**22266**	28258	25268
*spot*5-507	CPU	13.3	> 600	> 600
	UB	**30417**	37429	37420
*spot*5-509	CPU	19.5	> 600	> 600
	UB	**37469**	48475	46477
*spot*5-1401	CPU	45.6	> 600	> 600
	UB	**483110**	513097	516095
*spot*5-1403	CPU	85	> 600	> 600
	UB	**493265**	517260	507265
*spot*5-1407	CPU	334	> 600	> 600
	UB	**495615**	517623	507633
*spot*5-1502	CPU	2.47	0.98	0.02
	UB	28044	**28042**	**28042**
*spot*5-1504	CPU	40.9	> 600	> 600
	UB	**175308**	204314	198318
*spot*5-1506	CPU	207	> 600	> 600
	UB	**388556**	426551	399568

TABLE 1 – Temps CPU en secondes et borne trouvée avant le temps limite pour des instances *spot5*.

complexité en pratique de notre approche, nous proposons d'identifier un noyau minimalement inconsistant (MUC) pour chaque réseau de contraintes insatisfiable afin de concentrer le relâchement des contraintes du WCN sur les contraintes de ce MUC. L'approche est déclinée en un algorithme complet et un algorithme glouton.

De nombreuses perspectives restent encore à étudier. Tout d'abord, nous avons validé expérimentalement dans ce papier l'algorithme glouton mais il nous reste à valider l'approche complète. Enfin des améliorations sont à envisager dans la méthode d'extraction des MUCs. Par exemple, la prise en compte des caractéristiques des contraintes présentes dans le MUC permettrait de mieux cibler celles à relâcher. Ceci pourrait permettre d'éviter l'extraction de nombreux MUCs par la suite et par conséquent permettre d'obtenir une borne plus rapidement. Il est également envisageable d'étudier une version dynamique et incrémentale de notre approche afin de réutiliser le travail de recherche effectué lors de résolutions successives de CNs.

Pour finir, nous rappelons que la méthode proposée permet une intégration simple et naturelle des cadres CSP, (weighted) Max-CSP et WCSP. Il suffit pour cela de généraliser la méthode de construction des CNs générés pendant la résolution.

Remerciements

Ce travail bénéficie du soutien du CNRS et d'OSEO dans le cadre du projet ISI Pajero.

Références

[1] C. Ansotegui, M.L. Bonet, and J. Levy. A new algorithm for weighted partial maxsat. In *Proceedings of AAAI'10*, pages 3–8, 2010.

[2] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier. Semiring-based CSPs and valued CSPs : Frameworks, properties, and comparison. *Constraints*, 4(3) :199–240, 1999.

[3] M. Cooper. Reduction operations in fuzzy or valued constraint satisfaction. *Fuzzy Sets and Systems*, 134(3) :311–342, 2003.

[4] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, and M. Zytnicki. Virtual arc consistency for weighted CSP. In *Proceedings of AAAI'08*, pages 253–258, 2008.

[5] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8) :449–478, 2010.

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

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

[8] Z. Fu and S. Malik. On solving the partial max-sat problem. In *Proceedings of SAT'06*, pages 252–265, 2006.

[9] F. Hemery, C. Lecoutre, L. Sais, and F. Boussemart. Extracting MUCs from constraint networks. In *Proceedings of ECAI'06*, pages 113–117, 2006.

[10] J. Larrosa. Node and arc consistency in weighted CSP. In *Proceedings of AAAI'02*, pages 48–53, 2002.

[11] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2) :1–26, 2004.

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

[13] J.P. Marques-Silva and J. Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In *DATE'08*, pages 408–413, 2008.

[14] P. Meseguer, F. Rossi, and T. Schiex. Soft constraints. In *Handbook of Constraint Programming*, chapter 9, pages 281–328. Elsevier, 2006.

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

DGVNS guidée par les séparateurs

Samir Loudni[1,2] **Mathieu Fontaine**[1,2] **Patrice Boizumault**[1,2]

[1] Université de Caen Basse-Normandie, UMR 6072 GREYC, F-14032 Caen, France
[2] CNRS, UMR 6072 GREYC, F-14032 Caen, France
{prénom.nom}@unicaen.fr

Résumé

Dans nos précédents travaux, nous avons proposé la méthode DGVNS (Decomposition Guided VNS) permettant d'exploiter les clusters, issus de la décomposition arborescente du graphe de contraintes, pour guider l'exploration des voisinages dans les méthodes de type VNS. Dans cet article, nous proposons deux extensions de DGVNS permettant d'exploiter à la fois les clusters et les variables partagées entre ces clusters, appelées séparateurs. De telles variables jouent un charnière dans la résolution du problème. En effet, l'affectation de toutes les variables des séparateurs sépare le problème initial en plusieurs sous-problèmes qui peuvent ensuite être résolus indépendamment. Les expérimentations menées sur des instances aléatoires (GRAPH) et des instances réelles (RLFAP, SPOT5 et tagSNP) montrent la pertinence et l'intérêt de notre approche. Cet article est une version étendue de [19].

Abstract

This paper presents two extensions for DGVNS (Decomposition Guided VNS) method, that exploit both the *graph of clusters* and *separators* between these clusters, to efficiently guide the exploration of large neighborhoods in VNS. Experiments conducted on several difficult instances (RLFAP, SPOT5, GRAPHand tagSNP) show the relevance and the interest of our approach. This paper is an extended version of [19].

1 Introduction

La notion de décomposition arborescente, proposée par Robertson et Seymour [22], vise à découper un problème en sous-problèmes (*clusters*) constituant un graphe acyclique. Chaque cluster correspond à un sous-ensemble de variables fortement connectées. Chaque sous-problème, étant plus petit que le problème original, devient plus facile à résoudre. L'intérêt d'exploiter les propriétés structurelles d'un problème a été attestée dans divers domaines : pour déterminer la satisfiabilité en SAT [1, 13, 21], pour résoudre les CSP (CTE [6]), dans les réseaux bayesiens (AND/OR graph search [20]), en bases de données relationnelles [11, 12], pour les problèmes d'optimisation sous contraintes (BTD [25], Lc-BTD$^+$ [5], RDS-BTD [23], DB [17]). Toutes ces propositions exploitent la décomposition arborescente dans des méthodes de recherche complète.

Dans nos précédents travaux [8, 9], nous avons proposé la méthode DGVNS (Decomposition Guided VNS) permettant d'exploiter les clusters issus de la décomposition arborescente du graphe de contraintes, pour guider l'exploration des voisinages dans les méthodes de type VNS. Cependant, DGVNS ne permet pas de prendre en compte les variables partagées entre ces clusters, appelées *séparateurs*. De telles variables jouent un rôle charnière dans la résolution du problème. En effet, l'affectation de toutes les variables des séparateurs sépare le problème initial en plusieurs sous-problèmes qui peuvent ensuite être résolus indépendamment. De plus, les compatibilités entre affectations des différents sous-problèmes dépendent uniquement des séparateurs.

Dans cet article, nous proposons deux extensions de DGVNS, notées SGVNS (Separator-Guided VNS) et ISGVNS (Intensified SGVNS), qui exploitent à la fois le graphe de clusters et les séparateurs entre ces clusters, afin de mieux cibler les régions du problème qui vont être explorées durant la recherche. Notre idée est de tirer parti des affectations des variables des séparateurs pour guider DGVNS vers les clusters qui sont plus susceptibles de conduire à des améliorations importantes. Les expérimentations menées sur des instances aléatoires (GRAPH) et des instances réelles (RLFAP, SPOT5 et tagSNP) montrent la pertinence et l'intérêt de notre approche.

La section 2 présente le contexte de nos travaux. La section 3 détaille comment exploiter les séparа-

teurs dans DGVNS. Les sections 4 et 5 présentent les jeux de test et les résultats expérimentaux. Enfin, nous concluons et présentons des perspectives.

2 Contexte et définitions

2.1 Réseau de fonctions de coût

Un réseau de fonctions de coût (CFN) est un couple (X, W) où $X=\{x_1, \ldots, x_n\}$ est un ensemble de n variables et W est un ensemble de e fonctions de coût. Chaque variable $x_i \in X$ a un domaine fini D_i de valeurs qui peuvent lui être affectées. La taille du plus grand domaine est notée d. Une affectation de x_i à la valeur $a \in D_i$ est notée (x_i, a). Pour un sous-ensemble de variables $S \subseteq X$, on note D^S le produit cartésien des domaines des variables de S. Pour un n-uplet donné t, $t[S]$ représente la projection du n-uplet t sur l'ensemble de variables S. Une affectation *complète* $t=(a_1, ..., a_n)$ est une affectation de toutes les variables ; dans le cas contraire, on l'appelle affectation *partielle*. Une fonction de coût $w_S \in W$, de portée $S \subseteq X$, est une fonction $w_S : D^S \mapsto [0, k_\top]$ où $k_\top$ est un coût entier maximum (fini ou non) utilisé pour représenter les affectations interdites (exprimant des contraintes dures). Pour capturer fidèlement les contraintes dures, les coûts sont combinés par l'addition bornée $\oplus$, définie par $\alpha \oplus \beta = \min(k_\top, \alpha + \beta)$. Le problème consiste à trouver une affectation complète t de l'ensemble des variables minimisant la combinaison des fonctions de coût $\bigoplus_{w_S \in W} w_S(t[S])$.

2.2 Décomposition arborescente

Le graphe de contraintes d'un CFN est un graphe $G=(X,E)$ composé d'un sommet par variable et il existe une arête $\{u, v\} \in E$ ssi $\exists\, w_S \in W, u, v \in S$.

Définition 1 *Une décomposition arborescente [22] de $G=(X, E)$ est un couple (C_T, T) où $T = (I, A)$ est un arbre avec pour ensemble de nœuds I et pour ensemble d'arêtes A, et $C_T = \{C_i \mid i \in I\}$ est une famille de sous-ensembles de X (appelés clusters) telle que :*

- $\cup_{i \in I} C_i = X$,
- $\forall\, (u, v) \in E$, $\exists\, C_i \in C_T$ *t.q.* $u, v \in C_i$,
- $\forall\, i, j, k \in I$, *si j est sur le chemin de i à k dans T, alors $C_i \cap C_k \subseteq C_j$.*

Définition 2 *L'intersection entre deux clusters est appelée séparateur et notée $sep(C_i, C_j)$. Deux clusters C_i et C_j sont adjacents ssi $sep(C_i, C_j) \neq \emptyset$. Les variables appartenant à un et un seul cluster sont appelées variables propres.*

Définition 3 *Un graphe de clusters, pour une décomposition arborescente (C_T, T), est un graphe non-orienté $G_T = (C_T, E_T)$ dont les sommets sont les éléments de C_T et il existe une arête $(C_i, C_j) \in E_T$ entre les sommets C_i et C_j ssi $sep(C_i, C_j) \neq \emptyset$.*

La largeur d'une décomposition arborescente $T=(I, A)$ est définie par $w^-(T)=\max_{i \in I}(|C_i|-1)$. La largeur de décomposition $tw(G)$ d'un graphe G est la plus petite largeur de toutes les décompositions arborescentes possibles de G. Calculer la largeur de décomposition d'un graphe est un problème NP-complet [2]. Cependant, des heuristiques reposant sur la notion de *triangulation* de graphe [1] permettent le calcul de décompositions approchées. Elles fournissent un majorant de la largeur de décomposition.

Dans cet article, nous utilisons l'heuristique *Maximum Cardinality Search* (MCS) [24] qui constitue un bon compromis entre la largeur de la décomposition obtenue et le temps nécessaire à son calcul [16].

2.3 Decomposition Guided VNS

DGVNS (Decomposition Guided VNS) [8, 9] étend le principe de VNDS [14] (Variable Neighborhood Decomposition Search), en exploitant le graphe de clusters pour guider l'exploration de grands voisinages. Les voisinages sont obtenus en désaffectant une partie de la solution courante selon une heuristique de choix de variables. La reconstruction de la solution sur les variables désinstanciées est effectuée par une recherche arborescente partielle, LDS (*Limited discrepancy search*, [15]), aidée par la propagation des contraintes (CP) basée sur un calcul de minorants.

Définition 4 *Soit $G_T=(\mathcal{C}_T, E_T)$ le graphe de clusters associé à G. Soient $C_i \in \mathcal{C}_T$ un cluster de G_T et $k \in [1 \ldots n]$ la dimension du voisinage. La structure de voisinage $N_{k,i}$ désigne l'ensemble de toutes les combinaisons possibles de k variables parmi C_i.*

L'algorithme 1 présente le pseudo-code de DGVNS. Tout d'abord, il construit une décomposition arborescente de G (ligne 4), puis génère aléatoirement une solution initiale S (fonction genInitSol, ligne 6). Afin de favoriser les mouvements dans des régions fortement liées, DGVNS se base sur les structures de voisinages $N_{k,i}$ (cf. Définition 4). En effet, la notion de cluster permet d'exhiber ces régions, de part sa taille (plus petite que le problème initial), et de part la forte connexion entre les variables qu'il contient. Ainsi, l'ensemble de variables candidates C_s à désaffecter est

1. Un graphe est cordal ou triangulé si et seulement si tous ses cycles de taille supérieure à quatre ont une corde (une arête connectant deux sommets non-adjacents du cycle).

Algorithme 1: Pseudo-code de DGVNS

```
1  Function DGVNS(X, W, k_init, k_max, δ_max);
2  begin
3      let G be the constraints graph of (X, W) ;
4      let (C_T, T) be a tree decomposition of G ;
5      let C_T = {C_1, C_2, ..., C_p} ;
6      S ← genInitSol() ;
7      k ← k_init ;
8      i ← 1 ;
9      while (k < k_max) ∧ (notTimeOut) do
10         C_s ← CompleteCluster(C_i, k) ;
11         X_un ← Hneighborhood(C_s, N_{k,i}, S) ;
12         A ← S\{(x_i, a) | x_i ∈ X_un} ;
13         S' ← Rebuild(A, X_un, δ_max, f(S), S) ;
14         NeighbourhoodChangeDGVNS(S, S', k, i);
15     return S ;
16 Procedure NeighbourhoodChangeDGVNS (S, S', k, i);
17 begin
18     if f(S') < f(S) then
19         S ← S' ;
20         k ← k_init;
21         i ← succ(i) ;
22     else k ← k + 1 ; i ← succ(i) ;
```

sélectionné à partir du cluster C_i. Si $(k > |C_i|)$, C_s est étendu aux variables des clusters C_j voisins de C_i afin de prendre en compte la topologie du graphe de clusters. Ce traitement est réalisé par la fonction `CompleteCluster`(C_i, k) (ligne 10). De plus, grâce à la forte connexion entre les variables de C_s, l'étape de reconstruction pourra bénéficier d'un plus fort filtrage et d'un meilleur calcul de minorants. Un sous ensemble X_{un} de k variables est sélectionné aléatoirement dans C_s parmi les variables en conflit par l'heuristique de voisinage `Hneighborhood` (ligne 11). Une affectation partielle $\mathcal{A}$ est générée à partir de la solution courante S en désaffectant les k variables de X_{un} (ligne 12). Ensuite, ces variables sont reconstruites (ligne 13) par une recherche arborescente partielle `LDS` [15], aidée par la propagation de contraintes (CP) (voir [18] pour plus de détails). La recherche s'arrête dès que la dimension maximale de voisinage k_{max} ou le *TimeOut* est atteint (ligne 9).

La procédure `NeighborhoodChangeDGVNS` contrôle les mécanismes d'intensification et de diversification de `DGVNS` (cf. Algorithme 1). Soit p le nombre total de clusters, *succ* une fonction de succession [2], et $N_{k,i}$ la structure de voisinage courante. Si `LDS+CP` ne trouve pas de meilleure solution S' dans le voisinage de S, `DGVNS` cherche des améliorations dans $N_{(k+1),succ(i)}$ (la structure de voisinage où $(k+1)$ variables de C_s seront désaffectées) (ligne 22). Tout d'abord, la diversification réalisée par le déplacement du cluster C_i au cluster $C_{succ(i)}$ permet de favoriser l'exploration de nouvelles parties de l'espace de recherche et de rechercher de meilleures solutions dans celles-ci. Ensuite, quand un optimum local est atteint dans le voisinage courant, l'augmentation de la dimension du voisinage k permet aussi de diversifier la recherche en explorant de plus grandes régions.

Quand une solution de meilleure qualité S' est trouvée par `LDS+CP` dans le voisinage $N_{k,i}$, S' devient la solution courante (ligne 19), k est réinitialisé à k_{init} et le prochain cluster est considéré (ligne 21). En effet, rester dans le même cluster rend plus difficile la découverte de nouvelles solutions : se déplacer sur un nouveau cluster permet la diversification de la recherche autour de la nouvelle solution S'.

3 `VNS` guidée par les séparateurs

Dans cette section, nous présentons les deux extensions de `DGVNS`, notées `SGVNS` et `ISGVNS`, qui exploitent à la fois le graphe de clusters et les séparateurs entre ces clusters, afin de mieux cibler les régions du problème qui vont être explorées durant la recherche.

3.1 Separator-Guided `VNS` (`SGVNS`)

`SGVNS` exploite l'évolution de la solution au cours de la recherche afin de guider l'effort de diversification de `DGVNS` vers des clusters contenant, dans leurs séparateurs, **au moins une variable réinstanciée** impliquée dans l'amélioration de la solution courante. L'algorithme 2 présente le pseudo code de `SGVNS`. Son schéma général est similaire à celui de `DGVNS` (cf. algorithme 1). La différence principale, réside dans l'utilisation d'une **liste de prospection**, notée $\mathcal{C}and$, contenant les clusters à visiter en priorité. Cette liste est mise à jour à chaque fois que `LDS+CP` trouve une solution de meilleure qualité.

Soit C_i le cluster courant et S' une solution de meilleure qualité trouvée par `LDS+CP` dans le voisinage de S. Soit V_c l'ensemble des variables qui ont été réinstanciées dans S pour obtenir S' (ligne 23) et C_w l'ensemble des clusters C_j de $\mathcal{C}and$ tel que $sep(C_i, C_j)$ partage au moins une variable avec V_c [3] (ligne 24). Au début, $\mathcal{C}and$ est initialisé à l'ensemble des clusters $\mathcal{C}_T$ de la décomposition arborescente (ligne 6).

Contrairement à `DGVNS`, la diversification de `SGVNS` est réalisée en considérant successivement les clusters $C_j \in C_w$. En effet, à chaque fois que `LDS+CP` trouve une solution de meilleure qualité S' dans le voisinage de S (ligne 20), l'ensemble C_w est calculé (lignes

2. si $i < p$ alors $succ(i) = i + 1$ sinon $succ(p) = 1$.

3. En effet, comme V_c ne contient que des variables de C_s (cf. ligne 10), si $C_j \cap V_c \neq \emptyset$, alors $sep(C_i, C_j) \cap V_c \neq \emptyset$.

Algorithme 2: Pseudo-code de SGVNS

```
1  Function SGVNS (X, W, k_init, k_max, δ_max);
2  begin
3      let G be the constraints graph of (X, W) ;
4      let (C_T, T) be a tree decomposition of G ;
5      S ← genSolInit();
6      Cand ← C_T;
7      k ← k_init ;
8      C_i ← remove-first(Cand) ;
9      while (k < k_max) ∧ (notTimeOut) do
10         C_s ← CompleteCluster(C_i, k) ;
11         X_un ← Hneighborhood(C_s, N_{k,i}, S) ;
12         A ← S\{(x_i, a) | x_i ∈ X_un} ;
13         S' ← LDS + CP(A, X_un, δ_max, f(S), S) ;
14         NeighbourhoodChangeSGVNS(S, S', k, i) ;
15     return S ;
16 Procedure NeighbourhoodChangeSGVNS(S, S', k, i);
17 begin
18     if Cand = ∅ then
19         Cand ← C_T;
20     if f(S') < f(S) then
21         S ← S' ;
22         k ← k_init;
23         V_c ← {x | S'[x] ≠ S[x]};
24         C_w ← {C_j ∈ Cand | C_j ∩ V_c ≠ ∅};
25         Cand ← Cand\C_w;
26         foreach C_j ∈ C_w do
27             insert-first(Cand, C_j);
28     else
29         k ← k + 1;
30     C_i ← remove-first(Cand);
```

Algorithme 3: Pseudo-code de ISGVNS

```
1  Function ISGVNS (X, W, k_init, k_max, δ_max);
2  begin
3      let G be the constraints graph of (X, W) ;
4      let (C_T, T) be a tree decomposition of G ;
5      S ← genSolInit() ;
6      k ← k_init ;
7      i ← 1 ;
8      P_List ← ∅;
9      while (k < k_max) ∧ (notTimeOut) do
10         C_s ← CompleteCluster(C_i, k) ;
11         X_un ← Hneighborhood(N_{k,i}, C_s, W, S) ;
12         A ← S\{(x_i, a) | x_i ∈ X_un} ;
13         S' ← LDS + CP(A, X_un, δ_max, f(S), S) ;
14         NeighbourhoodChangeISGVNS(S, S', k, i);
15     return S ;
16 Procedure ChangeNeighborISGVNS(S, S', k, i);
17 begin
18     if f(S') < f(S) then
19         S ← S';
20         k ← k_init;
21         V_c ← {x | S'[x] ≠ S[x]};
22         C_w ← {C_j | C_j ∩ V_c ≠ ∅, j = i + 1, ..., | C_T |} ;
23         foreach C_j ∈ C_w do
24             insert-Queue (P_List, C_j) ;
25         make tabu each element x ∈ V_c for L next
           iterations;
26     else
27         k ← k + 1 ;
28     if P_List ≠ ∅ then
29         i ← remove-first(P_List)
30     else
31         i ← succ(i)
```

23-24), les clusters de C_w sont déplacés en début de la liste $\mathcal{C}and$ (lignes 25-27) et le prochain cluster C_i à considérer est retiré de la liste (ligne 30). Comme pour DGVNS, S' devient la solution courante et k est réinitialisé à k_{init} (ligne 22). À l'inverse, si aucune amélioration de la solution courante n'a été trouvée, k est incrémenté de 1 (ligne 29) et SGVNS considère le cluster courant en tête de la liste $Cand$ (ligne 30).

Tout d'abord, la suppression du cluster courant C_i de $\mathcal{C}and$ permet de ne maintenir dans celle-ci que la liste des clusters non encore visités, assurant ainsi une plus large couverture de l'espace de recherche. Ainsi, chaque cluster de $\mathcal{C}_T$ sera visité au moins une fois[4]. De plus, privilégier les clusters contenant, dans leurs séparateurs, au moins une variable réinstanciée permet de guider la recherche vers des régions du problème pouvant mener à de plus larges améliorations de la solution courante. Enfin, cela permet de propager, au travers des séparateurs, les nouvelles affectations de S' vers les clusters $C_j \in C_w$, lors des prochaines itérations de SGVNS.

4. Un cluster pourra être considéré plusieurs fois dès lors que $Cand$ devient vide, car réinitialisée à $\mathcal{C}_T$ (ligne 19).

3.2 Intensified Separator-Guided VNS (ISGVNS)

ISGVNS vise à intensifier l'exploration "autour" des clusters contenant des variables réinstanciées. À cet effet, une *liste de propagation* P_{List} dotée d'une *liste taboue dynamique* T_{List} de taille L sont utilisées. La liste de propagation contient l'ensemble des clusters candidats à examiner après chaque amélioration de la solution courante. La liste taboue assure que les variables impliquées dans la sélection des clusters candidats (i.e. variables de V_c) ne seront pas reconsidérées dans $N_{k,i}$ par la fonction Hneighborhood, lors des L prochaines itérations de ISGVNS.

L'algorithme 3 présente le pseudo-code de ISGVNS. **L'intensification est réalisée en exploitant la liste de propagation**. Comme pour SGVNS, V_c dé-

signe l'ensemble de toutes les variables qui ont été réinstanciées (ligne 21) et C_w est l'ensemble des clusters ayant au moins, dans leurs séparateurs, une variable de V_c (ligne 22). Contrairement à SGVNS, chaque cluster $C_j \in C_w$ est ajouté à P_{List} (ligne 24) et chaque variable $x \in V_c$ est rendue taboue pour les L prochaines itérations (ligne 25). La valeur de L est fixée à la taille de P_{List} afin d'éviter de réaffecter les variables de V_c tant que tous les clusters $C_j \in C_w$ n'ont pas été considérés. Enfin, le prochain cluster à examiner correpond au premier élément de P_{List}, si celle-ci n'est pas vide (ligne 29). Dans le cas contraire, le successeur de C_i dans $\mathcal{C}_T$ est considéré (ligne 31).

ISGVNS **permet de renforcer l'équilibre entre intensification et diversification**. En effet, tant qu'aucune amélioration n'est effectuée, ISGVNS se comporte comme DGVNS, en considérant successivement tous les clusters C_i. Cependant, dès que LDS+CP améliore la solution courante, ISGVNS bascule vers un schéma d'intensification, jusqu'à ce que tous les clusters de P_{List} aient été examinés.

4 Jeux de test

Les expérimentations ont été menées sur différentes instances de quatre problèmes différents.

- Instances RLFAP : Le CELAR (Centre d'électronique de l'Armement) a mis à disposition un ensemble d'instances du problème d'affectation de fréquences radio (RLFAP) [4]. L'objectif est d'assigner un nombre limité de fréquences à un ensemble de liens radios entre des couples de sites, afin de minimiser les interférences dues à la réutilisation des fréquences. Nous reportons les résultats sur les instances les plus difficiles : Scen06, Scen07 et Scen08.

- Instances GRAPH : Le générateur GRAPH (Generating Radio link frequency Assignment Problems Heuristically) a été développé par le projet CALMA [26] afin de proposer des instances aléatoires ayant une structure proche des instances RLFAP.

- Instances SPOT5 : La planification quotidienne d'un satellite d'observation de la terre (SPOT5) consiste à sélectionner les prises de vue à effectuer dans la journée en prenant en compte les limites matérielles du satellite, tout en maximisant l'importance des photographies sélectionnées [3]. Nous reportons les résultats sur six instances sans contraintes dures de capacité.

- Instances tagSNP : Un SNP (Single Nucleotide Polymorphism) -ou polymorphisme nucléotidique- est la variation d'une seule paire de nucléotides dans l'ADN de deux individus d'une même espèce ou dans une paire de chromosomes d'un même individu. Les SNP sont des marqueurs biologiques qui peuvent être utilisés pour la prédiction des risques de développement de certaines maladies [7, 10]. Le problème de sélection des tagSNP consiste à choisir un sous-ensemble de SNP, appelé tagSNP, qui permet de capturer le maximum d'information génétique. Ce problème est considéré comme très difficile du fait de sa proximité avec le problème de *set covering* (NP-difficile) [23]. Nous rapportons les résultats des expérimentations menées sur 13 instances issues des données du chromosome1 humain [5] avec r_0=0.5 (instances modélisées sous forme de CFN binaires [23] ayant jusqu'à $n = 1550$ variables, avec des domaines de taille d allant de 30 à 266 valeurs et jusqu'à $e = 250,000$ fonctions de coût). 7 instances sont de taille moyenne et 6 de grande taille.

5 Expérimentations

5.1 Protocole expérimental

Chaque méthode a été appliquée sur chaque instance, avec une *discrepancy* de 3 pour LDS, ce qui correspond à la meilleure valeur trouvée sur les instances RLFAP [18]. Les valeurs de k_{min} et k_{max} ont été respectivement fixées à 4 et n (nombre total de variables). Le $TimeOut$ à été fixé à 3 600 secondes pour les instances RLFAP et SPOT5. Pour les instances tagSNP de taille moyenne (resp. de grande taille), le $TimeOut$ a été fixé à 2 heures (resp. 4 heures). Un ensemble de 50 essais par instance a été réalisé sur un AMD opteron 2.1 GHz et 256 Go de RAM. Toutes les méthodes ont été implantées en C++ en utilisant la librairie toulbar2 [6].

Pour chaque instance et chaque méthode, nous reportons (i) le nombre de succès (l'optimum est atteint), (ii) le temps de calcul moyen pour atteindre l'optimum, ainsi que (iii) le coût moyen des solutions trouvées sur les 50 essais.

Dans cette section, nous reportons les résultats de nos expérimentations menées sur les instances RLFAP, GRAPH, SPOT5 et tagSNP. Tout d'abord, nous comparons SGVNS et ISGVNS avec DGVNS (cf. sections 5.2 et 5.3), puis nous comparons SGVNS avec ISGVNS (cf. section 5.4).

5.2 Comparaison entre SGVNS et DGVNS

5.2.1 Instances RLFAP

Sur les instances RLFAP (cf. tableau 1), SGVNS surclasse clairement DGVNS sur Scen07, et reste très comparable sur les deux autres instances. Pour la Scen07, SGVNS améliore le taux de succès de DGVNS de 16% (de

5. http://www.costfunction.org/benchmark
6. http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP

Instance	Méthode	Succ.	Temps	Moy.
Scen06, $n = 100$	SGVNS	50/50	111	3,389
$e = 1222$, $d = 44$	ISGVNS	**50/50**	**93**	**3,389**
S^* =3,389	DGVNS	50/50	112	3,389
Scen07, $n = 200$	SGVNS	48/50	652	343,802
$e = 2,665$, $d = 44$	ISGVNS	**49/50**	**806**	**343,794**
S^* =343,592	DGVNS	40/50	317	345,614
scen08, $n = 458$	SGVNS	2/50	532	281
$e = 5,286$, $d = 44$	ISGVNS	**4/50**	**1,408**	**276**
S^* =262	DGVNS	3/50	1,811	275
Graph05, $n = 100$	SGVNS	50/50	19	221
$e = 1,034$, $d = 44$	ISGVNS	50/50	16	221
S^* =221	DGVNS	**50/50**	**10**	**221**
Graph06, $n = 200$	SGVNS	50/50	292	4,123
$e = 1,970$, $d = 44$	ISGVNS	**50/50**	**240**	**4,123**
S^* =4,123	DGVNS	50/50	367	4,123
Graph11, $n = 340$	SGVNS	27/50	3,026	4,238
$e = 3417$, $d = 44$	ISGVNS	**39/50**	**2,762**	**3,349**
S^* =3,080	DGVNS	8/50	3,046	4,234
Graph13, $n = 458$	SGVNS	**6/50**	**3,260**	**14,707**
$e = 4,915$, $d = 44$	ISGVNS	1/50	3,196	17,085
S^* =10,110	DGVNS	0/50	-	22,489 (18,639)

TABLE 1 – Comparaison entre SGVNS, ISGVNS et DGVNS sur les instances RLFAP et GRAPH.

80% à 96%), et réduit la déviation moyenne de l'optimum de (0.58% à 0.06%).

5.2.2 Instances SPOT5

Sur les instances SPOT5 (cf. tableau 2), SGVNS se montre très efficace comparé à DGVNS, particulièrement sur les instances de grande taille (#412, #414, #507 et #509), pour lesquelles l'amélioration est très significative. Pour les instances #507 et #509, SGVNS procure un gain en termes de taux de succès de 12% (de 66% à 78%) et 16% (de 80% à 96%) respectivement. Ces gains atteignent respectivement 24% et 20% pour les instances #412 et #414. Sur les autres instances, les deux méthodes obtiennent des résultats similaires.

5.2.3 Instances GRAPH

La tendance se confirme sur les instances GRAPH (cf. tableau 1). Pour les instances réputées faciles (Graph05 et Graph06), les deux méthodes obtiennent les mêmes taux de succès (i.e. 100%). Toutefois, SGVNS est plus rapide que DGVNS sur Graph06 (gain en temps de calcul d'environ 20%). Pour les instances difficiles, les gains obtenus par SGVNS sont plus importants. Pour l'instance Graph11, SGVNS obtient 3 fois plus de succès que DGVNS (gain de 38% en termes de taux de succès). Pour l'instance Graph13, SGVNS atteint l'optimum (6 essais avec succès), surclassant nettement DGVNS. Ces résultats démontrent clairement la pertinence d'exploiter les séparateurs pour guider l'exploration des voisinages.

Instance	Méthode	Succ.	Temps	Moy.
#408, $n = 200$	SGVNS	**50/50**	**132**	**6,228**
$e = 2,232$, $d = 4$	ISGVNS	50/50	140	6,228
S^* =6,228	DGVNS	49/50	117	6,228
#412, $n = 300$	SGVNS	48/50	351	32,381
$e = 4,348$,$d = 4$	ISGVNS	**49/50**	**434**	**32,381**
S^* =32,381	DGVNS	36/50	84	32,381
#414, $n = 364$	SGVNS	**48/50**	**630**	**38,478**
$e = 2,242$, $d = 4$	ISGVNS	46/50	524	38,478
S^* =38,478	DGVNS	38/50	554	38,478
#505, $n = 240$	SGVNS	50/50	99	21,253
$e = 2,242$, $d = 4$	ISGVNS	50/50	67	21,253
S^* =21,253	DGVNS	**50/50**	**63**	**21,253**
#507, $n = 311$	SGVNS	39/50	576	27,390
$e = 5,732$, $d = 4$	ISGVNS	**45/50**	**694**	**27,390**
S^* =27,390	DGVNS	33/50	71	27,390
#509, $n = 348$	SGVNS	**48/50**	**412**	**36,446**
$e = 8,624$, $d = 4$	ISGVNS	46/50	499	36,446
S^* =36,446	DGVNS	40/50	265	36,446

TABLE 2 – Comparaison entre SGVNS, ISGVNS et DGVNS sur les instances SPOT55.

5.2.4 Instances tagSNP

Pour les instances de taille moyenne (cf. tableau 3), SGVNS et DGVNS atteignent l'optimum sur chacun des 50 essais. Cependant, SGVNS est plus rapide sur trois instances (#4449, #9313 et #16421), plus lent sur trois autres instances (#3792, #8956 et #15757), et très similaire à DGVNS sur l'instance #16706. Pour l'instance #16421, SGVNS améliore de 24% le temps de calcul de DGVNS. Le meilleur résultat est obtenu sur l'instance #9313, où SGVNS améliore le temps de calcul de DGVNS de 36%. Les moins bons résultats sont obtenus sur l'instance #8956, pour laquelle DGVNS est plus rapide.

Instance	Méthode	Succ.	Temps	Moy.
#3792, $n = 528$	DGVNS	50/50	954	6,359,805
$e = 12,084$, $d = 59$	SGVNS	50/50	1,033	6,359,805
$S^* = 6,359,805$	ISGVNS	**50/50**	**853**	**6,359,805**
#4449, $n = 464$	DGVNS	50/50	665	5,094,256
$e = 12,540$, $d = 64$	SGVNS	**50/50**	**661**	**5,094,256**
$S^* = 5,094,256$	ISGVNS	50/50	675	5,094,256
#8956, $n = 486$	DGVNS	50/50	4,911	6,660,308
$e = 20,832$, $d = 106$	SGVNS	50/50	5,483	6,660,308
$S^* = 6,660,308$	ISGVNS	**50/50**	**4,118**	**6,660,309**
#9319, $n = 562$	DGVNS	50/50	788	6,477,229
$e = 14,811$, $d = 58$	SGVNS	**50/50**	**500**	**6,477,229**
$S^* = 6,477,229$	ISGVNS	50/50	672	6,477,229
#15757, $n = 342$	DGVNS	**50/50**	**60**	**2,278,611**
$e = 5,091$, $d = 47$	SGVNS	50/50	104	2,278,611
$S^* = 2,278,611$	ISGVNS	50/50	80	2,278,611
#16421, $n = 404$	DGVNS	50/50	2,673	3,436,849
$e = 12,138$, $d = 75$	SGVNS	**50/50**	**2,025**	**3,436,849**
$S^* = 3,436,849$	ISGVNS	50/50	5,863	3,436,849
#16706, $n = 438$	DGVNS	50/50	153	2,632,310
$e = 6,321$, $d = 30$	SGVNS	50/50	159	2,632,310
$S^* = 2,632,310$	ISGVNS	**50/50**	**89**	**2,632,310**

TABLE 3 – Comparison entre SGVNS, ISGVNS et DGVNS sur les instances tagSNP de taille moyenne.

Instance	Méthode	Succ.	Temps	Moy.
#10442, $n = 908$	DGVNS	**50/50**	**4,552**	**21,591,913**
$e = 28,554$, $d = 76$	SGVNS	50/50	7,153	21,591,913
$S^* = 21,591,913$	ISGVNS	50/50	7,291	21,591,913
#14226, $n = 1,058$	DGVNS	46/50	7,606	25,688,751
$e = 36,801$, $d = 95$	SGVNS	40/50	7,646	25,805,242
$S^* = 25,665,437$	ISGVNS	**50/50**	**9,596**	**25,665,437**
#17034, $n = 1142$	DGVNS	**41/50**	**8,900**	**38,563,232**
$e = 47,967$,$d = 123$	SGVNS	33/50	10,212	38,869,514
$S^* = 38,318,224$	ISGVNS	36/50	10,579	38,746,957
#12976, $n = 920$	DGVNS	0/50	-	24,685,331 (24,483,048)
$e = 44,823$, $d = 128$	SGVNS	0/50	-	25,059,787 (24,483,414)
$S^* = 21,604,644$	ISGVNS	0/50	-	**24,627,760 (24,482,897)**
#13931, $n = 820$	DGVNS	0/50	-	**21,107,259 (21,106,760)**
$e = 38,775$, $d = 145$	SGVNS	0/50	-	21,107,723 (21,106,833)
$S^* = 21,106,731$	ISGVNS	0/50	-	21,331,064 (**21,106,770**)
#14007, $n = 1,554$	DGVNS	0/50	-	57,808,373 (53,753,706)
$e = 54,753$, $d = 100$	SGVNS	0/50	-	58,155,109 **(53,753,161)**
$S^* = 50,290,563$	ISGVNS	0/50	-	**57,565,916** (55,486,138)

TABLE 4 – Comparison entre SGVNS, ISGVNS et DGVNS sur les instances tagSNP de grande taille.

Pour les instances de grande taille (cf. tableau 4), SGVNS se montre moins compétitive en termes de taux de succès et de temps de calcul, en particulier sur les trois instances #10442, #14226 et #17034, où DGVNS est clairement le meilleur. Pour l'instance #17034, DGVNS améliore le taux de succès de SGVNS de 16% (de 66% à 82%), la déviation moyenne de l'optimum est réduite (de 1.43% à 0.63%) et DGVNS est 1.2 fois plus rapide que SGVNS. Pour l'instance #14226, le gain en taux de succès atteint 12% (de 80% à 92%) et DGVNS réduit la déviation moyenne de l'optimum (de 0.54% à 0.09%). Pour les autres instances, DGVNS trouve des solutions de meilleure qualité en moyenne que SGVNS.

5.3 Comparaison entre ISGVNS et DGVNS

5.3.1 Instances RLFAP

ISGVNS domine clairement DGVNS, notamment sur les deux instances réputées difficiles Scen07 et Scen08. Pour la Scen07, ISGVNS améliore le taux de succès de DGVNS de 18% (de 80% à 98%). Pour la Scen08, ISGVNS obtient un succès de plus que DGVNS et réduit le temps de calcul d'environ 22%. Pour la Scen06, ISGVNS est plus rapide que DGVNS.

5.3.2 Instances SPOT5

ISGVNS est meilleur que DGVNS sur la plupart des instances (cf. tableau 2). Pour les instances de grande

taille (#412, #414, #507 et #509), ISGVNS obtient un taux de succès moyen de 93% contre 73.5% pour DGVNS. Pour les autres instances (#408 et #505), les deux méthodes obtiennent les mêmes taux de succès, mais DGVNS est plus rapide.

5.3.3 Instances GRAPH

Une fois de plus, ISGVNS surclasse nettement DGVNS, particulièrement sur les deux instances Graph11 et Graph13 (cf. tableau 1). Pour l'instance Graph11, ISGVNS obtient 5 fois plus de succès que DGVNS. Pour l'instance Graph13, ISGVNS atteint une fois l'optimum et obtient des solutions de meilleure qualité en moyenne. Pour les instances faciles, ISGVNS est plus rapide que DGVNS.

5.3.4 Instances tagSNP

Pour les instances de taille moyenne (cf. tableau 3), ISGVNS est plus rapide que DGVNS sur quatre instances (#3792, #8956, #9313 et #16706), plus lent sur deux instances (#15757 et #16421) et similaire à DGVNS sur l'instance #4449. Pour les instances #8956 et #9313, ISGVNS améliore le temps de calcul de DGVNS d'environ 16% en moyenne. Les meilleurs résultats sont obtenus sur l'instance #16706, pour laquelle le gain est d'environ 41%. À l'inverse, pour l'instance #16421, ISGVNS obtient sa plus mauvaise performance : DGVNS est deux fois plus rapide.

Pour les instances de grande taille, ISGVNS est plus efficace que DGVNS sur l'instance #14226 et est moins compétitive sur l'instance #17034. Les deux méthodes obtiennent les mêmes taux de succès sur l'instance #10442, mais DGVNS est plus rapide. Pour les autres instances, si on compare la qualité moyenne des solutions obtenues, ISGVNS est meilleure sur 2 instances (#12976 et #14007) et reste comparable sur l'instance #13931. Par ailleurs, ISGVNS obtient les meilleurs coûts sur 2 instances parmi 3.

5.4 Comparaison entre SGVNS et ISGVNS

5.4.1 Instances RLFAP

ISGVNS devance SGVNS sur toutes les instances (cf. tableau 1). Pour la Scen06, ISGVNS est plus rapide en moyenne. Pour la Scen07, ISGVNS obtient un succès de plus que SGVNS. Pour la Scen08, ISGVNS obtient **deux fois plus de succès** que SGVNS et des solutions de meilleure qualité en moyenne.

5.4.2 Instances SPOT5

Sur les instances SPOT5, aucune des deux méthodes ne domine l'autre (cf. tableau 2). En effet, SGVNS obtient 2 succès de plus que ISGVNS sur deux instances (#414 et #509) et est plus rapide sur l'instance #408, tandis que ISGVNS devance SGVNS en termes de taux de succès sur deux instances (#412 et #507) et est plus rapide sur l'instance #505.

5.4.3 Instances GRAPH

Sur les instances faciles (Graph05 et Graph06), ISGVNS est légèrement plus rapide que SGVNS (cf. tableau 1). En revanche, sur les deux instances réputées difficiles, aucune méthode ne surclasse nettement l'autre. Pour l'instance Graph11, ISGVNS obtient 24% de succès en plus par rapport à SGVNS et réduit la déviation moyenne de l'optimum de (37% à 8%). Pour l'instance Graph13, SGVNS multiplie le nombre de succès de ISGVNS par 6 et réduit la déviation moyenne de l'optimum de (69% à 45%).

5.4.4 Instances tagSNP

Pour les instances de taille moyenne, SGVNS et ISGVNS obtiennent les mêmes taux de succès. Toutefois, ISGVNS est plus rapide que SGVNS sur quatre instances (#3792, #8956, #15757 et #16706), et est moins rapide sur trois autres instances. Pour les instances de grande taille, ISGVNS surclasse nettement SGVNS. ISGVNS améliore le taux de succès moyen de SGVNS d'environ 13% sur deux instances (#14226 et #17034) et obtient des solutions de meilleure qualité sur les instances #12976 et #14007, tandis que SGVNS est plus rapide sur l'instance #14226 et obtient des solutions de meilleure qualité sur l'instance #13931.

5.5 Bilan sur les apports de SGVNS et ISGVNS

Nous avons montré expérimentalement l'intérêt d'exploiter les séparateurs pour mieux guider DGVNS vers des voisinages susceptibles de conduire à des améliorations plus importantes. Sur la plupart des instances testées, SGVNS et ISGVNS surclassent nettement DGVNS, excepté sur les instances tagSNP de grande taille, où SGVNS est moins efficace que DGVNS. SGVNS reste toutefois plus compétitive sur les instances de taille moyenne.

De ces expérimentations, il ressort également que ISGVNS donne de meilleurs résultats en moyenne que SGVNS. ISGVNS surclasse SGVNS sur 17 instances parmi 26, alors que SGVNS obtient de meilleurs résultats sur 9 instances. Ces résultats montrent l'importance de la liste de propagation pour réaliser un meilleur compromis entre intensification et diversification.

6 Conclusions et perspectives

Nous avons proposé deux extensions de la méthode DGVNS qui tirent parti à la fois du graphe de clusters et des séparateurs entre ces clusters, pour guider efficacement l'exploration des grands voisinages dans DGVNS. Les expérimentations menées sur plusieurs instances difficiles ont montré que SGVNS et ISGVNS sont nettement plus performants que DGVNS, et que ISGVNS est très efficace par rapport à SGVNS. Nous travaillons actuellement sur une version parallèle pour l'exploration des clusters.

Acknowledgements. Ce travail à été soutenu par l'Agence Nationale de la Recherche, référence projet FiCOLOFO ANR-10-BLA-0214.

Références

[1] E. Amir and S. Mcllraith. Solving satisfiability using decomposition and the most constrained subproblem. *Electronic Notes in Discrete Mathematics*, 9 :329–343, 2001.

[2] S. Arnborg, D. G. Corneil, and A. Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM. J. on Algebraic and Discrete Methods*, 8 :277–284, 1987.

[3] E. Bensana, M. Lemaître, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4(3) :293–299, 1999.

[4] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4(1) :79–89, 1999.

[5] S. de Givry, T. Schiex, and G. Verfaillie. Exploiting tree decomposition and soft local consistency in weighted CSP. In *AAAI*, pages 22–27. AAAI Press, 2006.

[6] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artif. Intell.*, 38(3) :353–366, 1989.

[7] C.S. Carlson et al. Selecting a maximally informative set of single-nucleotide polymorphisms for association analyses using linkage disequilibrium. *Am. J. of Hum. Genetics*, 74(1) :106–120, 2004.

[8] M. Fontaine, S. Loudni, and P. Boizumault. Guiding VNS with tree decomposition. In *ICTAI*, pages 505–512, 2011.

[9] M. Fontaine, S. Loudni, and P. Boizumault. Exploiting tree decomposition for guiding neighborhoods exploration for VNS. *RAIRO Operations Research*, 47(2) :91–123, 2013.

[10] S. Gopalakrishnan and Z. S. Qin. Tagsnp selection based on pairwise ld criteria and power analysis in association studies. In *Pacific Symposium on Biocomputing*, pages 511–522, 2006.

[11] G. Gottlob, S. T. Lee, and G. Valiant. Size and treewidth bounds for conjunctive queries. In Jan Paredaens and Jianwen Su, editors, *PODS*, pages 45–54. ACM, 2009.

[12] G. Gottlob, R. Pichler, and F. Wei. Tractable database design through bounded treewidth. In Stijn Vansummeren, editor, *PODS*, pages 124–133. ACM, 2006.

[13] D. Habet, L. Paris, and C. Terrioux. A tree decomposition based approach to solve structured SAT instances. In *ICTAI*, pages 115–122, 2009.

[14] P. Hansen, N. Mladenovic, and D. Perez-Brito. Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4) :335–350, 2001.

[15] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995.

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

[17] M. Kitching and F. Bacchus. Exploiting decomposition on constraint problems with high tree-width. In *IJCAI*, pages 525–531, 2009.

[18] S. Loudni and P. Boizumault. Combining VNS with constraint programming for solving anytime optimization problems. *EJOR*, 191(3) :705–735, 2008.

[19] S. Loudni, M. Fontaine, and P. Boizumault. Exploiting separators for guiding VNS. *Electronic Notes in Discrete Mathematics*, 39 :265 – 272, 2012.

[20] R. Marinescu and R. Dechter. AND/OR branch-and-bound search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17) :1457–1491, 2009.

[21] I. Rish and R. Dechter. Resolution versus search : Two strategies for SAT. *J. Autom. Reasoning*, 24(1/2) :225–275, 2000.

[22] N. Robertson and P.D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3) :309–322, 1986.

[23] M. Sánchez, D. Allouche, S. de Givry, and T. Schiex. Russian doll search with tree decomposition. In *IJCAI*, pages 603–608, 2009.

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

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

[26] H. van Benthem. GRAPH : Generating radiolink frequency assignment problems heuristically, 1995.

Un Algorithme Optimal de Filtrage pour Contraintes Table

Jean-Baptiste Mairy[1] **Pascal Van Hentenryck**[2] **Yves Deville**[1]

[1] ICTEAM, Université catholique de Louvain, Belgique

[2] Optimization Research Group, NICTA, University of Melbourne, Australia

{jean-baptiste.mairy,yves.deville}@uclouvain.be pvh@nicta.com.au

Résumé

Les algorithmes de filtrage pour les contraintes table sont généralement basés sur les contraintes, ce qui signifie que la queue de propagation ne contient que des informations sur les contraintes qui doivent être reconsidérées. Cet article [1] propose quatre algorithmes efficaces pour des contraintes table basés sur les valeurs, ce qui signifie que la queue de propagation contient également des informations sur les valeurs retirées. L'un de ces algorithmes (AC5TC-Tr) a une complexité en temps prouvée optimale de $O(r \cdot t + r \cdot d)$ par contrainte table. Des résultats expérimentaux montrent que, sauf sur les instances purement aléatoires, tous nos algorithmes sont deux ou trois fois plus rapides que l'état de l'art (STR2+ et MDDC).

1 Introduction

Les algorithmes de domaine consistance peuvent généralement être classées comme basé sur les contraintes (la queue de propagation ne contient que des informations sur les contraintes qui doivent être reconsidérées) ou basée sur les valeurs (les valeurs retirées des domaines sont également stockées dans la queue de propagation). Pour les contraintes tables, qui ont fait l'objet de nombreuses recherches au cours de ces dernières années, tous les algorithmes existants avec une implémentation (sauf dans [14]) sont basés sur les contraintes. Ce papier propose quatre algorithmes originaux basés sur les valeurs pour contraintes table, qui sont touts des instances de l'algorithme générique AC5. Les propagateurs proposées dans ce papier maintiennent, pour chaque valeur des variables, l'index de son premier support valide dans la table. Ils utilisent également, pour chaque variable et chaque tuple, l'indice du tuple suivant ayant la même valeur pour cette variable. Les algorithmes diffèrent dans leur utilisation de l'information sur la validité des tuples. Trois des algorithmes proposés ont une complexité en temps de $O(r^2 \cdot t + r \cdot d)$ par contrainte table et l'un d'eux (AC5TC-Tr) a une complexité temporelle optimale de $O(r \cdot t + r \cdot d)$, où r est l'arité de la table, d la taille du plus grand domaine et t le nombre de tuples de la table. Un des algorithmes proposés, AC5TC-Recomp, est le propagateur table (non publié) du système Comet.

Les résultats expérimentaux montrent que, sur les instances non-aléatoires, nos algorithmes améliorent les propagateurs état de l'art STR2+ [11] et MDDC [3]. Notre algorithme (théoriquement) optimal n'est pas toujours le plus rapide en pratique. Fait intéressant, sur les tables purement aléatoires, nos algorithmes ne sont pas compétitifs par rapport à STR2+ et MDDC. Comme la plupart des problèmes réels ne sont pas purement aléatoires, nous pensons que nos algorithmes représentent une contribution intéressante. Le reste de cet article est organisé comme suit. La section 2 présente le contexte et travaux apparentés. La section 3 décrit les deux premiers propagateurs pour contraintes table. La section 4 présente notre propagateur optimal, tandis que la section 5 propose une variante efficace de nos premiers algorithmes. La Section 6 décrit les résultats expérimentaux.

1. Ce papier a été publié à CP2012, octobre 2012 à Québec City

2 Contexte

Un CSP $(X, D(X), C)$ se compose d'un ensemble de n variables $X = \{x_1, \ldots, x_n\}$, d'un ensemble de domaines $D(X) = \{D(x_1), \ldots, D(x_n)\}$ où $D(x)$ est l'ensemble des valeurs possibles pour la variable x, et d'un ensemble de contraintes $C = \{c_1, \ldots, c_e\}$, où $Vars(c_i) \subseteq X (1 \leq i \leq e)$. Nous avons pour convention que $d = max_{1 \leq i \leq n}(\#D(x_i))$ et $D(X)_{x_i=a}$ est l'ensemble des tuples $\mathbf{v}$ dans $D(X)$ où $\mathbf{v}_i = a$. Etant donné $Y = \{x_1, \ldots, x_k\} \subseteq X$, l'ensemble des tuples dans $D(x_1) \times \cdots \times D(x_k)$ est noté $D(X)[Y]$ ou simplement $D(Y)$. Un support dans une contrainte c pour une paire variable-valeur (x, a) est un tuple $\mathbf{v} \in D(Vars(c))$ tel que $c(\mathbf{v})$ et $\mathbf{v}[x] = a$. Les ensembles suivants (inconsistants et consistants) sont utiles pour spécifier la domaine consistance et les algorithmes de propagation. Soit c une contrainte d'un CSP $(X, D(X), C)$ avec $y \in Vars(c)$ et $B(X)$ un domaine quelconque.

$$\begin{aligned} Inc(c, B(X)) &= \{(x,a) |\ x \in Vars(c) \wedge a \in D(x) \wedge \\ &\qquad \forall \mathbf{v} \in B(Vars(c))_{x=a} : \neg c(\mathbf{v})\} \\ Cons(c, y, b) &= \{(x,a) | x \in Vars(c) \wedge\ a \in D(x) \wedge \\ &\qquad \exists \mathbf{v} : \mathbf{v}[x] = a \wedge \mathbf{v}[y] = b \wedge c(\mathbf{v})\} \\ Inc(c) &= Inc(c, D(X)) \end{aligned}$$

Une contrainte c dans un CSP $(X, D(X), C)$ est *domaine consistante* ssi $Inc(c) = \emptyset$. Un CSP $(X, D(X), C)$ est *domaine consistant* ssi toutes ses contraintes sont domaine consistantes.

Contraintes Table. Étant donné un ensemble de tuples T d'arité r, une contrainte table c sur T est respectée si $(x_1, \ldots, x_r) \in T$. La taille t d'une contrainte table c est le nombre de lignes de sa table, noté *c.length*. Nous supposons un ordre implicite des tuples : $\sigma_{c,i}$ désigne le i^{eme} élément de la table de c et $\sigma_{c,i}[x]$ est la valeur de $\sigma_{c,i}$ pour la variable x. Nous introduisons une valeur maximale $\top$ (resp. valeur minimale $\bot$) supérieure (resp. inférieure) à toute autre valeur. Nous introduisons également un tuple universel $\sigma_{c,\top}$, avec $\sigma_{c,\top}[x] = *$ pour tout $x \in X$ et postulons par abus de notation que $\forall a \in D(x), * = a$. Cela implique que, pour toute table T, $\sigma_{c,\top} \in T$. Pour une contrainte table, nous disons qu'un tuple σ est autorisée si il appartient à la table. Un tuple σ est valide si toutes ses valeurs appartiennent au domaine des variables correspondantes. Pour atteindre la domaine consistance, il faut au moins vérifier la validité de chaque tuple de la table et, dans le pire des cas, supprimer toutes les valeurs des domaines. Un algorithme de domaine consistance a donc une complexité temporelle en $\Omega(r \cdot t + r \cdot d)$ par contrainte table dans le pire des cas. Un algorithme de type AC5 avec une complexité en $O(r \cdot t + r \cdot d)$ par contrainte table est donc optimal. Comme toujours pour ces algorithmes, si un algorithme de domaine consistance a une complexité temporelle en $O(f)$, alors la complexité temporelle globale des exécutions de cet algorithme le long de tout chemin dans l'arbre de recherche est également en $O(f)$.

Travaux Apparentés. Beaucoup d'efforts de recherche ont été consacrés aux contraintes table. Les propagateurs existants peuvent être classés en 3 catégories : basés sur les indexes, basés sur la compression, et basés sur une table dynamique. Les approches basées sur les indices utilisent une indexation de la table pour accélérer son parcours. Des exemples de tels propagateurs sont GAC3-allowed, et autres variantes basées sur les contraintes (GAC3$_{rm}$-allowed, GAC2001-allowed) [10, 1, 12, 6]. Pour chaque littéral (x, a), la structure de données d'index a un tableau des indices de tuples où x prend la valeur a. La complexité spatiale de la structure de données est $O(r \cdot t)$. La complexité temporelle de GAC3-allowed est en $O(r^3 \cdot d \cdot t + r \cdot d^2)$ par contrainte table. GAC2001-allowed a une complexité temporelle de $O(r^3 \cdot d \cdot t + r^2 \cdot t)$ par contrainte table. L'indexation peut également être utilisée dans les propagateurs basés sur les valeurs. Dans [14], les auteurs proposent un propagateur basé sur les valeurs pour contraintes table implémentant GAC6. Il utilise une structure indexant, pour chaque paire variable-valeur (x, a), et chaque tuple, le tuple suivant dans la table avec a comme valeur pour x. La complexité spatiale de cette structure de données est en $O(r \cdot d \cdot t)$. Cette complexité spatiale peut être réduite en utilisant une structure de données appelée hologramme [13]. Un autre type d'index, proposé dans [7], indexe, pour chaque tuple et variable, le prochain tuple de la table ayant une valeur différente pour la variable. Les propagateurs basés sur la compression comprimment la table sous une forme qui permet de la traverser rapidement. L'une de ces formes comprimées utilise un trie pour chaque variable [7]. Un autre exemple de propagateurs utilisant la compression [3, 2] utilise un Diagrame de Décision Multi-Valué (MDD) pour représenter la table de manière plus efficace. Au cours de la propagation, les tries ou les MDD sont parcourus en utilisant les domaines courants pour effectuer le pruning. Ces algorithmes sont basés sur les contraintes et ont une complexité tmporelle de $O(r^2 \cdot d \cdot t)$ par contrainte table. Compression et parcours rapide peuvent aussi être obtenus en utilisant des tuples compressés, représentant chacun un ensemble de tuples [8, 17]. Les propagateurs basés sur des tables dynamiques maintiennent la table de la contrainte en en supprimant les tuples invalides.

L'algorithme STR [18] et sa version améliorée, STR2 [11], sont basés sur les contraintes et parcourent uniquement les tuples précédemment valides pour extraire de la table les valeurs domaine consistantes. La complexité temporelle de STR2 est en $O(r^2 \cdot d^2 + r^2 \cdot d \cdot t)$ par contrainte table. Le propagateur pour contrainte table de *or-tool* [16] maintient également un tableau dynamique. Il utilise un bitset sur les tuples de la table afin de maintenir leur validité. Un bitset par littéral (x, a) est également utilisé pour faciliter l'accès aux tuples avec a comme valeur pour x. Ce propagateur a une complexité temporelle en $O(r \cdot d \cdot t)$ par contrainte table.

L'Algorithme AC5. AC5 [20, 4] est un algorithme générique de domaine consistance basé sur les valeurs. Dans une approche basée sur les valeurs, de l'information sur les valeurs supprimées sont également stockées dans la queue de propagation. La Spécification 1 décrit les méthodes principales d'AC5, qui utilisent une queue Q de propagation contenant des triplets (c, x, a), indiquant que la domaine consistance de la contrainte c doit être reconsidérée car la valeur a a été retirée de $D(x)$. Lorsqu'une valeur est supprimée dans un domaine, la méthode `enqueue` met dans la queue de propagation les informations nécessaires. Dans la post-condition, Q_o représente la valeur de Q au moment de l'appel. La méthode `post`$(c, \triangle)$ est appelée une fois lors du post de la contrainte. Elle calcule les valeurs inconsistantes de la contrainte c et initialise les structures de données spécifiques nécessaires pour la propagation. Tant que (c, x, a) est présent dans la queue de propagation, il est algorithmiquement souhaitable de considérer que la valeur a est toujours dans $D(x)$ du point de vue de la contrainte c. Ce concept est capturé par la définition suivante.

Définition 1 *La vue locale d'un domaine $D(x)$ par rapport à une queue de propagation Q pour une contrainte c est définie comme $D(x, Q, c) = D(x) \cup \{a | (c, x, a) \in Q\}$.*

Pour des contraintes table, un triplet σ est Q-valide si toutes ses valeurs appartiennent à $D(X, Q, C)$. La méthode centrale de AC5 est la méthode `valRemove`, où l'ensemble $\triangle$ est l'ensemble des valeurs devenant inconsistentes parce que b est supprimé de $D(y)$. Dans la spécification, b est une valeur qui n'est plus dans $D(y)$ et `valRemove` calcule les valeurs (x, a) qui ne sont plus supportées dans la contrainte c à cause de la suppression de b de $D(y)$. Les valeurs dans la queue sont toujours considérées comme supports potentiels tant que leur suppression n'a pas été actée dans cette contrainte. Le pruning minimal $\triangle_1$ ne traite que les variables et valeurs précédemment suportées par (y, b). Cependant, `valRemove` a la possibilité de réaliser un pruning plus important ($\triangle_2$), ce qui est utile pour les contraintes table.

```
1  enqueue(in x: Variable;in a: Value; inout Q: Queue)
2  // Pre: x ∈ X, a ∉ D(x)
3  // Post: Q = Q0 ∪ {(c, x, a)|c ∈ C, x ∈ Vars(c)}
4  post(in c: Constraint;out △: Set of Values)
5  // Pre: c ∈ C
6  // Post: △ = Inc(c) + initialization of
7  //     specific data structures
8  valRemove(in c: Constraint; in y: Variable;
9            in b: Value; out △: Set of Values)
10 // Pre: c ∈ C, b ∉ D(y, Q, c)
11 // Post: △1 ⊆ △ ⊆ △2
12 //         with △1 = Inc(c, D(X, Q, c)) ∩ Cons(c, y, b)
13 //         and △2 = Inc(c)
```

Specification 1 – Les méthodes enqueue, post, et valRemove de AC5

3 Algorithmes Basés sur les Valeurs Efficaces pour Contraintes Table

Nos approches efficaces basées sur les valeurs utilisent une structure de données FS mémorisant les premiers supports des litéraux. Intuitivement $FS[x, a]$ est l'indice du premier support Q-valide de la paire valeur-variable (x, a). Pour accélérer le parcours du tableau, nos algorithmes utilisent une seconde structure de données appelée *next* qui relie tous les tuples de la table partagent la même valeur pour une variable donnée. La structure de données *next* est sémantiquement équivalente à l'index utilisé dans [12]. La structure de données *next*, illustrée Figure 1, est statique, car elle ne dépend pas du domaine des variables. Cependant, FS doit être trailé durant la recherche.

La méthode `post` de cet algorithme (`postTC`) initialise les structures de données FS et *next* et renvoie l'ensemble des valeurs inconsistantes. Ces valeurs sont celles n'ayant, d'entrée de jeu, pas de support valide dans la contrainte. La méthode `valRemove`(c, y, b) de

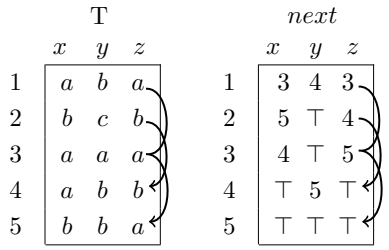


FIGURE 1 – Example pour la structure *next* d'une table T (flèches dessinées uniquement pour la variable z).

cet algorithme (`valRemoveTC`(c, y, b)) ne doit considérer que les tuples dans la chaîne de *next* à partir de $FS[y, b]$. Quand un de ces tuples $\sigma_{c,i}$ est le premier support d'un élément $a = \sigma_{c,i}[x]$, un nouveau support $FS[x, a]$ doit être trouvée. Si un tel support n'existe pas, (x, a) est domaine inconsistant. Sinon, FS est mis à jours. L'ensemble des algorithmes ainsi que des invariants sur les structures de données sont présentés dans [15].

L'algorithme AC5 avec les méthodes `postTC` et `valRemoveTC` pour les contraintes table est appelé AC5TC. AC5TC est générique par rapport aux structures utilisées pour maintenir la Q-validité des tuples de la table. En effet, à la place de tester chaque tuple qui est candidat support, il maintient leur Q-validité. Nous avons défini deux versions différentes de AC5TC. Elles diffèrent dans les structures utilisées. AC5TC-Bool, la première implémentation de AC5TC, utilise un tableau de booléens, $isQValid[i]$ enregistrant la Q-validité du tuple $\sigma_{c,i}$. Ce tableau est trailé car il dépend des domaines. La seconde implémentation, AC5TC-Cutoff, ne trail qu'un nombre entier, en s'appuyant sur une idée de STR et STR2 [18, 11] . Cette implémentation conserve simplement les tuples Q-invalides à la fin de la table, avec un seul entier représentant la frontière entre éléments Q-valides et Q-invalides. Lorsqu'un élément devient Q-invalide, il est échangé avec l'élément au niveau de la frontière et cette frontière est décrémentée d'une unité. La variable de taille doit être trailée mais le tableau n'a pas besoin de l'être : les éléments valides sont automatiquement rétablis en cas de backtrack, mais à une position différente dans la table. Cela est parfois appelé backtrack sémantique [19]. Au lieu d'échanger tuples, notre implémentation utilise deux tableaux Map et Dyn qui donnent la position virtuelle des tuples et les positions des tuples virtuels dans la table. Ces tableaux ne doivent pas être trailés.

Proposition 1 *AC5TC-Bool et AC5TC-Cutoff sont corrects et ont une complexité temporelle en* $O(r^2 \cdot t + r \cdot d)$ *par contrainte table.*

Tous les algorithmes et preuves de complexités peuvent être trouvés dans [15].

4 Un Propagateur Optimal

Dans les propagateurs de la section 3, la recherche d'un nouveau support pour un littéral peut revisiter certains tuples Q-invalides plusieurs fois, parce que la structure de données *next* est statique. L'idée, pour éviter ces visites inutiles, est de rendre la structure *next* dynamique et de toujours veiller à ce que l'élément suivant un élément Q-valide dans une chaîne *next* soit aussi Q-valide. Cela permet d'éviter de considérer à tort des tuples Q-invalides comme supports potentiels. Cette structure est implémentée avec une liste doublement chainée et appelée $nextTr$. Le propagateur utilisant cette structure de donnée (avec la structure FS de la section 3) est appelé AC5TC-Tr (pour AC5TC avec structure Trailée).

La méthode `post` de AC5TC-Tr (`postTC-Tr`) initialise la nouvelle structure $nextTr$ en parallèle de l'initialisation de FS. Comme précédemment, les valeurs n'ayant pas de support valide dans la contrainte sont retirées des domaines. La méthode `valRemove`(c, y, b) de AC5TC-Tr (`valRemoveTC-Tr`(c, y, b)), comme `valRemoveTC`, ne doit considérer que les tuples dans la chaîne de $nextTr$ à partir de $FS[y, b]$. À la différence que `valRemoveTC-Tr` ne doit jamais chercher un support (en parcourant une chaîne) vu que les éléments suivants dans les chaînes $nextTr$ sont nécessairement Q-valides. Toutefois, $nextTr$ doit être mis à jour par `valRemoveTC-Tr` afin de s'assurer que les tuples nouvellement Q-invalides ne soient plus dans les chaînes. Les éléments retirés des chaînes ne seront donc jamais revisités.

Proposition 2 *AC5TC-Tr est correct et a une complexité temporelle optimale en* $O(r \cdot t + r \cdot d)$ *par contrainte table.*

5 Une Variation basée sur du Recalcul

Nous proposons, dans cette Section, une variante de l'algorithme AC5TC, appelée AC5TC-Recomp, qui ne nécessite pas de structure de données pour maintenir la Q-validité des tuples. AC5TC-Recomp est le propagateur de contraintes table (non publié) du solver Comet. Il remplace les tests de Q-validité par un test de validité. Tester la validité d'un tuple correspond simplement à vérifier, pour toutes les valeurs du scope de la contrainte, que leur valeur est dans leur domaine.

La méthode valRemove de AC5TC-Recomp exploite la flexibilité de ses spécifications en calculant un ensemble $\triangle$ entre $\triangle_1$ et $\triangle_2$. Cela est du a son utilisation de la validité des tuples à la place de la Q-validité.

Proposition 3 *AC5TC-Recomp est correct et a une complexité temporelle en* $O(r^2 \cdot t + r \cdot d)$ *par contrainte table.*

Malgré sa sous-optimalité, AC5TC-Recomp améliore les algorithmes de l'état de l'art sur certaines classes de problèmes.

6 Résultats Exmpérimentaux

Tous les algorithmes proposés dans ce papier ont été implémentés en user-space en Comet, et AC5TC-Recomp y a été également réimplémenté. Pour comparaison, les algorithmes basés sur les contraintes classiques ont également été implémentés dans le user-space Comet. L'algorithme GAC3-allowed a été choisi parce qu'il est l'algorithme standard GAC3 pour les contraintes table [10]. Les deux méthodes à l'état de l'art ont également été ré-implémentées : l'algorithme STR2+ de [11] et l'algorithme de *mddc* [3]. Ils sont appelés respectivement STR et MDD dans les résultats expérimentaux. Toutes les expériences ont été menées sur un processeur Intel Xeon 2,53 GHz, utilisant Comet 2.1.1.

Les algorithmes sont comparés au sein d'une recherche où la domaine consistance est maintenue (MAC). Cette section présente les résultats expérimentaux sur des instances entièrement aléatoires et sur le problème du voyageur de commerce. Des résultats expérimentaux étendus sont présentés dans [15].

Pour chaque ensemble d'instances, nous reportons les temps moyens d'exécution en secondes (*totTime*), le temps moyen pour poster les contraintes (*postT*), le nombre moyen d'appels au propagateur (*nProp*), le pourcentage par rapport au meilleur propagateur pour le temps d'exécution (*% best*), la moyenne du pourcentage en temps par rapport au meilleur propagateur (μ*% best*), le nombre moyen de tests de validité (*valChk*), de tests de Q-validité (*QvalChk*), et le nombre de pointeurs suivis (*pFollow*). La différence entre le *% best* et μ*% best* la suivante : pour *% best*, les moyennes de temps d'exécution sont calculées avant de calculer *% best*. Il y a donc un propagateur qui est le meilleur sur chaque ensemble d'instances. Pour μ*% best*, les pourcentages sont calculés instance par instance et agrégés avec une moyenne géométrique à la fin. Cette mesure tient compte du fait que différentes instances peuvent donner lieu à différents meilleurs propagateurs. La mesure μ*% best* utilise une moyenne géométrique comme suggéré dans [5]. La dernière quantité, *pFollow*, a différentes significations pour différents algorithmes. Pour GAC3-allowed, elle correspond au nombre de fois où l'algorithme accède aux tuples. Pour les algorithmes AC5TC, elle est définie comme le nombre de fois où les structures *next* ou *nextTr* servent à traverser la table. Pour MDD, elle correspond au nombre d'arêtes suivies dans la structure MDD. Bien que se référant à des quantités différentes, *pFollow* est utile pour comparer le comportement des propagateurs, car cette mesure reflète l'utilisation de leurs structures spécifiques.

Instances Aléatoires Cet ensemble de test contient des contraintes table aléatoires. Les instances sont générées par le modèle RD [21]. Les paramètres sont choisis de manière à générer des instances dans la transition de phase, en utilisant les théorèmes 1 et 2 de [21]. Les instances ont 10 variables, une taille de domaine uniforme de 10 et 15 contraintes d'arité 5. Le nombre attendu de tuples dans chaque table est donc de 20000. Cet ensemble test contient 10 instances générées avec ces paramètres. La stratégie de recherche utilisée est minDom avec ordonnancement de valeurs lexicographique.

La Table 1 résume les résultats. Des résultats similaires sont observés pour d'autres valeurs de paramètres dans la transition de phase. Les propagateurs standard STR2 et MDD surpassent nos propagateurs basés sur les valeurs. Observons le grand nombre de tests de validité réalisés par AC5TC-Recomp, de tests de Q-validité de AC5TC-Bool et AC5TC-Cutoff, ainsi que le nombre de fois où ils suivent un de leurs pointeurs. AC5TC-Tr, le meilleur de nos propagateurs, suit beaucoup moins de pointeurs que nos autres propagateurs, car il ne suit aucun pointeur vers un tuple qui a déjà été inspecté. En raison de l'absence de structure dans l'ensemble de contraintes, les trois premiers propagateurs AC5TC semblent vérifier beaucoup de fois les mêmes tuples. En outre, ces instances aléatoires ont de grandes tables, ce qui rend le coût de la structure *nextTr* de AC5TC-Tr, qui doit être trailée, trop élevé.

Problème du Voyageur de Commerce Nous concluons les expériences avec les résultats des propagateurs sur une version satisfaction du problème du voyageur de commerce (TSP). Nous avons utilisé les ensembles d' instances *tsp-20* et *tsp-25* [9]. Ces instances sont composées de contraintes table très différentes. Leur arité varie entre 2 et 3 et elles peuvent compter jusqu'à 20 000 tuples mais également seulement 20. Les variables ont également des domaines assez différents : certaines ont des petits domaines, tandis que d'autres ont des domaines contenant jusqu'à 1000 valeurs. Il y a 61 variables et 230 contraintes table dans les instances *tsp-20*. Les instances *tsp-25* comptent 76 variables et 350 contraintes. Les contraintes table négatives trouvées dans ces instances ont été transformées en contraintes positives. La stratégie de recherche utilisée est dom/deg avec ordonnancement de valeur lexicographique.

Les Tables 2 et 3 présentent les résultats expérimentaux sur les instances TSP. Nous observons d'abord que STR2 et MDD obtiennent de moins bonnes performances que nos propagateurs. AC5TC-Recomp est la stratégie gagnante sur les instances *tsp-20* alors que

AC5TC-Tr est la plus rapide sur les instances *tsp-25*. Les dernières instances sont plus difficiles. Nous pouvons également voir que la vérification de la validité à la place de la Q-validité permet à AC5TC-Recomp de suivre moins de pointeurs et d'effectuer moins de tests de validité que de Q-validité pour AC5TC-Bool et AC5TC-Cutoff. En outre, sur ces instances, la petite arité rend le test de validité ($O(r)$) peu cher par rapport au test de Q-validité. Encore une fois, sur ces instances, les structures trailées de AC5TC-Cutoff, plus légères, le rendent plus rapide que AC5TC-Bool.

Lorsque nous fusionnons tables binaires des instances *tsp-20* en tables d'arités plus élevées, nous observons que AC5TC-Tr, notre algorithme optimal, permet de résoudre plus d'instances que STR2 lorsqu'un timeout de 20 minutes est imposé, avec un temps d'exécution total plus petit sur les instances résolues par les deux propagateurs. MDD est loin derrière sur ces instances. Sur les instances simples, STR2 est plus efficace que AC5TC-Recomp qui est lui même plus efficace que AC5TC-Tr. Cependant, sur les instances difficiles, l'optimalité de AC5TC-Tr paye et il devient le meilleur algorithme.

Résumé Nous concluons que, pour les instances totalement aléatoires, le manque de structure dans les tables empêche nos propagateurs de rivaliser avec les algorithmes de l'état de l'art. Cependant, pour les instances non aléatoires, nos propagateurs sont plus rapides. Globalement, AC5TC-Bool et AC5TC-Cutoff sont plus lents que AC5TC-Recomp car ils testent la Q-validité, non la validité, et donc ils effectuent de plus petits sauts dans la table. En outre, le maintien de leurs structures de données est coûteux. Seul notre propagateur optimal AC5TC-Tr surpasse AC5TC-Recomp sur les instances difficiles tout en utilisant la Q-validité. Cependant, sur les instances faciles, le coût de sa structure de données trailée $nextTr$ le rend plus lent que AC5TC-Recomp.

7 Conclusion

Ce papier propose quatre algorithmes de domaine consistance basés sur les valeurs pour les contraintes table, utilisant le cadre générique AC5. Ces nouveaux propagateurs maintiennent , pour chaque paire valeur-variable, l'indice de son premier support Q-valide dans le tableau. Ils utilisent également, pour chaque variable d'un tuple, l'indice de la prochaine ligne partageant la même valeur pour cette variable. Ils diffèrent par leur utilisation de l'information sur la validité des tuples. AC5TC-Tr et AC5TC-Recomp sont les deux meilleurs algorithmes : AC5TC-Recomp ne conserve pas d'information de Q-validité et la recalcule sur demande ; AC5TC-Tr incorpore les informations de Q-validité directement dans la structure d'indexe, en évitant les visites inutiles de tuples invalides et conduisant à un algorithme optimal avec une complexité temporelle en $O(r \cdot t + r \cdot d)$ par contrainte table. Nos autres algorithmes ont une complexité en temps en $O(r^2 \cdot t + r \cdot d)$ par contrainte table. Les résultats expérimentaux montrent que, sur des contraintes table purement aléatoires, nos algorithmes ne sont pas compétitifs par rapport aux propagateurs à l'état de l'art STR2+ et *mddc*. Sur instances non-aléatoires, nos propagateurs surpassent STR2+ et *mddc*.

Références

[1] Christian Bessière and Jean-Charles Régin. Arc consistency for general constraint networks : Preliminary results. In *IJCAI (1)*, pages 398–404, 1997.

[2] Mats Carlsson. Filtering for the case constraint. Talk given at the advanced school on global constraints, 2006.

[3] Kenil Cheng and Roland Yap. An mdd-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints. *Constraints*, 15 :265–304, 2010.

[4] Yves Deville and Pascal Van Hentenryck. Domain consistency with forbidden values. In David Cohen, editor, *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, volume 6308 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2010.

[5] Philip J. Fleming and John J. Wallace. How not to lie with statistics : the correct way to summarize benchmark results. *Commun. ACM*, 29(3) :218–221, March 1986.

[6] Ian P. Gent, Chris Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In *Proceedings of CP2006*, pages 182–197. Springer-Verlag, 2006.

[7] Ian P. Gent, Chris Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *Proceedings of the Twenty Second Conference on Artificial Intelligence*, pages 191–197. AAAI Press, 2007.

[8] George Katsirelos and Toby Walsh. A compression algorithm for large arity extensional

constraints. In *Proceedings of the 13th international conference on Principles and practice of constraint programming*, pages 379–393. Springer-Verlag, 2007.

[9] Ch. Lecoutre. *Instances of the Constraint Solver Competition*. http ://www.cril.fr/∼lecoutre/.

[10] Christophe Lecoutre. *Constraint Networks : Techniques and Algorithms*. ISTE/Wiley, 2009.

[11] Christophe Lecoutre. Str2 : optimized simple tabular reduction for table constraints. *Constraints*, 16 :341–371, 2011.

[12] Christophe Lecoutre and Radoslaw Szymanek. Generalized arc consistency for positive table constraints. In *Proceedings of CP 06*, pages 284–298, 2006.

[13] Olivier Lhomme. Arc-consistency filtering algorithms for logical combinations of constraints. In *CPAIOR*, pages 209–224, 2004.

[14] Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In *Proceedings of the Nationnal Conference on Artificial Intelligence*, pages 405–410. AAAI Press, 2005.

[15] Jean-Baptiste Mairy, Pascal Van Hentenryck, and Yves Deville. An optimal filtering algorithm for table constraints. In *18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*. Lecture Notes in Computer Science, Springer, 2012.

[16] Laurent Perron and Vincent Furnon. or-tools. http :// code.google.com/p/or-tools.

[17] Jean-Charles Régin. Improving the expressiveness of table constraints. In *Proceedings of workshop ModRef 11 at CP 11*, 2011.

[18] Julian R. Ullmann. Partition search for non-binary constraint satisfaction. *Inf. Sci.*, 177(18) :3639–3678, 2007.

[19] P. Van Hentenryck and V. Ramachandran. Backtracking without Trailing in CLP($\Re_{lin}$). *ACM Transactions on Programming Languages and Systems*, 17(4) :635–671, July 1995.

[20] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artif. Intell.*, 57(2-3) :291–321, 1992.

[21] Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre. Random constraint satisfaction : Easy generation of hard (satisfiable) instances. *Artif. Intell.*, 171(8-9) :514–534, 2007.

propagator	totTime	postT	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	3 000	1.5	614 k	2 725	2 660	523 M	0	523 M
AC5TC-Bool	4 636	1.0	2.8 M	4 211	4 070	19 k	257 M	481 M
AC5TC-CutOff	3 991	0.8	2.8 M	3 626	3 538	19 k	257 M	481 M
AC5TC-Tr	994	5.2	2.8 M	903	930	19 k	0	16 M
AC5TC-Recomp	3 874	0.8	2.4 M	3 519	3 357	98 M	0	305 M
STR2	483	0.7	614 k	439	455	22 M	0	0
MDD	**110**	12.4	614 k	100	100	0	0	12 M

TABLE 1 – Résultats des propagateurs sur les instances aléatoires (temps en secondes)

propagator	totTime	postT	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	797	1.7	6.7 M	733	587	11 M	0	11 M
AC5TC-Bool	186	0.8	21.2 M	171	187	2 k	1 M	2 M
AC5TC-CutOff	153	0.5	21.2 M	141	144	2 k	1 M	2 M
AC5TC-Tr	120	3.3	21.2 M	111	164	2 k	0	466 k
AC5TC-Recomp	**109**	0.3	20.9 M	100	104	391 k	0	1 M
STR2	398	1.4	6.7 M	366	353	803 k	0	0
MDD	456	19.0	6.7 M	419	769	0	0	7 M

TABLE 2 – Résultats des propagateurs sur les instances *tsp-20* (temps en secondes)

propagator	totTime	postT	nProp	%best	μ%best	valChk	QvalChk	pFollow
GAC3-Allowed	6 607	2.4	73 M	606	509	23 M	0	23 M
AC5TC-Bool	2 625	1.3	198 M	241	233	2 k	11 M	19 M
AC5TC-CutOff	1 937	0.7	198 M	178	175	2 k	11 M	19 M
AC5TC-Tr	**1 089**	5.2	198 M	100	100	2 k	0	3 M
AC5TC-Recomp	1 315	0.5	196 M	121	120	3 M	0	10 M
STR2	3 740	2.9	73 M	343	333	5 M	0	0
MDD	4 974	25.2	73 M	457	425	0	0	28 M

TABLE 3 – Résultats des propagateurs sur les instances *tsp-25* (temps en secondes)

Exploiter les séparateurs : une approche hybride combinant recherche locale et arborescente *

Jean-Philippe Métivier **Mathieu Fontaine** **Samir Loudni**

Université de Caen Basse-Normandie – GREYC – (CNRS UMR 6072)
Campus II – Côte de Nacre – Boulevard du Maréchal Juin 14000 Caen – France
prenom.nom@unicaen.fr

Résumé

La décomposition d'un graphe de contraintes permet de découper un problème en clusters (sous-ensembles de variables). Les variables partagées par les clusters sont appelées variables séparateurs. Celles-ci jouent un rôle majeur dans la phase de résolution. En effet, si toutes les variables des séparateurs sont instanciées alors chaque cluster peut être traité de façon indépendante. Dans ce papier, nous montrons comment exploiter ce principe dans les méthodes de recherche locale à voisinages larges. Notre approche procède en deux temps. Tout d'abord, on sélectionne un sous-ensemble de variables séparateurs, puis on les réaffectent de façon gloutonne. Ensuite, on reconstruit l'ensemble des clusters contenant ces variables séparateurs à l'aide de méthodes arborescentes. Les expérimentations menées sur des instances réelles (CELAR et SPOT5) montrent la pertinence et l'efficacité de notre approche.

1 Introduction

Un grand nombre de problèmes réels, tels que les problèmes d'affectations de fréquence [2] ou la planification de satellites d'observation [1], sont de très grande taille et exhibent un graphe de contraintes très structuré.

La décomposition, proposée par Robertson et Seymour [12], vise à découper un problème en sous-problèmes (clusters) constituant un graphe acyclique. Chaque cluster représente un ensemble de variables fortement connexes. Chaque sous-problème étant plus petit que le problème d'origine est plus facile à résoudre. Les variables partagées par plusieurs clusters sont appelées séparateurs. Ces séparateurs jouent un rôle charnière dans la résolution des sous-problèmes associés. En effet, toute instanciation d'un séparateur induit plusieurs sous-problèmes indépendants. Cette propriété est au coeur des méthodes complètes exploitant une décomposition arborescente comme `RDS-BTD` [13] ou `AND/OR graph search` [8].

Dans les méthodes de recherche locale qui utilisent de grands voisinages tels que *Large Neighborhood Search* (`LNS`) [14], *Propagation Guided Large Neighborhood Search* (`PGLNS`) [11] ou encore *Variable Neighborhood Search* (`VNS`) [9], la définition de structures de voisinages pertinentes est une étape cruciale, car elle permet d'intensifier et de diversifier la recherche dans différentes régions de l'espace de recherche. À notre connaissance, il n'existe aucun travail exploitant les séparateurs pour identifier des structures de voisinages.

Dans ce papier, nous montrons comment les séparateurs peuvent être utilisés afin de guider une recherche à grand voisinage comme `VNS`. Les expérimentations menées sur des instances réelles (CELAR et SPOT5) montrent que l'utilisation des séparateurs procure des performances bien meilleures comparé à `VNS/LDS+CP` [7] ou `ID-Walk` [10].

La section 2 présente le contexte de nos travaux. La section 3 montre comment exploiter les séparateurs dans une méthode à grands voisinages. Les sections 4 et 5 présentent les résultats expérimentaux. Enfin, nous concluons et présentons des perspectives.

2 Définition et notations

2.1 Réseau de fonctions de coûts

Un réseau de fonctions de coûts est une paire (X,W) où $X=\{1,..,n\}$ est un ensemble de n variables et de W est un ensemble de fonctions de coûts. Chaque variable

*Ce travail a été soutenu par l'Agence Nationale de la Recherche, référence ANR-10-BLA-0214

Algorithme 1 : VNS/LDS+CP

```
1  fonction VNS/LDS+CP(X,W,k_init,k_max)
2  début
3      S ← genInitSol();
4      k ← k_init;
5      tant que k ≤ k_max faire
6          X_unassign ← Hneighborhood(X,N_k,S);
7          A ← S \ X_unassign;
8          S' ← LDS(A);
9          si f(S') < f(S) alors
10             S ← S';
11             k ← k_init;
12         sinon k ← k + 1;
13     retourner S;
14 fin
```

$X_i \in X$ a un domaine fini D_i de valeurs qui peuvent lui être affectées. Pour un sous-ensemble de variables $S \subseteq X$, on note D^S le produit cartésien des domaines des variables de S. Pour un n-uplet t donné, $t[S]$ représente la projection de t sur l'ensemble de variables S. Une fonction de coûts $w_s \in W$, de portée $S \subseteq X$, est une fonction $w_s : D^s \rightarrow [0, k_\top]$ où $k_\top$ est un coût entier maximum représentant les affectations interdites. Les coûts sont combinés par l'addition bornée $\oplus$, définie par $\alpha \oplus \beta = \min(\alpha+\beta, k_\top)$.

Résoudre un réseau de fonctions de coûts consiste à trouver une affectation complète t de l'ensemble des variables minimisant la combinaison des fonctions de coûts $\bigoplus_{w_s \in W} w_s(t[S])$.

2.2 Décomposition

Un réseau de fonctions de coûts peut être vu comme un graphe $G=(X,W)$ composé d'un sommet par variable et d'une arête pour chaque fonction de coûts (reliant les variables de sa portée).

Définition 1 *Une **décomposition** d'un réseau de fonctions de coûts consiste à former un ensemble de clusters $C=\{C_1,..,C_m\}$ tel que :*
- $\bigcup_{i \in [1..m]} C_i = X$,
- $\forall (u,v) \in W$, $\exists C_i \in C$ *t.q.* $u \in C_i$ *et* $v \in C_i$.

Définition 2 *L'intersection de deux clusters est appelée **séparateur**. Sep désignera l'ensemble des variables séparateurs.*

Définition 3 *Les variables d'un cluster contenues dans aucun séparateur sont appelées **variables propres**. C^i_{pr} désignera l'ensemble des variables propres du cluster C_i.*

Pour des raisons pratiques, nous utiliserons ici une décomposition arborescente obtenue en utilisant l'heuristique MCS [15].

2.3 VNS/LDS+CP

VNS/LDS+CP [7] est une méthode de recherche locale qui exploite le principe de VNDS [4]. Les voisinages sont obtenus en désaffectant heuristiquement une partie de la solution courante. Ensuite, les variables désinstanciées sont réaffectées à l'aide d'une recherche arborescente partielle (LDS [5]) aidée par des mécanismes de propagation basée sur le calcul de minorants.

L'algorithme 1 présente le pseudo-code de VNS/LDS+CP. N_k désigne l'ensemble des sous-ensembles de k variables de X. Un sous-ensemble de k variables est sélectionné dans X, puis réaffecté avec LDS+CP. Si une solution de meilleure qualité S' est trouvée, k est réinitialisé à k_{init}. Sinon, k est incrémenté de 1. La recherche s'arrête quand elle a atteint k_{max} la taille maximale de voisinage.

3 Guider VNS grâce aux séparateurs

Dans cette section, nous présentons notre contribution : sep-VNS qui utilise les séparateurs pour définir de nouveaux de voisinages.

3.1 Choix des variables à désinstancier

Tout comme VNS/LDS+CP, les voisinages sont obtenus en désaffectant une partie de la solution courante. Toutefois, les variables à désinstancier sont gérés différemment afin de tirer parti des séparateurs.

Les variables à désinstancier sont sélectionnées en deux temps. Tout d'abord, k variables séparateurs sont sélectionnées aléatoirement et désinstanciées (algo. 2, ligne 6). Puis les clusters partageant ces k variables sont, à leur tour, désinstanciés (algo. 2, ligne 7).

Les k variables séparateurs désinstanciés permettent de décrire un sous-espace de recherche intéressant. En effet, ces variables constituent une sorte de « résumé » du problème : lorsqu'elles sont instanciées, les différents clusters sous-jacents peuvent être traités individuellement. De ce fait, il est nécessaire de désinstancier les clusters partageant les k variables sélectionnées car toute nouvelle affectation de ces k variables remettra en question la valuation des clusters associés. Notons que les variables appartenant aux clusters ne contenant pas de variables séparateurs désinstanciées sont conservés dans l'instanciation partielle (algo. 2, ligne 9).

3.2 Reconstruction de la solution partielle

La reconstruction est effectuée en deux temps. Les premières variables à réinstancier sont celles des séparateurs, puis les variables propres de chaque cluster désinstancié.

Algorithme 2 : Sep-VNS

```
1  fonction Sep-VNS(X,W,k_init,k_max,C,Sep)
2  début
3      S ← genInitSol() ;
4      k ← k_init ;
5      tant que k ≤ k_max faire
6          Sep_k ← select(k,Sep) ;
7          X_pr ← {X_i | X_i∈C^j_pr, Cj∩Sep_k≠∅ et j∈[1..m]} ;
8          X_unassign ← Sep_k ∪ X_pr ;
9          A ← S \ X_unassign ;
10         pour chaque X_i ∈ Sep_k faire
11             value ← mincost(D_{X_i}) ;
12             A ← A ∪ {X_i=value} ;
13         pour chaque C_j t.q. Cj ∩ Sep_k ≠ ∅ faire
14             A_j ← A[C_j ∩ Sep_k] ;
15             S_j ← DFBB(C_j,A_j) ;
16             A ← A ∪ S_j ;
17         S' ← A ;
18         si f(S') < f(S) alors
19             S ← S' ;
20             k ← k_init ;
21         sinon k ← k + 1 ;
22     retourner S ;
23 fin
```

Reconstruction des variables séparateurs (algo. 2, lignes 10 à 12) : Chaque variable séparateur désinstanciée va être réinstanciée (de façon gloutonne) à sa valeur engendrant le moins de violations. Afin d'éviter d'être bloqué dans un optimum local, une liste taboue dynamique [3] est utilisée pour diversifier les instanciations des variables séparateurs. Cette liste va interdire une instanciation donnée pour les $(0.1 \times \#conflict + rand(5))$ prochaines iterations.

Reconstruction des clusters désinstanciés (algo. 2, lignes 13 à 16) : La reconstruction des clusters désinstanciés ne commence qu'une fois toutes les variables séparateurs instanciées. Chaque cluster est réinstancié de façon individuelle (en ne considérant que les variables et contraintes de ce cluster) et cela à l'aide d'une recherche arborescente. Lorsque le cluster est suffisament petit (moins de 30 variables) DFBB est utilisé, sinon le cluster est reconstruit en utilisant un LDS.

3.3 Accelérer la reconstruction des clusters

La reconstruction des variables séparateurs peut nous conduire à explorer deux fois le même sous-espace de recherche. En effet, si pour un cluster donné, la reconstruction réinstancie les variables séparateurs à des valeurs déjà explorées précédemment, alors la phase de reconstruction des clusters va explorer le même espace de recherche. Afin d'éviter cela, nous avons mis en place un système de *cacher* [6]. Ces cachers enregistrent, pour chaque cluster et instanciation des séparateurs, la meilleure instanciation des variables propres de ce cluster. Ainsi, lors de la reconstruction des variables d'un cluster, si l'affectation courante des variables séparateurs a été déjà examinée, alors cette reconstruction se fait via le cacher plutôt que par une recherche arborescente.

4 Jeux de test

Les expérimentations ont été menées sur les instances de deux problèmes différents.

Instance RLFAP : Le centre d'électronique de l'armement a mis à disposition un ensemble d'instances du problème d'affectation de fréquences radio [2]. Dans ce problème, le but est d'assigner un nombre limité de fréquences à un ensemble de liens radios, et cela en minimisant les interférences dues à la réutilisation d'une fréquence. Nous reportons ici les résultats sur les instances les plus difficiles : Scen06, Scen07 et Scen08.

Instance SPOT5 : La planification quotidienne de satellite d'observation consiste à sélectionner les prises de vue à effectuer durant une journée en tenant compte des limites matérielles du satellite tout en maximisant l'importance des photographies sélectionnées [1]. Nous reportons les résultats de cinq instances sans contraintes dures de capacité.

5 Expérimentations

5.1 Protocole expérimental

Les différents paramètres de Sep-VNS ont été fixés comme suit : k_{min} à 1, k_{max} au nombre total de variables dans les séparateurs, et la valeur de la *discrepancy* à 3.

Un ensemble de 50 essais par instance a été réalisé sur un AMD Opteron 2,1 GHz et 256 Go de RAM. Toutes les méthodes ont été implantées en C++ en utilisant la librairie toulbar2 [1]. Pour ID-Walk, nous avons utilisé la librairie INCOP [10]. Pour chaque instance et chaque méthode, nous reportons le nombre de succès, le temps moyen nécessaire pour atteindre l'optimum (incluant le temps nécessaire à la décomposition), ainsi que le coût moyen des solutions retournées ainsi que le coût de la meilleure solution (entre parenthèses) si l'optimum n'est pas atteint. Nous avons limité le temps d'exécution de chaque méthode à une heure. Les résultats des autres méthodes ont été obtenus dans les mêmes conditions expérimentales que les nôtres.

5.2 Résultats expérimentaux

Le tableau 1 reporte les résultats obtenus sur les instances RLFAP. On peut constater sur ces instances

1. http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP

Instance	Méthode	Succés	Temps	Moyenne
Scen06	Sep-VNS	46/50	1026	3390
Opt=3389	VNS/LDS+CP	15/50	83	3399
	ID-Walk	NA	840	3447 (3389)
Scen07	Sep-VNS	2/50	3233	384065
Opt=343592	VNS/LDS+CP	1/50	461	355982
	ID-Walk	NA	360	373334 (343998)
Scen08	Sep-VNS	0/50	–	318 (269)
Opt=262	VNS/LDS+CP	0/50	–	397 (357)
	ID-Walk	NA	–	291 (267)

Table 1 – Résultats sur les instances RLFAP.

Instance	Méthode	Succés	Temps	Moyenne
408	Sep-VNS	38/50	546	6228
Opt=6228	VNS/LDS+CP	26/50	142	6228
	ID-Walk	50/50	3	6228
412	Sep-VNS	34/50	820	32381
Opt=32381	VNS/LDS+CP	32/50	130	32381
	ID-Walk	10/50	2102	32381
414	Sep-VNS	18/50	1761	38479
Opt=38478	VNS/LDS+CP	12/50	434	38481
	ID-Walk	0/50	–	38481
507	Sep-VNS	23/50	925	27390
Opt=27390	VNS/LDS+CP	11/50	232	27391
	ID-Walk	7/50	1862	27391
509	Sep-VNS	19/50	2370	36447
Opt=36446	VNS/LDS+CP	12/50	598	36448
	ID-Walk	0/50	–	36450

Table 2 – Résultats sur les instances SPOT5.

que notre approche surclasse VNS/LDS+CP et ID-Walk sur les instances Scen06 et Scen07. Sur l'instance Scen06, Sep-VNS obtient un taux de réussite de 92% contre 30% pour VNS/LDS+CP. Sur l'instance Scen08, Sep-VNS obtient des performances bien meilleures que VNS/LDS+CP. En effet, la déviation moyenne à l'optimum est réduite de 51.5% à 21.3% et la meilleure solution est très proche de l'optimum.

Une fois encore, cette tendance se confirme sur les instances SPOT5 (tableau 2), sur lesquelles Sep-VNS permet d'améliorer très nettement le taux de réussite (avec un gain de 55% en moyenne par rapport à VNS/LDS+CP).

Notons toutefois, que notre approche a besoin de plus de temps de calcul en moyenne pour obtenir l'optimum, cela à cause de temps de reconstruction plus important.

6 Conclusions et futurs travaux

Dans cet article, nous avons proposé une approche hybride combinant une recherche locale (pour la sélection et la reconstruction des variables des séparateurs) et une recherche arborescente (pour la reconstruction des clusters). Nous avons aussi montré expérimentalement que l'exploitation des séparateurs permet de dépasser les performances de VNS/LDS+CP et ID-Walk. Nous travaillons actuellement sur deux pistes.

i) La première consiste en l'étude de l'influence de la décomposition sur les performances de notre approche. La taille des clusters et le nombre de variables dans l'union des séparateurs semble jouer un rôle majeur pour notre approche (nombre des voisinages, temps de reconstruction, ...). De plus, nous utilisons une décomposition arborescente, alors que la décomposition n'a pas besoin d'être arborescente dans notre cas.

ii) La seconde piste vise à utiliser de manière plus intelligente les cachers. En effet, actuellement ceux-ci sont simplement utilisés pour éviter de re-explorer un sous-espace de recherche. Pourquoi ne pas utiliser ceux-ci pour guider la reconstruction des variables séparateurs.

Références

[1] E. Bensana, M. Lemaître, and G. Verfaillie. Earth observation satellite management. *Constraints*, 4(3) :293–299, 1999.

[2] B. Cabon, S. de Givry, L. Lobjois, T. Schiex, and J.P. Warners. Radio link frequency assignment. *Constraints*, 4(1) :79–89, 1999.

[3] F. Glover and M. Laguna. *TABU search*. Kluwer, 1999.

[4] P. Hansen, N. Mladenovic, and D. Perez-Brito. Variable neighborhood decomposition search. *Journal of Heuristics*, 7(4) :335–350, 2001.

[5] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995.

[6] M. Kitching and F. Bacchus. Symmetric component caching. In *IJCAI*, pages 118–124, 2007.

[7] S. Loudni and P. Boizumault. Combining vns with constraint programming for solving anytime optimization problems. *European Journal of Operational Research*, 191(3) :705–735, 2008.

[8] R. Marinescu and R. Dechter. And/or branch-and-bound search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17) :1457–1491, 2009.

[9] N. Mladenovic and P. Hansen. Variable neighborhood search. *Computers & OR*, 24(11) :1097–1100, 1997.

[10] B. Neveu and G. Trombettoni. Incop : An open library for incomplete combinatorial optimization. In *CP*, pages 909–913, 2003.

[11] L. Perron, P. Shaw, and V. Furnon. Propagation guided large neighborhood search. In *CP*, pages 468–481, 2004.

[12] N. Robertson and P.D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3) :309–322, 1986.

[13] M. Sánchez, D. Allouche, S. de Givry, and T. Schiex. Russian doll search with tree decomposition. In *IJCAI*, pages 603–608, 2009.

[14] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP*, pages 417–431, 1998.

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

Des Propagateurs Indépendants des Solveurs *

Jean-Noël Monette **Pierre Flener** **Justin Pearson**

Uppsala University, Department of Information Technology, Uppsala, Suède
{jean-noel.monette,pierre.flener,justin.pearson}@it.uu.se

Résumé

Cet article présente une extension des indexicals pour décrire des propagateurs de contraintes globales. Ce nouveau langage produit par compilation des propagateurs pour différents solveurs et est indépendant d'un solveur particulier. De plus, nous montrons que cette description de haut niveau aide à prouver certaines propriétés telles que correction et monotonie. Des résultats expérimentaux montrent que les propagateurs compilés à partir de descriptions par indexicals sont parfois presque aussi rapides que des propagateurs natifs de Gecode. Cela montre que notre langage peut être utilisé pour prototyper facilement de nouvelles contraintes globales.

1 Introduction

L'un des principaux atouts de la programmation par contraintes (CP) est l'existence de nombreux algorithmes de filtrage, appelés propagateurs, conçus spécialement pour les contraintes globales et qui permettent de résoudre efficacement des problèmes combinatoires. Les solveurs CP permettent la définition de nouvelles contraintes et de leurs propagateurs respectifs. Cependant, il peut être fastidieux de mettre en oeuvre un propagateur correct, efficace, se conformant à l'interface du solveur et codé à la main dans le langage d'implémentation du solveur, par exemple en C++.

Dans cet article, nous proposons un langage indépendant des solveurs pour décrire une grande classe de propagateurs. Notre contribution est double.

Tout d'abord, nous facilitons la mise en oeuvre et le partage de propagateurs. Les propagateurs sont décrits de façon concise et sans référence aux détails d'implémentation des solveurs. L'implémentation des propagateurs est générée à partir de leurs descriptions. Cela permet de prototyper rapidement un propagateur pour une nouvelle contrainte pour laquelle il n'existe pas de propagateur natif ou de décomposition (assez bonne). Le propagateur généré peut même servir de base à des raffinements manuels. Cela permet aussi d'intégrer une contrainte existante dans un nouveau solveur, comme chaque solveur peut être équipé de son propre générateur de code à partir du langage de description de propagateurs. Nous pensons qu'un tel langage peut jouer le même rôle pour le partage de propagateurs entre solveurs que des langages de modélisation jouent pour le partage de modèles entre solveurs.

Ensuite, nous facilitons la preuve des propriétés d'un propagateur, telles que la correction et la monotonie. Le niveau d'abstraction plus élevé de notre langage de description de propagateurs nous permet de concevoir des outils d'analyse et de transformation de propagateurs. Cette méthode permet d'appliquer des résultats théoriques aux propagateurs de manière systématique.

Notre approche est basée sur le travail fondateur au sujet des indexicals [28]. En résumé, un indexical définit une restriction sur le domaine d'une variable de décision en tenant compte des domaines actuels des autres variables de décision. Les indexicals ont été utilisés pour implémenter des contraintes définies par l'utilisateur dans divers systèmes, tel que SICStus Prolog [6]. Alors que les indexicals originaux ne peuvent faire face qu'à des contraintes d'arité fixe, nous les étendons pour traiter des contraintes d'arité non-fixe (appelées communément contraintes globales) en ajoutant des tableaux de variables de décision et des opérations sur de tels tableaux (itérations et opérateurs n-aire). De plus, contrairement aux implémentations classiques des indexicals, les indexicals ne sont pas interprétés ici, mais compilés en instructions pour le langage source du solveur cible.

*Cet article est une traduction de "Towards Solver-Independent Propagators" publié dans les actes de CP2012[16]. Nous conseillons la lecture de l'article original. Ce travail est supporté par la bourse 2011-6133 du Conseil Suédois de la Recherche (VR). Nous remercions les relecteurs anonymes et Christian Schulte pour leurs commentaires constructifs, et Marie Battisti et Loïc Blet pour l'aide à la traduction.

Le document est structuré comme suit. Après avoir présenté les bases nécessaires (Section 2) et des exemples de descriptions de propagateurs dans notre langage (Section 3), nous donnons la liste de nos décisions pour la conception du langage (Section 4). Ensuite, nous décrivons notre langage (Section 5), montrons comment analyser les propagateurs écrits dans le langage (Section 6), discutons notre implémentation actuelle (Section 7) et l'évaluons expérimentalement (Section 8). Nous terminons le papier par une revue des travaux apparentés et un regard sur nos futures directions de recherche (Section 9).

2 Bases

Soit X un ensemble de variables de décision entières qui prennent leurs valeurs dans un univers $\mathcal{U}$, où $\mathcal{U}$ peut être $\mathbb{Z}$ mais en pratique en est un sous-ensemble. Dans un solveur à domaines finis (FD), un store est une fonction $S\colon X \to \mathcal{P}(\mathcal{U})$, où $\mathcal{P}(\mathcal{U})$ est l'ensemble des sous-ensembles de $\mathcal{U}$. Pour une variable $x \in X$, l'ensemble $S(x)$ est appelé le *domaine* de x et est l'ensemble des valeurs possibles pour x. Un store S est une *affectation* si chaque variable a un domaine singleton ; de telles variables sont dites *déterminées*. Un store est *échoué* si une variable a un domaine vide. Un store S *domine* un store T (noté $S \sqsubseteq T$) si le domaine de chaque variable dans S est un sous-ensemble de son domaine dans T.

Une *contrainte* $C(Y)$ sur une séquence Y de X est une restriction des valeurs que ces variables peuvent prendre en même temps. La contrainte $C(Y)$ est un sous-ensemble de $\mathcal{U}^n$, où n est la longueur de Y. Une affectation A *satisfait* une contrainte $C(Y)$ si la séquence $[v \mid \{v\} = A(y) \wedge y \in Y]$ est membre de $C(Y)$. Étant donné un store S, une valeur $v \in S(y)$ pour un $y \in Y$ quelconque est *cohérente* avec une contrainte C si il existe une affectation $A \sqsubseteq S$ avec $A(y) = \{v\}$ qui satisfait C. Une contrainte C est *satisfaisable* dans un store S si il existe une affectation $A \sqsubseteq S$ qui satisfait C. Une contrainte C est *impliquée* dans un store S si toutes les affectations $A \sqsubseteq S$ satisfont C.

Un *vérificateur* pour une contrainte C est une fonction qui décide si une affectation satisfait C. Un *propagateur* pour une contrainte C est une fonction de stores à stores dont le rôle est de retirer du domaine les valeurs non cohérentes avec C. Nous sommes intéressés ici par l'écriture de propagateurs. Dans de vrais solveurs, les propagateurs ne renvoient pas un store mais modifient le store courant. Il y a un certain nombre de propriétés désirables pour un propagateur :

- *Correct* : Un propagateur correct ne retire jamais de valeurs qui sont cohérentes avec sa contrainte. C'est requis pour tout propagateur.
- *Vérifiant* : A propagateur vérifiant décide si une affectation satisfait sa contrainte. Cela peut se diviser en *correction de singleton* (accepte toutes les affectations satisfaisantes) et *complètude de singleton* (rejette toutes les affectations non satisfaisantes).
- *Contractant* : Un propagateur P est contractant si $P(S) \sqsubseteq S$ pour tout store S.
- *Monotone* : Un propagateur P est monotone si $S \sqsubseteq T$ implique $P(S) \sqsubseteq P(T)$ pour tous stores S et T.
- *Domaine cohérent* (DC) : Un propagateur DC retire toutes les valeurs non cohérentes. Des cohérences plus faibles existent, telles que la cohérence de bornes et la cohérence de valeur.
- *Idempotent* : Un propagateur P est idempotent si $P(S) = P(P(S))$ pour tout store S. Cela permet un meilleur ordonnancement du solveur [23].

De plus, pour des raisons d'efficacité, il est bon d'avoir des propagateurs avec une faible complexité temporelle. C'est la raison pour laquelle la cohérence de domaine est souvent remplacée par des cohérences plus faibles. Une autre préoccupation est d'éviter d'exécuter un propagateur qui ne peut retirer de valeur du store courant. Plusieurs mécanismes peuvent être implémentés pour éviter de telles exécutions : déclarer l'idempotence et l'implication, souscrire seulement à des modifications de domaine pertinentes.

Les indexicals ont été introduits dans [28] pour décrire des propagateurs pour le solveur cc(FD). Un(e) (*expression*) *indexical* est de la forme x `in` σ, qui signifie que le domaine de la variable x doit être réduit à l'intersection de son domaine courant avec l'ensemble σ ; cet ensemble peut dépendre d'autres variables et est calculé en fonction de leurs domaines dans le store *courant*. Un indexical est exécuté chaque fois que le domaine d'une des variables apparaissant dans σ est modifié. Un propagateur est typiquement décrit par plusieurs indexicals, un par variable de la contrainte. Les indexicals ont été inclus dans plusieurs autres systèmes, étendus avec des indexicals vérifiant [6], des expressions conditionnelles [24, 6], et des gardes [29].

3 Exemples de Descriptions de Propagateurs

Dans ce papier, nous étendons la syntaxe des indexicals pour utiliser des tableaux de variables et des opération sur ces tableaux (itérations et opérateurs n-aire). Nous commençons par présenter quelques exemples de descriptions de propagateurs qui montrent les principales caractéristiques de notre langage. Cela nous amènera à expliquer les décisions que nous avons prises pendant la conception et à donner une définition

```
1 def SUM(vint[] X,vint N){
2   propagator(v1){
3     N in sum(i in rng(X))(dom(X[i]));
4     forall(i in rng(X))
5       X[i] in dom(N) -
6           sum(j in rng(X):j!=i)(dom(X[j]));
7   }
8   propagator(v2){
9     N in sum(i in rng(X))(min(X[i])) ..
10            sum(i in rng(X))(max(X[i]));
11    forall(i in rng(X))
12      X[i] in min(N) - sum(j in rng(X):j!=i)(max(X[j]))
13          .. max(N) - sum(j in rng(X):j!=i)(min(X[j]));
14  }
15  checker{ val(N) = sum(i in rng(X))(val(X[i])) }
16}
```

FIGURE 1 – Code pour la contrainte SUM, avec deux propagateurs

```
1 def EXACTLY(vint[] X, vint N, int v){
2   propagator{
3     N in sum(i in rng(X))(b2i(entailed(EQ(X[i],v)))) ..
4        sum(i in rng(X))(b2i(satisfiable(EQ(X[i],v))));
5     forall(i in rng(X)){
6       once(val(N) <=
7         sum(j in rng(X):i!=j)
              (b2i(entailed(EQ(X[j],v)))))
8           post(NEQ(X[i], v));
9       once(val(N) >
10        sum(j in rng(X):i!=j)
              (b2i(satisfiable(EQ(X[j],v)))))
11          post(EQ(X[i], v));
12    }
13  }
14        checker{val(N) = sum(i in
              rng(X))(b2i(val(X[i])=v))}
15}
```

FIGURE 2 – Code pour la contrainte EXACTLY

plus précise du langage.

La Figure 1 présente la contrainte $\text{SUM}(X, N)$, vérifiée si la somme des valeurs dans le tableau X égale N. Il est possible de décrire plusieurs propagateurs et un vérificateur pour une contrainte. En particulier, il y a ici deux propagateurs : `v1` utilise l'entièreté des domaines des variables (p.ex., `dom(N)`), tandis que `v2` n'utilise que leurs bornes (p.ex., `min(N)`). Une description de propagateur est une liste d'indexicals. Il est possible d'accéder aux domaines des variables avec les quatre fonctions `dom`, `min`, `max` et `val`. Les opérateurs arithmétiques peuvent être appliqués à des entiers ou à des ensembles. L'opérateur `sum` est n-aire, il opère sur une séquence de valeurs de longueur arbitraire. L'opérateur `rng` représente l'intervalle des indices d'un tableau, et $\ell..u$ l'intervalle d'entiers de ℓ à u inclus. Les lignes 9–10 signifient "Le domaine de `N` doit être intersecté avec l'intervalle dont la borne inférieure est la somme des plus petites valeurs du domaine de chaque variable de `X`, et dont la borne supérieure est la somme des plus grandes valeurs du domaine de chaque variable de `X`." Cet exemple montre aussi comment écrire des boucles (`forall` aux lignes 4–6 et 11–13).

La figure 2 présente la contrainte $\text{EXACTLY}(X, N, v)$, vérifiée si exactement N variables du tableau X sont égales à la valeur v. Cet exemple illustre l'utilisation de conditions (`when`), de conversions de booléen vers entier (`b2i`$(\mathit{false}) = 0$ et `b2i`$(\mathit{true}) = 1$), et de références à d'autres contraintes (`EQ` et `NEQ`, contraignant une variable à être respectivement égale à et différente d'une certaine valeur). Les fonctions `entailed` et `satisfiable` vérifient l'état d'une contrainte en fonction du domaine courant des variables et `post` invoque un propagateur d'une certaine contrainte. Les lignes 3–4 restreignent le domaine de `N` entre deux bornes. La borne inférieure est calculée comme le nombre de variables de `X` qui *doivent* être assignées à `v`, et la borne supérieure comme le nombre de variables qui *peuvent* être assignées à `v`. Le corps de la boucle retire `v` du domaine d'une variable (lignes 6–8), ou fixe une variable à `v` (lignes 9–11), quand certaines conditions sur les autres variables sont respectées. Les modifications des domaines sont effectuées en invoquant la propagation d'autres contraintes.

4 Décisions de Conception du Langage

Le langage, tel que présenté dans la section précédente et défini plus précisément dans la suivante, a été conçu selon les décisions suivantes.

Le langage est basé sur les indexicals. Les indexicals ont déjà été utilisés avec succès dans plusieurs solveurs, et de nombreux travaux ont été effectués à leur sujet, e.g. in [5] and [9]. De plus, les indexicals sont très simples à comprendre et sont souvent proches du premier raisonnement que quelqu'un ferait au sujet d'un propagateur. Une des restrictions que nous maintenons est que les indexicals sont *sans état*, et ne peuvent donc pas représenter des propagateurs compliqués tels que le propagateur DC pour ALLDIFFERENT [20]. C'est une limitation forte mais un choix doit être fait entre la simplicité du langage et la complication des propagateurs. Un grand nombre de contraintes ont cependant des propagateurs sans état efficaces.

Le langage est fortement typé, pour simplifier la compréhension et la compilation des propagateurs. Cela nécessite l'ajout de l'opérateur `b2i`.

Nous introduisons des tableaux et des opérateurs n-aires pour pouvoir traiter efficacement des contraintes globales. Par exemple, l'expression `sum(i in`

`rng(X))(val(X[i]))` est paramétrée par l'indice de boucle (`i` ici), son domaine (`rng(X)` ici), et l'expression dépendant de l'indice (`val(X[i])` ici) qui doit être agrégée (sommée ici).

Nous introduisons aussi des méta-contraintes, des invocations de contraintes et des variables locales pour aider à écrire des propagateurs plus concis. Voir Section 5 pour les détails.

Pour faciliter l'utilisation, nous ne voulons qu'un petit nombre d'opérateurs et primitives. Cela nous permet aussi d'avoir une procédure de compilation relativement simple. Cependant, il n'est pas impossible que de nouvelles constructions soient ajoutées dans le futur, mais avec précaution.

Pour être général (dans l'approche FD), nous évitons d'ajouter des primitives spécifiques à un solveur. En particulier, notre langage n'a que quatre fonctions pour accéder au domaine d'une variable (voir Section 5). Les autres moyens de communiquer avec un solveur sont les réductions de domaine, fournies par tout solveur FD, et un mécanisme d'échec (qui peut être imité en vidant un domaine).

Notre langage n'a pas non plus certaines constructions présentes dans certaines implémentations de contraintes, telles que la détection d'implication, les littéraux gardés [13], et des évènements avec un grain plus fin [15]. Nous comptons étudier comment ceux-ci peuvent être inclus sans compliquer de trop le langage.

```
CSTR    ::= def CNAME(ARGS){ PROPAG+ CHECKER?)}
PROPAG  ::= propagator(PNAME?){ INSTR* }
CHECKER ::= checker{ BOOL }
INSTR   ::= VAR in SET ; | post(CINVOKE,PNAME?); | fail;
            | once(BOOL){ INSTR* } | forall(ID in SET){
            INSTR* }
SET     ::= univ | emptyset | ID | INT..INT | rng(ID) |
            NSETOP(ID in SET)(SET) | dom(VAR) | -SET |
            SET BSETOP SET | {INT+} | {ID in SET:BOOL}
INT     ::= inf | sup | NUM | ID | card(SET)| min(SET)
            | max(SET) | min(VAR) | max(VAR) | val(VAR)
            | INT BINTOP INT | - INT | NINTOP(ID in
            SET)(INT) | b2i(BOOL)
BOOL    ::= true | false | ID | INT INTCOMP INT |
            INT memberOf SET | SET SETCOMP SET | not
            BOOL | BOOL BBOOLOP BOOL | NBOOLOP(ID
            in SET)(BOOL) | check(CINVOKE) | en-
            tailed(CINVOKE) | satisfiable(CINVOKE)
BINTOP  ::= + | - | * | / | mod
NINTOP  ::= sum | min | max
BSETOP  ::= union | inter | minus | BINTOP
NSETOP  ::= union | inter | sum
INTCOMP ::= = | != | <= | < | >= | >
SETCOMP ::= = | subseteq
BBOOLOP ::= and | or | =
NBOOLOP ::= and | or
CINVOKE ::= CNAME | CNAME(ARGS)
```

FIGURE 3 – Grammaire "à la BNF" de notre langage. Les constructions en gris se trouvaient déjà dans d'autres définitions des indexicals [28], [6]. Les règles correspondant à `ARGS` (liste d'arguments), `CNAME`, `PNAME`, `ID`, `VAR` (respectivement identificateur de contrainte, propagateur, constante, et variable), et `NUM` (littéral entier) ne sont pas montrées.

5 Définition du Langage

Dans la Section 5.1, nous définissons la syntaxe et la sémantique de notre langage. Il est fortement typé et a cinq types de base : entiers (`int`), booléens (`bool`), ensembles d'entiers (`set`), variables de décision entières (`vint`) et contraintes (`cstr`). Ce dernier type est discuté à la Section 5.2. Nous supportons des tableaux de n'importe quel type de base (mais pas de tableaux de tableaux pour le moment). Les identificateurs de (tableaux de) variables de décision commencent avec un lettre majuscule. Les identificateurs de constantes dénotant des (tableaux de) entiers, booléens et ensembles commencent avec une lettre minuscule.

5.1 Syntaxe et Sémantique

La Figure 3 présente la grammaire de notre langage. Nous passons en revue les règles de production. La règle principale (`CSTR`) définit une contrainte. Une contrainte est définie par son nom et sa liste d'arguments. Une définition de contrainte peut contenir la description d'un ou plusieurs propagateurs et un vérificateur.

Un *propagateur* possède un identificateur facultatif et contient une liste d'instructions. Une instruction (`INSTR`) peut être un *indexical*, x `in` σ, dont la signification est que le domaine de la variable x doit être réduit à l'intersection de son domaine actuel avec l'ensemble σ. Les autres instructions sont `fail` et `post`. L'effet de `fail` est de transformer le store courant en store échoué. L'instruction `post(`C`,` P`)` invoque le propagateur P de la contrainte C ; si P n'est pas spécifié, alors le premier (ou le seul) propagateur de C est invoqué. Il existe deux structures de contrôle. Le `forall` crée une itération sur un ensemble, et `once` crée un bloc conditionnel ; la raison de ne pas nommer ce dernier `if` est de souligner que les propagateurs doivent être monotone, et qu'une fois que la condition devient vraie, elle doit le rester. Voir Section 6 pour une discussion sur la monotonie.

La plupart des règles sur les ensembles, les entiers et les booléens n'ont pas besoin d'explications ou ont déjà été expliquées dans la Section 3. Certaines constantes sont définies : `univ` désigne l'univers $\mathcal{U}$, `inf` sa borne inférieure, `sup` sa borne supérieure et `emptyset` l'ensemble vide. Les opérations arithmétiques sur les entiers sont étendues point par point aux ensembles.

Il y a quatre moyens d'accéder au domaine d'une

variable : `dom(`x`)`, `min(`x`)`, `max(`x`)` et `val(`x`)` désignent respectivement le domaine de la variable x, son minimum, son maximum et sa valeur unique. Comme `val(`x`)` n'est déterminé que quand la variable x est fixée, le compilateur doit ajouter des gardes pour assurer un traitement correct quand x n'est pas déterminée.

Alors que l'instruction `post` invoque le propagateur d'une autre contrainte, les fonctions `entailed`, `satisfiable`, et `check` interrogent l'état d'un autre contrainte. Soit S le store actuel : `entailed(`C`)` et `satisfiable(`C`)` décident si la contrainte C est impliquée (respectivement, satisfaisable) dans S. Si S est une affectation, la fonction `check(`C`)` peut être appelée et décide si S satisfait la contrainte C (un exemple sera donné dans la sous-section suivante).

5.2 Méta-contraintes

Une nouvelle fonctionnalité de notre langage est ce que nous appelons une méta-contrainte, qui est une contrainte qui prend d'autre(s) contrainte(s) comme argument(s). Les méta-contraintes permettent d'écrire des propagateurs plus concis en encapsulant des fonctionnalités récurrentes.

Par exemple, la contrainte Among(X, N, S), vérifiée s'il y a N éléments dans le tableau X qui ont une valeur dans l'ensemble S, serait décrite presque identiquement à la contrainte Exactly(X, N, v) de la Figure 2. Le code commun peut être factorisé dans la méta-contrainte `Count(Vint [] X, vint N, cstr C, cstr NC)`, dont la description n'est pas donnée ici, mais dont le sens est défini par son vérificateur : `val(N) = sum(i in rng(X))(b2i(check(C(X[i]))))`, c'est-à-dire exactement `N` variables du tableau `X` satisfont la contrainte `C`. L'argument `NC` est la négation de la contrainte `C` (voir [3] pour savoir comment nier même des contraintes globales) et est utilisé dans le propagateur de Count (comme `NEQ` est utilisé à la ligne 8 de la Figure 2). Nous pouvons alors décrire Exactly et Among comme le montre la Figure 4. La méta-contrainte Count est étroitement liée à l'opérateur de cardinalité [26] mais nous permettons à l'utilisateur de décrire d'autres méta-contraintes.

6 Analyse Syntaxique et Outils

Un de nos objectifs est de faciliter la preuve de propriétés des propagateurs. Avant de passer à la compilation proprement dite, nous montrons comment notre langage contribue à cela et à d'autres fonctionnalités liées à l'écriture de propagateurs.

```
def EXACTLY(vint[] X, vint N, int v){
  propagator{
    cstr EQv(vint V):= EQ(V,v);
    cstr N_EQv(vint V):= NEQ(V,v);
    post(COUNT(X,N,EQv,N_EQv));
  }
}

def AMONG(vint N, vint[] X, set s){
  propagator{
    cstr INs(vint V):= INSET(V,s);
    cstr NINs(vint V):= NOTINSET(V,s);
    post(COUNT(X,N,INs,NINs));
  }
}
```

Figure 4 – Exactly et Among, décrits en utilisant la méta-contrainte Count

6.1 Analyse

Parmi les propriétés d'un propagateur présentées à la Section 2, la plupart sont difficiles à prouver pour un propagateur donné (sauf la contraction, que les indexicals satisfont par définition). Toutefois, comme cela a été démontré dans [5], il est possible de prouver la monotonie des indexicals. Ce résultat peut être étendu à notre langage plus général. Nous montrons en outre comment prouver la correction de certains propagateurs par rapport à leurs contraintes

Monotonie. La procédure pour vérifier la monotonie des indexicals est combinée à l'ajout de gardes pour `val`. L'arbre syntaxique représentant les indexicals est traversé par un ensemble de fonctions mutuellement récursives. Pour assurer la monotonie de l'ensemble du propagateur, chaque fonction vérifie un comportement *attendu* de la sous-arborescence auquel il est appliquée. Les fonctions récursives sont étiquetées *monotone anti-monotone* ou *fixe* pour les expressions booléennes ; *croissante*, *décroissante* ou *fixe* pour les expressions entières ; *croissante*, *décroissante* ou *fixe* pour les expressions d'ensemble ; et *monotone* pour les instructions. Par exemple, la fonction *croissante* vérifie que son expression entière en argument est (non strictement) croissante entre un store et un store dominant. Ces fonctions renvoient deux valeurs : si l'expression respecte effectivement son comportement attendu, et l'ensemble des variables qui doivent être fixées afin d'assurer une utilisation sûre de `val`. Cet ensemble de variables est utilisé pour ajouter des gardes dans le propagateur généré. Ces gardes sont ajoutées non seulement aux instructions, mais aussi à l'intérieur du corps de l'expression `b2i`.

Par manque de place, nous ne pouvons pas exposer toutes les règles qui composent les fonctions récursives

Expression originale	Expression gardée	Mono
`X in {val(Y)}`	`once(ground(Y)) X in {val(Y)}`	true
`B in b2i(val(X)=v) ..` `  b2i(v memberOf dom(X))`	`B in b2i(ground(X) and val(X)=v) ..` `  b2i(v memberOf dom(X))`	true
`once(min(B)=1) X in {v}`	`once(min(B)=1) X in {v}`	false
`once(min(B)>=1)X in {v}`	`once(min(B)>=1) X in {v}`	true
`once(val(B)=1) X in {v}`	`once(ground(B) and val(B)=1)X in {v}`	true

TABLE 1 – Exemples d'ajout de garde et de vérification de monotonie

(il y a environ 200 règles). Au lieu de cela, nous montrons quelques exemples. A titre d'exemple d'une règle, considérez l'appel à la fonction *croissante* sur une expression de la forme `min(`i `in` σ`)(`e`)`. Pour que cette expression soit croissante, σ doit être *décroissante* et e doit être *croissante*. En outre, l'ensemble des variables de garde est l'union des variables qui doivent être gardées pour ces deux sous-expressions.

Le tableau 1 montre quelques exemples de résultats de la procédure. Dans ce tableau, `ground(`x`)` représente un opérateur (qui ne fait pas partie de notre langage) qui décide si la variable x est déterminée. La deuxième colonne indique où les gardes sont ajoutées aux indexicals donnés dans la première colonne. La troisième colonne indique si l'indexical est prouvé monotone ou non. La première ligne ajoute juste une garde. La deuxième ligne montre qu'une garde peut être ajoutée à l'intérieur d'une expression `b2i` afin qu'il retourne `0` tant que `X` n'est pas fixé. Les trois dernières lignes montrent comment de petites variations modifient la monotonie d'une expression. La condition `min (B) = 1` n'est (syntaxiquement) pas monotone, car en général cette condition pourrait être vraie dans un store et devenir fausse dans un store plus fort, mais si nous savons que `B` représente un booléen (avec un domaine `0..1`), alors nous pouvons remplacer l'égalité par une inégalité comme cela est fait dans le quatrième exemple. La dernière ligne montre une autre façon d'obtenir la monotonie : remplacer le `min` par `val`, ce qui nécessite l'ajout d'un garde.

La correction de la procédure de contrôle de monotonie peut être démontrée par induction sur les règles récursives, comme suggéré dans [5]. Toutefois, cette procédure est syntaxique, donc incomplète. Un exemple de propagateur pour $\text{EQ}(X, Y)$ qui est monotone mais non reconnu comme tel est donné en haut de la Figure 5. La monotonie n'est pas reconnue, car il faut que les ensembles sur lesquelles les boucles `forall` itèrent soient croissants, tandis que `dom(`x`)` ne peut que rétrécir. Cependant, ce n'est pas le plus simple pour décrire la propagation de cette contrainte : un propagateur de 2 lignes se trouve en bas à gauche de la Figure 5.

```
def EQ(vint X1, vint X2){
 propagator{
   forall(i in dom(X1)) once(not i memberOf dom(X2))
     X1 in univ minus {i};
   forall(i in dom(X2)) once(not i memberOf dom(X1))
     X2 in univ minus {i};
 }
}
```

```
def EQ(vint X, vint Y){
 propagator{
   X in dom(Y);
   Y in dom(X);
 }
 checker{val(X) = val(Y)}
}
```

```
def EQ(vint X, int cY){
 propagator{
   X in {cY};
   once(not cY
       memberOf dom(X))
     fail;
 }
 checker{ val(X) = cY }
}
```

FIGURE 5 – Variations sur la contrainte EQ

Correction. Pour prouver algorithmiquement si un propagateur est correct par rapport à sa contrainte, nous utilisons le fait connu qu'un propagateur est correct si il est singleton correct et monotone. Nous proposons une procédure incomplète mais correcte pour prouver qu'un propagateur P est singleton correct par rapport à son vérificateur C, et donc par rapport à sa contrainte (en supposant que le vérificateur est correct). Nous devons prouver que si une affectation satisfait C, alors elle n'est pas supprimée par P. A cette fin, à partir de la description par indexicals de P, on déduit une formule $C(P)$ qui définit les affectations acceptées par le propagateur. La correction de singleton est vraie si la formule $C \wedge \neg C(P)$ est insatisfaisable (c'est à dire, si $C \rightarrow C(P)$). Pour calculer $C(P)$, nous transformons les indexicals en une formule équivalente à l'aide des règles de réécriture suivantes :

`dom(`x`)` $\rightarrow$ `{val(`x`)}`
`min(`x`)` $\rightarrow$ `val(`x`)`
`max(`x`)` $\rightarrow$ `val(`x`)`
`fail` $\rightarrow$ `false`
x `in` σ $\rightarrow$ `val(`x`) memberOf` σ
`once(`b`){`y`}` $\rightarrow$ `(not` b`) or` y
`forall(`i `in` σ`){`y`}` $\rightarrow$ `and(`i `in` σ`)(`y`)`

Notre procédure actuelle pour prouver l'insatisfais-

abilité de $C \wedge \neg C(P)$ essaye de simplifier la formule en `false` en utilisant des règles de réécriture. Nous avons environ 240 règles de réécriture, allant de la simplification des booléens (par exemple, `faux ou` $b \rightarrow G$), à celle d'entiers et d'ensembles (par exemple, `min(`i `in` $\ell..u$`)(`i`)` $\rightarrow$ ℓ) et à l'évaluation partielle (par exemple, $2 + x + 3 \rightarrow x + 5$). Comme cette procédure incomplète n'est en mesure de prouver la correction de singleton que pour une petite partie des propagateurs (voir Section 7.2), nous avons l'intention de l'améliorer en appelant un solveur externe.

À titre d'exemple, l'application de la transformation propagateur-à-vérificateur sur le propagateur dans le coin inférieur gauche de la Figure 5 résulte en la formule `val(X) memberOf {val(Y)} and val(Y) memberOf {val(X)}`. Cette formule peut être démontrée équivalente au vérificateur de la contrainte (en utilisant les règles suivantes : x `memberOf {`y`}` $\rightarrow$ x `=` y, b `and` $B \rightarrow b$ et b `and not` $b \rightarrow$ `false`). En résumé, ce propagateur peut être prouvé automatiquement monotone, singleton correct, singleton complet (voir ci-dessous), et donc correct.

Vérifiant. L'approche ci-dessus peut également être utilisée pour prouver qu'un propagateur est vérifiant. En effet, la complétude de singleton est prouvée par l'implication $C(P) \rightarrow C$ (l'inverse de la correction de singleton), et un propagateur qui est singleton correct et singleton complet est vérifiant ($C(P) \Leftrightarrow C$).

6.2 Transformation

En plus de l'analyse, nous pouvons transformer algorithmiquement un propagateur. Nous avons mis au point deux premières transformations qui semblent intéressantes : modifier le niveau de raisonnement, et fixer des variables de décisions. Par manque de place, cette section ne figure pas dans la traduction en français.

7 Compilation

Nous discutons maintenant nos décisions de conception du compilateur et notre compilateur actuel.

7.1 Décisions de conception du compilateur

Au lieu d'interprétation, nous avons pris la décision de compiler vers le langage dans lequel les propagateurs sont écrits pour un solveur. Cela a le double avantage d'avoir une infrastructure qui est relativement indépendante des solveurs (seule la partie de génération de code dépend des solveurs), et d'avoir du code compilé qui est plus efficace que du code interprété. Le code généré est aussi autonome (il peut être distribué sans le compilateur).

Les propagateurs compilés sont actuellement sans état (comme le sont les indexicals) et utilisent des événements de réveil à gros grain. Ce choix vise à simplifier la compilation. Toutefois, après une analyse appropriée, il devrait être possible de produire des propagateurs qui incorporent certains états ou d'utiliser des événements plus fins.

La compilation produit un propagateur pour chaque description `propagator`. La compilation ne modifie pas l'ordre des indexicals à l'intérieur d'un propagateur. En outre, pour obtenir l'idempotence, le propagateur complet est répété jusqu'à ce qu'il atteigne son point fixe interne. Un autre choix valable aurait été de créer un propagateur pour chaque indexical et laisser le solveur effectuer l'ordonnancement. Nous n'avons pas évalué tous les compromis de ce choix. Une approche intermédiaire serait d'analyser la structure interne de la description du propagateur pour générer une bonne politique d'ordonnancement des indexicals à l'intérieur du propagateur. Cela nécessite beaucoup plus de travail et reste comme travail futur.

Les invocations des propagateurs (`post`) ou vérificateurs (`check`) sont remplacés par le code correspondant. C'est similaire à l'inlining de fonctions dans des langages de programmation classiques. Pour les fonctions `entailed` et `satisfiable`, la description du vérificateur correspondant est d'abord transformée afin de traiter des stores qui ne sont pas des affectations. Cela se fait avec environ 130 règles de réécriture formant une procédure récursive similaire à celle de la vérification de monotonie. Comme différences, les accesseurs `val` sont remplacés par `min`, `max`, ou `dom` lorsque cela est possible, ou sont convenablement gardés autrement. Par exemple, appeler la fonction *décroissante* sur le singleton `{val(`x`)}` renvoie `dom(`x`)`, mais l'appel à la fonction *croissante* ajoute une garde. L'implication nécessite une formule booléenne *monotone* et la satisfaisabilité une formule *antimonotone*. Par exemple, le vérificateur de satisfaisabilité généré pour `EQ(X, Y)` est `not dom(X) inter dom(Y) = emptyset`. À son tour, le vérificateur d'implication généré pour `EQ(X, Y)` est `ground(X) and ground(Y) and val(X) = val(Y)`.

Le choix mettre en ligne les invocations de contraintes simplifie beaucoup le processus de compilation. Toutefois, cela signifie que tous les contraintes référencées doivent être décrites par indexicals. Nous avons l'intention d'explorer comment supprimer cette limitation afin d'être en mesure d'invoquer des propagateurs intégrés dans le solveur cible.

7.2 Implémentation et Solveurs Cibles

Nous avons écrit un prototype en Java. Il utilise Antlr [19] pour l'analyse et StringTemplate [18] pour la génération de code. Actuellement, nous compilons en propagateurs pour Comet [10], Gecode [12] et OscaR [22]. Une grande partie du processus de compilation consiste à réécrire les opérateurs n-aires en boucles. Certaines optimisations sont faites, comme un pré-calcul des tableaux par programmation dynamique (en remplaçant de boucles imbriquées par des boucles successives) [25, Section 9] et la factorisation d'expressions répétées. Le compilateur détecte les événements qui devraient réveiller le propagateur, ce qui est fait en traversant l'arbre syntaxique et collectant les accès aux variables. Le compilateur ajoute également la détection d'implication au propagateur d'une contrainte C, en testant si `entailed(`C`)` est vrai.

Actuellement, nous avons écrit à peu près 700 lignes d'indexicals décrivant 76 propagateurs pour 48 contraintes, dont 14 sont des méta-contraintes, 17 sont des contraintes globales, et 17 sont binaires ou ternaires. Sur les 76 propagateurs, 69 sont prouvés monotones, parmi lesquels 16 sont prouvés singleton corrects et 29 sont prouvés singleton complets, faisant 16 propagateurs prouvés corrects. Ces chiffres pourraient être améliorées avec une meilleure procédure pour prouver l'insatisfaisabilité.

Pour avoir une idée de la concision du langage, notez que notre compilateur produit à partir de la description de 15 lignes de Exactly à la Figure 2 un propagateur pour Comet qui est d'environ 150 lignes de code, et un autre pour Gecode d'environ 170 lignes. Nous estimons que le code pour le propagateur intégré de cette contrainte est environ 150 lignes dans Gecode.

Le prototype, ainsi que les descriptions de propagateurs, sont disponible sur demande auprès du premier auteur ou sur `http://user.it.uu.se/~jeamo371/indexicals`.

8 Evaluation Expérimentale

Pour évaluer si les propagateurs décrits par indexicals se comportent raisonnablement, nous comparons quelques propagateurs générés avec des propagateurs natifs de Gecode et des décompositions simples. Nous ne nous attendons pas à ce que les propagateurs générés soient aussi efficaces que ceux faits à la main, mais le but est de montrer qu'ils sont une alternative viable lorsque l'on a peu de temps pour développer un propagateur d'une contrainte.

Notre cadre expérimental est comme suit. Nous utilisons Gecode 3.7.3. Pour chaque contrainte, nous recherchons l'ensemble de ses solutions. Nous répétons la recherche en utilisant plusieurs heuristiques de branchement pour essayer d'exercer autant de parties des propagateurs que possible.

Les contraintes étudiées sont Sum, Maximum Exactly et Element. Leurs descriptions par indexicals sont représentatives des autres propagateurs que nous avons implémentés. En outre, elles partagent la propriété que l'une des variables dépend fonctionellement des autres. Cela nous permet de comparer les différents propagateurs d'une contrainte avec un problème de référence où la contrainte est absente, mais la variable dépendante est fixée à une valeur arbitraire. Comme l'utilité de la contrainte est seulement de définir la dépendance fonctionnelle, le nombre de solutions est le même et la taille de l'arbre de recherche est la même, mais le temps passé en propagation est nul. On peut alors calculer le temps d'exécution d'un propagateur en soustrayant les totaux des temps d'exécution.

Pour Sum et Maximum, nous utilisons des versions qui raisonnent sur les bornes pour les indexicals et les propagateurs natifs ; pour Exactly et Element, nous utilisons le raisonnement sur le domaine. Les descriptions par indexicals de Sum et Exactly se trouve aux Figures 1 (`v2`) et 2 respectivement. Par manque de place, Maximum et Element ne sont pas montrées. Les décompositions de $\textsc{Sum}(X, N)$ et $\textsc{Maximum}(X, N)$ introduisent un tableaux A de $n = \|X\|$ variables auxiliaires. La décomposition de Sum est exprimée comme $A[1] = X[1] \wedge \forall_{i \in 2..n} (A[i-1] + X[i] = A[i]) \wedge A[n] = N$, et celle de Maximum est $A[1] = X[1] \wedge \forall_{i \in 2..n} (\max(A[i-1], X[i]) = A[i]) \wedge A[n] = N$. La décomposition de $\textsc{Exactly}(X, N, v)$ introduit un tableau B de variables booléennes et est défini comme $\forall_{i \in 1..n} (B[i] \equiv X[i] = v) \wedge N = \sum_{i \in 1..n} B[i]$; la somme des variables booléennes est implémentée par un propagateur natif. La contrainte $\textsc{Element}(X, Y, Z)$ (vraie si $X[Y] = Z$) est décomposée en $Y \in 1..n \wedge \forall_{i \in 1..n} (Y = i \Rightarrow X[i] = Z)$. De plus, nous utilisons des formulations par automate de Maximum et Element, qui se trouvent dans le catalogue des contraintes globales [4]. Les automates de Maximum et Element et la décomposition de Element ne font pas toutes les réductions possibles (mais bien les autres décompositions). Cela génère un surcoût d'environ 15% de noeuds en plus visités pour les automates, et 7% pour la décomposition de Element.

Le Tableau 2 présente les temps relatifs des différentes implémentations des contraintes pour des tableaux de 9 variables avec des domaines de 9 valeurs. Les heuristiques de recherche utilisées sont quelques combinaisons de l'ordre des variables (ordre des arguments de la contrainte, dans un tableau le plus petit ou le plus grand domaine en premier) et de l'or-

TABLE 2 – Temps d'exécution Relatifs (en pourcents)

	MAX.	SUM	EXAC.	ELEM.
Natif	100	100	100	100
Indexicals	125	269	252	118
Décomposition	195	296	313	204
Automate	675	n/a	n/a	487

dre des valeurs (fixer au minimum, diviser en deux, fixer à la médiane). Pour chaque contrainte et chaque propagateur, les temps sont additionnés pour les différentes heuristiques de recherche. Puis la somme des temps pour explorer l'arbre de recherche est soustraite. Enfin, pour chaque contrainte, la somme des temps de chaque propagateur est divisée par la somme des temps des propagateurs de Gecode. Par rapport aux propagateurs natifs, ceux générés induisent uniquement une petite surcharge pour MAXIMUM et ELEMENT, mais ne se comportent pas aussi bien pour SUM et EXACTLY. Cependant, dans tous les cas, les indexicals ont un meilleur temps que la décomposition, mais parfois seulement légèrement. En particulier, pour la contrainte EXACTLY, la décomposition a un meilleur temps pour certains cas plus petits (non présentés). Nous expliquons le comportement des indexicals sur cette contrainte par le fait qu'ils sont éveillés à chaque fois que le domaine d'une variable change même si certaines variables ne peuvent plus affecter le statut de la contrainte. Le propagateur natif et la décomposition sont plus intelligents et ignorent les variables qui ne peuvent plus prendre la valeur donnée. Les indexicals ont un bien meilleur comportement que les automates.

Les propagateurs compilés à partir des indexicals ont un temps d'exécution moyen par appel qui augmente (nécessairement) de façon linéaire avec le nombre de variables. Les temps d'exécution des propagateurs natifs augmentent également mais avec une pente beaucoup plus douce.

Ces expériences montrent que les descriptions de propagateurs par indexicals sont utiles, mais qu'il y a encore de la place pour améliorer la compilation.

9 Conclusion

Nous avons présenté un langage pour décrire des propagateurs indépendemment des solveurs. L'objectif est de faciliter l'écriture et le partage de propagateurs et de faciliter la preuve de leurs propriétés formelles. Le langage qui en résulte, basé sur les indexicals, est de suffisamment haut niveau pour cacher les détails d'implémentation et permettre certaines analyses et transformations. Il est compilé en code source pour des solveurs cibles.

L'idée de laisser l'utilisateur écrire ses propres propagateurs n'est pas neuve. Le système cc(FD) [28] est l'un des premiers à avoir proposé cela grâce à l'utilisation d'indexicals. Depuis lors, beaucoup de solveurs CP sont ouverts, dans le sens où n'importe quel utilisateur peut ajouter de nouveaux propagateurs (surtout pour des contraintes globales) au noyau de propagateurs. Dans de tels systèmes, l'utilisateur écrit le code dans le langage de programmation du solveur et l'intègre au solveur grâce à une interface définissant principalement comment interagir avec les variables et le coeur du solveur. Certains solveurs prennent une approche différente en proposant un langage pour définir la propagation ; citons les constraints handling rules [11] et les actions rules [29]. Notre langage est un niveau d'abstraction au-dessus de ces approches, car il peut être traduit en code pour *n'importe quel* solveur. De ce point de vue, il est proche en esprit de ce qu'est un langage de modélisation indépendant des solveurs : une couche au dessus des solveurs pour décrire facilement des modèles de problèmes qui sont ensuite compilés dans le langage du solveur. Dans notre cas, des propagateurs sont décrits au lieu des modèles.

Des propagateurs ont également été décrits en utilisant des contraintes atomiques et des règles de propagation [8]. Cependant, ils ont été conçus seulement pour raisonner sur les propagateurs, et il ne sont pas pratique pour effectivement implémenter des propagateurs, sauf pour les approches basées sur des solveurs SAT qui utilisent des contraintes de bas niveau (par exemple, la génération paresseuse de clauses [17]).

Les contraintes enfichables de [21] découplent également la mise en oeuvre des contraintes de l'architecture solveur, en utilisant une interface indépendante du solveur pour la communication entre les deux composants. Notre approche vise un niveau plus élevé d'abstraction, avec la possibilité de perdre un certain niveau de contrôle.

Cet article ouvre plusieurs pistes de recherche intéressantes, en plus de celles déjà citées dans les Sections 4 et 7.1 pour surmonter les décisions initiales de conception : le langage doit permettre un meilleur contrôle de l'algorithme de propagation, tout en restant simple et générale. La simplicité est importante car nous croyons que la présence d'outils *automatisés* pour l'analyse facilite l'écriture de propagateurs.

Des futures cibles de compilation sont d'autres solveurs FD (par exemple, Choco [7] et Jacop [14]), les générateurs paresseux de clauses SAT [17], les générateurs de coupes en MIP [1] et les fonctions de pénalité et invariants en CBLS [27], [2].

Références

[1] Achterberg, T. : SCIP : Solving constraint integer programs. Mathematical Programming Computation 1, 1–41 (2009)

[2] Ågren, M., Flener, P., Pearson, J. : Inferring variable conflicts for local search. In : Benhamou, F. (ed.) Proceedings of CP'06. LNCS, vol. 4204, pp. 665–669. Springer (2006)

[3] Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J. : On the reification of global constraints. Tech. Rep. T2012 :02, Swedish Institute of Computer Science (February 2012), available at `http://soda.swedish-ict.se/view/sicsreport/`

[4] Beldiceanu, N., Carlsson, M., Rampon, J.X. : Global constraint catalog, 2nd Edition (revision a). Tech. Rep. T2012 :03, Swedish Institute of Computer Science (February 2012), available at `http://soda.swedish-ict.se/view/sicsreport/`

[5] Carlson, B., Carlsson, M., Diaz, D. : Entailment of finite domain constraints. In : Proceedings of ICLP'94. pp. 339–353. MIT Press (1994)

[6] Carlsson, M., Ottosson, G., Carlson, B. : An open-ended finite domain constraint solver. In : Proceedings of PLILP'97. LNCS, vol. 1292, pp. 191–206. Springer (1997)

[7] CHOCO : An open source Java CP library, `http://www.emn.fr/z-info/choco-solver/`

[8] Choi, C.W., Lee, J.H.M., Stuckey, P.J. : Removing propagation redundant constraints in redundant modeling. ACM Transactions on Computational Logic 8(4) (2007)

[9] Dao, T.B.H., Lallouet, A., Legtchenko, A., Martin, L. : Indexical-based solver learning. In : Proceedings of CP'02. LNCS, vol. 2470, pp. 541–555. Springer (2002)

[10] Dynadec, Dynamic Decision Technologies Inc. : Comet tutorial, v2.0 (2009)

[11] Frühwirth, T.W. : Theory and practice of constraint handling rules. Journal of Logic Programming 37(1–3), 95–138 (1998)

[12] Gecode Team : Gecode : A generic constraint development environment (2006), available from `http://www.gecode.org/`

[13] Gent, I.P., Jefferson, C., Miguel, I. : Watched literals for constraint propagation in minion. In : Proceedings of CP'06. LNCS, vol. 4204, pp. 182–197 (2006)

[14] JaCoP : Java constraint programming solver, `http://jacop.osolpro.com/`

[15] Mohr, R., Henderson, T. : Arc and path consistency revisited. Artificial Intelligence 28, 225–233 (1986)

[16] Monette, J.-N., Flener, P., Pearson, J. : Towards solver-independent propagators. In : Proceedins of CP'12. LNCS, vol. 7514, pp. 544–560 (2012)

[17] Ohrimenko, O., Stuckey, P., Codish, M. : Propagation via lazy clause generation. Contraintes 14, 357–391 (2009)

[18] Parr, T.J. : Enforcing strict model-view separation in template engines. In : Proceedings of the 13th intl. conf. on the World Wide Web. pp. 224–233. ACM (2004)

[19] Parr, T.J. : The Definitive ANTLR Reference : Building Domain-Specific Languages. The Pragmatic Bookshelf (2007)

[20] Régin, J.C. : A filtering algorithm for constraints of difference in CSPs. In : Proceedings of AAAI'94. pp. 362–367. AAAI Press (1994)

[21] Richaud, G., Lorca, X., Jussien, N. : A portable and efficient implementation of global constraints : The tree contrainte case. In : Proceedings of CICLOPS 2007. pp. 44–56 (2007)

[22] OscaR, `https://bitbucket.org/oscarlib/oscar/`

[23] Schulte, C., Stuckey, P.J. : Speeding up constraint propagation. In : Proceedings of CP'04. LNCS, vol. 3258, pp. 619–633. Springer (2004)

[24] Sidebottom, G., Havens, W.S. : Nicolog : A simple yet powerful cc(FD) language. Journal of Automated Reasoning 17, 371–403 (1996)

[25] Tack, G., Schulte, C., Smolka, G. : Generating propagators for finite set constraints. In : Proceedings of CP'06. LNCS, vol. 4204, pp. 575–589. Springer (2006)

[26] Van Hentenryck, P., Deville, Y. : The cardinality operator : A new logical connective for constraint logic programming. In : Proceedings of ICLP'91. pp. 745–759 (1991)

[27] Van Hentenryck, P., Michel, L. : Differentiable invariants. In : Proceedings of CP'06. LNCS, vol. 4204, pp. 604–619. Springer (2006)

[28] Van Hentenryck, P., Saraswat, V., Deville, Y. : Design, implementation, and evaluation of the constraint language cc(FD). Tech. Rep. CS-93-02, Brown University, Providence, USA (January 1993), revised version in *Journal of Logic Prog.* 37(1–3) :293–316, 1998. Based on the unpublished manuscript *Constraint Processing in cc(FD)*, 1991.

[29] Zhou, N.F. : Programming finite-domain constraint propagators in action rules. Theory and Practice of Logic Prog. 6, 483–507 (2006)

Sur une classe polynomiale hybride pour les CSP d'arité quelconque *

Achref El Mouelhi **Philippe Jégou** **Cyril Terrioux**

LSIS - UMR CNRS 7296
Aix-Marseille Université
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)

{achref.elmouelhi, philippe.jegou, cyril.terrioux}@lsis.org

Abstract

Exhiber de nouvelles classes polynomiales pour les CSP, c'est-à-dire, des classes de problèmes de satisfaction de contraintes pour lesquelles il existe des algorithmes de reconnaissance et de résolution de complexité polynomiale, constitue un axe de recherche fondamental dans l'étude des CSP. Dans ce domaine, le concept de classe hybride, qui permet de combiner à la fois des restrictions de langages et par exemple la vérification des propriétés structurelles, est une approche qui a déjà montré son intérêt. Ici, nous étudions une classe hybride pour les CSP non binaires. Pour cela, nous revenons sur la classe BTP proposée dans [5], au niveau binaire et pour laquelle les auteurs ont proposé une version concernant les CSP non binaires. Nous développons ce travail d'abord en fournissant une nouvelle définition, certes équivalente, mais cette fois-ci, énoncée dans les termes d'une propriété sémantique associée aux relations de compatibilités. Cette classe, appelée *DBTP*, est ensuite comparée à certaines classes parmi les plus connues de la littérature. En particulier, nous démontrons que DBTP, bien que s'appuyant sur BTP lui est incomparable, et qu'elle capture certaines classes bien connues dont notamment les CSP β-acycliques, lui conférant ainsi le statut de classe hybride, au sens où elle conjugue à la fois des aspects sémantiques mais aussi structurels.

1 Introduction

Une instance de CSP $P = (X, D, C)$ est définie par la donnée d'un ensemble X de n variables (notées $x_1, ..., x_n$), d'un ensemble de domaines $D = \{d_1, ..., d_n\}$ (d_i est l'ensemble des valeurs possibles pour la variable x_i) et un ensemble C de e contraintes (notées $c_1, ..., c_e$). Chaque contrainte c_i porte sur un ensemble $S(c_i)$ de r_i variables (appelé *portée* de c_i) et autorise un ensemble de tuples défini sur $\prod_{x_j \in S(c_i)} d_j$ exprimé par une relation de compatibilité $R(c_i)$. r_i note l'*arité* de la contrainte c_i. Nous noterons r l'arité maximum et $\rho = max\{|R(c_i)|\}$. En général, on distingue les contraintes binaires dont l'arité vaut 2 des contraintes d'arité quelconque (parfois dite n-aires). Aussi, les CSP binaires (CSP dont toutes les contraintes sont binaires) sont généralement considérés de façon différente des CSP dont les contraintes sont d'arité quelconque. Pour les CSP binaires, nous noterons c_{ij} la contrainte portant sur x_i et x_j. Que ce soit pour les CSP binaires ou les CSP d'arité quelconque, le problème d'existence d'une solution (i.e. une affectation de valeur à chaque variable qui satisfait toutes les contraintes) est NP-Complet.

Bien que ce problème soit NP-complet, il n'en demeure pas moins qu'il existe des classes d'instances qui peuvent être à la fois reconnues et résolues en temps polynomial. Cela leur confère le statut de "classes polynomiales". Elles se définissent sur la base de propriétés vérifiées par les instances. On peut recenser deux grands types de propriétés. Les premières concernent des traits structurels du réseau de contraintes. Par exemple, il est bien connu qu'un CSP binaire arborescent peut être résolu en temps linéaire [8]. Un autre type de propriétés s'énonce en termes de restrictions sur le langage définissant les contraintes. Ces

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

restrictions concernent les domaines et/ou les relations de compatibilité associées aux contraintes. C'est par exemple le cas des contraintes de la classe "0-1-Tous" ("ZUT") [4]. Plus récemment, un autre type de classes polynomiales a été mis en évidence, comme par exemple la classe BTP [5]. Leur intérêt porte notamment sur le fait qu'elle peuvent prendre en compte à la fois des restrictions relevant du langage qui les exprime ainsi que des aspects structurels. Ces classes sont souvent identifiées sous le vocable de "classes hybrides".

Dans cette contribution, nous étudions une classe hybride appelée DBTP pour "*Dual BTP.*" Elle s'appuie sur le concept de "Triangle Cassé" qui est à la base de BTP, et constitue cependant une classe polynomiale très différente. Alors que BTP n'est définie qu'au niveau des contraintes binaires, DBTP est définie pour les CSP dont les contraintes sont d'arité quelconque.

DBTP peut s'exprimer via l'expression d'une propriété relative à la compatibilité entre triplets de tuples figurant dans des triplets de relations de compatibilité. Cependant, cette propriété peut également être considérée comme l'expression de BTP appliquée sur la représentation duale d'un CSP. C'est d'ailleurs ainsi qu'elle a pour la première fois été évoquée dans [5]. Néanmoins, nous verrons que DBTP ne constitue pas pour autant une généralisation de BTP aux contraintes d'arité quelconque puisque pour le cas particulier des CSP binaires, BTP et DBTP sont formellement différentes (cf. théorème 12). Nous verrons également que cette classe polynomiale couvre simultanément des classes structurelles comme les CSP β-*acycliques* ainsi que des classes définies par des restrictions de langages. Nous montrerons également que DBTP est incomparable avec plusieurs autres classes bien connues de la littérature (par exemple *ZUT*, *row-convex* ou *max-closed* [4, 18, 11]).

En plus de ces résultats théoriques, nous avons prouvé que DBTP constitue une propriété conservatoire pour les filtrages classiques comme notamment la cohérence d'arc. Il semblerait ainsi que DBTP recèle un réel intérêt pratique puisque les instances DBTP peuvent être résolues en temps polynomial par l'usage d'algorithmes similaires à MAC [17] notamment.

Dans la partie 2, nous introduisons la classe DBTP dont nous fournissons les principales caractéristiques. Puis, dans la partie 3, nous étudions les relations qui peuvent exister entre BTP et DBTP pour le cas notamment des CSP binaires, et nous montrons que DBTP inclut la classe des CSP β-acycliques. La partie 4 examine les liens entre DBTP et d'autres classes polynomiales. Enfin, du fait des relations entre BTP et l'hyper-3-cohérence, nous étudions les liens entre (D)BTP et l'(hyper-)k-cohérence avant de conclure et de proposer différentes perspectives.

2 DBTP : définition et propriétés

Nous présentons d'abord la propriété DBTP d'un point de vue relationnel :

Définition 1 (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* $t_j[S(c_j) \cap S(c_k)] = t_k[S(c_j) \cap S(c_k)]$

où $t[Y]$ note la restriction du tuple t aux variables du sous-ensemble $Y \subseteq X$.

Nous notons DBTP l'ensemble de ces instances.

Une caractérisation alternative s'appuie sur la propriété BTP [5] et la notion d'instance duale [6]. Le *dual* du CSP $P = (X, D, C)$ est le CSP binaire $P^d = (X^d, D^d, C^d)$ où chaque contrainte c_i de C correspond à la variable x_i^d de X^d dont le domaine d_i^d est défini par l'ensemble des tuples t_i de $R(c_i)$, et une contrainte c_{ij}^d de C^d relie deux variables x_i^d et x_j^d de X^d si les contraintes associées c_i et c_j de C partagent au moins une variable. La relation $R(c^d)$ est alors définie par les tuples $(t_i, t_j) \in d_i^d \times d_j^d$ tels que $t_i[S(c_i) \cap S(c_j)] = t_j[S(c_i) \cap S(c_j)]$. Il est bien connu que tout CSP P possède une solution ssi P^d possède une solution.

Nous rappelons maintenant la propriété BTP :

Définition 2 (Broken Triangle Property [5]) *Une instance de CSP (X, D, C) vérifie 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 $x_i < x_j < x_k$, telles que $(v_i, v_j) \in R(c_{ij})$, $(v_i, v_k) \in R(c_{ik})$ 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 aucun de ces deux tuples n'existe, (v_i, v_j), (v_i, v_k) et (v_j, v'_k) est appelé **Triangle Cassé sur** x_k.*
Nous notons BTP l'ensemble de ces instances.

Le théorème suivant met en évidence les liens entre DBTP et BTP sur l'instance duale :

Théorème 1 *Un CSP $P = (X, D, C)$ vérifie DBTP par rapport à un ordre $\prec$ sur les contraintes ssi le dual de P vérifie BTP par rapport à l'ordre $\prec$.*

Preuve : P vérifie DBTP par rapport à l'ordre $\prec$

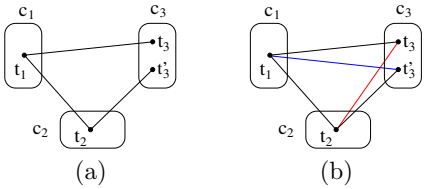


FIGURE 1 – Illustration de la propriété DBTP sur trois contraintes c_1, c_2 et c_3.

$\Leftrightarrow$ 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 $t_i[S(c_i) \cap S(c_j)] = t_j[S(c_i) \cap S(c_j)]$, $t_i[S(c_i) \cap S(c_k)] = t_k[S(c_i) \cap S(c_k)]$ et $t'_k[S(c_j) \cap S(c_k)] = t_j[S(c_j) \cap S(c_k)]$ alors soit $t'_k[S(c_i) \cap S(c_k)] = t_i[S(c_i) \cap S(c_k)]$, soit $t_j[S(c_j) \cap S(c_k)] = t_k[S(c_j) \cap S(c_k)]$

$\Leftrightarrow$ pour tout triplet de variables (x_i^d, x_j^d, x_k^d) tel que $x_i^d \prec x_j^d \prec x_k^d$, pour tout $t_i \in d_i^d$, $t_j \in d_j^d$ et $t_k, t'_k \in d_k^d$ tels que $(t_i, t_j) \in R(c_{ij}^d)$, $(t_i, t_k) \in R(c_{ik}^d)$ et $(t_j, t'_k) \in R(c_{jk}^d)$ alors soit $(t_i, t'_k) \in R(c_{ik}^d)$, soit $(t_j, t_k) \in R(c_{jk}^d)$

$\Leftrightarrow$ P^d vérifie BTP par rapport à l'ordre $\prec$. □

C'est d'ailleurs en termes d'expression duale que DBTP a d'abord été exprimée dans [5] puisque dans cet article, les auteurs ont évoqué le fait qu'un CSP dual, et donc binaire, pouvait vérifier BTP.

Sur la base de cette propriété, nous pouvons observer graphiquement la propriété DBTP sur la microstructure de l'instance duale. La *microstructure* [13] d'un CSP binaire $P = (X, D, C)$ est le graphe non-orienté $\mu(P) = (V, E)$ où $V = \{(x_i, v_i) : x_i \in X, v_i \in d_i\}$ et $E = \{ \{(x_i, v_i), (x_j, v_j)\} : i \neq j, c_{ij} \notin C$ ou $(v_i, v_j) \in R(c_{ij})\}$. La figure 1 présente la microstructure de l'instance duale d'un instance P concernant trois contraintes. Dans la figure 1(a), nous pouvons observer la présence d'un triangle cassé sur c_3 si nous considérons l'ordre $c_1 \prec c_2 \prec c_3$ et ainsi, P ne vérifie pas DBTP par rapport à cet ordre. Au contraire, dans la figure 1(b), si soit t_1 et t'_3 (arête bleue), soit t_2 et t_3 (arête rouge) sont compatibles, alors P vérifie DBTP relativement à l'ordre $\prec$.

Nous pouvons noter que DBTP est très différente de BTP. En particulier, une instance binaire peut vérifier DBTP tout en ne vérifiant pas BTP. Par exemple, l'instance binaire décrite dans la figure 2 est DBTP par rapport à l'ordre $c_{ij} \prec c_{jk} \prec c_{ik}$ mais elle n'est pas BTP. Il n'est pas surprenant que DBTP n'implique pas BTP car, même si l'instance originale et son expression duale représentent le même problème, leurs structures et microstructures sont très différentes. Les liens entre DBTP et BTP seront étudiés de façon plus détaillée dans la partie 3 qui leur est dédiée.

Nous montrons maintenant que la classe des CSPs vérifiant DBTP constitue une classe polynomiale. Pour

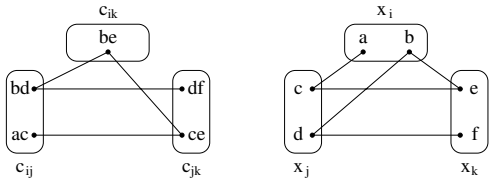


FIGURE 2 – Une instance vérifiant DBTP (a) mais pas BTP (b).

cela, et du fait du théorème 1, les preuves s'appuient sur les mêmes schémas que ceux utilisés dans [5].

Lemme 1 *Tout CSP $P = (X, D, C)$ qui vérifie DBTP par rapport à un ordre $\prec$ sur les contraintes peut être résolu en $O(e^2.r.\rho^2)$.*

Preuve : La première étape consiste à construire le dual de P, ce qui peut être réalisé en $O(e^2.r.\rho^2)$. Ensuite, comme le dual de P est BTP, nous savons qu'il peut être résolu en $O(e^2.\rho^2)$ [5]. Ainsi, la complexité globale est en $O(e^2.r.\rho^2)$. □

Le lemme 2 exprime le fait qu'un ordre $\prec$ sur les contraintes et associé à DBTP peut être calculé (le cas échéant) en temps polynomial.

Lemme 2 *Étant donné un CSP $P = (X, D, C)$, déterminer si un ordre $\prec$ sur les contraintes tel que P est DBTP par rapport à $\prec$ existe (et le trouver le cas échéant) peut être réalisé en temps polynomial.*

Preuve : Un algorithme possible consiste à calculer d'abord le dual de P, puis à déterminer si un ordre $\prec$ tel que le dual de P est BTP existe comme proposé dans [5]. Chaque étape est polynomiale (voir la preuve précédente et [5]). Par conséquent, la complexité globale est polynomiale. □

Du fait de ces deux lemmes, nous pouvons déduire le théorème suivant :

Théorème 2 *DBTP est une classe polynomiale.*

Nous étudions maintenant ce qu'il en est de la propriété DBTP dans le cas de l'application d'un algorithme de filtrage sur une instance DBTP. Une classe $\mathcal{C}$ d'instances de CSP est dite **conservatoire** par rapport à un filtrage de cohérence ϕ si elle est fermée pour ϕ, c'est-à-dire, si le problème obtenu après l'application de ϕ à toute instance de $\mathcal{C}$ appartient à la classe $\mathcal{C}$. Une propriété est dite conservatoire si elle définit une classe d'instances conservatoires.

Propriété 1 *DBTP est conservatoire pour tout filtrage qui se limite à supprimer des valeurs dans les domaines ou des tuples dans les relations.*

Preuve : Considérons un CSP P vérifiant DBTP par rapport à un ordre donné sur les contraintes. La suppression d'une valeur du domaine d'une variable x de P conduit à une suppression de tuples pour les contraintes dont la portée contient x. En d'autres termes, cela revient à supprimer des valeurs dans les domaines de variables du dual de P. Par conséquent, dans les deux cas, les suppressions de valeurs ou de tuples dans l'instance d'origine conduisent à supprimer les valeurs des variables duales. Comme BTP est conservatoire pour les cohérences filtrant les domaines, le dual de P, après ces suppressions, vérifie encore BTP. Par conséquent, P demeure DBTP. □

Cette propriété est donc valable pour tout filtrage de domaine (par exemple pour la cohérence d'arc généralisée ou la cohérence inverse de chemin [2]), qu'il soit appliqué sur l'instance originale ou son instance duale. C'est également le cas pour l'intercohérence ou pairwise-consistency (introduite dans le cadre de la Théorie des Bases de Données Relationelles [1]) dont l'application est équivalente à celle de l'application de la cohérence d'arc sur l'instance duale.

Définition 3 (Intercohérence [10]) *Un CSP $P = (X, D, C)$ est* ***intercohérent ou pairwise-consistant*** *ssi $\forall\ 1 \leq i \leq e$, $R(c_i) \neq \emptyset$ et $\forall\ 1 \leq i < j \leq e$, $R(c_i)[S(c_i) \cap S(c_j)] = R(c_j)[S(c_i) \cap S(c_j)]$.*

Comme MAC [17] maintient la cohérence d'arc (notée AC) à chaque étape de la recherche, nous pouvons définir MPWC comme l'algorithme correspondant au maintien de l'intercohérence.

Théorème 3 *Si $P = (X, D, C)$ vérifie DBTP, alors MPWC résout P en temps polynomial pour tout ordre.*

Preuve : Comme l'intercohérence sur P est équivalente à la cohérence d'arc sur le dual de P [10], l'application de MPWC sur P est équivalente à celle de MAC sur le dual de P. De plus, comme P est DBTP, P^d est BTP et ainsi, selon le théorème 7.6 de [5], MPWC résout P en temps polynomial. □

Ce résultat est également vérifié pour MAC mais dans le cas particulier où tout couple de contraintes partage au plus une variable. Avant de le montrer, il nous faut rappeler deux résultats sur la cohérence d'arc et l'intercohérence.

Lemme 3 (Propriété 8.1 page 146 dans [12]) *Soit $P = (X, D, C)$ un CSP tel que $\forall c_i, c_j \in C, |S(c_i) \cap S(c_j)| \leq 1$. Si l'instance P vérifie la cohérence d'arc, alors elle vérifie l'intercohérence.*

Lemme 4 *Soit $P = (X, D, C)$ un CSP vérifiant la cohérence d'arc et tel que $\forall c_i, c_j \in C, |S(c_i) \cap S(c_j)| \leq 1$. Si l'instance P' obtenue à partir de P en supprimant certaines valeurs et en appliquant AC ne possède pas de domaine vide, alors son dual vérifie la cohérence d'arc.*

Preuve : Considérons P et P' obtenu à partir de P en supprimant certaines valeurs et en appliquant AC, tel qu'il ne possède pas de domaine vide. Puisque P' vérifie la cohérence d'arc, d'après le lemme 3, il vérifie aussi l'intercohérence. Ainsi, comme l'intercohérence sur P est équivalente à la cohérence d'arc sur le dual de P [10], le dual de P' vérifie la cohérence d'arc. □

Théorème 4 *Si $P = (X, D, C)$ est tel que $\forall c_i, c_j \in C, |S(c_i) \cap S(c_j)| \leq 1$, et qu'il vérifie la cohérence d'arc ainsi que DBTP, alors MAC peut résoudre P en temps polynomial.*

Preuve : Si après l'obtention de la cohérence d'arc, aucun domaine, ni relation n'est vide, alors P vérifie l'intercohérence et possède une solution. D'après le lemme 4, le problème obtenu après la suppression de valeurs et le filtrage par cohérence d'arc demeure intercohérent. Par conséquent, lors de l'application de MAC sur le problème initial, nous maintenons également l'intercohérence. De plus, comme l'intercohérence est équivalente à la cohérence d'arc sur le problème dual [10], le théorème 7.6 de [5] fait que MAC résout P en temps polynomial puisque le dual est BTP. □

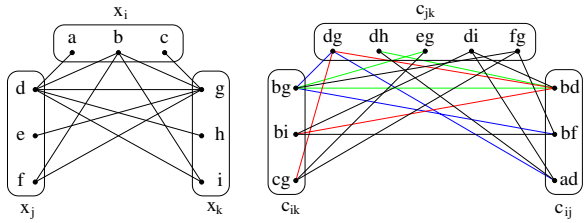
Ce théorème est bien entendu vérifié pour tout CSP binaire.

3 Relations entre DBTP et BTP

La classe $DBTP$ diffère nécessairement de la classe BTP puisque $DBTP$ peut contenir des instances non-binaires alors que BTP n'a été définie qu'au niveau binaire, ce qui conduit à ne se poser la question de leur comparaison que dans ce cadre. Comme le montre la figure 2, une instance peut vérifier DBTP mais pas BTP. Inversément, une instance peut vérifier BTP sans qu'elle ne vérifie DBTP. Ce cas est illustré dans la figure 3 (où les triangles cassés coloriés prouvent que DBTP n'est pas vérifiée). Ces résultats étaient prévisibles, puisque, même si l'instance d'origine et son dual représentent le même problème, leur structure et leur microstructure sont en fait différentes. Ainsi, à partir des exemples des figures 2 et 3, nous obtenons :

Théorème 5 *Soit $P = (X, D, C)$ un CSP binaire.*
- *P vérifie DBTP $\not\Rightarrow$ P vérifie BTP,*
- *P vérifie BTP $\not\Rightarrow$ P vérifie DBTP.*

Les résultats précédents reposent sur la présence de triangles cassés dans la microstructure de l'instance ou



(a) (b)

FIGURE 3 – Une instance vérifiant BTP (a) mais pas DBTP (b).

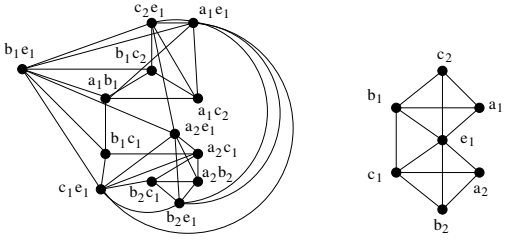
de son instance duale. Dans chaque cas, ces triangles cassés concernent des valeurs qui pourraient être supprimées par un certain filtrage comme la cohérence d'arc notamment. Ainsi, comme DBTP et BTP sont conservatoires par rapport aux filtrages de domaines, nous focalisons notre étude sur les instances binaires qui satisfont la cohérence d'arc et ainsi que l'intercohérence (du fait du lemme 4). Sous ces hypothèses, nous déduisons le lemme suivant :

Lemme 5 *Étant donné un CSP binaire* $P = (X, D, C)$ *vérifiant la cohérence d'arc, si pour un triplet* (x_i, x_j, x_k) *de variables, nous disposons d'un triangle cassé, alors il existe deux triangles cassés pour le triplet* (c_{ij}, c_{ik}, c_{jk}) *dans l'instance duale.*

Preuve : Soient x_i, x_j, $x_k \in X$ telles que $(v_i, v_j) \in R(c_{ij})$, $(v_i, v_k) \in R(c_{ik})$, $(v_j, v'_k) \in R(c_{jk})$, $(v_i, v'_k) \notin R(c_{ik})$ et $(v_j, v_k) \notin R(c_{jk})$. Comme P est intercohérent, il existe des valeurs $v'_i \in d_i$ et $v'_j \in d_j$ telles que $v_i \neq v'_i$, $v_j \neq v'_j$, $(v'_i, v'_k) \in R(c_{ik})$ et $(v'_j, v_k) \in R(c_{jk})$. Aussi, $((v_i, v_j), (v_i, v_k))$, $((v_i, v_j), (v_j, v'_k))$ et $((v_i, v_k), (v'_j, v_k))$ forment un triangle cassé sur c_{jk} pour le triplet (c_{ij}, c_{ik}, c_{jk}). Il en est de même pour $((v_i, v_j), (v_j, v'_k))$, $((v_i, v_j), (v_i, v_k))$ et $((v_j, v'_k), (v'_i, v'_k))$ sur c_{ik}. □

Par conséquent, quand un triangle cassé pour un triplet (x_i, x_j, x_k) impose la condition $x_k < max(x_i, x_j)$ sur l'ordre $<$ des variables, cela revient à imposer les deux conditions $c_{jk} \prec max(c_{ij}, c_{ik})$ et $c_{ik} \prec max(c_{ij}, c_{jk})$ pour le triplet (c_{ij}, c_{ik}, c_{jk}) sur l'ordre $\prec$ des contraintes. Il s'ensuit que toute instance binaire arc-cohérente et intercohérente qui satisfait BTP et dispose de deux triangles cassés pour deux variables différentes d'une même triplet de variables ne peut satisfaire DBTP puisque nous obtiendrions tous les triangles cassés possibles pour le triplet correspondant de contraintes.

Inversément, considérons une instance binaire avec neuf variables $\{x_a, x_b, \ldots, x_i\}$. Nous définissons cet exemple en reproduisant plusieurs fois un même motif qui est tel que chaque valeur apparaissant dans une occurrence de ce motif n'apparaît dans aucune autre occurrence.



(a) (b)

FIGURE 4 – Morceau d'une instance non BTP mais vérifiant DBTP, la cohérence d'arc et l'intercohérence.

Ce motif consiste en un triangle cassé sur une variable z pour un triplet (x, y, z) (c'est-à-dire qui impose la condition $z < max(x, y)$ sur $<$) et chaque valeur des variables x, y et z est liée à une valeur donnée d'une variable qui n'est pas impliquée dans ce triplet. Nous reproduisons ce schéma 9 fois de telle sorte que les conditions suivantes soient imposées : $x_a < max(x_b, x_c)$, $x_b < max(x_e, x_h)$, $x_c < max(x_e, x_g)$, $x_d < max(x_a, x_g)$, $x_e < max(x_a, x_i)$, $x_f < max(x_d, x_e)$, $x_g < max(x_h, x_i)$, $x_h < max(x_b, x_d)$ et $x_i < max(x_c, x_f)$. La figure 4(b) décrit ce motif pour le triplet (x_a, x_b, x_c), un triangle cassé sur x_a (correspondant à la condition $x_a < max(x_b, x_c)$) et une variable indépendante x_e tandis que la figure 4 (a) décrit la partie correspondante dans l'instance duale. En faisant cela, la microstructure de notre CSP binaire et celle de son instance duale ont 9 composantes connexes. On peut noter que cette instance n'est pas BTP parce que les 9 conditions rendent impossible la construction d'un ordre approprié sur les variables. En revanche, il est DBTP (par rapport à l'ordre $c_{ab} \prec c_{ac} \prec c_{ad} \prec c_{bf} \prec c_{bh} \prec c_{ci} \prec c_{df} \prec c_{dh} \prec c_{ef} \prec c_{ei} \prec c_{gh} \prec c_{gi} \prec c_{af} \prec c_{bc} \prec c_{bd} \prec c_{ce} \prec c_{cg} \prec c_{de} \prec c_{dg} \prec c_{fi} \prec c_{hi} \prec c_{ae} \prec c_{ag} \prec c_{be} \prec c_{bg} \prec c_{cd} \prec c_{cf} \prec c_{di} \prec c_{fh} \prec c_{ai} \prec c_{bi} \prec c_{eg} \prec c_{eh} \prec c_{fg} \prec c_{ah} \prec c_{ch}$), et il vérifie la cohérence d'arc et l'intercohérence.

Néanmoins, pour des niveaux plus élevés de cohérence, il n'en est pas nécessairement de même. Nous pouvons notamment démontrer qu'une instance binaire chemin-cohérente et DBTP est BTP :

Théorème 6 *Si un CSP binaire* P *vérifie DBTP par rapport à un ordre* $\prec$ *sur les contraintes et s'il vérifie la chemin-cohérence, alors* P *vérifie BTP par rapport à tout ordre* $<$ *sur les variables.*

Preuve : Supposons que P ne vérifie pas BTP. Alors, pour tout ordre $<$ sur les variables, il existe un triplet $x_i < x_j < x_k$ tel que $\exists v_i \in d_i, v_j \in d_j$ et $v_k, v'_k \in d_k$, $(v_i, v_j) \in R(c_{ij}), (v_i, v_k) \in R(c_{ik})$ et $(v_j, v'_k) \in R(c_{jk})$, $(v_j, v_k) \notin R(c_{jk})$ et $(v_i, v'_k) \notin R(c_{ik})$. Comme P vérifie la cohérence de chemin, $\exists v'_i \in d_i, v'_j \in d_j$ et $v''_k \in d_k, (v_i, v'_j) \in R(c_{ij}), (v'_i, v_j) \in R(c_{ij})$, $(v'_i, v'_k) \in R(c_{ik})$, $(v_i, v''_k) \in R(c_{ik})$, $(v'_j, v_k) \in R(c_{jk})$ et

$(v_j, v''_k) \in R(c_{jk})$. Par conséquent, il est facile de voir qu'il n'y a aucun ordre sur les contraintes tel que P satisfasse DBTP, et donc P ne satisfait pas DBTP. Ainsi, nous obtenons une contradiction et P satisfait BTP. □

Nous étudions maintenant le cas des CSP acycliques pour lesquels [5] a déjà prouvé que ces CSP binaires vérifient BTP. Nous allons montrer que cela est également vrai pour DBTP. Soit $TREE$ l'ensemble des CSP binaires dont le graphe de contraintes est acyclique.

Théorème 7 *$TREE \subsetneq DBTP$.*

Preuve : Soit *DUAL-TREE* l'ensemble des CSP binaires dont chaque élément est le dual d'une instances de $TREE$. Comme cela a été montré dans [5], $DUAL\text{-}TREE \subsetneq BTP$. Par conséquent, $TREE \subsetneq DBTP$. □

Ce résultat peut être étendu aux CSP d'arité quelconque. Pour cela, nous devons considérer la notion de cyclicité dans les hypergraphes, pour lesquels différents degrés ont été définis [1]. Ici, nous nous intéressons en particulier à l'α-acyclicité et à la β-acyclicité. Nous allons montrer que les CSP β-acycliques vérifient DBTP, alors que ce n'est pas le cas pour les CSP α-acycliques. Nous rappelons d'abord la définition de la β-acyclicité d'un hypergraphe (de contraintes).

Définition 4 ([9]) *Une séquence $(c_1, ..., c_m, c_{m+1})$ avec $m \geq 3$ telle que $(c_1, ..., c_m)$ sont distincts et $c_1 = c_{m+1}$ est un cycle de Graham si chaque $\Delta_i = S(c_i) \cap S(c_{i+1})$ $(1 \leq i \leq m)$ n'est pas vide, et chaque fois que $i \neq j$, Δ_i et Δ_j sont incomparables (i.e. $\Delta_i \not\subseteq \Delta_j$ et $\Delta_j \not\subseteq \Delta_i$). $H = (X, C)$ est un hypergraphe **β-acyclique** ssi il ne possède pas de cycle de Graham.*

Il a récemment été montré dans [7] que les hypergraphes β-acycliques peuvent être définis en appliquant les deux règles suivantes, qui doivent mener à l'hypergraphe vide :

(1) Si une hyperarête est vide, elle est retirée de C.

(2) Si un sommet est un point de type "nest" (i.e. l'ensemble des hyperarêtes le contenant est une chaîne pour la relation d'inclusion), alors il est retiré de H (i.e. il est retiré de X ainsi que des hyperarêtes qui le contiennent).

Théorème 8 ([7]) *Un hypergraphe H est β-acyclique ssi, après l'application successive des deux règles jusqu'à ce qu'aucune ne puisse l'être, nous obtenons l'hypergraphe vide.*

En utilisant ces définitions, nous pouvons maintenant établir le théorème suivant :

Théorème 9 *Étant donné un CSP (X, D, C), il existe un ordre sur les contraintes $\prec$, tel que $\forall c_i, c_j, c_k \in C$ tel que $c_i \prec c_j \prec c_k$, nous avons $S(c_i) \cap S(c_k) \subseteq S(c_j) \cap S(c_k)$ ou $S(c_j) \cap S(c_k) \subseteq S(c_i) \cap S(c_k)$ ssi (X, D, C) possède un hypergraphe de contraintes β-acyclique.*

Preuve : ($\Rightarrow$) Par contraposition. Nous montrons que si l'hypergraphe de contraintes (X, C) est β-cyclique, alors, il ne peut exister d'ordre sur les contraintes.

Considérons un hypergraphe de contraintes (X, C) qui est β-cyclique. Il possède donc un cycle de Graham, que l'on notera par la séquence d'hyperarêtes $(c_1, ..., c_m, c_{m+1})$. Considérons un ordre quelconque sur les contraintes $\prec$. Nécessairement, parmi les contraintes de ce cycle, il existe une contrainte maximum c_k par rapport à l'ordre $\prec$. Considérons ses deux voisines dans le cycle, notées c_i et c_j (avec $c_i, c_j \prec c_k$). Par définition des cycles de Graham, nous savons que $S(c_i) \cap S(c_k)$ et $S(c_j) \cap S(c_k)$ sont incomparables. Aussi, nous n'avons ni $S(c_i) \cap S(c_k) \subseteq S(c_j) \cap S(c_k)$ ni $S(c_j) \cap S(c_k) \subseteq S(c_i) \cap S(c_k)$, et donc aucun ordre correct sur les contraintes $\prec$ ne peut exister.

($\Leftarrow$) Nous utilisons ici le théorème 8. Étant donné un CSP β-acyclique (X, D, C) d'hypergraphe H qui admet un ordre $\prec$ correct sur les contraintes, nous allons montrer que :

(1) Une hyperarête $S(c_i)$ est vide ssi H sans $S(c_i)$ admet un ordre et est β-acyclique.

(2) Un sommet x de H est un point de type "nest" tel que H admet un ordre ssi H sans x admet un ordre et est β-acyclique.

On voit immédiatement que la propriété est vérifiée par l'application de la règle (1). Considérons donc la règle (2). Supposons que pour un hypergraphe H nous disposons d'un ordre $\prec$. Aussi, $\forall c_i, c_j, c_k \in C$ tel que $c_i \prec c_j \prec c_k$, nous avons $S(c_i) \cap S(c_k) \subseteq S(c_j) \cap S(c_k)$ ou $S(c_j) \cap S(c_k) \subseteq S(c_i) \cap S(c_k)$. Nous avons 5 cas à considérer :

1. $x \notin S(c_i) \cup S(c_j) \cup S(c_k)$: après la suppression de x, ni $S(c_i) \cap S(c_k)$, ni $S(c_j) \cap S(c_k)$ n'ont changées. La propriété est donc vérifiée.
2. $x \in S(c_i) \cap S(c_j) \cap S(c_k)$: après la suppression de x, ce sommet disparaît de chaque intersection $S(c_i) \cap S(c_k)$ et $S(c_j) \cap S(c_k)$, et ainsi, la propriété est vérifiée.
3. x n'appartient qu'à un seul des ensembles $S(c_i)$ ou $S(c_j)$ ou $S(c_k)$: donc x n'appartient à aucune intersection, et donc la propriété est vérifié après la suppression.
4. $x \in S(c_i) \cap S(c_j)$ et $x \notin S(c_k)$: donc x n'appartient ni à $S(c_i) \cap S(c_k)$, ni à $S(c_j) \cap S(c_k)$, et donc la propriété est vérifiée après la suppression.

5. $x \in S(c_i) \cap S(c_k)$ et $x \notin S(c_j)$ (ou symétriquement $x \in S(c_j) \cap S(c_k)$ et $x \notin S(c_i)$) : donc, avant la suppression, nous avions nécessairement $S(c_i) \cap S(c_k) \not\subseteq S(c_j) \cap S(c_k)$ et $S(c_j) \cap S(c_k) \subsetneq S(c_i) \cap S(c_k)$. Ainsi, après la suppression de x, nous avons au moins $S(c_j) \cap S(c_k) \subseteq S(c_i) \cap S(c_k)$.

Donc, nous avons montré que si nous pouvons réduire tout hypergraphe, ce qui ne contredit pas la propriété sur l'ordre, et donc, nécessairement, l'hypergraphe d'origine est β-acyclique et admet un ordre approprié sur les contraintes. □

Nous pouvons noter que ce théorème explique pourquoi la condition énoncée dans le lemme 4.6 de [5] est vérifiée indépendamment de la portée des contraintes. Ce lemme et le théorème précédent nous permettent d'obtenir le théorème 10 où β-$ACYCLIC$ est l'ensemble des CSP pour lesquels l'(hyper)graphe de contraintes est β-acyclique.

Théorème 10 *$TREE \subsetneq \beta$-$ACYCLIC \subsetneq DBTP$.*

Toutefois, l'équivalence proposée dans le lemme 4.6 [5] est seulement vérifiée dans le sens inverse (ce qui ne compromet pas la preuve du théorème 10). Pour s'en rendre compte, il suffit de considérer une instance binaire avec 3 variables monovalentes (une seule valeur par domaine) mutuellement connectées (chaque contrainte admet l'unique tuple possible). Son dual satisfait BTP bien que le graphe de contraintes ne soit pas β-acyclique. Cet exemple de CSP peut bien entendu être généralisé à des tailles de domaines et à un nombre de variables quelconques en reproduisant un motif identique.

Nous montrons maintenant que si α-$ACYCLIC$ est l'ensemble des CSP dont l'(hyper)graphe de contraintes est α-acyclique, alors les ensembles α-$ACYCLIC$ et $DBTP$ sont incomparables. Pour cela, nous rappelons que l'α-acyclicité d'un (hyper)graphe de contraintes peut être définie en utilisant la "running intersection property" [1], à savoir :

Définition 5 *(X, C) est un hypergraphe **α-acyclique** ssi il existe un ordre $(c_1, ..., c_e)$ tel que*

$$\forall k, 1 < k \leq e, \exists j < k, (S(c_k) \cap \bigcup_{i=1}^{k-1} S(c_i)) \subseteq S(c_j).$$

Considérons un CSP possédant six variables $x_a, \ldots x_f$ et quatre contraintes dont les portées sont respectivement $\{x_a, x_b, x_c\}$, $\{x_a, x_b, x_d\}$, $\{x_a, x_c, x_e\}$ et $\{x_b, x_c, x_f\}$. La figure 5 présente son hypergraphe de contraintes (a) et la microstructure de son dual (b). Nous pouvons constater que cette instance est α-acyclique mais qu'elle ne vérifie pas DBTP puisqu'aucun ordre approprié sur les contraintes ne peut exister. De plus, il est bien connu que β-$ACYCLIC \subsetneq \alpha$-$ACYCLIC$ [1]. Par conséquent, si l'on note par $A \perp B$ le fait que deux classes polynomiales sont incomparables (i.e. nous n'avons ni $A \subseteq B$ ni $B \subseteq A$), nous obtenons le théorème suivant :

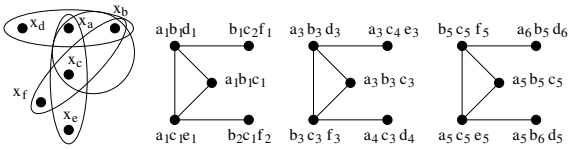


FIGURE 5 – Un CSP α-acyclique (a) qui ne vérifie pas DBTP (b).

Théorème 11 *α-$ACYCLIC \cap DBTP \neq \emptyset$ et α-$ACYCLIC \perp DBTP$.*

Dans cette partie, nous avons donc globalement établi le résultat suivant :

Théorème 12 *$BTP \cap DBTP \neq \emptyset$ et $BTP \perp DBTP$.*

Dans la partie suivante, nous étudions le lien entre DBTP et quelques autres classes polynomiales.

4 DBTP vs Quelques classes polynomiales

4.1 Cas des CSP binaires

Comme DBTP et BTP sont deux classes différentes, nous nous concentrons d'abord sur certaines classes traitables incluses dans BTP. Ces classes dont les définitions sont rappelées ci-dessous s'appuient sur des restrictions de langages de contraintes.

Définition 6 (Row-convex [18]) *Un CSP binaire $P = (X, D, C)$ est dit **row-convex** par rapport à un ordre $<$ sur les variables et à un ordre sur les valeurs, si, pour chaque contrainte c_{ij} de C avec $x_i < x_j$, $\forall v_i \in d_i$, $\{v_j \in d_j | (v_i, v_j) \in R(c_{ij})\} = [a_j..b_j]$ pour a_j, $b_j \in d_j$ où $[a_j..b_j]$ représente le valeurs de d_j entre a_j et b_j par rapport à l'ordre sur les valeurs.*
Nous noterons RC l'ensemble des instances row-convex.

Définition 7 (0-1-tous [4]) *Un CSP binaire CSP $P = (X, D, C)$ est dit **0-1-tous** 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) pour chaque valeur $v_j \in d_j, (v_i, v_j) \notin R(c_{ij})$,*
- *(UN) il y a une valeur unique $v_j \in d_j$ telle que $(v_i, v_j) \in R(c_{ij})$,*
- *(TOUS) pour chaque valeur $v_j \in d_j, (v_i, v_j) \in R(c_{ij})$.*

Nous noterons ZUT l'ensemble des instances vérifiant la propriété 0-1-tous.

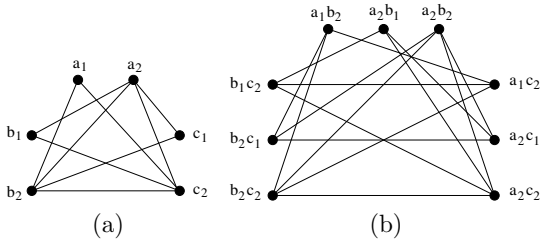


FIGURE 6 – Un CSP appartenant à RC, ZUT et RRM (a) mais qui ne vérifie pas DBTP (b).

Définition 8 (Renamable right monotone [5]) *Un CSP binaire $P = (X, D, C)$ est dit **renamable right monotone** par rapport à un ordre $<$ sur les variables, si, pour $2 \leq j \leq n$, chaque domaine d_j peut-être ordonné par $\lessdot_j$ de telle sorte que pour chaque contrainte c_{ij} de C avec $x_i < x_j$, $\forall v_i \in d_i, v_j, v'_j \in d_j$, si $(v_i, v_j) \in R(c_{ij})$ et $v_j \lessdot_j v'_j$ alors $(v_i, v'_j) \in R(c_{ij})$.*
Nous noterons RRM l'ensemble de ces instances.

Le théorème suivant montre que ces classes polynomiales partagent des instances avec DBTP mais sont cependant différentes.

Théorème 13 *$RC \cap DBTP \neq \emptyset$ et $RC \perp DBTP$.*
$ZUT \cap DBTP \neq \emptyset$ et $ZUT \perp DBTP$.
$RRM \cap DBTP \neq \emptyset$ et $RRM \perp DBTP$.

Preuve : Si l'on considère le CSP binaire de la figure 6(a), il est *0-1-tous*, *row convex*, et *renamable right monotone* par rapport aux ordres lexicographique sur les valeurs et les variables. Cependant, comme le montre la figure 6(b), cette instance n'est pas DBTP. Inversement, toute instance non binaire DBTP ne peut appartenir à RC, ZUT ou RRM.

Afin de prouver que $DBTP$ intersecte RC, ZUT et RRM, il suffit de considérer un CSP binaire monovalent à trois variables et trois contraintes qui est cohérent. En effet, ce type d'instance satisfait à la fois $DBTP$, RC, ZUT et RRM. □

La classe suivante repose sur le nombre de cliques maximales figurant dans la microstructure :

Définition 9 (Maximal clique bounded [15]) *Un CSP $P = (X, D, C)$ est dit **maximal clique bounded** si sa microstructure possède un nombre polynomial de cliques maximales.*
Nous noterons CL l'ensemble de ces instances.

Théorème 14 *$CL \cap DBTP \neq \emptyset$ et $CL \perp DBTP$.*

Preuve : Tout CSP binaire monovalent et cohérent possède une seule clique maximale et il est DBTP. Donc, l'intersection n'est pas vide.

Considérons maintenant un CSP binaire tel que sa microstructure recèle un nombre polynomial de cliques maximales. Nous rajoutons à ce CSP binaire des variables supplémentaires avec d'autres valeurs et contraintes supplémentaires correspondant à l'instance représentée dans la figure 1, de telle sorte que ces valeurs ne soient pas compatibles avec celles de la première partie de ce CSP. De cette façon, il possède un nombre polynomial de cliques maximales dans sa microstructure mais il n'est pas DBTP. Inversement, toute instance de $DBTP$ non binaire ne peut appartenir à CL. □

En ce qui concerne les classes basées sur des structures restreintes, nous avons prouvé dans le théorème 10 que $TREE \subsetneq DBTP$.

4.2 CSP d'arités quelconques

Nous considérons d'abord certaines classes polynomiales connues basées sur des restrictions de langages de contraintes comme la classe *max-closed*.

Définition 10 (Max-closed [11]) *Un CSP $P = (X, D, C)$ est dit **max-closed** si pour chaque contrainte c d'arité r_c, $\forall (v_1, v_2, \ldots, v_{r_c}), (v'_1, v'_2, \ldots, v'_{r_c}) \in R(c)$, $(max(v_1, v'_1), max(v_2, v'_2), \ldots, max(v_{r_c}, v'_{r_c})) \in R(c)$.*
Nous notons MC l'ensemble de ces instances.

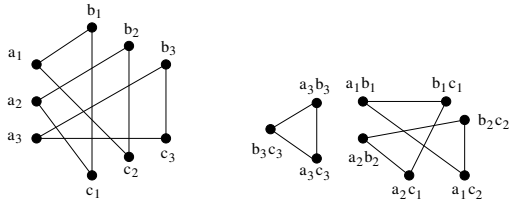
Théorème 15 *$MC \cap DBTP \neq \emptyset$ et $MC \perp DBTP$.*

Preuve : La preuve de $MC \cap DBTP \neq \emptyset$ et de $MC \not\subseteq DBTP$ est similaire à celle du théorème 13. En ce qui concerne $DBTP \not\subseteq MC$, tout CSP ayant deux variables et une contrainte binaire est DBTP mais pas nécessairement *max-closed*. □

Définition 11 (incrementally functional [3]) *Un $P = (X, D, C)$ est dit **incrementally functional** s'il existe un ordre $<$ sur les variables, tel que pour $1 \leq i < n$, chaque solution de $P[\{x_1, \ldots, x_i\}]$ peut s'étendre au plus à une solution de $P[\{x_1, \ldots, x_{i+1}\}]$ où, pour $X' \subseteq X, P[X']$ notant le CSP (X', D', C') où $D' = \{d_i | x_i \in X'\}$ et $C' = \{c' | c \in C$ tel que $S(c) \cap X' \neq \emptyset, S(c') = S(c) \cap X'$ et $R(c') = \{t[S(c')] | t \in R(c)\}\}$.*
Nous notons $IFUN$ l'ensemble de ces instances.

Théorème 16 *$IFUN \cap DBTP \neq \emptyset$ et $IFUN \perp DBTP$.*

Preuve : Afin de prouver que l'intersection n'est pas vide, nous considérons un CSP avec quatre variables monovalentes $x_1, \ldots, x_4$ et trois contraintes ternaires c_1, c_2 et c_3 telles que $S(c_1) = \{x_1, x_2, x_3\}$, $R(c_1) = \{(v_1, v_2, v_3)\}$, $S(c_2) = \{x_1, x_2, x_4\}$, $R(c_2) = \{(v_1, v_2, v_4)\}$, $S(c_3) = \{x_2, x_3, x_4\}$ et $R(c_3) = \{(v_2, v_3, v_4)\}$. Cette instance est *incrementally functional* (en utilisant la numérotation des



(a) (b)

FIGURE 7 – Une instance *incrementally functional* (a) mais qui ne vérifie pas DBTP (b).

variables comme ordre) et DBTP. L'instance présentée dans la figure 7 est *incrementally functional* mais pas DBTP. Inversement, toute instance DBTP ayant plusieurs solutions ne peut pas être *incrementally functional*. □

Définition 12 (Dual CB [15]) *$P = (X, D, C)$ est dit **dual maximal clique bounded (DMCB)** si la microstructure de son instance duale possède un nombre polynomial de cliques maximales. Nous notons DCL l'ensemble de ces instances.*

Théorème 17 *$DCL \cap DBTP \neq \emptyset$ et $DCL \perp DBTP$.*

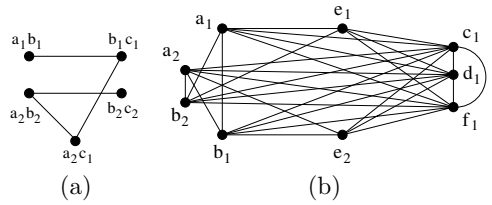
Preuve : Considérons la première instance définie dans la preuve du théorème 16. La microstructure de son instance duale possède une seule clique maximale et l'instance est DBTP. Ainsi, l'intersection n'est pas vide. En ce qui concerne l'exemple représenté dans la figure 7, elle possède un nombre polynomial de cliques maximales dans la microstructure de son instance duale mais n'est pas DBTP. Inversement, une instance binaire dont le graphe de contraintes est une étoile et pour lequel chaque domaine dispose de plusieurs valeurs est DBTP mais possède un nombre de cliques maximales dans sa microstructure duale non borné polynomialement. □

Maintenant, nous introduisons une nouvelle classe polynomiale basée sur une restriction du langage de contraintes.

Définition 13 (Triangulaire) *Un $P = (X, D, C)$ est dit **triangulaire** par rapport à un ordre sur les contraintes $\prec$ ssi $\forall c_i, c_j, c_k$, $c_i \prec c_j \prec c_k$, $\forall t_i \in R(c_i), t_j \in R(c_j), t_k \in R(c_k)$, if*
- *$t_i[S(c_i) \cap S(c_j)] = t_j[S(c_i) \cap S(c_j)]$ et*
- *$t_i[S(c_i) \cap S(c_k)] = t_k[S(c_i) \cap S(c_k)]$.*

alors, $t_j[S(c_j) \cap S(c_k)] = t_k[S(c_j) \cap S(c_k)]$.
Nous notons TR l'ensemble de ces instances.

Théorème 18 *Si un CSP P est triangulaire par rapport à un ordre $\prec$, alors P vérifie DBTP par rapport à $\prec$.*



(a) (b)

FIGURE 8 – (a) Une instance vérifiant DBTP mais qui n'est pas triangulaire. (b) Partie d'une instance vérifiant l'hyper-3-cohérence orientée mais pas BTP.

Preuve : Supposons que P soit triangulaire mais pas DBTP. Alors, il existe trois contraintes c_i, c_j et c_k, $c_i \prec c_j \prec c_k$, $t_i \in R(c_i)$, $t_j \in R(c_j)$ et $t_k, t'_k \in R(c_k)$ telles que $t_i[S(c_i) \cap S(c_j)] = t_j[S(c_i) \cap S(c_j)]$, $t_i[S(c_i) \cap S(c_k)] = t_k[S(c_i) \cap S(c_k)]$, $t'_k[S(c_j) \cap S(c_k)] = t_j[S(c_j) \cap S(c_k)]$, $t'_k[S(c_i) \cap S(c_k)] \neq t_i[S(c_i) \cap S(c_k)]$ et $t_j[S(c_j) \cap S(c_k)] \neq t_k[S(c_j) \cap S(c_k)]$. Comme P est triangulaire par rapport à l'ordre $\prec$, nous devons avoir $t'_k[S(c_i) \cap S(c_k)] = t_i[S(c_i) \cap S(c_k)]$ et $t_j[S(c_j) \cap S(c_k)] = t_k[S(c_j) \cap S(c_k)]$, ce qui n'est pas possible puisque P ne vérifie pas DBTP. □

Théorème 19 *$TR \subsetneq DBTP$.*

Preuve : Le théorème 18 montre que $TR \subseteq DBTP$. L'instance présentée dans la figure 8(a) vérifie DBTP mais n'est pas triangulaire. □

En ce qui concerne les classes basées sur des restrictions de structures, nous avons prouvé dans les théorèmes 10 et 11 que *β-ACYCLIC $\subsetneq$ DBTP*, *α-ACYCLIC $\cap$ DBTP $\neq \emptyset$* et *α-ACYCLIC $\perp$ DBTP*.

Une autre classe polynomiale importante basée sur une restriction de structure porte sur la largeur d'arbre. Avant de l'évoquer, nous rappelons d'abord la notion de décomposition arborescente de graphe [16].

Définition 14 (décomposition arborescente) *Une **décomposition arborescente** d'un graphe $G = (X, E)$ est une paire (N, T) où $T = (I, F)$ est un arbre avec des nœuds I et des arêtes F et $N = \{N_i : i \in I\}$ est une famille de sous-ensembles de X, telle que chaque sous-ensemble N_i est un nœud de T et vérifie (i) $\cup_{i \in I} N_i = X$, (ii) pour chaque arête $\{x, y\} \in E$, il existe $i \in I$ avec $\{x, y\} \subseteq N_i$, et (iii) pour tout $i, j, k \in I$, si k est un chemin de i à j dans T, alors $N_i \cap N_j \subseteq N_k$.*

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

Cette définition est souvent étendue aux hypergraphes en se référant au *graphe primal* d'un hypergraphe (X, E), qui est le graphe (X, E') où $E' = \{\{x, y\} | \exists e \in E \text{ tel que } x, y \in e\}$.

Définition 15 (largeur arborescente bornée) *Une instance de CSP possède une* ***largeur arborescente bornée*** *si sa largeur arborescente est bornée par une constante. Nous notons BTW l'ensemble de ces instances, et qui constitue une classe polynomiale.*

Théorème 20 $BTW \cap DBTP \neq \emptyset$ *et* $BTW \perp DBTP$.

Preuve : Tout CSP binaire acyclique CSP possède une largeur arborescente bornée par 1 et vérifie DBTP. Donc, l'intersection n'est pas vide. Maintenant, considérons une instance ayant n variables avec $n \gg 3$, dont la largeur arborescente est bornée par une constante $k \geq 3$ et qui contient le sous-problème présenté dans la figure 7. Cette instance possède une largeur arborescente bornée mais ne vérifie pas DBTP. Inversement, une instance ayant n variables et une contrainte d'arité n est DBTP mais n'a pas une largeur arborescente bornée. □

5 (D)BTP et l'(Hyper-)k-Cohérence

Dans cette partie, nous étudions les liens existant entre (D)BTP et l'(Hyper-)k-cohérence orientée. Cette étude nous semble naturelle dans la mesure où, comme évoqué dans [5], BTP est une propriété moins forte que l'hyper-3-cohérence [14].

Comme l'hyper-3-cohérence qui implique BTP, est trop forte, nous définissons une forme relâchée en prenant en compte un ordre sur les contraintes :

Définition 16 (Hyper-k-Cohérence Orientée) *Étant donnés un CSP* $P = (X, D, C)$*, un ordre sur les contraintes* $\prec$ *et un entier* k *tel que* $1 \leq k \leq e$*,* P *vérifie l'****Hyper-k-cohérence orientée*** *si pour tout sous-ensemble de* k *contraintes tel que* $c_1 \prec c_2 \prec \ldots \prec c_{k-1} \prec c_k$*, on a* $\bowtie_{i=1}^{k-1} R(c_i)[(\bigcup_{i=1}^{k-1} S(c_i)) \cap S(c_k)] \subseteq R(c_k)[(\bigcup_{i=1}^{k-1} S(c_i)) \cap S(c_k)]$

Théorème 21 *Si un CSP* P *vérifie DBTP par rapport à un ordre* $\prec$ *et qu'il est Hyper-k-cohérent orienté par rapport à* $\prec$ *pour* $2 \leq k < e$*, alors* P *est Hyper-*$(k+1)$*-cohérent orienté par rapport à* $\prec$*.*

Preuve : Supposons que P ne soit pas Hyper-$(k+1)$-cohérent. Alors, il existe un sous-ensemble de $k+1$ contraintes tel que $c_1 \prec \ldots \prec c_k \prec c_{k+1}$, $\exists (t_1, \ldots, t_k) \in R(c_1) \times \ldots \times R(c_k), \forall t_{k+1} \in R(c_{k+1})$, $\bowtie_{i=1}^{k} t_i[(\bigcup_{i=1}^{k} S(c_i)) \cap S(c_{k+1})] \neq t_{k+1}[(\bigcup_{i=1}^{k} S(c_i)) \cap S(c_{k+1})]$. Considérons les k sous-ensembles de k contraintes $\{c_1, \ldots c_{j-1}, c_{j+1}, \ldots c_k, c_{k+1}\}$, pour $1 \leq j \leq k$. Comme P est Hyper-k-cohérent orienté, pour $1 \leq j \leq k$, il existe un tuple t_{k+1}^{j} of $R(c_{k+1})$ tel que $\bowtie_{i=1, i\neq j}^{k} t_i[(\bigcup_{i=1, i\neq j}^{k} S(c_i)) \cap S(c_{k+1})] = t_{k+1}^{j}[(\bigcup_{i=1, i\neq j}^{k} S(c_i)) \cap S(c_{k+1})]$. Considérons $1 \leq j < j' \leq k$. Nous avons deux cas :

(1) $t_{k+1}^{j} = t_{k+1}^{j'}$. Alors $t_{j'}[S(c_{j'}) \cap S(c_{k+1})] = t_{k+1}^{j}[S(c_{j'}) \cap S(c_{k+1})]$ et $t_j[S(c_j) \cap S(c_{k+1})] = t_{k+1}^{j'}[S(c_j) \cap S(c_{k+1})]$. Donc $\bowtie_{i=1}^{k} t_i[(\bigcup_{i=1}^{k} S(c_i)) \cap S(c_{k+1})] = t_{k+1}^{j}[(\bigcup_{i=1}^{k} S(c_i)) \cap S(c_{k+1})]$ et nous avons une contradiction.

(2) $t_{k+1}^{j} \neq t_{k+1}^{j'}$. Nous avons $t_j[S(c_j) \cap S(c_{j'})] = t_{j'}[S(c_j) \cap S(c_{j'})]$, $t_{j'}[S(c_{j'}) \cap S(c_{k+1})] = t_{k+1}^{j}[S(c_{j'}) \cap S(c_{k+1})]$, $t_j[S(c_j) \cap S(c_{k+1})] = t_{k+1}^{j'}[S(c_j) \cap S(c_{k+1})]$, $t_{j'}[S(c_{j'}) \cap S(c_{k+1})] \neq t_{k+1}^{j'}[S(c_{j'}) \cap S(c_{k+1})]$ et $t_j[S(c_j) \cap S(c_{k+1})] \neq t_{k+1}^{j}[S(c_j) \cap S(c_{k+1})]$. Par conséquent P ne vérifie pas DBTP par rapport à $\prec$ et nous avons encore une contradiction.

Donc, P est Hyper-$(k+1)$-cohérent orienté par rapport à l'ordre $\prec$. □

Comme l'intercohérence correspond à l'hyper-2-cohérence, nous pouvons en déduire ce corollaire :

Corollaire 1 *Si un CSP* P *vérifie DBTP par rapport à un ordre* $\prec$ *sur les contraintes et qu'il est intercohérent orienté par rapport à* $\prec$*, alors* P *est Hyper-*$(k+1)$*-cohérent orienté par rapport à* $\prec$ *pour* $1 \leq k < e$*.*

Donc, un CSP qui est à la fois DBTP et intercohérent orienté par rapport à un ordre donné sur les contraintes est cohérent. Il s'ensuit également qu'un tel CSP vérifie l'hyper-3-cohérence orientée.

De plus, comme l'Hyper-$(k+1)$-cohérence correspond à la k-cohérence sur le problème dual, nous pouvons également en déduire le théorème et le corollaire suivants en appliquant un raisonnement similaire.

Théorème 22 *Si un CSP binaire* P *vérifie BTP par rapport à un ordre* $<$ *sur les variables et qu'il est k-cohérent orienté par rapport à* $<$ *pour* $2 \leq k < n$*, alors* P *est* $(k+1)$*-cohérent orienté par rapport à* $<$*.*

Corollaire 2 *Si un CSP binaire* P *vérifie BTP par rapport à un ordre* $<$ *sur les variables et qu'il est k-cohérent orienté par rapport à* $<$*, alors* P *est* $(k+1)$*-cohérent orienté par rapport à* $<$ *pour* $1 \leq k < n$*.*

Comme conséquence, un CSP binaire qui est à la fois BTP et DAC par rapport à un ordre donné sur les variables est cohérent.

En ce qui concerne le positionnement de l'Hyper-3-cohérence orientée par rapport à BTP, nous pouvons prouver que l'hyper-3-cohérence n'implique pas nécessairement BTP. Par exemple, nous pouvons considérer une instance binaire avec 6 variables $\{x_a, x_b, \ldots, x_f\}$. Nous définissons cette instance en reproduisant plusieurs fois un même motif tel que chaque valeur

apparaissant dans une occurrence du motif n'apparaisse dans aucune autre de ses occurrences. Ce motif consiste en un triangle cassé sur une variable z pour un triplet (x, y, z) (i.e. ce qui impose la condition $z < max(x, y)$ sur $<$) et chaque valeur des variables x, y et z est liée à une valeur donnée d'une variable qui n'est pas impliqué dans ce triplet. Nous reproduisons ce motif six fois de telles sorte que les conditions suivantes soient vérifiées : $x_a < max(x_b, x_c)$, $x_b < max(x_d, x_e)$, $x_c < max(x_e, x_f)$, $x_d < max(x_a, x_b)$, $x_e < max(x_a, x_b)$ et $x_f < max(x_a, x_b)$. La figure 8(b) présente ce motif pour les triplets (x_a, x_b, x_e), un triangle cassé sur x_e (correspondant à la condition $x_e < max(x_a, x_b)$) et sur les variables indépendantes x_c, x_d et x_f. En faisant cela, la microstructure de notre CSP binaire possède six composantes connexes. On peut noter que cette instance n'est pas BTP parce que les six conditions rendent impossible la construction d'un ordre approprié sur les variables. Néanmoins, il est hyper-3-cohérent orienté ainsi qu'arc-cohérent.

6 Conclusion et perspectives

Dans cet article, nous avons étudié une classe polynomiale hybride dont les instances peuvent être résolues en temps polynomial par des algorithmes de type MAC. Nous avons prouvé qu'elle est incomparable avec plusieurs classes polynomiales connues (notamment BTP) et qu'elle inclut des classes polynomiales à la fois structurelles et relationnelles (à savoir les CSP β-acycliques et les CSP Triangulaires). Enfin, nous avons mis en évidence les liens existant entre (D)BTP et la k-cohérence ainsi que l'hyper-k-cohérence orientée.

Une première extension devrait consister en l'étude des liens entre DBTP et d'autres classes polynomiales non prises en compte dans cette contribution. Puis, dans le même esprit, nous pourrions explorer la possibilité de définir de nouvelles classes polynomiales basées sur d'autres codages de CSP non binaires.

Références

[1] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30 :479–513, 1983.

[2] C. Bessière, K. Stergiou, and T. Walsh. Domain filtering consistencies for non-binary constraints. *Artificial Intelligence*, 172 :800–822, 2008.

[3] D. Cohen, M. Cooper, M. Green, and D. Marx. On guaranteeing polynomially bounded search tree size. In *Proc. of CP*, pages 160–171, 2011.

[4] M. Cooper, D. Cohen, and P. Jeavons. Characterising Tractable Constraints. *Artificial Intelligence*, 65(2) :347–361, 1994.

[5] M. Cooper, Peter Jeavons, and Andras Salamon. Generalizing constraint satisfaction on trees : hybrid tractability and variable elimination. *Artificial Intelligence*, 174 :570–584, 2010.

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

[7] D. Duris. Some characterizations of γ and β-acyclicity of hypergraphs. *Information Processing Letters*, 112 (16) :617–620, 2012.

[8] E. Freuder. A Sufficient Condition for Backtrack-Free Search. *J. ACM*, 29 (1) :24–32, 1982.

[9] M. H Graham. On the universal relation. Technical report, University of Toronto, 1979.

[10] P. Janssen, P. Jégou, B. Nouguier, and M. C. Vilarem. A filtering process for general constraint satisfaction problems : achieving pairwise-consistency using an associated binary representation. In *Proc. of IEEE Workshop on Tools for Artificial Intelligence*, pages 420–427, 1989.

[11] P. Jeavons and M. Cooper. Tractable constraints on ordered domains. *Artificial Intelligence*, 79(2) :327–339, 1995.

[12] 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, 1991.

[13] P. Jégou. Decomposition of Domains Based on the Micro-Structure of Finite Constraint Satisfaction Problems. In *Proc. of AAAI*, pages 731–736, 1993.

[14] P. Jégou. On the Consistency of General Constraint-Satisfaction Problems. In *Proc. of AAAI*, pages 114–119, 1993.

[15] A. El Mouelhi, P. Jégou, C.Terrioux, and B. Zanuttini. Some New Tractable Classes of CSPs and their Relations with Backtracking Algorithms. In *Proc. of CP-AI-OR*, 2013.

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

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

[18] P. van Beek and R. Dechter. On the minimality and decomposability of row-convex constraint networks. *J. ACM*, 42(3) :543–561, 1995.

Cohérence d'arc virtuelle dynamique

Hiep Nguyen[1] **Christian Bessiere**[2] **Thomas Schiex**[1]

[1] Unité de Biométrie et Intelligence Artificielle, INRA, Toulouse, France
[2] CNRS, Université de Montpellier, France
{hnguyen,tschiex}@toulouse.inra.fr bessiere@lirmm.fr

Résumé

La cohérence d'arc virtuelle (VAC) est une cohérence récente pour les réseaux de fonctions de coût (ou réseaux de contraintes pondérées). VAC exploite une relation simple mais puissante avec les réseaux de contraintes classiques. VAC a permis de clore des problèmes d'affectation de fréquences difficiles, et est capable de résoudre directement les réseaux de fonctions de coût sous-modulaires. L'algorithme pour établir VAC est un algorithme itératif qui résout une séquence de réseaux de contraintes classiques. Dans cet article, nous montrons que les techniques utilisées dans les algorithmes de cohérence d'arc dynamique peuvent être injectées dans l'algorithme itératif de cohérence d'arc virtuelle. Cette intégration donne une accélération importante.

Abstract

Virtual Arc Consistency (VAC) is a recent local consistency for processing Cost Function Networks (or Weighted Constraint Networks) that exploits a simple but powerful connection with classical Constraint Networks. It has allowed to close hard frequency assignment benchmarks and is capable of directly solving networks of submodular functions. The algorithm enforcing VAC is an iterative algorithm that solves a sequence of classical Constraint Networks. In this work, we show that Dynamic Arc Consistency algorithms can be suitably injected in the virtual arc consistency iterative algorithm, providing noticeable speedups.

1 Introduction

L'analyse des modèles graphiques, et en particulier des réseaux de contraintes, est un problème important en intelligence artificielle. L'optimisation du coût combiné de fonctions de coût locales, problème central dans le cadre des CSP valués [11], permet de capturer des problèmes variés, tels que MaxSAT pondéré, CSP pondéré ou le problème de recherche d'une explication de probabilité maximum dans les modèles graphiques stochastiques (réseaux bayésiens, champs aléatoires de Markov). Elle a des applications en *allocation de ressources*, *enchères combinatoires*, *bioinformatique*, etc.

Les approches de programmation dynamique, basées sur l'élimination de variables ou sur les arbres de jonction, ont été largement utilisées pour résoudre de tels problèmes. Cependant, elles sont intrinsèquement limitées par leur complexité exponentielle en espace et en temps dans la largeur d'arbre (*treewidth*) des problèmes traités. Au contraire, la recherche arborescente en profondeur d'abord par "Branch and Bound" (séparation et évaluation) permet de conserver une complexité en espace raisonnable. Elle nécessite cependant de bons minorants (forts et peu coûteux) pour être efficace.

Ces dernières années, des minorants de qualité croissante ont été définis en appliquant des cohérences locales dans des Réseaux de Fonctions de Coût (ou CFN pour "Cost Function Networks"). Ces cohérences sont établies en appliquant itérativement des opérations appelées *Transformations Préservant l'Equivalence* (EPT, *Equivalence Preserving Transformations*, [7]) qui étendent les opérations de cohérence locale usuelles utilisées dans les CSP classiques. Les EPT déplacent des coûts entre des fonctions d'arité différente tout en préservant l'équivalence du problème. En déplaçant finalement des coûts vers une fonction d'arité nulle, les EPT sont capables de fournir un minorant du coût optimum qui peut être maintenu de manière incrémentale pendant la recherche arborescente.

Les cohérences locales traditionnelles des CFN telles qu'AC*, DAC*, FDAC* ou EDAC* [10] appliquent les EPT dans un ordre arbitraire. Par contre, la Cohérence d'Arc Virtuelle (VAC pour *Virtual Arc Consistency* [5,6]) planifie une séquence d'EPT à appliquer. Elle le fait en se basant sur le résultat de l'établissement de la cohérence d'arc classique dans un réseau

de contraintes classique qui interdit toutes les combinaisons de valeurs de coût non nul. VAC est non seulement plus fort que ces cohérences locales mais est capable de résoudre les réseaux formés de fonctions de coût sous-modulaires. Une itération de VAC peut être appliquée via un algorithme polynomial d'ordre faible et VAC a permis de clore des problèmes d'affectation de fréquences difficiles [5]. Cependant, il est souvent trop couteux pour être utilisé de façon générale.

Dans cet article, l'efficacité de VAC est améliorée en exploitant son comportement itératif. Chaque itération de VAC demande d'établir la cohérence d'arc classique dans la version durcie du réseau. Néanmoins, chacun de ces réseaux est juste le résultat des modifications incrémentales effectuées par les EPT appliquées dans l'itération précédente. Cette situation, où la cohérence d'arc est établie itérativement dans les versions modifiées de manière incrémentale d'un réseau de contraintes, a été déjà considérée dans les algorithmes de Cohérence d'Arc Dynamique [1, 3] pour les CSP dynamiques [8].

En pratique, nous observons que l'utilisation de la cohérence d'arc dynamique dans VAC accélère la recherche pour de nombreux problèmes. Ceci est peut-être l'une des premières applications réelles des algorithmes de Cohérence d'Arc Dynamique.

2 Préliminaires

2.1 Réseaux de Fonctions de Coût

Un réseau de fonctions de coût, ou CSP pondéré, est quadruplet (X, D, W, m) où X est un ensemble de n variables. Chaque variable $i \in X$ a un domaine $D_i \in D$ d'au plus d valeurs. Étant donné un sous-ensemble de variables $S \subseteq X$, on note $\ell(S)$ l'ensemble des n-uplets définis sur S. W est un ensemble de e fonctions de coût. Chaque fonction de coût $w_S \in W$ est définie sur un ensemble de variables S, appelé sa portée, et supposé différent pour chaque fonction de coût. Une fonction de coût w_S assigne un coût à chaque affectation de variables dans S, c'est à dire $w_S : \ell(S) \rightarrow [0..m]$ où $m \in \{1, ..., +\infty\}$. Le coût m représente un coût intolérable, associé à une valeur ou un n-uplet interdit. Les coûts sont combinés par l'addition bornée $\oplus$, définie par $a \oplus b = \min(a + b, m)$. Un coût b peut être soustrait d'un coût plus grand a en utilisant la soustraction $\ominus$ définie par : $a \ominus b = a - b$ si $a < m$ et m sinon. Le coût d'un n-uplet complet t est la somme des coûts $Val_P(t) = \bigoplus_{w_S \in W} w_S(t[S])$ où $t[S]$ est la projection de t sur S. Dans cet article, nous considérons des réseaux binaires, formés de fonctions d'arité au plus 2. Nous dénotons par w_i et w_{ij} les fonctions de coût unaires et binaires définies sur la variable i et sur les variables i, j respectivement. Nous faisons l'hypothèse qu'une fonction de coût unaire, notée w_i, existe pour toute variable i ainsi qu'une fonction de coût d'arité nulle notée $w_\varnothing$. Ce coût positif constant définit un minorant du coût de toutes les solutions.

L'établissement d'une cohérence locale donnée dans un CFN P consiste à transformer le CFN P en un problème P' équivalent à P : $(Val_P(t) = Val_{P'}(t)\ \forall t)$ menant à un éventuel accroissement du minorant de coût optimum défini par $w_\varnothing$. Il s'appuie sur l'application des transformations préservant l'équivalence (EPT) qui déplacent les coûts entre les fonctions de portée différente. L'algorithme 1 introduit trois EPT élémentaires. Project(w_{ij}, i, a, α) déplace une quantité de coût α d'une fonction binaire w_{ij} vers une fonction unaire w_i, sur une valeur $a \in D_i$. Un appel à Extend(i, a, w_{ij}, α) effectue le travail inverse. Enfin, UnaryProject(i, α) projette une quantité de coût α d'une fonction unaire vers la fonction de coût d'arité nulle $w_\varnothing$.

Algorithme 1 : Les EPTs élémentaires

```
1 Procédure Project(w_ij, i, a, α)
2     w_i(a) ⟵ w_i(a) ⊕ α ;
3     foreach b ∈ D_j do w_ij(a, b) ⟵ w_ij(a, b) ⊖ α;
4 Procédure Extend(i, a, w_ij, α)
5     foreach b ∈ D_j do w_ij(a, b) ⟵ w_ij(a, b) ⊕ α;
6     w_i(a) ⟵ w_i(a) ⊖ α ;
7 Procédure UnaryProject(i, α)
8     foreach a ∈ D_i do w_i(a) ⟵ w_i(a) ⊖ α;
9     w_∅ ⟵ w_∅ ⊕ α ;
```

Notez qu'un CSP binaire classique peut être représenté comme un CFN avec $m = 1$ (le coût 1 est assigné aux n-uplets interdits). Comme d'habitude, une valeur (i, a) est dite arc cohérente (AC) sur w_{ij} ssi il y a une paire (a, b) (support) qui satisfait w_{ij} et que $b \in D_j$ (valide). Un CSP est AC si toutes ses valeurs sont AC sur toutes les contraintes. L'établissement de AC dans un CSP P produit sa fermeture AC, un CSP qui est équivalent à P et qui est AC.

2.2 Cohérence d'Arc Virtuelle

Définition 1. *Étant donné un CFN* $P = (X, D, W, m)$*, le problème* $Bool(P) = (X, D, \overline{W}, 1)$ *est un CSP qui satisfait :* $\exists \overline{w}_S \in \overline{W}$ *ssi* $\exists w_S \in W$, $S \neq \varnothing$ *et* $\overline{w}_S(t) = 1 \Leftrightarrow w_S(t) \neq 0$*. Un CFN* P *est arc cohérent virtuel (VAC) ssi la fermeture arc cohérente classique du CSP* $Bool(P)$ *n'est pas vide [6].*

Bool(P) est un CSP dont les solutions sont exactement les n-uplets complets ayant un coût de $w_\varnothing$

dans P. Si P n'est pas VAC, l'établissement de AC dans Bool(P) va générer un domaine vide. Dans ce cas, comme indiqué dans [6], il existe une séquence d'EPT dont l'application mène à une augmentation de $w_\varnothing$. L'algorithme d'établissement de VAC utilise cette propriété dans un processus itératif en trois phases.

La phase 1 consiste à établir AC dans le CSP Bool(P). Dans cette phase, les effacements de valeurs sont mémorisés dans une structure de données notée tueur. Lorsqu'une valeur (i, a) n'a pas de support valide sur $\overline{w}_{ij}$, on affecte tueur $(i, a) = j$ et on efface (i, a). Si aucun domaine de variable n'est vidé, P est VAC et le processus d'établissement de VAC s'arrête.

Sinon, la phase 2 identifie le sous-ensemble des valeurs effacées qui sont nécessaires pour produire le domaine vide et les stocke dans une pile R. En retraçant l'histoire des propagations de la phase précédente définie par tueur, à partir de la variable dont le domaine a été vidé jusqu'aux coûts non nuls, il est possible de déterminer les effacements indispensables. Par ailleurs, cette phase évalue la quantité de coût maximale qu'il est possible de déplacer sur $w_\varnothing$ (notée λ) et définit l'ensemble d'EPT à appliquer dans P afin d'atteindre cette augmentation. Comme présenté dans [6], toutes les quantités de coût déplacées par EPT sont stockées dans deux tableaux de nombres entiers $k(j, b)$ et $k_{ij}(j, b)$. Ils représentent respectivement le nombre de demandes de coût d'un montant λ à projeter sur (j, b) et à étendre de (j, b) sur w_{ij}. Notez que le déplacement de ces coûts respecte une règle de conservation simple. Pour chaque valeur (j, b) d'une variable non vide j qui n'est pas la source de coût ($w_j(b) = 0$), la quantité de coût que (j, b) reçoit par Project est exactement la quantité de coût sortant de (j, b) par Extend (voir [6], page 465).

$$\forall (j,b) \ s.t. \ w_j(b) = 0, k(j,b) = \sum_{w_{ij} \in W} k_{ij}(j,b) \quad (1)$$

La phase 3 de VAC, comme décrite en détail dans l'algorithme 2, modifie le CFN original en appliquant les EPT définies par les structures de données k et k_{ij} sur toutes les valeurs effacées indispensables stockées dans R. Une valeur (j, b) effacée par $\overline{w}_{ij}$ va recevoir un coût de $k(j, b) \times \lambda$ par Project depuis w_{ij} (ligne 7). Néanmoins, il faut tout d'abord étendre un coût de $k_{ij}(i, a) \times \lambda$ à partir des supports invalides (i, a) sur w_{ij} (ligne 5). Cette phase produit finalement un nouveau problème P' équivalent à P avec un minorant $w_\varnothing$ augmenté (ligne 8).

Notez qu'à la fin de la phase 1, lorsqu'il y a une variable de domaine vide i_0, le problème obtenu n'est pas forcément la fermeture arc cohérente maximum de Bool(P). Il y a peut-être quelques valeurs qui n'ont pas été encore supprimées bien qu'elles n'aient aucun support valide, parce que le domaine concerné n'a pas encore été révisé. Une telle variable est encore dans la file de propagation Q_{AC} qui contient les variables restant à propager. Quant aux autres valeurs, soit elles ont un support valide, soit elles ont été supprimées et leur tueur associé n'est pas vide. Un tel problème est appelé une fermeture arc cohérente *partielle justifiée* de Bool(P). Aucun domaine dans ce problème ne peut être plus large que dans Bool(P), les valeurs effacées n'ont pas de support valide et sont justifiés correctement par les tueur.

Algorithme 2 : Phase 3 de VAC : Application des EPTs

```
1 tant que R ≠ ∅ faire
2     (j, b) ⟵ R.pop();
3     i ⟵ tueur [j, b];
4     pour chaque a ∈ D_i s.t. k_ij(i, a) ≠ 0 faire
5         Extend(i, a, w_ij, λ × k_ij(i, a));
6         k_ij(i, a) ⟵ 0;
7     Project(w_ij, j, b, λ × k(j, b));
8 UnaryProject(i_0, λ);
```

L'algorithme de VAC a une complexité spatiale en $O(ed)$ parce que les structures de données tueur, k et k_{ij} nécessitent une mémoire de $O(nd)$, $O(nd)$ et $O(ed)$ respectivement. La complexité en temps de chaque itération de VAC est en $O(ed^2)$ tant qu'un algorithme de AC optimal est utilisé dans la phase 1.

Les itérations de VAC établissent AC sur une séquence de CSP légèrement modifiés : Bool(P), Bool(P'), ... Cela motive l'idée d'utiliser des algorithmes d'AC dynamique pour appliquer AC à chaque itération.

2.3 Cohérence d'Arc Dynamique

Définition 2 ([8]). *Un CSP dynamique est une séquence $P_0, P_1, ..., P_n$ de CSP où chaque P_i est un CSP résultant de l'addition ou du retrait d'une contrainte dans P_{i-1}.*

Les algorithmes de Cohérence d'Arc Dynamique (DnAC) consistent à maintenir la cohérence d'arc dans la séquence des problèmes P_i. L'établissement d'AC est naturellement incrémentale pour la restriction (addition d'une contrainte) : il est suffisant d'établir AC (phase 1) avec une file de propagation Q_{AC} contenant les variables de la contrainte qui vient d'être ajoutée. Cependant, AC n'est pas incrémental pour la relaxation (retrait d'une contrainte). On a donc besoin d'un processus supplémentaire pour régler ce cas : il faut

restaurer les valeurs supprimées directement ou indirectement à cause de la contrainte retirée dès lors qu'il n'y pas d'autre raison de les supprimer.

Différentes familles d'algorithmes ont été proposées pour DnAC. Dans cet article, nous nous appuyons sur AC/DC2 [1] qui est une amélioration de AC/DC [2] s'appuyant sur des structures de données persistantes très limitées. La plus importante d'entre-elles est un tableau *justification*(i, a) (comme dans DnAC4 [3]) qui mémorise la source de l'effacement de (i, a). Cette structure correspond exactement à la structure tueur de VAC [1].

Lors du retrait d'une contrainte donnée w_{ij}, AC/DC2 se déroule en 3 étapes : 1) initialisation : toutes les valeurs des variables i et j supprimées par w_{ij} sont candidates pour être restaurées et marquées en "Propageable". On peut déterminer exactement ces valeurs en se basant sur le tableau *justification*. 2) propagation : chaque valeur propageable est propagée vers ses variables voisines en vérifiant si elle offre un nouveau support aux valeurs dont elle était le tueur (valeurs qui ont été effacées par le manque de support sur la variable propageable) et qui seront alors aussi marquées comme "Propageable". Notez que grâce à *justification*, quand on propage la variable "Propageable" i, on ne considère que les valeurs (j, b) effacées par w_{ji} comme candidates à la restauration, au lieu de tester toutes les valeurs du voisinage. Lorsqu'une valeur est propagée sur toutes les variables de son voisinage, elle est marquée "Restaurable" et sera restaurée. 3) filtrage : toutes les valeurs restaurées ont besoin d'être revérifiées pour la cohérence d'arc. Ceci est accompli en établissant l'AC via la file de propagation Q_{AC}, initialisée de façon à imposer la révision des variables ayant des valeurs restaurées.

La complexité spatiale d'un algorithme AC/DC basé sur AC3 est en $O(nd + e)$. La complexité en temps de AC/DC2 est définie par l'algorithme de filtrage utilisé dans la dernière étape : en $O(ed^3)$ pour AC-3 et en $O(ed^2)$ pour AC2001 [1].

3 Algorithme de VAC Dynamique

En établissant VAC, le CFN P est incrémentalement modifié dans la phase 3 de chaque itération de VAC. Nous proposons donc une version améliorée de VAC, appelée VAC dynamique (DynVAC), qui utilise AC dynamique pour maintenir AC dans les Bool(P) successifs au lieu de filtrer à partir de zéro à chaque itération comme le fait l'algorithme VAC standard. Nous utilisons AC/DC2 [1] basé sur AC2001, pour son optimalité. Un avantage de l'utilisation d'AC/DC2 est que la structure de données *justification* est fournie gratuitement par la structure tueur de VAC.

3.1 Propriétés et algorithme

Dans les CSP dynamiques traditionnels, les algorithmes de DnAC sont appliqués après chaque addition ou suppression d'une contrainte. Dans le cas de VAC, la situation est plus compliquée parce qu'une série de modifications de Bool(P) a lieu durant la phase 3 via l'application des EPT. Un appel de Project(w_{ij}, i, a, α) a deux effets : 1) l'augmentation du coût unaire $w_i(a)$ et 2) la diminution de coûts dans la fonction de coût binaire w_{ij}. Si un coût $w_i(a)$ nul augmente et devient strictement positif, la valeur (i, a) est enlevée de Bool(P), et ceci correspond à une restriction. Au contraire, si la paire (a, b) de coût strictement positif voit son coût atteindre zéro, cette paire précédemment interdite dans $\overline{w}_{ij}$ devient autorisée et ce fait correspond à une relaxation. Au lieu d'appliquer un algorithme de DnAC à chaque opération Project, Extend et UnaryProject, une approche plus rationnelle consiste à appliquer les principes de DnAC une seule fois après la phase 3 pour éviter les restaurations/suppressions inutiles.

Chaque itération de VAC transforme le CFN P actuel en un problème modifié P' dont les fonctions de coûts sont notées w'_i et w'_{ij}. Il est possible de calculer les valeurs w'_i et w'_{ij} dès la fin de la phase 2 puisqu'elles sont définies par une séquence connue d'opérations Project, Extend et UnaryProject appliquées sur w_i et w_{ij}. Par exemple, toutes les valeurs a d'une variable non vide i satisfont l'équation suivante :

$$w'_i(a) = w_i(a) \oplus (k(i,a).\lambda) \bigominus_{w_{ij} \in W} (k_{ij}(i,a).\lambda)$$

De manière similaire, il est possible de calculer les coûts de la variable de domaine vide et des w'_{ij}. Nous allons maintenant prouver que l'effet global des toutes les EPT sur Bool(P) dans la phase 3 est simplement un ensemble de relaxations au niveau des fonctions unaires et binaires.

Propriété 1. *Dès la phase 2, nous savons que :*

(a) $\forall (i, a)$ *:* $w'_i(a) \leq w_i(a)$.

(b) $\forall (i, a)$ *et* (j, b) *: si* $w'_{ij}(a, b) \neq w_{ij}(a, b)$ *alors* (i, a) *ou* (j, b) *est effacée de la fermeture arc cohérente partielle justifiée actuelle de Bool(*P*).*

Démonstration. (a) Dans VAC, la seule opération qui peut augmenter les coûts unaires est l'opération Project Ċependant, selon l'équation 1, si une valeur (i, a) reçoit un coût via Project, elle va faire un Extend de

1. AC/DC2 introduit une autre structure *time-stamp*(i, a) qui mémorise l'ordre des effacements. Le premier auteur de AC/DC2 nous a confirmé que cette structure est subsumée par *justification*(i, a) (communication privée) et est donc inutile.

la même quantité de coût (vers d'autres fonctions de coût binaires ou vers $w_\varnothing$). En conséquence, les coûts unaires ne peuvent pas augmenter.

(b) Il n'y a que deux façons pour changer un coût binaire $w_{ij}(a,b)$. C'est d'exécuter une opération Project à partir de w_{ij} ou une opération Extend sur lui. Néanmoins, la phase 3 de VAC n'applique Project et Extend qu'avec des valeurs extraites de la file des valeurs supprimées R (construite dans la phase 2). Donc, quand le coût d'une paire (a,b) change, il faut que (i,a) ou (j,b) soit effacée. □

Corollaire 1. *Les EPT appliquées dans la phase 3 de VAC transformant Bool(P) en Bool(P') ne génèrent que les types de relaxation suivants :*

(1) *des valeurs (i,a) deviennent autorisées $(w_i(a) > w'_i(a) = 0)$.*

(2) *des paires $((i,a),(j,b))$ deviennent autorisées $(w_{ij}(a,b) > w'_{ij}(a,b) = 0)$.*

Démonstration. Selon la Propriété 1(a), nous savons que les coûts unaires ne peuvent que diminuer. Certains coûts non nuls peuvent diminuer à zéro. Donc, les valeurs correspondantes réapparaissent dans Bool(P'). Ce fait peut être considéré comme la rétraction de contraintes unaires.

Selon la Propriété 1(b), le coût des paires peut augmenter ou diminuer. Si un coût binaire (a,b) augmente de zéro à non nul, cela ne peut détruire aucun support valide parce qu'une des deux valeurs est effacée dans la fermeture partielle actuelle. Le support n'est donc pas valide. Au contraire, si le coût de (a,b) diminue, cela peut créer un nouveau support pour a ou b. □

En conséquence, l'algorithme DnAC utilisé peut être spécialisé pour un ensemble de relaxations (Algorithme 3). Le protocole de restauration se compose de 3 étapes, comme dans AC/DC2. Notez que Bool(P) est maintenu après la phase 3 de chaque itération. $\overline{D}_i$ mentionné dans l'algorithme 3 représente le domaine de la variable i dans la fermeture arc cohérente partielle justifiée finale obtenue après la phase 1.

L'étape **initialisation** parcourt toutes les valeurs dans la file R dans le même ordre de parcours que la phase 3 pour identifier les valeurs qui doivent être restaurées (ligne 2). La variable i_0, dont le domaine a été vidé, est traitée séparément. Selon le corollaire 1, il y a deux cas possibles : (1) quand une valeur (i,a) devient autorisée $w_i(a) > w'_i(a) = 0$, elle sera restaurée (ligne 5), (2) quand un nouveau support valide apparaît pour la valeur (j,b) en satisfaisant $(w_{ij}(a,b) \oplus w_i(a)) > (w'_{ij}(a,b) \oplus w'_i(a)) = 0$ et tueur $[j,b] = i$, (j,b) sera restaurée (ligne 7). Lorsqu'une valeur (i,a) est restaurée, elle est stockée dans un tableau *restauré*$[i]$ et la variable i est ajoutée dans la liste RL qui contient les variables dont la restauration doit être propagée.

Algorithme 3 : Mise à jour de Bool(P)

```
1  Procédure Initialisation
2      pour chaque (j,b) ∈ R faire
3          i ⟵ tueur [j,b];
4          pour chaque a ∈ D_i − D̄_i faire
5              si (w_i(a) > 0) ∧ (w'_i(a) = 0) alors
               Restaurer(i,a);
6              si b ∉ D̄_j ∧ w'_i(a) = 0 ∧ w'_ij(a,b) = 0
               alors Restaurer(j,b);
7      pour chaque a ∈ D_{i_0} tel que
       w_{i_0}(a) > 0 ∧ w'_{i_0}(a) = 0 faire Restaurer(i_0,a);
8  Procédure Restaurer(i,a)
9      ajouter a dans D̄_i et restauré[i];
10     ajouter i dans RL;
11     tueur [i,a] ← nil;
12 Procédure Propagation
13     tant que RL ≠ ∅ faire
14         i ← RL.pop();
15         pour chaque w_ij ∈ W faire
16             pour chaque b ∈ D_j − D̄_j s.t. tueur
               [j,b]= i faire
17                 si ∃a ∈ restauré[i] s.t. w'_ij(a,b) = 0
                   alors Restaurer(j,b);
18         restauré[i] ← ∅;
19         Q_AC ← Q_AC ∪ {j | w_ij ∈ W}
```

L'étape **propagation** propage les restaurations de valeurs vers les variables voisines, comme dans AC/DC2 (ligne 14,15). Chaque variable i peut restaurer une valeur (j,b) dans une variable voisine j si cette valeur a été supprimée à cause de w_{ij} (ligne 16) et qu'elle maintenant supportée par une valeur restaurée dans i (ligne 17). Après avoir propagé toutes les valeurs restaurées, la liste *restauré* est vidée (ligne 18) pour éviter de repropager les valeurs qui ont été déjà propagées.

L'étape **filtrage** doit éliminer les valeurs restaurées (i,a) qui ne sont pas arc cohérentes sur une contrainte $\overline{w}_{ij}$ et assigner j à tueur $[i,a]$. Cette tâche est exactement le travail de la phase 1 de VAC. En conséquence, nous avons intégré cette étape dans la phase 1 en ajoutant les variables voisines des variables ayant des valeurs restaurées dans la file de révision Q_{AC} (ligne 19).

Il est possible de prouver la correction de l'algorithme DynVAC en montrant que cet algorithme fournit exactement une fermeture arc cohérente partielle justifiée de Bool(P') pour l'itération suivante de VAC.

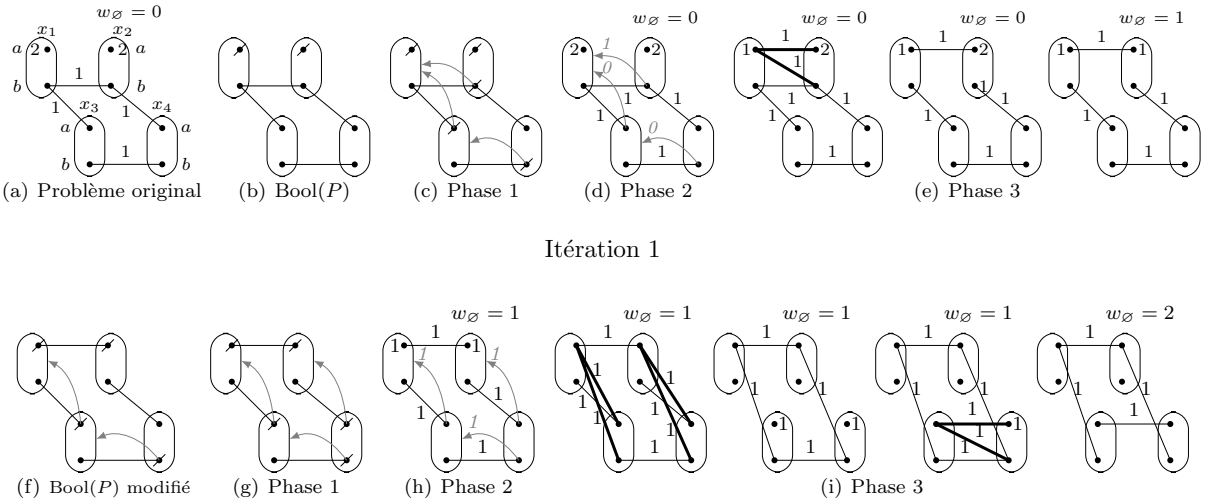


FIGURE 1 – Exemple du fonctionnement de VAC dynamique durant les itérations

Nous ne présentons pas la preuve dans cet article.

3.2 Exemple

L'avantage de DynVAC par rapport à VAC est que la liste des variables à réviser utilisée dans la phase 1 est maintenue pendant toutes les itérations, au lieu d'être réinitialisée à l'ensemble complet de toutes les variables X à chaque itération, comme fait l'algorithme VAC de [6]. Ce fait, illustré dans l'exemple suivant, permet d'éviter de répéter des filtrages inutiles.

Soit le CFN binaire de la Figure 1(a). Chaque variable a deux valeurs a et b, représentées par des sommets. Les coûts unaires non nuls sont affichés à côté des valeurs. Une arête entre deux sommets implique un coût binaire non nul de la paire correspondante. Les coûts nuls ne sont pas affichés. Dans Bool(P) (Figure 1(b)), les valeurs interdites sont barrées et les arêtes représentent les paires interdites.

Supposons que les contraintes soient révisées dans la phase 1 dans l'ordre (w_{13}, w_{34}, w_{12}, w_{24}). Après avoir révisé w_{13}, w_{34}, w_{12}, les valeurs $(3, a)$, $(4, b)$ et $(2, b)$ sont respectivement effacées de Bool(P). Dès que la variable 2 a un domaine vide, la phase 1 s'arrête (Figure 1(c)). Les flèches en gris représentent la structure de données tueur pointant sur la variable qui a causé la perte de support valide. Dans la phase 2 (Figure 1(d)), l'effacement de $(2, b)$ est suffisant pour produire un domaine vide. $w_\varnothing$ peut augmenter d'une quantité de coût d'au maximum $\lambda = 1$ en utilisant les coûts non nuls de $w_{12}(b, b)$ et $w_1(a)$. Les nombres en italique associés aux flèches indiquent la valeur correspondante de $k(i, a)$. En appliquant les EPT précédemment identifiées, la phase 3 (Figure 1(e)) transforme P en un problème équivalent P' avec $w'_\varnothing = 1$. Les coûts étendus sont affichés en gras.

L'établissement de VAC se poursuit car P' n'est pas encore VAC. Bool(P') n'est pas construit à partir de P' mais inféré à partir de Bool(P). Pour mettre à jour Bool(P), au lieu de considérer toutes les valeurs effacées, nous considérons uniquement les valeurs $(1, a), (2, a)$ et $(2, b)$ pour restauration car seule la contrainte w_{12} a été modifiée par des EPT dans la phase 3. Parmi les trois valeurs effacées considérées, seule $(2, b)$ est restaurée car elle a un coût final nul et un nouveau support (b, b) dans w_{12}. Cette restauration ne provoque aucune autre restauration. Les contraintes du problème Bool(P') sont inférées directement à partir des coûts non nuls de P' [2]. En fait, le résultat de la mise à jour, tel que décrit dans la Figure 1(f), est déjà une fermeture arc cohérente partielle justifiée de Bool(P'), avec deux valeurs effacées supplémentaires $(3, a)$ et $(4, b)$, et les tueur associés définis. La phase 1 de l'itération suivante peut démarrer à partir de ce problème. $(4, a)$ est effacée après la révision de w_{24} et le domaine de la variable 4 devient vide (Figure 1(g)). La phase 2 et la phase 3 fonctionnent de la même manière qu'à l'itération précédente. Le problème final (Figure 1(i)) avec $w''_\varnothing = 2$ est VAC.

2. En pratique, aucune contrainte n'est créée. On se contente de tester la positivité stricte du coût dans P'.

3.3 Maintien de VAC dynamique pendant la recherche

On peut maintenir VAC durant la recherche arborescente en maintenant le problème Bool(P), défini par la liste des valeurs effacées et leurs justifications, de manière incrémentale. Lors d'une affectation $i = a$, il est habituel de propager les fonctions binaires contenant i dans leur portée et de les déconnecter du réseau car la propagation en a extrait toute l'information. Dans Bool(P), la valeur (i, a) sélectionnée pouvait être effacée par absence de support valide. Après la suppression des fonctions binaires contenant i dans leur portée, une telle valeur peut redevenir viable dans Bool(P) et il faut donc envisager sa restauration. Lors d'une restriction de domaine ($i \neq a$, $i > a$ ou $i < a$), des valeurs de variables voisines peuvent perdre leur support, il faut donc refaire une partie de la propagation. Au lieu de reconstruire Bool(P) pour chaque nouveau nœud de l'arbre de recherche, le problème Bool(P) du nœud parent est mis à jour en ajoutant une variable affectée dans la liste RL ou les variables réduites dans la file de propagation Q_{AC}. En remontant dans l'arbre de recherche, la fermeture de Bool(P) est reconstruite via la seule restauration des `tueur` qui ont été sauvegardés par "trailing". En effet, dans une fermeture justifiée, `tueur` $[i, a]$ = null si et seulement si (i, a) n'a pas été effacée.

Comme dans [6], afin d'accélérer l'algorithme de VAC dynamique pendant le recherche, nous avons remplacé Bool(P) par une version relaxée mais de plus en plus stricte, notée Bool$(P)_\theta$, qui permet de collecter rapidement des contributions importantes au minorant $w_\varnothing$. Un n-uplet t est interdit dans Bool$(P)_\theta$ ssi son coût dans P est plus grand que θ. L'ensemble des coûts binaires c_{ij} non nuls du problème sont triés dans un nombre fixé k de groupes en temps linéaire (via l'utilisation d'un algorithme de type "*bucket sort*"). Les coûts minimum de chaque groupe définissent une séquence de seuils $(\theta_1, \theta_2, ...\theta_k)$. En partant de θ_1, les itérations de VAC fonctionnent avec un seuil fixé jusqu'à ce qu'il n'y ait plus de domaine vide. On contraint alors le problème en utilisant le seuil θ_{i+1}. Après le dernier θ_k, une stratégie géométrique, définie par $\theta_{i+1} = \theta_i/2$, est utilisée et s'arrête lorsque θ_i est plus petit qu'une valeur T donnée. Ce seuil T permet d'obtenir un algorithme qui se termine dans tous les cas. Dans VAC dynamique, le seuil θ est maintenu de manière incrémentale pendant la recherche avec l'objectif d'hériter du travail fait dans les nœuds parents. θ est également sauvegardé/restauré durant les backtracks par "*trailing*".

3.4 Complexité

DynVAC a la même complexité spatiale que VAC, en $O(ed)$. L'étape initialisation a une complexité en temps de $O(nd)$ parce qu'il y a au maximum $n \times d$ valeurs ayant un coût strictement positif (ligne 5, Algorithme 3). La complexité en temps dans le pire des cas de l'étape "propagation" est en $O(ed^2)$ parce que chaque paire de valeurs de chaque contrainte est testée au maximum une fois (ligne 17, 18, Algorithme 3). Donc, la mise à jour de Bool(P) a la même complexité en temps que la phase 3 de VAC : $O(ed^2)$. En conséquence, chaque itération de DynVAC a une complexité en temps en $O(ed^2)$ comme VAC, tant qu'un algorithme AC optimal est utilisé dans la phase 1.

Bien que DynVAC n'améliore pas la complexité asymptotique de VAC, les expérimentations présentées dans la prochaine section montrent qu'il peut fournir des accélérations importantes en pratique.

4 Expérimentations

Dans cette section, nous comparons l'efficacité de DynVAC à VAC que ce soit en pré-traitement ou maintenu pendant la recherche, sur un ensemble de problèmes extraits de la "Cost Function Library"[3]. Les expérimentations sont implémentées dans le solveur `toulbar2` et exécutées sur un processeur Intel(R) Core(TM)2@2.66GHz avec 4GB de RAM. Comme dans [5,6], pour chaque problème, nous établissons une version limitée de VAC qui s'arrête dès que l'augmentation de $w_\varnothing$ reste plus petite qu'une valeur ε donnée sur un nombre donné d'itérations successives. Nous avons fixé $\varepsilon = 0.05$ pour les expérimentations de pré-traitement (Table 1) tandis que dans les expérimentations en recherche (Table 2) $\varepsilon = \frac{1}{10000}$ à la racine de l'arbre de recherche et $\varepsilon = 0.1$ pendant la recherche.

Pour le simple pré-traitement des problèmes, comme espéré, DynVAC est plus rapide que VAC sur une large partie des instances extraites de l'entrepôt CFLib. La table 1 affiche les moyennes du temps total (en secondes), du minorant (lb) et du nombre d'itérations (iter) pour l'établissement de VAC en utilisant l'algorithme VAC statique usuel et le nouvel algorithme DynVAC. Chaque ligne correspond à une classe de #inst problèmes. Le meilleur temps ou minorant est indiqué à chaque fois en gras. Les expérimentations montrent que DynVAC est respectivement 1,6, 3 et 5 fois plus rapide que VAC pour les classes *celar, tagsnp, warehouse* tandis qu'il fournit des minorants de même qualité. Nous observons que le nombre d'itérations en moyenne de DynVAC peut augmenter par rapport à

3. https ://mulcyber.toulouse.inra.fr/scm/viewvc.php/trunk/ ?root=costfunctionlib

TABLE 1 – Les valeurs du minorant, le temps, et le nombre d'itérations de VAC nécessaires pour traiter des problèmes réels et fabriqués de la "*Cost Function Library*".

classe	#inst	VAC_ε			$DynVAC_\varepsilon$			$DynVAC_\varepsilon$ avec heuristique		
		lb	temps	iter	lb	temps	iter	lb	temps	iter
celar	32	6.180	3,14	382	**6.204**	**1,92**	418	5.892	1,12	319
prot_maxclique	10	1.016	**51,00**	1.022	1.016	364,00	1.022	1.016	56,95	1.022
tagsnp_r0.5	25	$1{,}43\times10^6$	364,31	8.798	$1{,}43\times10^6$	**116,57**	4.653	$1{,}43\times10^6$	81,46	5.810
tagsnp_r0.8	82	$1{,}11\times10^6$	4,64	155	$1{,}11\times10^6$	**1,53**	120	$1{,}11\times10^6$	2,54	150
dimacs_maxclique	65	266	**0,78**	284	266	3,65	284	266	0,96	284
planning	68	1.074	0,25	46	1.074	**0,19**	50	1.072	0,23	76
warehouse	57	$7{,}23\times10^6$	341,00	946	$\mathbf{7{,}24}\times10^6$	**66,00**	719	$7{,}25\times10^6$	114,17	790

celui de VAC. Mais le temps total en moyenne diminue comme espéré : il y a moins de travail à faire à chaque itération dans DynVAC parce que les effacements de valeurs sont hérités des itérations précédentes. Notez que le minorant final obtenu par DynVAC et VAC n'est pas obligatoirement identique. Après la première itération, VAC et DynVAC peuvent réviser les contraintes dans un ordre différent dans la phase 1 menant à des séquences d'EPT différentes. Le critère d'arrêt peut alors agir différemment selon les cas.

Néanmoins, DynVAC est plus lent que VAC (7 et 4 fois respectivement) pour toutes les catégories des problèmes de type "clique maximum" testées : *protein_maxclique* et *dimacs_maxclique*. Ces problèmes ont une structure spécifique avec des domaines booléens, des contraintes de différence binaires et des fonctions de coût (non dures) seulement au niveau unaire. Du fait de cette structure spécifique, chaque itération restaure de nombreuses valeurs en cascade mais de façon inutile, car elles vont être à nouveau effacées, pour une autre raison, dans l'itération suivante. Afin d'améliorer l'efficacité de DynVAC sur ces problèmes défavorables, nous avons utilisé une heuristique d'ordonnancement des révisions orientée variable, proposée dans [12]. Nous l'appliquons durant l'établissement de AC dans la phase 1. L'heuristique sélectionne premièrement dans la file Q_{AC} la variable dont la taille de domaine est la plus petite dans Bool(P). Ensuite, les contraintes sont traitées en ordre ascendant en terme de la taille des domaines des variables opposées. Le résultat obtenu, présenté dans la dernière colonne de la table 1, montre que cette heuristique permet d'améliorer considérablement la performance de DynVAC sur les problèmes de type "clique maximum". L'heuristique intégrée dans DynVAC fournit une performance comparable à celle de VAC statique dans ces cas extrêmement défavorables (20% de dépassement en temps qui est probablement causé par le calcul de l'heuristique).

Finalement, nous testons les performances de DynVAC et VAC lorsqu'elles sont maintenues pendant la recherche d'une solution de coût optimal, sur des problèmes d'affectation de fréquence (*celar*, [4]) et des problèmes d'affectation d'entrepôts (uncapacited warehouse location problems *warehouse*, [9]). Ces problèmes, qui ont été déjà utilisés pour l'évaluation de VAC dans [6], sont issus de problèmes difficiles, de taille parfois importante et possèdent de grands domaines, ce qui constitue sans doute un terrain de prédilection pour DynVAC. Sur ces problèmes, VAC est effectivement dominé par DynVAC au niveau du prétraitement. La table 2 affiche le temps (en secondes) et le nombre de nœuds visités durant la recherche, jusqu'à l'obtention et la preuve d'une solution optimale, en utilisant VAC statique ou DynVAC. La dernière colonne présente le ratio de temps CPU de DynVAC par rapport à celui de VAC. Comme dans le cas du pré-traitement, VAC dynamique continue à être plus efficace sur la plupart des instances de ces classes. Il est en moyenne 40% plus rapide que VAC et visite 18% de moins de nœuds sur des instances *celar* (correspondant à la section en haut de la table). Cependant, il y a quelques instances *celar* qui sont plus longues à résoudre avec DynVAC. Le résultat est encore plus visible avec la classe *warehouse* où DynVAC est de 2 à 5 fois plus rapide que VAC, sur toutes les instances de cette classe (sauf pour "capb") en visitant souvent un peu moins de nœuds.

5 Conclusion

Cet article présente une approche incrémentale pour l'établissement de VAC dans les réseaux de fonctions de coût. Notre approche combine l'idée des algorithmes d'arc consistance dynamique avec l'algorithme itératif VAC afin de maintenir efficacement la cohérence d'arc dans le CSP Bool(P) durant l'établissement de VAC.

Le nouvel algorithme fournit un minorant de même qualité qu'avec l'algorithme VAC statique mais il est plus efficace sur de nombreux problèmes, particuliè-

TABLE 2 – Résultats de DynVAC et VAC pendant la recherche sur un sous-ensemble des instances.

	VAC_ε		$DynVAC_\varepsilon$		ratio
	temps	nœuds	temps	nœuds	temps
cl6-1-24	12	9.686	18	13.962	1,5
cl6-1	26	14.871	38	17.131	1,46
cl6-2	29	11.228	46	13.081	1,59
cl6-3	161	54.083	133	35.614	0,83
$cl6\text{-}4_r$	160	47.646	138	30.399	0,86
cl6-4	210	52.724	270	47.806	1,29
cl7-1	194	17.951	84	8.733	0,43
cl7-2	1.076	33.415	773	33.545	0,72
cl7-3	4.983	142.755	2.979	85.626	0,6
cl7-4	9.193	380.566	7.426	207.207	0,81
$gr11_{rm}$	6	618	5	639	0,83
gr11	470	6.600	366	4.292	0,78
$gr13_{rm}$	34	50	20	11	0,59
gr13	1.431	5.921	1.144	3.022	0,8
$sc06\text{-}16_r$	884	142.740	625	170.851	0,71
sc06-16	1.873	283.872	1.052	290.993	0,56
$sc06\text{-}18_r$	677	143.386	360	118.060	0,53
sc06-18	1.511	276.610	736	213.282	0,49
$sc06\text{-}20_r$	595	125.674	358	124.856	0,6
sc06-20	765	130.260	508	164.515	0,66
$sc06\text{-}22_r$	374	96.270	177	79.211	0,47
sc06-22	433	91.202	235	74.481	0,54
$sc06\text{-}24_r$	156	45.558	81	36.407	0,52
sc06-24	165	40.045	89	37.170	0,54
$sc06\text{-}30_r$	5	1826	3	1.697	0,6
sc06-30	14	6.990	10	6.184	0,71
$sc06_r$	3.629	310.034	2.183	270.387	0,6
sc06	5.227	347.925	2.454	275.992	0,47
capa	2.462	1.100	1.013	1.101	0,41
capb	3.019	1.350	6.168	1.826	2,04
capc	2.027	1.650	1.228	1.233	0,61
capmo1	123	2.254	95	2.671	0,77
capmo2	64	407	21	328	0,33
capmo3	93	1.105	32	993	0,34
capmo4	74	846	18	574	0,24
capmo5	75	412	14	305	0,19
capmp1	1.573	3.030	920	3.554	0,58
capmp2	1.063	1.611	452	2.130	0,43
capmp3	1.123	1.454	443	1.474	0,39
capmp4	1.340	2.585	680	1.518	0,51
capmp5	984	1.932	311	1.541	0,32
capmq1	5.111	3.546	2.374	2.419	0,46
capmq2	6.520	4.408	3.209	4.633	0,49
capmq3	6.310	3.943	2.743	3.115	0,43
capmq4	7.530	7.208	4.295	6.124	0,57
capmq5	13.938	13.980	7.219	11.629	0,52

rement sur les problèmes avec coûts élevés et grands domaines. Cependant, DynVAC peut être ralenti pour certains problèmes spécifiques comme les problèmes de la classe *clique maximum*. L'intégration d'une heuristique de révision orientée variable dans l'algorithme AC permet d'éviter ce comportement pathologique.

Dans l'avenir, nous comptons étendre DynVAC aux fonctions de coût d'arité arbitraire et identifier des heuristiques de révision appropriées qui pourraient améliorer la performance de DynVAC et en même temps améliorer le minorant produit.

Références

[1] Roman Barták and Pavel Surynek. An improved algorithm for maintaining arc consistency in dynamic constraint satisfaction problems. In *Proc. of the* 18^{th} *International FLAIRS Conference*, pages 161–166, Menlo Park, CA, USA, 2005. AAAI Press.

[2] P. Berlandier and B. Neveu. Maintaining Arc Consistency through Constraint Retraction. In *Proc. of the* 6^{th} *IEEE International Conference on Tools with Artificial Intelligence (TAI94)*, New Orleans, LA, 1994.

[3] Christian Bessière. Arc-consistency in dynamic constraint satisfaction problems. In *Proc. of AAAI'91*, pages 221–226, Anaheim, CA, 1991.

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

[5] M Cooper, S de Givry, M Sanchez, T Schiex, and M Zytnicki. Virtual Arc Consistency for Weighted CSP. In *Proc. of AAAI'2008*, Chicago, USA.

[6] M. Cooper, S. de Givry, M. Sanchez, T. Schiex, M. Zytnicki, and T. Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174 :449–478, 2010.

[7] M C. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2) :199–227, 2004.

[8] Rina Dechter and Avi Dechter. Belief maintenance in dynamic constraint networks. In *Proc. of AAAI'88*, pages 37–42, St. Paul, MN, 1988.

[9] J. Kratica, D. Tosic, V. Filipovic, and I. Ljubic. Solving the Simple Plant Location Problems by Genetic Algorithm. *RAIRO Operations Research*, 35 :127–142, 2001.

[10] 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 the* 19^{th} *IJCAI*, pages 84–89, Edinburgh, Scotland, August 2005.

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

[12] R.J. Wallace and E.C. Freuder. Ordering heuristics for arc consistency algorithms. In *Proceedings of the Biennial Conference-Canadian Society for Computational Studies of Intelligence*, pages 163–163, 1992.

Un solveur de contraintes basé sur les domaines abstraits

Marie Pelleau[1] **Antoine Miné**[2] **Charlotte Truchet**[1] **Frédéric Benhamou**[1]

[1] Université de Nantes, 2 rue de la Houssinière, Nantes
[2] École normale supérieure, 45, rue d'Ulm, Paris
{Prénom.Nom}@univ-nantes.fr Antoine.Mine@ens.fr

Résumé

Dans cet article, nous utilisons des techniques de l'interprétation abstraite (une théorie d'approximation des sémantiques) dans le cadre de la programmation par contraintes (basée sur la logique du premier ordre qui permet de résoudre des problèmes combinatoires). Nous mettons en évidence certains liens et différences entre ces domaines de recherches : tous deux calculent itérativement des points fixes mais emploient des extrapolations et stratégies de raffinement différentes. De plus, nous pouvons mettre en correspondance les consistances en programmation par contraintes et les domaines abstraits non relationnels. Nous utilisons ensuite ces correspondances pour construire un solveur de contraintes abstrait qui s'appuie sur des techniques d'interprétation abstraite (comme les domaines relationnels) pour aller au-delà des solveurs classiques. Les résultats expérimentaux obtenus avec notre prototype sont encourageants.

1 Introduction

L'interprétation abstraite (IA) est une méthode d'approximation des sémantiques de programmes. Elle fournit des réponses garanties à des questions sur leur comportements lors de l'exécution [7, 6]. La programmation par contraintes (PPC) vise à résoudre, avec des techniques génériques, des problèmes combinatoires exprimés de manière déclarative. Cet article étudie l'utilisation de techniques de l'IA en PPC.

1.1 État de l'art

Introduite par Montanari [16], la PPC repose sur l'idée que de nombreux problèmes peuvent être exprimés comme une conjonction de formules logiques du premier ordre, appelées contraintes, chacune représentant une caractéristique combinatoire spécifique du problème [19]. Chaque contrainte vient avec des opérateurs *ad hoc* exploitant sa structure interne afin de réduire la combinatoire. Les contraintes sont ensuite combinées dans des algorithmes de résolution. Une grande partie de l'effort de recherche en PPC porte sur la définition et l'amélioration de contraintes [1] et des algorithmes de résolution. La PPC propose désormais de puissantes techniques d'optimisation combinatoire, avec de nombreuses applications pratiques pour la planification, l'emballage, l'attribution des fréquences, etc. Il subsiste cependant des limitations aux solveurs. Ils sont limités à des domaines non relationnels, tels que des boîtes ou des produits cartésiens d'ensembles d'entiers. De plus, les algorithmes de résolution sont séparés en deux familles : l'une manipule des variables discrètes et l'autre des variables continues. Plusieurs méthodes ont été proposées pour traiter les problèmes mixtes, tels que la discrétisation de variables continues afin de les manipuler dans un solveur discret (comme dans Choco [21]). Malheureusement, le solveur reste purement discret et ne bénéficie pas d'heuristiques développées pour les solveurs continus. Alternativement, il est possible d'ajouter des contraintes mixtes [5] ou des contraintes d'intégrité [3] à un solveur continu, avec des inconvénients similaires.

Dans un autre domaine de recherche, celui de la vérification de programmes, l'interprétation abstraite (IA) est utilisée pour concevoir des analyseurs statiques de programmes qui sont corrects et se terminent toujours (comme Astrée [4]) en développant des approximations calculables de problèmes essentiellement indécidable. La sémantique concrète (incalculable) exprime sous forme de point fixe l'ensemble des comportements

1. Voir `http://www.emn.fr/z-info/sdemasse/gccat` pour un catalogue de contraintes globales existantes.

observables du programme. Une approximation de la sémantique est calculée dans un domaine abstrait limitant l'expressivité à un ensemble de propriétés intéressantes. Les domaines abstraits fournissent des structures de données pour les représentations, des algorithmes efficaces pour calculer des versions abstraites de la sémantique concrète, des opérateurs de points fixes et des opérateurs d'accélération pour calculer des approximations en un temps fini.

La sûreté garantit que l'analyseur observe un ensemble plus grand que celui des comportements du programme. Les domaines abstraits numériques mettent l'accent sur les variables et les propriétés numériques. Ils en existent de nombreux et ils sont particulièrement bien développés. Les plus connus sont les intervalles [6] et les polyèdres [8]. Ces dernières années, de nouveaux domaines ont été développés tels que les octogones [15] et des bibliothèques de domaines abstraits telles que Apron [12] ont vu le jour. Les domaines abstraits numériques peuvent traiter tous les types de variables numériques, y compris les entiers, les rationnels, les réels et les nombres flottants, et même exprimer des relations entre les variables de différents types [14, 4]. Chaque domaine correspond à un certain compromis entre le coût et la précision. Enfin, les domaines peuvent être modifiés et combinés par des opérateurs génériques, tels que la complétion disjonctive et les produits réduits.

Dans cet article, nous cherchons à utiliser les techniques d'IA pour construire un solveur de PPC qui soit abstrait et générique.

Cet articles est organisé comme suit. La section 2 introduit brièvement l'IA et la PPC, et donne des éléments de comparaison. La section 3 définit la PPC à l'aide de techniques de l'IA et présente notre solveur abstrait. Notre prototype ainsi que des résultats expérimentaux sont présentés section 4.

1.2 Travaux connexes

Certaines interactions entre la PPC et les techniques de vérification ont déjà été explorées dans des travaux antérieurs. Par exemple, la PPC a été utilisée pour générer automatiquement des tests de configuration [11], ou pour vérifier des modèles en PPC [13].

Plus récemment, des travaux tels que [9, 22], établissent des liens entre l'IA et les algorithmes de résolution SAT. Notre but est similaire mais lie la PPC à l'IA. Bien que connexes, la PPC et SAT diffèrent assez significativement dans les modèles choisis et les algorithmes de résolution pour que les résultats précédents ne s'appliquent pas. Notre travail s'inscrit dans la continuité de [18] qui étend les méthodes de résolution en PPC afin d'utiliser des représentations plus expressives, telles que les octogones. Cependant, l'ajout d'un nouveau domaine abstrait demande de définir des opérateurs *ad hoc* (consistance, coupe, ...) Dans cet article, nous effectuons le processus inverse : nous concevons un solveur abstrait reposant sur les domaines abstraits et utilisant les opérateurs existants en IA. Ceci permet entre autres choses de définir la consistance de façon unique pour tous les domaines abstraits.

2 Notations et rappels

Dans cette section nous présentons uniquement les notions d'IA et de PPC nécessaires pour la suite (voir [7, 6] et [19] pour une présentation plus détaillée).

2.1 Bases de l'interprétation abstraite

Nous présentons d'abord des éléments d'IA nécessaire pour la conception de notre solveur.

2.1.1 Abstractions de point fixe

Le domaine concret, noté $\mathcal{D}$, correspond aux valeurs que peuvent prendre les variables tout au long du programme. Calculer la sémantique concrète d'un programme peut être indécidable et une approximation est acceptée. L'approximation du domaine concret est appelée domaine abstrait et est notée $\mathcal{D}^\sharp$. Les domaines abstraits sont regroupés en deux familles, les domaines abstraits non-relationnels et les domaines abstraits relationnels. Un domaine abstrait non-relationnel associe une propriété indépendante à chaque variable (comme ses bornes), tandis qu'un domaine abstrait relationnel exprime des relations entre les variables (comme des relations linéaires), ce qui est bien plus expressif mais aussi plus coûteux.

Une fonction de concrétisation $\gamma : \mathcal{D}^\sharp \to \mathcal{D}$ associe à un élément abstrait un élément concret. Un opérateur abstrait $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ est une abstraction correcte de F si $F \circ \gamma \sqsubseteq \gamma \circ F^\sharp$, avec $\sqsubseteq$ la relation d'inclusion. Il peut exister une fonction d'abstraction $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ telle que (α, γ) forme une correspondance de Galois (notée $\mathcal{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$), assurant que chaque élément concret X admet une abstraction optimale $\alpha(X)$. Si une correspondance de Galois existe entre le domaine concret et le domaine abstrait, toute fonction F dans $\mathcal{D}$ admet une abstraction $F^\sharp$ dans $\mathcal{D}^\sharp$ telle que $\forall X^\sharp \in \mathcal{D}^\sharp, (\alpha \circ F \circ \gamma)(X^\sharp) \sqsubseteq F^\sharp(X^\sharp)$. La fonction abstraite $F^\sharp$ est dite optimale ssi $\alpha \circ F \circ \gamma = F^\sharp$.

Afin d'analyser un programme, chaque ligne de code est analysée. Pour cela, chaque instruction du programme est associée à une fonction, appelée fonction de transfert, qui modifie les valeurs possibles pour les variables.

Définition 2.1 (Fonction de transfert) *Soit C une ligne de code à analyser. Une fonction de transfert $F : \mathcal{P}(\mathcal{D}) \to \mathcal{P}(\mathcal{D})$ retourne, en fonction d'un ensemble de départ, un ensemble d'environnements correspondant à tous les états accessibles après avoir exécuté C.*

Exemple 2.1 *Considérons une expression booléenne, la fonction de transfert ne conserve que les environnements satisfaisant l'expression booléenne.*

Soit deux variables x et y dont les valeurs possibles sont comprises dans $[-10, 10]$, après l'expression booléenne $x \leq 0$ la fonction de transfert filtre les valeurs de x afin de satisfaire la condition. On a maintenant x dans $[-10, 0]$ et y dans $[-10, 10]$.

Chaque instruction du programme est associée à une fonction de transfert, le programme est donc associé à une composition de ces fonctions. Prouver que le programme est correct revient à calculer le plus petit point fixe de cette composition de fonctions. La sémantique concrète d'un programme est donc donnée comme le plus petit point fixe $\mathrm{lfp}_\perp F$ d'un opérateur $F : \mathcal{D} \to \mathcal{D}$ dans une structure partiellement ordonnée $(\mathcal{D}, \sqsubseteq, \perp, \sqcup)$, avec $\sqcup$ l'opérateur pour l'union, telle qu'un treillis ou un ordre partiel complet. De la même façon, notons $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \perp^\sharp, \sqcup^\sharp)$ le domaine abstrait. Le plus petit point fixe, $\mathrm{lfp}_\perp F$ peut être approximé par $\bigsqcup^\sharp_{i \in \mathrm{Ord}} F^{\sharp i}(\perp^\sharp)$. Cette limite peut ne pas être calculable, même si $F^\sharp$ l'est, ou peut nécessiter un très grand nombre d'itérations. Elle est donc généralement remplacée par la limite d'une séquence croissante : $X^\sharp_0 = \perp^\sharp$, $X^\sharp_{i+1} = X^\sharp_i \triangledown F^\sharp(X^\sharp_i)$ utilisant un opérateur d'élargissement $\triangledown$ afin d'accélérer la convergence. L'élargissement est conçu afin de sur-approximer $\sqcup$ et converger en un temps fini δ en un post point fixe $X^\sharp_\delta$ de $F^\sharp$, ce qui implique $\gamma(X^\sharp_\delta) \sqsupseteq \mathrm{lfp}_\perp F$. Cette limite est généralement améliorée par des itérations décroissantes : $Y^\sharp_0 = X^\sharp_\delta, Y^\sharp_{i+1} = Y^\sharp_i \triangle F^\sharp(Y^\sharp_i)$, utilisant un opérateur de rétrécissement $\triangle$ conçu pour rester au-dessus de tout point fixe de F et converger en un temps fini. Comme tous les $Y^\sharp_i$ sont des abstractions de $\mathrm{lfp}_\perp F$, les itérations peuvent être arrêtées à tout moment.

2.1.2 Itérations locales

Les itérations décroissantes permettent d'améliorer le point fixe calculé, mais ont aussi été utilisées localement par Granger [10], au sein du calcul de $F^\sharp$. Granger observe que l'opérateur concret F utilise souvent des opérateurs de clôture inférieure, *i.e.,* des opérateurs ρ qui sont croissants, idempotents ($\rho \circ \rho = \rho$) et contractants ($\rho(X) \sqsubseteq X$). Étant donnée une abstraction $\rho^\sharp$ de ρ, la limite $Y^\sharp_\delta$ de la séquence $Y^\sharp_0 = X^\sharp$, $Y^\sharp_{i+1} = Y^\sharp_i \triangle \rho^\sharp(Y^\sharp_i)$ est une abstraction de $\rho(\gamma(X^\sharp))$. Si $\rho^\sharp$ n'est pas une abstraction optimale de ρ, $Y^\sharp_\delta$ peut être significativement plus précis que $\rho^\sharp(X^\sharp)$. Une application pertinente est celle de l'analyse de conjonctions de tests complexes $C_1 \wedge \cdots \wedge C_p$ où chaque test atomique C_i est modélisé dans l'abstraction par $\rho^\sharp_i$. Généralement, $\rho^\sharp = \rho^\sharp_1 \circ \cdots \circ \rho^\sharp_p$ n'est pas optimal, même si chaque $\rho^\sharp_i$ l'est. Une application complémentaire est l'analyse d'un seul test C_i utilisant une séquence non-optimale d'abstractions de tests relaxés. Par exemple, les expressions non-linéaires peuvent être remplacées par des intervalles calculés avec les bornes des variables [14]. Comme l'exécution des tests relaxés améliore ces bornes, la relaxation n'est pas idempotente et peut être améliorée par les itérations locales. Le lien entre les itérations locales et l'amélioration du plus petit point fixe repose sur le fait que $\rho(X)$ calcule un point fixe trivial : le plus grand point fixe de ρ plus petit que X : $\mathrm{gfp}_X \rho$. Dans ces deux cas, une itération décroissante commence par une abstraction d'un point fixe ($\mathrm{lfp}_\perp F$ dans un cas, $\mathrm{gfp}_X \rho$ dans l'autre) et calcule une abstraction plus petite de ce point fixe.

2.2 Programmation par contraintes

Nous présentons maintenant des définitions basiques de PPC. Dans cette section, nous utilisons la terminologie de PPC, et précisons les termes avec un sens différents en IA et PPC.

Les problèmes sont modélisés sous forme d'un problème de satisfaction de contraintes (CSP) tel que défini ci-dessous :

Définition 2.2 *Un CSP est défini par un ensemble de variables $(v_1, \ldots, v_n)$ prenant leurs valeurs dans les domaines $(\hat{D}_1, \ldots, \hat{D}_n)$ et un ensemble de contraintes $(C_1, \ldots, C_p)$.*

Soit D_i le domaine de la variable v_i, notons $D = D_1 \times \cdots \times D_n$ l'*espace de recherche*. Comme l'espace de recherche est modifié durant la résolution, nous distinguons l'espace de recherche initial et le notons $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$. Les problèmes peuvent être discrets ($\hat{D} \subseteq \mathbb{Z}^n$) ou continus ($\hat{D} \subseteq \mathbb{R}^n$). Cependant, les domaines sont toujours bornés.

Soit D_i le domaine de la variable v_i, et $x_i \in D_i$. On note $C(x_1, \ldots, x_n)$ le fait que la contrainte est satisfaite quand chaque variable v_i prend la valeur x_i. L'ensemble de solutions est $S = \{(s_1, \ldots, s_n) \in \hat{D} \,|\, \forall i \in [\![1, p]\!], C_i(s_1, \ldots, s_n)\}$, avec p le nombre de contraintes et où $[\![a, b]\!] = \{x \in \mathbb{Z} \,|\, a \leq x \leq b\}$ correspond à l'intervalle des entiers entre a et b.

Pour les problèmes discrets, deux représentations des domaines sont généralement utilisées : les ensembles et les intervalles. Nous ne présentons que la représentation ensembliste.

Définition 2.3 (Produit cartésien d'entiers) *Soient $v_1 \dots v_n$ des variables de domaines discrets finis $\hat{D}_1 \dots \hat{D}_n$. Un produit cartésien d'ensembles d'entiers est appelé produit cartésien d'entiers et est exprimé dans $\mathcal{D}$ par :* $\mathcal{S}^\sharp = \left\{ \prod_i X_i \,|\, \forall i, X_i \subseteq \hat{D}_i \right\}$

Pour les problèmes continus, les domaines sont représentés par des intervalles à bornes flottantes. Soit $\mathbb{F}$ l'ensemble des flottants représentables en machine. Soit $a, b \in \mathbb{F}$, notons $[a, b] = \{x \in \mathbb{R} \,|\, a \leq x \leq b\}$ l'intervalle des réels compris entre a et b, et $\mathbb{I} = \{[a, b] \,|\, a, b \in \mathbb{F}\}$ l'ensemble de tous ces intervalles.

Définition 2.4 (Boîte) *Soient $v_1 \dots v_n$ des variables de domaines continus $\hat{D}_1 \dots \hat{D}_n$. Un produit cartésien d'intervalles est appelé boîte et est exprimé dans $\mathcal{D}$ par :* $\mathcal{B}^\sharp = \left\{ \prod_i I_i \,|\, I_i \in \mathbb{I}, I_i \subseteq \hat{D}_i \right\} \cup \emptyset$

Résoudre un CSP revient à calculer exactement ou une approximation de l'ensemble des solutions S.

Définition 2.5 (Approximation) *Une* approximation complète *(resp.* correcte*), de l'ensemble des solutions est un ensemble $\mathcal{A}$ de produits de domaines telle que $\forall (D_1, \dots, D_n) \in \mathcal{A}$, $\forall i$, $D_i \subseteq \hat{D}_i$ et $S \subseteq \bigcup_{(D_1,\dots,D_n)\in\mathcal{A}} D_1 \times \dots \times D_n$ (resp. $\bigcup_{(D_1,\dots,D_n)\in\mathcal{A}} D_1 \times \dots \times D_n \subseteq S$).*

La correction garantit que les points trouvés sont bien des solutions, alors que la complétude garantit qu'aucune solution n'est perdue. En général, un solveur de contraintes sur les domaines finis est complet et correct, ce qui revient à calculer les solutions. Cela n'est pas possible dans le cas de domaines continus car les réels ne sont pas représentables exactement en machine, et un solveur continu sera soit complet (la plupart du temps) soit correct. Les notions de correction et complétude sont différentes en IA et en PPC. Afin d'éviter toute ambiguïté, le terme sur-approximation (resp. sous-approximation) sera utilisé pour une approximation PPC-complète IA-correcte (resp. PPC-correcte IA-complète).

Dans cet article, nous considérons les méthodes de résolution qui sur-approximent les solutions des problèmes continus et calculent les solutions exactes pour les problèmes discrets. Ces méthodes alternent deux étapes : la propagation et l'exploration.

2.2.1 Propagation

Le but de la propagation est d'utiliser les contraintes afin de réduire les domaines. Intuitivement, nous supprimons des domaines les valeurs inconsistantes, *i.e.,* les valeurs ne pouvant pas être dans une solution. Plusieurs formes de consistance ont été proposées. Nous présentons ici les plus communes.

Définition 2.6 (Consistance d'arc généralisée) *Soient $v_1 \dots v_n$ des variables de domaines discrets finis $D_1 \dots D_n, D_i \subseteq \hat{D}_i$, et C une contrainte. Les domaines sont dits* arc-consistants généralisés *(GAC) pour C ssi $\forall i \in [\![1, n]\!], \forall x_i \in D_i, \forall j \neq i, \exists x_j \in D_j$ tel que $C(x_1, x_2, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_n)$.*

Définition 2.7 (Consistance d'enveloppe) *Soient $v_1 \dots v_n$ des variables de domaines continus représentés par les intervalles $D_1 \dots D_n \in \mathbb{I}, D_i \subseteq \hat{D}_i$, et C une contrainte. Les domaines $D_1 \dots D_n$ sont dits Hull-consistant (HC) pour C ssi $D_1 \times \dots \times D_n$ est la plus petite boîte à bornes flottantes contenant les solutions de C dans $D_1 \times \dots \times D_n$.*

Chaque contrainte C et consistance vient avec un algorithme appelé *propagateur*, qui essaie d'atteindre la consistance. Quand plusieurs contraintes sont considérées, une *boucle de propagation* exécute les propagateurs des contraintes jusqu'à ce qu'un point fixe soit atteint. Comme indiqué dans [2], l'ordre d'exécution des propagateurs n'a pas d'importance car l'ensemble des domaines forme un treillis fini ($\mathcal{B}^\sharp$, $\mathcal{I}^\sharp$ ou $\mathcal{S}^\sharp$) et le point fixe consistant est son unique plus petit élément. Lorsque la consistance est trop coûteuse à réaliser, les propagateurs et les boucles de propagations calculent une sur-approximation (*e.g.,* enlève seulement quelques valeurs inconsistantes). En plus de restreindre l'espace de recherche, la propagation est parfois en mesure de découvrir qu'il n'existe pas de solution, ou que tous les points sont des solutions.

2.2.2 Exploration

En règle générale, la propagation seule ne permet pas de calculer les solutions exactes (dans le cas discret) ou une sur-approximation assez précise (dans le cas continu). Ainsi, dans une deuxième étape, différentes hypothèses sur les valeurs des variables sont testées. Dans le cas discret, une variable est choisie et est instanciée pour chaque valeur dans son domaine. Dans le cas continu, son domaine est divisé en deux sous-domaines plus petits. L'algorithme de résolution continue en sélectionnant un espace de recherche et applique une nouvelle étape de propagation (car il peut ne plus être consistant), puis fait d'autres choix. Ces itérations de propagation et de choix s'arrêtent lorsqu'il peut être prouvé que l'espace de recherche ne contient pas de solution, seulement des solutions ou, dans le cas continu, lorsque sa taille est inférieure à un seuil spécifié par l'utilisateur. Dans le cas discret, dans le pire des cas, toutes les variables sont instanciées. Après avoir exploré une branche, en cas d'erreur ou si toutes les solutions doivent être calculées, l'algorithme revient à un point de choix précédent (instanciation ou coupe) par *backtracking* et tente une autre hypothèse.

2.3 Comparaison entre l'IA et la PPC

Cette section présente, de manière informelle, certains liens entre l'IA et la PPC. La section suivante formalisera ces liens en exprimant la PPC dans le cadre de l'IA.

Les techniques de ces deux domaines reposent sur la théorie des points fixes dans les treillis. Ils poursuivent des objectifs similaires : calculer ou sur-approximer les solutions d'équations complexes en manipulant des vues abstraites d'ensembles de solutions potentielles, comme les boîtes (appelés domaines en PPC et éléments de domaines abstraits en IA). Cependant, les méthodes diffèrent, les méthodes de résolution visent la complétude et améliore donc jusqu'à une précision arbitraire. Au contraire, la précision des analyseurs abstraits est fixée par le choix du domaine abstrait : ils peuvent rarement calculer des sur-approximations de précision arbitraire. Le choix du domaine abstrait définit le coût et la précision d'un analyseur, alors que le choix du domaine en PPC définit le coût d'un solveur pour atteindre une précision donnée.

Même si ils visent la complétude, les solveurs utilisent néanmoins des domaines non relationnels simples. Ils s'appuient sur les collections de domaines simples (disjonctions en IA) pour atteindre la précision souhaitée. Les domaines sont homogènes et ne peuvent pas mélanger des variables de type différent. Au contraire, l'IA possède de nombreux domaines abstraits, pouvant être relationnels et hétérogènes.

Sur le plan algorithmique, l'IA et la PPC partagent des idées communes. L'itération des propagations en PPC est similaires aux itérations locales en IA. En effet, approximer une consistance en PPC est similaire à approximer l'effet d'un test complexe en IA. Cependant, les méthodes de résolution en PPC utilisent des techniques telles que les points de choix et le backtracking, qui n'ont pas d'équivalent en IA. Inversement, l'élargissement de l'IA n'a pas d'équivalent en PPC, car la PPC n'emploie pas d'itérations croissantes.

Enfin, alors que les analyseurs abstraits sont généralement définis de façon très générique et paramétrable grâce aux domaines abstraits, les solveurs sont bien moins flexibles et incorporent les domaines et les consistances ainsi que le type des variables dans leur conception. Par la suite, nous concevons un solveur abstrait qui permet d'éviter ces désavantages et permet de bénéficier de la grande bibliothèque de domaines abstraits conçus pour l'IA.

3 Un solveur de contraintes abstrait

Nous présentons maintenant notre principale contribution : exprimer la résolution de contraintes comme un analyseur abstrait. Nous définissons donc les domaines concrets et abstraits, les opérateurs abstraits pour la consistance et le choix, et le schéma itératif.

3.1 Résolution concrète

Un CSP est similaire à l'analyse d'une conjonction de tests et peut être formalisé en terme d'itérations locales. Considérons comme domaine concret $\mathcal{D}$ les sous-ensembles de l'espace de recherche $\hat{D} = \hat{D}_1 \times \cdots \times \hat{D}_n$ du CSP, *i.e.,* $(\mathcal{P}(\hat{D}), \subseteq, \emptyset, \cup)$. Chaque contrainte C_i vient avec un opérateur de clôture inférieure $\rho_i : \mathcal{P}(\hat{D}) \to \mathcal{P}(\hat{D})$, tel que $\rho_i(X)$ conserve uniquement les points de X satisfaisant C_i. Les solutions concrètes du problèmes sont $S = \rho(\hat{D})$, où $\rho = \rho_1 \circ \cdots \circ \rho_p$, et peuvent être exprimées sous forme de point fixe $\mathrm{gfp}_{\hat{D}}\, \rho$.

3.2 Domaines abstraits

Les solveurs ne manipulent pas des points dans $\hat{D}$, mais plutôt des collections de points d'une certaine forme, telles que des boîtes, appelées domaines en PPC. Nous montrons que les domaines en PPC sont des éléments d'un domaine abstrait $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \bot^\sharp, \sqcup^\sharp)$ en IA, dépendant d'une consistance. Nous ajoutons aussi une fonction *de précision* $\tau : \mathcal{D}^\sharp \to \mathbb{R}^+$ qui sera utilisée dans la condition d'arrêt (Def. 3.2).

Exemple 3.1 *La consistance d'arc généralisée (Def. 2.6) correspond au domaine abstrait des produits cartésiens d'entiers $\mathcal{S}^\sharp$ (Def. 2.3), ordonné par l'inclusion sur les ensembles. Elle est liée au domaine concret $\mathcal{D}$ par une correspondance de Galois standard :*
$\mathcal{D} \underset{\alpha_a}{\overset{\gamma_a}{\leftrightarrows}} \mathcal{S}^\sharp$
$\gamma_a(S_1, \ldots, S_n) = S_1 \times \cdots \times S_n$
$\alpha_a(X) = \lambda i.\{x \mid \exists (x_1, \ldots, x_n) \in X,\, x_i = x\}$

La fonction de précision τ_a utilise la taille du plus grand ensemble d'entiers, moins un, de sorte que les singletons aient une taille 0 : $\tau_a(S_1, \ldots, S_n) = \max_i(|S_i| - 1)$

Exemple 3.2 *La consistance d'enveloppe (Def. 2.7) correspond au domaine des boîtes à bornes flottantes $\mathcal{B}^\sharp$ (Def. 2.4). Nous utilisons la correspondance de Galois et la fonction de précision suivantes :*
$\mathcal{D} \underset{\alpha_h}{\overset{\gamma_h}{\leftrightarrows}} \mathcal{B}^\sharp$
$\gamma_h([a_1, b_1], \ldots, [a_n, b_n]) = [a_1, b_1] \times \cdots \times [a_n, b_n]$
$\alpha_h(X) = \lambda i.[\ \max\{x \in \mathbb{F} \mid \forall (x_1, \ldots, x_n) \in X,\, x_i \geq x\},$
$\qquad \min\{x \in \mathbb{F} \mid \forall (x_1, \ldots, x_n) \in X,\, x_i \leq x\}]$
$\tau_h([a_1, b_1], \ldots, [a_n, b_n]) = \max_i(b_i - a_i)$

Notons qu'à chacun des choix correspond un domaine abstrait non-relationnel classique en IA. De

plus, il correspond à un produit cartésien homogène d'un seul et même ensemble de base (type de représentation pour une variable). Cependant, ceci n'est pas obligatoire, et de nouveaux solveurs peuvent être conçus en modifiant davantage les domaines abstraits. Une première idée est d'appliquer différentes consistances à différentes variables, ce qui permettrait en particulier de mélanger des variables discrètes avec des variables continues. Une seconde idée est de paramétrer le solveur avec d'autres domaines abstraits existant en IA, en particulier les domaines abstraits relationnels ce que nous illustrons ci-dessous.

Exemple 3.3 *Le domaine abstrait des octogones $\mathcal{O}^\sharp$ [15] assigne une borne (flottante) supérieure à chaque expression binaire unitaire $\pm v_i \pm v_j$ sur les variables $v_1, \ldots, v_n$. Il admet une correspondance de Galois et la fonction de précision définie dans [18] :*
$\mathcal{D} \underset{\alpha_o}{\overset{\gamma_o}{\leftrightarrows}} \mathcal{O}^\sharp$
$\mathcal{O}^\sharp = \{\alpha v_i + \beta v_j \,|\, i,j \in [\![1,n]\!],\, \alpha,\beta \in \{-1,1\}\} \to \mathbb{F}$
$\gamma_o(X^\sharp) = \{(x_1,\ldots,x_n) \in \mathbb{R}^n \,|\, \forall i,j,\alpha,\beta, \alpha x_i + \beta x_j \leq X^\sharp(\alpha v_i + \beta v_j)\}$
$\alpha_o(X) = \lambda(\alpha v_i + \beta v_j).\min\{x \in \mathbb{F} \,|\, \forall (x_1,\ldots,x_n) \in X,\, \alpha x_i + \beta x_j \leq x\}$

$$\tau_o(X^\sharp) = \min(\max_{i,j,\beta}(X^\sharp(v_i+\beta v_j)+X^\sharp(-v_i-\beta v_j)), \max_i(X^\sharp(v_i+v_i)+X^\sharp(-v_i-v_i))/2)$$

En représentant un octogone comme une intersection de boîtes, la fonction de précision retourne la plus grande dimension de la plus petite boîte.

Le domaine abstrait des polyèdres $\mathcal{P}^\sharp$ peut aussi être défini comme montré dans [17].

3.3 Contraintes et consistance

À partir de maintenant, nous supposons qu'un domaine abstrait $\mathcal{D}^\sharp$ sous-jacent au solveur est fixé. Pour une sémantique concrète de contraintes $\rho = \rho_1 \circ \cdots \circ \rho_p$ donnée, et si $\mathcal{D}^\sharp$ admet une correspondance de Galois $\mathcal{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$, alors la sémantique du propagateur optimal réalisant la consistance pour toutes les contraintes est simplement : $\alpha \circ \rho \circ \gamma$. Les solveurs réalisent ceci de façon algorithmique, ils exécutent le propagateur de chaque contrainte jusqu'à atteindre un point fixe ou, quand ce processus est estimé trop coûteux, s'arrêtent avant que le point fixe ne soit atteint. En observant que chaque propagateur correspond à une fonction abstraite modélisant un test $\rho_i^\sharp$ dans $\mathcal{D}^\sharp$, nous retrouvons les itérations locales de Granger pour analyser des conjonctions de tests [10].

3.4 Complétion disjonctive et coupe

Afin d'approximer les solutions à une précision arbitraire, les solveurs utilisent un pavage fini d'éléments abstraits de $\mathcal{D}^\sharp$. En IA, un pavage fini correspond à une complétion disjonctive [7]. Considérons le domaine abstrait $\mathcal{E}^\sharp = \mathcal{P}_{\text{fini}}(\mathcal{D}^\sharp)$, et équipons-le d'un ordre de Smyth $\sqsubseteq^\sharp_\mathcal{E}$, un ordre classique pour les complétions disjonctives défini ainsi : $X^\sharp \sqsubseteq^\sharp_\mathcal{E} Y^\sharp \iff \forall B^\sharp \in X^\sharp, \exists C^\sharp \in Y^\sharp, B^\sharp \sqsubseteq^\sharp C^\sharp$

Définition 3.1 (Opérateur de coupe) *Un* opérateur de coupe *est un opérateur $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ tel que*

1. $\forall e \in \mathcal{D}^\sharp, |\oplus(e)|$ *est fini,*
2. $\forall e \in \mathcal{D}^\sharp, \forall e_i \in \oplus(e),\, e_i \sqsubseteq^\sharp e$, *et*
3. $\forall e \in \mathcal{D}^\sharp, \gamma(e) = \bigcup\{\gamma(e_i) \,|\, e_i \in \oplus(e)\}$.

Chaque élément de $\oplus(e)$ étant inclus dans e (condition 2), on a $\oplus(e) \sqsubseteq^\sharp_\mathcal{E} \{e\}$. De plus, la condition 3 montre que $\oplus$ est une abstraction de l'identité. Et donc, $\oplus$ peut être utilisé à n'importe quel moment de la résolution sans en altérer la correction. Nous en présentons ici quelques exemples.

Exemple 3.4 (Coupe dans $\mathcal{S}^\sharp$) *L'instanciation d'une variable v_i de domaine discret $X^\sharp = (S_1,\ldots,S_n) \in \mathcal{S}^\sharp$ est un opérateur de coupe : $\oplus_a(X^\sharp) = \{(S_1,\ldots,S_{i-1},x,S_{i+1},\ldots,S_n) \,|\, x \in S_i\}$*

Exemple 3.5 (Coupe dans $\mathcal{B}^\sharp$) *Couper une boîte en deux le long d'une variable v_i de domaine continu $X^\sharp = (I_1,\ldots,I_n) \in \mathcal{B}^\sharp$ est un opérateur de coupe : $\oplus_h(X^\sharp) = \{(I_1,\ldots,I_{i-1},[a,h],I_{i+1},\ldots,I_n),(I_1,\ldots,I_{i-1},[h,b],I_{i+1},\ldots,I_n)\}$ où $I_i = [a,b]$ et $h = (a+b)/2$ arrondi dans $\mathbb{F}$ de manière quelconque.*

Exemple 3.6 (Coupe dans $\mathcal{O}^\sharp$) *Étant donnée une expression binaire $\alpha v_i + \beta v_j$, nous définissons l'opérateur de coupe pour les octogones $X^\sharp \in \mathcal{O}^\sharp$ le long de cette expression comme : $\oplus_o(X^\sharp) = \{X^\sharp[(\alpha v_i+\beta v_j) \mapsto h],\, X^\sharp[(-\alpha v_i - \beta v_j) \mapsto -h]\}$ où $h = (X^\sharp(\alpha v_i + \beta v_j) - X^\sharp(-\alpha v_i - \beta v_j))/2$, arrondi dans $\mathbb{F}$ de manière quelconque.*

De même, nous pouvons définir un opérateur de coupe pour les polyèdres (voir [17]).

Ces opérateurs de coupe sont paramétrés par le choix d'une direction de coupe (le long d'une variable ou d'une expression). Pour un domaine non-relationnel, nous pouvons utiliser les deux stratégies classiques en PPC : définir un ordre et couper les variables tour à tour suivant l'ordre choisi, ou couper le long de la variable ayant le plus grand ou le plus petit domaine (*i.e.*, $|S_i|$ ou $b_i - a_i$). On peut naturellement étendre ces stratégies aux octogones en remplaçant l'ensemble des variables par l'ensemble (fini) des expressions binaires ou utiliser un opérateur de coupe plus rapide et plus simple tel que couper le long de la variable ayant le plus grand domaine.

Afin d'assurer la terminaison du solveur, nous imposons que toute suite de réductions, coupes et sélections retourne à partir d'un moment un élément assez petit pour τ :

Définition 3.2 *Les deux opérateurs $\tau : \mathcal{D}^\sharp \to \mathbb{R}^+$ et $\oplus : \mathcal{D}^\sharp \to \mathcal{E}^\sharp$ sont dits compatibles si, pour tout opérateur de réduction $\rho^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ (i.e., $\forall X^\sharp \in \mathcal{D}^\sharp, \rho^\sharp(X^\sharp) \sqsubseteq^\sharp X^\sharp$) et pour toute famille d'opérateurs de sélection $\pi_i : \mathcal{E}^\sharp \to \mathcal{D}^\sharp$ (i.e., $\forall Y^\sharp \in \mathcal{E}^\sharp, \pi_i(Y^\sharp) \in Y^\sharp$) : $\forall e \in \mathcal{D}^\sharp, \forall r \in \mathbb{R}^{>0}, \exists K$ tel que $\forall j \geq K$, $(\tau \circ \pi_j \circ \oplus \circ \rho^\sharp \circ \cdots \circ \pi_1 \circ \oplus \circ \rho^\sharp)(e) \leq r$.*

Les opérateurs de coupes définis précédemment, $\oplus_a$, $\oplus_h$ et $\oplus_o$ sont compatibles avec les fonctions de précision τ_a, τ_h et τ_o.

La procédure d'exploration peut être représentée par un arbre de recherche où chaque nœud correspond à un espace de recherche et dont les fils sont construits par l'opérateur de coupe. Avec cette représentation, l'ensemble des nœuds à une profondeur donnée correspond à une disjonction sur-approximant l'ensemble des solutions. De plus, une série d'opérateurs de réduction (ρ), de sélection (π) et de coupe ($\oplus$) correspond à une branche. La définition 3.2 stipule que chaque branche de l'arbre de recherche est finie.

3.5 Résolution abstraite

Notre algorithme de résolution est présenté Fig. 1. Il maintient dans toExplore et sols deux disjonctions dans $\mathcal{E}^\sharp$, et itère les étapes suivantes : choisir un élément abstrait e de toExplore (**pop**), appliquer la consistance ($\rho^\sharp$), et soit supprimer e, soit l'ajouter à la liste de solutions sols, soit le couper ($\oplus$). La résolution commence avec le plus grand élément $\top^\sharp$ de $\mathcal{D}^\sharp$ tel que $\gamma(\top^\sharp) = \hat{D}$.

3.5.1 Correction et terminaison

À chaque itération, $\bigcup\{\gamma(x) \,|\, x \in \text{toExplore} \cup \text{sols}\}$ est une sur-approximation de l'ensemble des solutions, car la consistance $\rho^\sharp$ est une abstraction de la sémantique concrète ρ pour les contraintes et l'opérateur de coupe $\oplus$ est une abstraction de l'identité. Notons que les éléments abstraits dans sols sont consistants et soit contiennent uniquement des solutions soit sont plus petits que r. L'algorithme termine quand toExplore est vide, et donc sols sur-approxime l'ensemble des solutions avec des éléments consistants contenant uniquement des solutions ou plus petits que r. Afin de calculer exactement l'ensemble des solutions dans le cas discret, il suffit de choisir $r = 0$.

La terminaison est assurée par la proposition suivante :

Proposition 3.1 *Si τ et $\oplus$ sont compatibles, l'algorithme 1 se termine.*

Preuve Supposons par l'absurde que l'arbre de recherche est infini. Sa largeur étant finie par Def. 3.1, il devrait exister une branche infinie (lemme de König), ce qui contredit Def. 3.2.

L'arbre de recherche étant fini, l'algorithme 1 se termine.

L'algorithme 1 utilise une structure de données de file et coupe le plus ancien élément abstrait en premier. Des stratégies de sélection plus élaborées, telles que couper l'élément le plus grand pour τ, peuvent être définies. Quelle que soit la stratégie choisie, l'algorithme reste correct et termine.

3.5.2 Comparaison avec l'IA

Comme les itérations locales en IA, notre solveur effectue des itérations abstraites décroissantes. Plus précisément, toExplore$\cup$sols est décroissant pour $\sqsubseteq^\sharp_{\mathcal{E}}$ dans le domaine de complétion disjonctive $\mathcal{E}^\sharp$ à chaque itération de la boucle. En effet, $\rho^\sharp$ est contractant dans $\mathcal{D}^\sharp$, et $\oplus(e) \sqsubseteq^\sharp_{\mathcal{E}} \{e\}$. Cependant, notre solveur diffère de l'IA classique sur deux points. Premièrement, il n'existe pas d'opérateurs de coupe en IA, les nouveaux éléments d'une disjonction étant généralement ajoutés au niveau des jointures dans le graphe de flot de contrôle. Deuxièmement, la stratégie de résolution itérative est plus élaborée qu'en IA. L'utilisation de l'opérateur de rétrécissement est remplacée par une structure de données permettant de maintenir une liste ordonnée des éléments abstraits et une stratégie de coupe permet une réduction et assure sa terminaison.

```
liste de domaines abstraits sols ← ∅
queue de domaines abstraits toExplore ← ∅
domaine abstrait e ∈ D♯
push ⊤♯ dans toExplore
while toExplore ≠ ∅ do
    e ← ρ♯(pop(toExplore))
    if e ≠ ∅ then
        if τ(e) ≤ r ou isSol(e) then
            sols ← sols ∪ e
        else
            push ⊕(e) dans toExplore
        end
    end
end
```

Algorithm 1: Solveur abstrait générique

4 Le solveur Absolute

Nous avons implanté un prototype de solveur abstrait, d'après les idées ci-dessus. Les principales caractéristiques et résultats préliminaires sont présentés ici.

4.1 Implémentation

Notre prototype de solveur, appelé Absolute, est implanté au-dessus de Apron, une bibliothèque OCaml de domaines abstraits numériques pour l'analyse statique [12]. Différents domaines abstraits sont implantés dans Apron, tels que les intervalles, les octogones et les polyèdres. De plus, Apron offre une API uniforme permettant de cacher les algorithmes internes des domaines abstraits, et permet de traiter les variables entières et réelles et les contraintes non linéaires.

4.1.1 Consistance

Apron fournit un langage de contraintes arithmétiques suffisant pour exprimer la plupart des CSP numériques : égalités et inégalités d'expressions numériques, comprenant les opérateurs $+$, $-$, $\times$, $/$, $\sqrt{}$, puissance, modulo et arrondi vers des entiers. Les fonctions de transfert de tests fournissent naturellement des propagateurs pour les contraintes. En interne, chaque domaine abstrait implante ses propres algorithmes afin de gérer les tests, comprenant des méthodes sophistiquées permettant de traiter les contraintes non-linéaires (telles que HC4 et la linéarisation [14]). Notre solveur effectue des itérations locales jusqu'à ce qu'un point fixe ou un nombre maximum d'itérations soit atteint (fixé à trois afin d'assurer une résolution rapide). Dans les solveurs de PPC, seules les contraintes contenant au moins une variable dont le domaine a été modifié à l'itération précédente sont propagées. Cependant, pour des raisons de simplicité, notre solveur propage toutes les contraintes à chaque itération.

4.1.2 Opérateur de coupe

Notre solveur utilise un opérateur de coupe très simple, consistant à couper selon la variable ayant le plus grand domaine, y compris dans le cas d'éléments abstraits relationnels. Pour des raisons de simplicité, nous ne coupons pas ceux-ci selon des directions non parallèles aux axes, alors que c'est autorisé par l'exemple 3.6. Ceci sera traité dans des travaux futurs, de même que la prise en compte des stratégies sophistiquées de choix de variables utilisées dans les solveurs PPC actuels.

4.2 Résultats

Nous avons testé Absolute sur deux classes de problèmes : la première est constituée de problèmes continus, le but étant de comparer l'efficacité d'Absolute à celle d'un solveur classique en PPC. La deuxième est composée de problèmes mixtes, que les solveurs classiques ne peuvent pas traiter.

4.2.1 Résolution continue

Nous avons utilisé des problèmes du benchmark COCONUT [2] contenant uniquement des variables continues et comparé les résultats obtenus avec Absolute à ceux obtenus avec Ibex [3], un solveur continu basé sur les intervalles. De plus, nous avons comparé Absolute à notre extension d'Ibex pour les octogones [18], ce qui nous permet de comparer le choix du domaine abstrait (intervalles ou octogones) indépendamment du choix du solveur (solveur classique de PPC ou solveur basé sur l'IA). Le tableau 1 donne les temps de calculs en secondes et le nombre de nœuds crées pour trouver toutes les solutions pour chacun des problèmes.

En moyenne, Absolute est compétitif comparé à l'approche de PPC. Plus précisément, il est globalement plus lent sur les problèmes contenant des égalités et plus rapide sur les problèmes contenant des inégalités. Ces différences de performances ne semblent pas être dues au type des contraintes mais plutôt au ratio : nombre de contraintes dans lesquelles une variable apparait sur le nombre total de contraintes. Comme précisé précédemment, à chaque itération, toutes les contraintes sont propagées, même celles dont aucune des variables n'a été modifiée. Ceci augmente le temps de calcul à chaque itération et donc augmente le temps de calcul total. Ceci explique les mauvaises performances d'Absolute sur les problèmes `brent-10` et `nbody5.1`.

De plus, dans Absolute, la boucle de propagation est arrêtée après trois itérations, alors que dans l'approche de PPC la boucle de propagation atteint le point fixe. De ce fait, la consistance dans Absolute peut être moins précise que celle utilisée dans Ibex, ce qui réduit le temps passé lors de la propagation mais peut augmenter la phase d'exploration. Ceci explique le fait que le nombre de nœuds crées par Absolute est souvent plus grand que celui par Ibex.

Ces expériences montrent que notre prototype, bien que n'ayant que très peu des stratégies de PPC, se comporte raisonnablement bien sur un benchmark classique. De futurs travaux comprendront une analyse plus poussée des performances et bien sûr des

2. Disponible à l'adresse `http://www.mat.univie.ac.at/~neum/glopt/coconut/`.

3. Disponible à l'adresse `http://www.emn.fr/z-info/ibex/`.

nom	# vars	ctr type	$\mathcal{B}^\sharp$				$\mathcal{O}^\sharp$			
			Ibex		Absolute		Ibex		Absolute	
b	4	=	*0,02*	551	*0,10*	577	*0,26*	147	*0,14*	1057
nbody5.1	6	=	*96,0*	598 521	*1538,3*	5 536 283	*27,1*	7 925	$\geq 1h$	-
ipp	8	=	*38,8*	237 445	*39,2*	99 179	*279,4*	39 135	*817,9*	2 884 925
brent-10	10	=	*21,6*	211 885	*263,9*	926 587	*330,7*	5 527	$\geq 1h$	-
KP	2	$\leq$	*59,0*	847 643	*23,1*	215 465	*60,8*	520 847	*31,1*	215 465
biggsc4	4	$\leq$	*800,9*	3 824 249	*414,9*	6 038 844	*1772,5*	2 411 741	*688,6*	6 037 260
o32	5	$\leq$	*27,4*	161 549	*22,7*	120 842	*40,7*	84 549	*33,2*	111 194

TABLE 1 – Comparaison du temps CPU, en secondes et du nombre de nœuds crées, pour trouver toutes les solutions, avec l'implantation en Ibex et Absolute.

nom	# vars		ctr type	First Solution				All solutions			
	int	real		$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$		$\mathcal{B}^\sharp$		$\mathcal{O}^\sharp$	
gear4	4	2	=	*0,02*	43	*0,04*	226	*0,02*	67	*0,05*	501
st_miqp5	2	5	$\leq$	*0,67*	2 247	*1,15*	2 247	*2,64*	7 621	*3,64*	7 621
ex1263	72	20	= $\leq$	*8,75*	8544	$\geq 1h$	-	*473,93*	493 417	$\geq 1h$	-
antennes_4_3	6	2	$\leq$	*3,30*	17 625	*22,55*	40 861	*520,77*	2 959 255	*1562,34*	6 657 237

TABLE 2 – Comparaison du temps CPU, en secondes, et du nombre de nœuds crées par Absolute.

améliorations d'Absolute sur ses faiblesses identifiées (stratégie de coupe, boucle de propagation).

4.2.2 Résolution mixte discret-continu

Comme les solveurs en PPC traitent rarement des problèmes mixtes, il n'existe pas de benchmark standard. Nous avons donc rassemblé des problèmes de MinLPLib [4], une bibliothèque de problèmes mixtes d'optimisation issus de la communauté de Recherche Opérationnelle. Ces problèmes ne sont pas des CSP mais des problèmes d'optimisation, c'est-à-dire des problèmes ayant une fonction à minimiser. Nous les avons donc traduits en problèmes de satisfaction en utilisant la même méthode que celle présentée dans [3]. Nous avons remplacé chaque critère d'optimisation $\min f(x)$ par une contrainte $|f(x) - \text{best_known_value}| \leq \epsilon$. Nous comparons les résultats obtenus par Absolute au schéma de résolution proposé dans [3], utilisant le même ϵ et les mêmes problèmes, et trouvons des résultats similaires en termes de temps d'exécution (nous ne fournissons pas une comparaison détaillée, ce serait apporter de l'information dénuée de sens en raison des différences de machines). Par manque de place, les résultats obtenus avec les polyèdres n'apparaissent pas ici.

Plus intéressant, nous observons qu'Absolute peut résoudre des problèmes mixtes en un temps raisonnable et se comporte mieux avec les intervalles qu'avec les domaines relationnels. Une raison possible est que les propagations et heuristiques actuelles ne sont pas en mesure d'utiliser pleinement l'information relationnelle disponible avec les octogones ou les polyèdres. De précédents travaux [18] suggèrent qu'un opérateur de coupe soigneusement conçu est la clef pour obtenir une résolution octogonale efficace. De futurs travaux incorporeront les idées développées pour la résolution octogonale en PPC dans notre solveur. Cependant, Absolute est en mesure de faire face naturellement à des problèmes mixtes de PPC en un temps raisonnable, ouvrant la voie à de nouvelles applications de la PPC telles que le problème de restauration et réparation du réseau électrique après une catastrophe naturelle [20], ou des problèmes géométriques [1].

5 Conclusion

Dans cet article nous avons exploré certains liens entre l'IA et la PPC, et les avons utilisés pour concevoir un schéma de résolution en PPC entièrement basé sur les domaines abstraits. Les résultats obtenus avec notre prototype sont encourageants et ouvrent la voie au développement de solveurs PPC–IA permettant de traiter naturellement les problèmes mixtes. Dans de futurs travaux, nous souhaitons améliorer notre solveur en adaptant et intégrant les méthodes avancées

4. Disponible à l'adresse `http://www.gamsworld.org/minlp/minlplib.htm`.

de la littérature en PPC. Les améliorations incluent des opérateurs de coupe pour les domaines abstraits, des propagations spécialisées, et plus important encore, l'amélioration de la boucle de propagation. Nous avons conçu notre solveur sur des abstractions de façon modulaire, ainsi les nouvelles méthodes et les méthodes existantes peuvent être combinées, comme ce qui est fait pour le produit réduit en IA. Finalement, chaque problème sera résolu dans le domaine abstrait qui lui sied le mieux comme c'est la norme en IA.

Un autre développement intéressant est d'utiliser certaines méthodes de la PPC dans un analyseur statique d'IA, telles que l'utilisation d'opérateurs de coupe dans les complétions disjonctives, ainsi que la capacité de la PPC d'affiner un élément abstrait pour atteindre ou approcher la complétude.

Au final, il reste à comprendre comment les opérateurs d'élargissement très utilisés en IA pourraient être utilisés dans des solveurs de PPC.

Références

[1] Nicolas Beldiceanu, Mats Carlsson, Emmanuel Poder, Rida Sadek, and Charlotte Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In *Proc. of CP'07*, pages 180–194, 2007.

[2] Frédéric Benhamou. Heterogeneous constraint solvings. In *Proc. of ALP'96*, pages 62–76, 1996.

[3] Nicolas Berger and Laurent Granvilliers. Some interval approximation techniques for minlp. In *Proc. of SARA'09*, 2009.

[4] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA Infotech@Aerospace 2010*. American Institute of Aeronautics and Astronautics, 2010.

[5] Gilles Chabert, Luc Jaulin, and Xavier Lorca. A constraint on the number of distinct vectors with application to localization. In *Proc. of CP'09*, pages 196–210, 2009.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conf. Rec. of POPL'77*, pages 238–252, 1977.

[7] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4) :511–547, August 1992.

[8] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of POPL'78*, pages 84–96, 1978.

[9] Vijay D'Silva, Leopold Haller, and Daniel Kroening. Satisfiability solvers are static analysers. In *Proc. of SAS'12*, 2012.

[10] Philippe Granger. Improving the results of static analyses of programs by local decreasing iterations. In *Proc. of FSTTCS'92*, 1992.

[11] Aymeric Hervieu, Benoit Baudry, and Arnaud Gotlieb. Pacogen : Automatic generation of pairwise test configurations from feature models. In *Proc. of ISSRE 2011*, pages 120–129, 2011.

[12] Bertrand Jeannet and Antoine Miné. Apron : A library of numerical abstract domains for static analysis. In *Proc. of CAV 2009*, volume 5643, pages 661–667, 2009.

[13] Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A cp framework for testing cp. *Constraints*, 17(2) :123–147, 2012.

[14] Antoine Miné. *Domaines numériques abstraits faiblement relationnels*. PhD thesis, École Normale Supérieure, December 2004.

[15] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1) :31–100, 2006.

[16] Ugo Montanari. Networks of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7(2) :95–132, 1974.

[17] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A constraint solver based on abstract domains. In *Proc. of VMCAI 2013*, 2013.

[18] Marie Pelleau, Charlotte Truchet, and Frédéric Benhamou. Octagonal domains for continuous constraints. In *Proc. of CP'11*, volume 6876, pages 706–720, 2011.

[19] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier, 2006.

[20] Ben Simon, Carleton Coffrin, and Pascal van Hentenryck. Randomized adaptive vehicle decomposition for large-scale power restoration. In *Proc. of CPAIOR'12*, pages 379–394, 2012.

[21] Choco Team. Choco : an open source java constraint programming library. Research report 10-02-INFO, Ecole des Mines de Nantes, 2010.

[22] Aditya Thakur and Thomas Reps. A generalization of stØAlmarck's method. In *Proc. of SAS'12*, 2012.

Raffiner l'analyse par interprétation abstraite des programmes à l'aide des techniques de programmation par contraintes *

Olivier Ponsini **Claude Michel** **Michel Rueher**

Université de Nice–Sophia Antipolis, I3S/CNRS, BP 121, 06903 Sophia Antipolis, France
prénom.nom@unice.fr

Résumé

L'analyse statique des valeurs des variables par interprétation abstraite est une approche classique pour la vérification des programmes qui comportent des calculs en virgule flottante. Cependant, même les outils les plus avancés calculent une sur-approximation des valeurs des variables qui peut être très grossière. Dans cet article, nous montrons que les solveurs de contraintes peuvent raffiner substantiellement les approximations calculées par les outils basés sur l'interprétation abstraite. Nous décrivons une approche hybride qui combine interprétation abstraite et techniques issues de la programmation par contraintes dans une même analyse statique et automatique des programmes. rAiCp, le système que nous avons développé, est significativement plus précis que Fluctuat, un analyseur par interprétation abstraite de référence. De ce fait, rAiCp a pu éliminer les 11 fausses alarmes que Fluctuat génère sur un ensemble de programmes de référence.

Abstract

Abstract interpretation based value analysis is a classical approach for verifying programs with floating-point computations. However, state-of-the-art tools compute an over-approximation of the variable values that can be very coarse. In this paper, we show that constraint solvers can significantly refine the approximations computed with abstract interpretation tools. We introduce a hybrid approach that combines abstract interpretation and constraint programming techniques in a single static and automatic analysis. rAiCp, the system we developed is substantially more precise than Fluctuat, a state-of-the-art static analyser. Moreover, it could eliminate 11 false alarms generated by Fluctuat on a standard set of benchmarks.

*Ce travail a été financé en partie par les projets ANR VACSIM (ANR-11-INSE-0004), ANR AEOLUS (ANR-10-SEGI-0013), et OSEO ISI PAJERO.

1 Introduction

Des programmes manipulant des nombres à virgule flottante sont utilisés pour contrôler des systèmes physiques complexes et critiques dans des domaines aussi variés que le transport, l'énergie nucléaire ou la santé. Les calculs en virgule flottante sont une source additionnelle d'erreurs dans les programmes et de célèbres bugs informatiques sont dus à des erreurs dans les calculs avec les flottants, e.g., l'échec du missile Patriot à Dhahran. Les calculs en virgule flottante sont généralement dérivés de modèles mathématiques sur les nombres réels [14]. Cependant, arithmétiques réelle et flottante sont différentes : pour la même séquence d'opérations, les nombres à virgule flottante n'ont pas un comportement identique aux nombres réels. Par exemple, certains nombres réels décimaux ne sont pas représentables avec les nombres binaires à virgule flottante (e.g., 0.1 n'a pas de représentation exacte), les opérateurs arithmétiques ne sont pas associatifs et sont sujets aux phénomènes d'absorption (e.g., $a+b$ est arrondi à a quand a est très grand par rapport à b) ou d'annulation (soustraction de deux nombres proches après arrondi qui ne conserve que l'erreur d'arrondi).

L'analyse des valeurs des variables est souvent utilisée pour vérifier dans les programmes l'absence de certaines erreurs à l'exécution, telles que des opérations invalides sur les entiers ou les flottants, ainsi que des assertions utilisateurs simples [8]. Cette analyse peut aussi aider à estimer la précision des calculs en virgule flottante par rapport à la même séquence d'opérations avec une sémantique idéalisée sur les nombres réels. Les outils automatiques existants sont principalement basés sur des techniques d'interprétation abstraite. Par exemple, Fluctuat [10], un analyseur statique de ré-

férence, calcule une sur-approximation des domaines des variables pour un programme C considéré avec une sémantique sur les nombres réels. Il calcule simultanément une sur-approximation de l'erreur due aux opérations en virgule flottante en divers points du programme. Toutefois, ces sur-approximations peuvent être très grossières même pour des expressions et des constructions courantes des programmes. Par conséquent, de nombreuses fausses alarmes [1] (appelées aussi faux positifs) peuvent être générées.

Dans cet article, nous décrivons une approche hybride pour l'analyse des programmes avec des flottants qui combine interprétation abstraite (IA) et techniques de programmation par contraintes (PPC). Nous montrons que les solveurs de contraintes sur les nombres à virgule flottante et réels peuvent raffiner substantiellement les sur-approximations calculées par interprétation abstraite. RAICP, le système que nous avons développé, utilise à la fois FLUCTUAT et les solveurs de contraintes suivants :

- REALPAVER [17], un solveur correct pour contraintes sur les nombres réels,
- FPCS [21, 20], un solveur correct pour contraintes sur les nombres à virgule flottante.

Les expérimentations montrent que RAICP est significativement plus précis que FLUCTUAT, en particulier sur des programmes C difficiles à traiter par les techniques d'interprétation abstraite. Cela provient principalement des capacités de réfutation des algorithmes de filtrage sur les nombres réels et sur les nombres à virgule flottante utilisés dans RAICP. RAICP a aussi pu éliminer 11 fausses alarmes générées par FLUCTUAT sur un ensemble de 55 programmes proposés par D'Silva et al [12] pour l'évaluation de CDFL, un outil d'analyse de programmes qui intègre un domaine abstrait dans l'algorithme d'apprentissage de clauses dirigé par les conflits d'un solveur SAT. De plus, RAICP est en moyenne 5 fois plus rapide que CDFL sur cet ensemble de programmes.

La Section 2 illustre notre approche sur un petit exemple. Des notions de base sur les techniques et les outils que nous employons sont présentées en Section 3. La section suivante présente l'état de l'art. La Section 5 détaille notre approche et les expérimentations sont analysées en Section 6.

1. Une fausse alarme correspond au cas où la sémantique abstraite intersecte une zone interdite, i.e., des états erronés d'un programme, alors que la sémantique concrète du programme n'intersecte pas cette zone interdite. Ainsi, une erreur potentielle est reportée alors qu'elle ne peut pas se produire dans les faits (cf. `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`).

FIGURE 1 – Fonction `conditional`.

```
1 /* Pre−condition :  x ∈ [0 .. 10] */
2 double conditional(double x) {
3   double y = x*x - x;
4   if (y >= 0)
5     y = x/10;
6   else
7     y = x*x + 2;
8   return y; }
```

2 Motivation

Dans cette section, nous illustrons notre approche sur un petit exemple. Le programme en Figure 1 est mentionné dans [13] comme un programme difficile pour les analyses par interprétation abstraite. Sur les nombres à virgule flottante, ainsi que sur les nombres réels, ce programme retourne une valeur dans l'intervalle $[0\,..\,3]$. En effet, à partir de l'instruction conditionnelle de la ligne 4, nous pouvons déduire les informations suivantes :

- branche `if` : $x = 0$ ou $x \geq 1$, et ainsi $y \in [0\,..\,1]$ à la fin de cette branche ;
- branche `else` : $x \in]0\,..\,1[$, et ainsi $y \in]2\,..\,3[$ à la fin de cette branche.

Cependant, les domaines abstraits classiques (e.g., les intervalles, les polyèdres), ainsi que le domaine abstrait des *zonotopes* utilisé dans FLUCTUAT, ne parviennent pas à produire une bonne approximation du résultat de ce programme. Le meilleur intervalle obtenu avec ces abstractions est $[0\,..\,102]$, à la fois sur les réels et sur les flottants. La difficulté pour ces analyses est d'intersecter les domaines abstraits calculés pour y aux lignes 3 et 4. En effet, elles sont incapables de déduire de ces instructions une contrainte sur x. Par conséquent, dans la branche `else`, x est toujours dans l'intervalle $[0\,..\,10]$ pour ces analyses.

Dans notre approche, nous proposons de calculer une approximation des domaines des variables dans les deux chemins d'exécutions de ce programme. Sur cet exemple, les techniques de filtrage de la PPC sont suffisamment fortes pour réduire les domaines. Considérons par exemple le système de contraintes sur les nombres réels $\{y_0 = x_0 * x_0 - x_0, y_0 < 0, y_1 = x_0 * x_0 + 2, x_0 \in [0\,..\,10]\}$ qui correspond au chemin d'exécution [2] passant par la branche `else` de la fonction. A partir des contraintes $y_0 = x_0 * x_0 - x_0$ et $y_0 < 0$, le solveur d'intervalle sur les nombres réels que nous utilisons peut réduire le domaine initial de

2. Les instructions sont mises en forme DSA (Dynamic Single Assignment) dans laquelle chaque variable n'est affectée qu'une seule fois par chemin dans le programme [2].

TABLE 1 – Domaine résultat du programme `conditional`.

	Domaine	Temps
Exact (réel et flottant)	$[0\,..\,3]$	–
FLUCTUAT (réel et flottant)	$[0\,..\,102]$	0.1 s
FPCS (flottant)	$[0\,..\,3{,}027]$	0.2 s
REALPAVER (réel)	$[0\,..\,3{,}001]$	0.3 s

x_0 à $[0\,..\,1]$. Ce domaine réduit est ensuite utilisé pour calculé celui de y_1 *via* la contrainte $y_1 = x_0 * x_0 + 2$, ce qui donne $y_1 \in [2\,..\,3{,}001]$. De la même façon, notre solveur de contraintes sur les nombres à virgule flottante réduit les domaines de x_0 à $[4{,}94 \cdot 10^{-324}\,..\,1{,}026]$ et y_1 à $[2\,..\,3{,}027]$.

Pour résumer, nous explorons le graphe de flot de contrôle (GFC) d'un programme en stoppant à chaque jonction de deux branches. Nous construisons alors un système de contraintes pour chacune des deux branches. Nous appliquons ensuite des techniques de filtrage sur ces systèmes afin de réduire les domaines des variables préalablement calculés par FLUCTUAT à ce point de jonction. L'exploration du GFC continue alors avec les domaines réduits. Cette exploration est réalisée à la volée : des branches sont coupées aussitôt qu'une inconsistance du système de contraintes associé est détectée par un algorithme de filtrage local. La Table 1 rassemble les résultats obtenus par les différentes techniques sur l'exemple du programme `conditional`. Sur cet exemple, contrairement à FLUCTUAT, notre approche calcule de très bonnes approximations avec des temps d'analyse similaires. Dans [13], les auteurs proposent une extension des zonotopes, les *zonotopes contraints*, qui améliore le traitement des instructions conditionnelles. Cette extensions est définie sur les nombres réels et n'est pas encore disponible dans FLUCTUAT. L'approximation calculée avec les *zonotopes contraints* est meilleure que celle de FLUCTUAT (la borne supérieure est réduite à 9.72) mais elle reste moins précise que celle calculée avec REALPAVER.

3 Notions de base

Nous rappelons ici quelques notions de base sur l'interprétation abstraite et FLUCTUAT, ainsi que sur les solveurs de contraintes REALPAVER et FPCS utilisés dans notre implémentation.

L'interprétation abstraite [3] repose sur une sémantique abstraite des programmes qui est un surensemble de la sémantique concrète. Puisque la sémantique abstraite couvre tous les comportements possibles dans la sémantique concrète, si les comportements possibles dans la sémantique abstraite ne conduisent pas à des erreurs, il en va alors de même pour les comportements dans la sémantique concrète.

Fluctuat est un analyseur statique de programmes C spécialisé dans l'estimation de la précision des calculs en virgule flottante [4] [10]. FLUCTUAT compare le comportement sur les nombres réels au comportement sur les nombres à virgule flottante du programme analysé. L'outil permet de spécifier des intervalles de valeurs pour les variables d'entrée du programme et il calcule pour chaque variable v :

- les bornes du domaine de v considérée comme un nombre réel ;
- les bornes du domaine de v considérée comme un nombre à virgule flottante ;
- les bornes de l'erreur maximale entre les valeurs sur les réels et sur les flottants associées à v ;
- la contribution de chaque instruction à l'erreur associée à v ;
- la contribution des variables d'entrée à l'erreur associée à v.

FLUCTUAT procède par interprétation abstraite. Il utilise le domaine faiblement relationnel des zonotopes [15]. Les zonotopes sont des ensembles de formes affines qui préservent les corrélations linéaires entre variables. Ils offrent un bon compromis entre performance et précision pour les calculs sur les nombres réels et à virgule flottante. En effet, l'analyse est rapide et passe bien à l'échelle, elle traite avec précision les expressions linéaires et elle détermine les instructions impliquées dans la perte de précision des calculs en virgule flottante. Afin d'augmenter la précision de l'analyse, FLUCTUAT permet d'utiliser des nombres en précision arbitraire et de subdiviser les domaines d'au plus deux variables d'entrée. Cependant, les surapproximations calculées par FLUCTUAT peuvent être très grandes car le domaine abstrait n'est pas précis sur les instructions conditionnelles et les expressions non-linéaires.

RealPaver est un solveur sur intervalles pour des systèmes de contraintes numériques sur les nombres réels [5] [17]. Les contraintes peuvent être non-linéaires et contenir les opérations arithmétiques usuelles et les fonctions transcendantes élémentaires.

3. Voir `http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html` pour une introduction informelle.

4. FLUCTUAT est développé au CEA-LIST (`http://www-list.cea.fr/validation_en.html`) et a été utilisé avec succès pour des applications industrielles de plusieurs dizaines de milliers de lignes de code dans le domaine de l'aviation notamment.

5. `http://pagesperso.lina.univ-nantes.fr/info/perso/permanents/granvil/realpaver/`

RealPaver calcule des approximations fiables d'ensembles continus de solutions en utilisant des méthodes issues de l'arithmétique des intervalles correctement arrondies et des techniques de satisfaction de contraintes. Plus précisément, les domaines calculés sont des intervalles fermés dont les bornes sont des flottants. RealPaver implémente plusieurs consistances partielles : box, hull et $3B$. L'approximation d'une solution est décrite par une boîte, i.e., le produit cartésien des domaines des variables. Soit RealPaver prouve qu'un système de contraintes est inconsistant, soit il calcule de petites boîtes susceptibles de contenir des solutions et dont l'ensemble contient toutes les solutions du système

Le langage de modélisation de RealPaver ne fournit pas les opérateurs d'inégalités strictes, qui existent dans les expressions conditionnelles des programmes. Par conséquent, dans les systèmes de contraintes que nous générons pour RealPaver, les inégalités strictes sont remplacées par des non-strictes et les contraintes avec l'opérateur de différence $\neq$ sont ignorées. Ceci peut conduire à des sur-approximations, mais reste sans perte de solution.

FPCS est un solveur de contraintes conçu pour résoudre un ensemble de contraintes sur les flottants sans perdre de solutions [21, 20]. Il utilise une $2B$-consistance [19] basée sur des fonctions de projections adaptées aux flottants [22, 4]. La principale difficulté réside dans le calcul de fonctions de projections inverses conservatives de toutes les solutions. En effet, les projections directes ne nécessitent qu'une légère adaptation des résultats classiques de l'arithmétique des intervalles, mais les projections inverses ne suivent pas les mêmes règles en raison des propriétés de l'arithmétique flottante. Plus précisément, chaque contraintes est décomposée en un ensemble de contraintes binaires ou ternaires équivalent en introduisant de nouvelles variables si nécessaire. Une contrainte ternaire $x = y \odot_f z$, où $\odot_f$ est un opérateur arithmétique sur les flottants, est décomposée en trois fonctions de projection :

- la projection directe, $\Pi_x(x = y \odot_f z)$;
- la première projection inverse, $\Pi_y(x = y \odot_f z)$;
- la seconde projection inverse, $\Pi_z(x = y \odot_f z)$.

Une contrainte binaire de la forme $x \odot_f y$, où $\odot_f$ est un opérateur relationnel (parmi `==`, `!=`, `<`, `<=`, `>`, and `>=`), est décomposée en deux fonctions de projection : $\Pi_x(x \odot_f y)$ et $\Pi_y(x \odot_f y)$. Le calcul de l'approximation de ces fonctions de projection est principalement inspiré de l'arithmétique des intervalles et bénéficie du fait que les flottants sont un ensemble fini totalement ordonné.

FPCS implémente aussi des consistances plus fortes, comme les kB -consistances [19], pour traiter les classiques problèmes d'occurrences multiples, mais aussi pour réduire plus substantiellement les bornes des domaines des variables.

Les domaines des flottants manipulés par FPCS incluent aussi les infinis. De plus, le solveur FPCS traite les opérations arithmétiques de base ainsi que la plupart des fonctions mathématiques usuelles. Les conversions de type sont aussi correctement gérées.

4 État de l'art

Différentes méthodes ont été appliquées à la validation statique des programmes comportant des calculs en virgule flottante : les analyses par interprétation abstraite, les preuves de programmes avec des assistants de preuve ou avec des procédures de décision dans des solveurs automatiques.

Les analyses par interprétation abstraite intègrent les erreurs d'arrondi des calculs en virgule flottante dans leurs domaines abstraits. Elles sont en général rapides, automatiques et passent bien à l'échelle. En revanche, elles peuvent manquer de précision et ne sont pas adaptées à la génération automatique de contre-exemples, c.-à-d., des valeurs des variables d'entrée qui violent une assertion dans le programme. Astrée [8] est sans doute l'un des outils les plus connus de cette famille de méthodes. En estimant la valeur des variables en tout point d'un programme, l'outil peut prouver l'absence d'erreur d'exécution, au sens de comportement non défini par le langage de programmation (e.g., division par zéro, débordement de capacité). Fluctuat calcule en plus une estimation de l'exactitude des calculs en virgule flottante, c.-à-d., une borne de la différence entre les valeurs des variables lorsqu'on donne au programme une sémantique sur les réels et lorsqu'on lui donne une sémantique sur les flottants [10].

Les assistants de preuve tels que Coq [3] ou HOL [18] permettent à l'utilisateur de formaliser l'arithmétique flottante. Les preuves des propriétés des programmes sont alors effectuées interactivement dans l'assistant, qui garantit la correction des preuves. Bien que des parties de la preuve puissent être automatisées, ces outils nécessitent généralement beaucoup d'interaction avec l'utilisateur. De plus, lorsqu'une stratégie de preuve échoue, l'utilisateur ne sait pas si la propriété est fausse ou si une autre stratégie pourrait la prouver. Comme pour l'interprétation abstraite, les assistants de preuve ne fournissent généralement pas automatiquement de contre-exemple. L'outil Gappa [9] combine le calcul d'intervalle et l'application de théorèmes prédéfinis par réécriture. Les théorèmes permettent

de réécrire les expressions arithmétiques afin de compenser certaines faiblesses du calcul d'intervalle, e.g., perte des dépendances entre variables. Lorsque les intervalles calculés ne sont pas assez précis, il est possible d'ajouter manuellement des théorèmes ou de subdiviser les domaines d'entrée. Le coût de cette méthode semi-automatique est alors important. Dans [1], les auteurs proposent une axiomatisation de l'arithmétique flottante dans la logique du premier ordre qui permet d'automatiser les preuves dans un assistant tel que Coq en appelant des solveurs SMT (Satisfiability Modulo Theories) et Gappa. Leurs expérimentations montrent qu'une interaction humaine avec l'assistant de preuve demeure nécessaire.

Du fait de la taille du domaine des variables flottantes et du coût des opérations sur les vecteurs de bits, l'approche classique par vecteurs de bits des solveurs SAT est inopérante. Une technique d'abstraction a été proposée pour CBMC [5]. Elle est basée sur la sous et la sur-approximation d'un flottant selon une précision exprimée en nombre de bits de la mantisse. Cette technique reste cependant lente. D'Silva et al [12] ont récemment développé CDFL, un outil d'analyse de programmes qui intègre un domaine abstrait dans l'algorithme d'apprentissage de clauses dirigé par les conflits d'un solveur SAT. CDFL dispose d'une analyse correcte et complète pour déterminer les intervalles de valeurs des variables flottantes dans les programmes. Dans [12], les auteurs affirment que CDFL est plus de 200 fois plus rapide que CBMC. En Section 6, nous comparons les performances de CDFL et de rAiCp sur un ensemble de programmes proposé par D'Silva et al.

Les liens entre interprétation abstraite et programmation logique et par contraintes ont été étudiés au niveau théorique (e.g., [6]) et de récents travaux s'intéressent à l'utilisation de l'interprétation abstraite et des domaines abstraits dans le contexte de la programmation par contraintes. Dans [11], les auteurs introduisent une nouvelle contrainte globale pour modéliser les relations arithmétiques itératives entre variables entières. L'algorithme de filtrage associé est basé sur l'interprétation abstraite avec des polyèdres. Dans [23], les auteurs proposent d'utiliser le domaine abstrait des octogones, qui s'est révélé efficace en interprétation abstraite, pour représenter les domaines des variables dans les problèmes de satisfaction de contraintes sur le continu. Ils généralisent alors une consistance locale et le découpage des domaines à leur représentation octogonale. Dans cet article, nous montrons comment interprétation abstraite et techniques de programmation par contraintes peuvent se compléter dans l'analyse statique de programmes avec des flottants.

5 rAiCp, une approche hybride

L'approche que nous proposons est basée sur une succession d'étapes d'exploration du graphe de flot de contrôle et de fusion. Plus en détail, nous appelons Fluctuat pour calculer une première approximation des domaines des variables à chaque nœud du GFC où deux branches se rejoignent. Ensuite, nous construisons un système de contraintes par branche et nous utilisons des techniques de filtrage pour réduire les domaines des variables calculés par Fluctuat. Les domaines réduits obtenus pour chaque branche sont fusionnés et l'exploration du GFC reprend avec le résultat de la fusion.

5.1 Exploration du graphe de flot de contrôle

Le GFC d'un programme est exploré par une analyse en avant allant du début vers la fin du programme. Les instructions sont mises en forme DSA (Dynamic Single Assignment) dans laquelle chaque variable n'est affectée qu'une seule fois dans un chemin donné du programme [2]. La longueur des chemins est bornée car les boucles sont dépliées un nombre fini de fois, au delà duquel les boucles sont abstraites par les domaines calculés par interprétation abstraite. En tout point d'un chemin d'exécution, les états possibles du programme sont représentés par un système de contraintes sur les variables du programme. Les domaines des variables sont des intervalles sur les nombres réels dans le système de contraintes géré par RealPaver ; les domaines sont des intervalles sur les flottants qui correspondent aux types machine `int`, `float` et `double` du langage C [6] dans le système de contraintes géré par FPCS. Chaque instruction du programme ajoute de nouvelles contraintes et variables dans ces systèmes de contraintes. Cette technique de représentation des programmes par des systèmes de contraintes a été introduite pour la vérification bornée des programmes dans CPBPV [7]. L'implémentation de l'approche proposée dans cet article repose sur des bibliothèques développées pour CPBPV.

L'exploration du GFC est réalisée à la volée et l'exploration des branches inatteignables est stoppée dès qu'une inconsistance est détectée lors de la construction des systèmes de contraintes. Nous collectons les contraintes entre deux nœuds de jonction du GFC. Si, pour tout chemin exécutable entre ces nœuds, les systèmes de contraintes sont inconsistants pour un in-

6. Il convient de noter que le comportement des programmes contenant des calculs en virgule flottante est susceptible de varier selon le langage de programmation ou le compilateur, mais aussi, selon le système d'exploitation et l'architecture matérielle. Nous considérons ici des programmes C, compilés avec GCC sans option d'optimisation et destinés à s'exécuter sur une architecture x86 dans un environnement Linux 32 bits.

tervalle I d'une variable de sortie x du programme, alors nous pouvons retirer l'intervalle I du domaine de x. Nous différencions les variables d'*entrée* du programme, dont les domaines ne peuvent pas être réduits, des variables de *sortie*, dont les domaines dépendent des domaines des variables d'entrée et des calculs dans le programme, et qui peuvent donc être réduits.

La fusion des états du programme à chaque nœud de jonction permet non seulement une coopération étroite entre FLUCTUAT et les solveurs de contraintes, mais limite aussi le nombre de chemins exécutables à explorer.

5.2 Techniques de filtrage

Dans RAICP, nous utilisons les techniques de filtrage des contraintes pour deux objectifs différents :

- l'élimination des branches inatteignables lors de l'exploration du GFC ;
- la réduction des domaines des variables aux nœuds de jonction du GFC.

Pour les systèmes de contraintes sur les flottants, nous effectuons un filtrage par $3B(w)$-consistance avec FPCS ; pour les systèmes de contraintes sur les réels, nous effectuons un filtrage par $BC5$-consistance en mode pavage avec REALPAVER [7].

6 Expérimentations

Dans cette section, nous comparons en détail FLUCTUAT et RAICP sur des programmes représentatifs des limitations de FLUCTUAT. Nous comparons aussi RAICP à CDFL, un outil représentatif de l'état de l'art, sur des programmes fournis par les auteurs de cet outil.

Tous les résultats ont été obtenus sur un Intel Core 2 Duo à 2.8 GHz avec 4 Go de mémoire exécutant Linux en utilisant FLUCTUAT version 3.8.73, REALPAVER version 0.4 et la version disponible au téléchargement de CDFL. Tous les programmes de test sont disponibles sur la page `http://users.polytech.unice.fr/~rueher/Benchs/RAICP`.

6.1 Améliorations par rapport à Fluctuat

Nous montrons ici que notre approche améliore les approximations calculées par FLUCTUAT sur des programmes comportant des conditionnelles, des non-linéarités et des boucles.

7. La $BC5$-consistance est une combinaison de la méthode de Newton pour les intervalles, de hull et de box.

6.1.1 Conditionnelles :

Ce premier programme de test concerne les instructions conditionnelles, pour lesquelles les domaines abstraits de FLUCTUAT doivent être intersectés la conditions des instructions. La fonction `gsl_poly_solve_quadratic` provient de la bibliothèque scientifique GNU et contient de nombreuses instructions conditionnelles. Elle calcule les racines réelles de l'équation du second degré $ax^2 + bx + c$ et met les résultats dans les variables `x0` et `x1`.

La Table 2 montre les temps d'analyse et les approximations des domaines des variables `x0` et `x1` pour deux configurations des variables d'entrée. Les deux premières lignes présentent les résultats de FLUCTUAT et RAICP (avec REALPAVER) sur les nombres réels. Les deux lignes suivantes présentent les résultats de FLUCTUAT et RAICP (avec FPCS) sur les flottants.

Dans la première configuration, la sur-approximation de FLUCTUAT est si grande qu'elle ne donne aucune information sur le domaine des racines, alors que RAICP réduit significativement ces domaines à la fois sur $\mathbb{R}$ et sur $\mathbb{F}$. Néanmoins, l'intersection des domaines abstraits n'a pas toujours un impact aussi important sur les bornes de tous les domaines comme illustré par le domaine sur $\mathbb{F}$ de `x0` dans la seconde configuration.

Pour augmenter la précision de l'analyse, FLUCTUAT permet de subdiviser les domaines d'au plus deux variables d'entrée en un nombre donné de sous-domaines. Des analyses sont alors lancées sur chaque combinaison de sous-domaines et les résultats sont fusionnés. Déterminer le nombre approprié de subdivisions est difficile : subdiviser peut ne pas améliorer la précision de l'analyse, en revanche cela augmente toujours la durée d'analyse. La Table 3 rapporte les résultats sur $\mathbb{F}$ avec 50 subdivisions quand un seul domaine est subdivisé, et avec 30 subdivisions quand deux domaines sont subdivisés. Sur $\mathbb{R}$, dans la première configuration, les subdivisions n'apportent aucune amélioration, et dans la seconde configuration, les résultats sont identiques à ceux sur $\mathbb{F}$.

Subdiviser les domaines peut être très coûteux pour peu de gains en précision :

- Dans la première configuration, les subdivisions du domaine de la variable a conduisent à une réduction significative du domaine de `x0`. Aucune combinaison de subdivisions n'a permis de réduire le domaine de `x1`.
- Dans la seconde configuration, la meilleure réduction du domaine de `x1` est obtenu en subdivisant les domaines de a et de b. Le gain reste cependant relativement faible et aucune combinaison de subdivisions n'a permis de réduire le domaine de `x0`.

RAICP s'avère plus efficace : il améliore souvent la pré-

TABLE 2 – Domaines des racines de la fonction `gsl_poly_solve_quadratic`.

		conf. #1 : $a \in [-1\,..\,1]$ $b \in [0{,}5\,..\,1]$ $c \in [0\,..\,2]$			conf. #2 : $a, b, c \in [1\,..\,1 \cdot 10^6]$		
		`x0`	`x1`	Temps	`x0`	`x1`	Temps
$\mathbb{R}$	FLUCTUAT	$[-\infty\,..\,\infty]$	$[-\infty\,..\,\infty]$	0.14 s	$[-2 \cdot 10^6\,..\,0]$	$[-1 \cdot 10^6\,..\,0]$	0.14 s
	RAICP	$[-\infty\,..\,0]$	$[-8{,}006\,..\,\infty]$	1.55 s	$[-1 \cdot 10^6\,..\,0]$	$[-5{,}186 \cdot 10^5\,..\,0]$	0.58 s
$\mathbb{F}$	FLUCTUAT	$[-\infty\,..\,\infty]$	$[-\infty\,..\,\infty]$	0.13 s	$[-2 \cdot 10^6\,..\,0]$	$[-1 \cdot 10^6\,..\,0]$	0.13 s
	RAICP	$[-\infty\,..\,0]$	$[-8{,}125\,..\,\infty]$	0.39 s	$[-1 \cdot 10^6\,..\,0]$	$[-3\,906{,}26\,..\,0]$	0.39 s

TABLE 3 – Domaines sur $\mathbb{F}$ pour la fonction `gsl_poly_solve_quadratic` avec les domaines d'entrée subdivisés.

	conf. #1		conf. #2	
	`x0`	Temps	`x1`	Temps
FLUCTUAT a subdivisée	$[-\infty\,..\,-0]$	> 1 s	$[-1 \cdot 10^6\,..\,0]$	> 1 s
FLUCTUAT b subdivisée	$[-\infty\,..\,\infty]$	> 1 s	$[-5 \cdot 10^5\,..\,0]$	> 1 s
FLUCTUAT c subdivisée	$[-\infty\,..\,\infty]$	> 1 s	$[-1 \cdot 10^6\,..\,0]$	> 1 s
FLUCTUAT a & b subdivisées	$[-\infty\,..\,-0]$	> 10 s	$[-1{,}834 \cdot 10^5\,..\,0]$	> 10 s
FLUCTUAT a & c subdivisées	$[-\infty\,..\,-0]$	> 10 s	$[-1 \cdot 10^6\,..\,0]$	> 10 s
FLUCTUAT b & c subdivisées	$[-\infty\,..\,\infty]$	> 10 s	$[-5 \cdot 10^5\,..\,0]$	> 10 s

cision des approximations et demande moins de temps que le processus de subdivisions de FLUCTUAT. De plus, RAICP pourrait aussi tirer bénéfice de la technique des subdivisions.

6.1.2 Non-linéarités :

Le domaine abstrait utilisé par FLUCTUAT est basé sur des formes affines qui ne permettent pas une représentation exacte des opérations non-linéaires : l'image d'un zonotope par une fonction non-linéaire n'est pas un zonotope en général. Les opérations non-linéaires sont alors sur-approximées. FPCS gère mieux les expressions non-linéaires. Ceci est illustré avec le développement de Taylor d'ordre 7 de la fonction sinus (cf. Table 4, colonne `sinus`).

FPCS et REALPAVER utilisent aussi des approximations pour traiter les termes non-linéaires, et ne sont donc pas toujours plus précis que FLUCTUAT. La seconde ligne de la Table 4 montre que RAICP n'a pas pu réduire le domaine calculé par FLUCTUAT pour le programme du polynôme de `rump` [24], un polynôme très particulier conçu pour mettre en valeur le phénomène d'annulation catastrophique.

6.1.3 Boucles :

FLUCTUAT déplie les boucles un nombre borné de fois [8] avant d'appliquer l'opérateur d'élargissement de l'interprétation abstraite. L'opérateur d'élargissement trouve un point fixe pour la boucle sans la déplier complètement. Dans RAICP, l'utilisateur définit aussi le nombre de dépliages des boucles, au-delà duquel la boucle est abstraite par l'invariant calculé par interprétation abstraite. FLUCTUAT pourrait aussi être utilisé pour estimer une borne supérieure du nombre de dépliages nécessaires [16].

`sqrt` est un programme qui calcule, selon la méthode dite babylonienne, une valeur approchée, à $1 \cdot 10^{-2}$ près, de la racine carrée d'un nombre supérieur à 4. Pour l'analyse de ce programme avec deux domaines d'entrée différents (cf. Table 5), dix dépliages sont suffisants pour toujours terminer la boucle. FLUCTUAT et RAICP obtiennent tous deux des résultats précis sur $\mathbb{R}$. Sur $\mathbb{F}$, dans la seconde configuration, RAICP réduit le domaine à $[2{,}232\,..\,3{,}168]$ alors que FLUCTUAT ne parvient à aucune réduction.

Le programme `bigLoop` contient des expressions non-linéaires suivies d'une boucle qui itère un million de fois. Pour de tels programmes, il n'est pas réaliste de complètement déplier les boucles. FLUCTUAT ne parvient pas à analyser précisément la boucle de ce programme à cause des sur-approximations faites sur les expressions non-linéaires. RAICP raffine significativement les sur-approximations calculées par FLUCTUAT, même sans aucun dépliage initial. Cet exemple montre qu'une coopération étroite entre les techniques de l'IA et de la PPC peut s'avérer très efficace.

8. 10 fois par défaut.

TABLE 4 – Domaines du résultat des fonctions `sinus` et `rump`.

		`sinus` $x \in [-1\,..\,1]$		`rump` $x \in [7 \cdot 10^4\,..\,8 \cdot 10^4]$ $y \in [3 \cdot 10^4\,..\,4 \cdot 10^4]$	
		Domaine	Temps	Domaine	Temps
$\mathbb{R}$	FLUCTUAT	$[-1{,}009\,..\,1{,}009]$	0.12 s	$[-1{,}168 \cdot 10^{37}\,..\,1{,}992 \cdot 10^{37}]$	0.13 s
	RAICP	$[-0{,}842\,..\,0{,}843]$	0.34 s	$[-1{,}144 \cdot 10^{36}\,..\,1{,}606 \cdot 10^{37}]$	1.26 s
$\mathbb{F}$	FLUCTUAT	$[-1{,}009\,..\,1{,}009]$	0.12 s	$[-1{,}168 \cdot 10^{37}\,..\,1{,}992 \cdot 10^{37}]$	0.13 s
	RAICP	$[-0{,}853\,..\,0{,}852]$	0.22 s	$[-1{,}168 \cdot 10^{37}\,..\,1{,}992 \cdot 10^{37}]$	0.22 s

TABLE 5 – Domaines du résultat des fonctions `sqrt` et `bigLoop`.

		sqrt #1 : $x \in [4{,}5\,..\,5{,}5]$		sqrt #2 : $x \in [5\,..\,10]$		bigLoop	
		Domaine	Temps	Domaine	Temps	Domaine	Temps
$\mathbb{R}$	FLUCTUAT	$[2{,}116\,..\,2{,}354]$	0.13 s	$[2.098\,..\,3.435]$	0.2 s	$[-\infty\,..\,\infty]$	0.15 s
	RAICP	$[2{,}121\,..\,2{,}346]$	0.35 s	$[2{,}232\,..\,3{,}165]$	0.57 s	$[0\,..\,10]$	0.8 s
$\mathbb{F}$	FLUCTUAT	$[2{,}116\,..\,2{,}354]$	0.13 s	$[-\infty\,..\,\infty]$	0.2 s	$[-\infty\,..\,\infty]$	0.15 s
	RAICP	$[2{,}121\,..\,2{,}347]$	0.81 s	$[2{,}232\,..\,3{,}168]$	1.59 s	$[0\,..\,10]$	0.7 s

6.1.4 Contributions de l'IA et de la PPC

FLUCTUAT produit souvent une première approximation qui est assez fine pour permettre un filtrage efficace avec des consistances locales. Bien que des réductions de domaine identiques puissent parfois être obtenues sans partir des approximations calculées par FLUCTUAT (i.e., en prenant comme domaine de départ $[-\infty\,..\,\infty]$), nos expérimentations montrent qu'en général notre approche bénéficie de l'approximation calculée par FLUCTUAT.

Le filtrage par $3B$-consistance donne de bons résultats avec FPCS. La $2B$-consistance n'est pas assez forte pour réduire les domaines calculés par FLUCTUAT alors qu'une kB-consistance plus forte est trop coûteuse en temps. Nous avons aussi essayé les différentes consistances disponibles dans REALPAVER : $BC5$, une combinaison de hull et de box avec la méthode de Newton pour les intervalles, $HC4$ et $3B$. La $3B$-consistance était en général trop lente. La $BC5$-consistance s'est révélée être le meilleur compromis entre temps et réduction des domaines.

6.2 Comparaison avec CDFL

CDFL [12] est un outil d'analyse des programmes critiques conçu pour prouver l'absence d'erreurs à l'exécution. Dans [12], les auteurs montrent que CDFL est bien plus efficace que CBMC et beaucoup plus précis que ASTRÉE [8] pour déterminer le domaine des variables flottantes dans divers programmes.

Nous comparons dans cette section RAICP et CDFL sur l'ensemble de programmes[9] utilisé dans [12]. L'ensemble est constitué de 57 tests formés à partir de 12 programmes en faisant varier les domaines des variables d'entrée, les bornes des boucles et les constantes dans les propriétés à vérifier. Nous avons écarté deux tests relatifs aux calculs sur les entiers qui ne sont pas l'objet de cet article. Tous les programmes proviennent d'algorithmes numériques académiques, à l'exception de `Sac` qui est généré à partir du modèle Simulink d'un contrôleur. Les propriétés de programme sont de simples assertions sur les domaines des variables.

La Table 6 fournit les temps d'exécution de RAICP, FLUCTUAT et CDFL. Pour ces expérimentations, RAICP n'a pas été exécuté avec REALPAVER puisque les propriétés et les programmes sont définis sur les flottants.

Les trois systèmes sont susceptibles d'émettre des fausses alarmes : i.e., ils peuvent répondre qu'une propriété est fausse alors qu'elle ne l'est pas. Dans les faits, RAICP et CDFL ont correctement analysé les 33 propriétés vraies. FLUCTUAT a émis 11 fausses alarmes (elles sont notées avec une * dans les colonnes de la Table 6). Le raffinement des domaines opéré par RAICP a ainsi éliminé avec succès les fausses alarmes produites par FLUCTUAT.

En moyenne, RAICP est 5 fois plus rapide que CDFL pour la même précision. Pour certains programmes, le gain de rapidité atteint même un facteur de 25. En moyenne, RAICP est 2,2 fois plus lent que FLUCTUAT utilisé seul mais ceci est compensé par le gain en précision.

7 Conclusion

Dans cet article, nous avons introduit une nouvelle approche pour calculer des intervalles resserrés des domaines des variables flottantes dans les programmes C. RAICP, le prototype que nous avons développé, s'ap-

9. Ces programmes sont disponibles sur `http://www.cprover.org/cdfpl`

TABLE 6 – Temps d'exécution (s) de CDFL, FLUCTUAT et RAICP.

	CDFL	FLUCTUAT	RAICP		CDFL	FLUCTUAT	RAICP
newton.1.1	0.5	0.12	0.26	eps_line1	0.12	0.11	0.28
newton.1.2	1.64	0.13	0.4	muller	0.13	0.11	0.2
newton.1.3	4.6	0.21	0.76	sac.10	2.49	1.25	1.6
newton.2.1	0.95	0.11	0.27	sac.20	2.46	1.38	1.75
newton.2.2	3.44	0.14	0.43	sac.30	2.49	1.39	1.68
newton.2.3	9.32	0.21	1.12	sac.40	2.47	1.38	1.68
newton.3.1	1.95	0.12*	0.33	sac.50	2.46	1.38	1.71
newton.3.2	5.61	0.13	0.5	sac.60	2.48	1.4	1.76
newton.3.3	15.9	0.22	1.16	sac.70	2.46	1.37	1.7
newton.4.1	1.07	0.12	0.61	sac.80	2.48	1.37	1.7
newton.4.2	8.4	0.13	0.63	sac.90	2.47	1.37	1.67
newton.4.3	23.63	0.22	1.18	sine.1	0.68	0.12	0.31
newton.5.1	1.76	0.12	0.64	sine.2	0.96	0.11	0.28
newton.5.2	14.61	0.13*	1.02	sine.3	0.5	0.11	0.28
newton.5.3	38.19	0.23*	1.53	sine.4	7.89	0.12*	0.3
newton.6.1	1.28	0.12	0.83	sine.5	0.68	0.12*	0.23
newton.6.2	2.33	0.13	2.43	sine.6	0.3	0.12*	0.26
newton.6.3	3.59	0.15	1.75	sine.7	0.13	0.12*	0.22
newton.7.1	1.8	0.12	0.74	sine.8	0.08	0.12	0.23
newton.7.2	1.57	0.14	0.82	square.1	0.16	0.12	0.26
newton.7.3	19.45	0.15	0.84	square.2	0.32	0.12	0.25
newton.8.1	0.41	0.11	0.45	square.3	0.7	0.11	0.25
newton.8.2	1.67	0.12	0.56	square.4	1.05	0.12*	0.22
newton.8.3	7.49	0.12	0.71	square.5	0.68	0.12*	0.22
GC4	0.04	0.14	0.23	square.6	0.55	0.11*	0.23
Poly	0.16	0.11	0.23	square.7	0.36	0.12*	0.23
Rump	0.02	0.11	0.21	square.8	0.06	0.12	0.21
Sterbenz	0	0.12	0.2	**Total**	**208.99**	**18.37**	**40.55**

puie sur l'analyseur statique FLUCTUAT et sur FPCS et REALPAVER, deux solveurs de contraintes qui sont respectivement corrects sur les flottants et sur les réels. Ainsi, RAICP peut exploiter les capacités de réfutation des consistances locales pour raffiner les domaines calculés par FLUCTUAT.

Nous avons montré que RAICP est rapide et efficace sur des programmes représentatifs des limitations de FLUCTUAT (instructions conditionnelles et non-linéarités). Les expérimentations sur un autre ensemble de programmes de référence ont aussi montré que RAICP pouvait être aussi précis et plus rapide que CDFL, un outil avancé de vérification d'assertions pour les programmes comportant des flottants.

Notre approche de l'intégration de l'interprétation abstraite avec la programmation par contraintes fonctionne bien car souvent la première approximation des bornes des domaines calculée par interprétation abstraite est suffisamment petite pour permettre un filtrage efficace avec des consistances locales. Dans le cas de FLUCTUAT, des ensembles de formes affines sont utilisées pour abstraire les expressions non-linéaires et les contraintes. Ces ensembles constituent une meilleure approximation des systèmes de contraintes linéaires que les boîtes utilisées dans les solveurs de contraintes sur intervalles. Toutefois, ils sont moins bien adaptés aux systèmes de contraintes non-linéaires pour lesquels les techniques de filtrage utilisées dans les solveurs de contraintes numériques offrent un cadre plus flexible et extensible.

Une extension de ces travaux consisterait en une intégration plus étroite entre interprétation abstraite et solveurs de contraintes, par exemple, au niveau du domaine abstrait de FLUCTUAT directement plutôt qu'au niveau des intervalles des domaines des variables.

Remerciements

Les auteurs remercient Sylvie Putot, Éric Goubault et Franck Védrine pour leurs conseils et leur aide dans l'utilisation de FLUCTUAT.

Références

[1] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In *IJCAR*, volume 6173 of *LNCS*, pages 127–141. Springer, 2010.

[2] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *Information Processing Letters*, 93(6) :281–288, 2005.

[3] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In *18th IEEE Symposium on Computer Arithmetic*, pages 187–194. IEEE, 2007.

[4] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2) :97–121, 2006.

[5] Angelo Brillout, Daniel Kroening, and Thomas Wahl. Mixed abstractions for floating-point arithmetic. In *9th International Conference on Formal Methods in Computer-Aided Design*, pages 69–76. IEEE, 2009.

[6] Philippe Codognet and Gilberto Filé. Computations, abstractions and constraints in logic programs. In *International Conference on Computer Languages (ICCL'92)*, pages 155–164. IEEE, 1992.

[7] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. A constraint-programming framework for bounded program verification. *Constraints Journal*, 15(2) :238–264, 2010.

[8] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers : A comparison with ASTRÉE. In *1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 3–20. IEEE, 2007.

[9] Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Transactions on Computers*, 60(2) :242–253, 2011.

[10] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *FMICS*, volume 5825 of *LNCS*, pages 53–69. Springer, 2009.

[11] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. An abstract interpretation based combinator for modeling while loops in constraint programming. In *Principles and Practices of Constraint Programming (CP'07)*, volume 4741 of *LNCS*, pages 241–255. Springer Verlag, 2007.

[12] Vijay D'Silva, Leopold Haller, Daniel Kroening, and Michael Tautschnig. Numeric bounds analysis with conflict-driven learning. In *Proc. TACAS*, volume 7214 of *Lecture Notes in Computer Science*, pages 48–63. Springer, 2012.

[13] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. A logical product approach to zonotope intersection. In *CAV*, volume 6174 of *LNCS*, pages 212–226. Springer, 2010.

[14] David Goldberg. What every computer scientist should know about floating point arithmetic. *ACM Computing Surveys*, 23(1) :5–48, 1991.

[15] Eric Goubault and Sylvie Putot. Static analysis of numerical algorithms. In *SAS*, volume 4134 of *LNCS*, pages 18–34. Springer, 2006.

[16] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *VMCAI*, volume 6538 of *LNCS*, pages 232–247. Springer, 2011.

[17] Laurent Granvilliers and Frédéric Benhamou. Algorithm 852 : RealPaver : an interval solver using constraint satisfaction techniques. *ACM Transactions on Mathematical Software*, 32(1) :138–156, 2006.

[18] John Harrison. A machine-checked theory of floating-point arithmetic. In *TPHOLs*, volume 1690 of *LNCS*, pages 113–130. Springer-Verlag, 1999.

[19] Olivier Lhomme. Consistency techniques for numeric CSPs. In *13th International Joint Conference on Artificial Intelligence*, pages 232–238, 1993.

[20] Bruno Marre and Claude Michel. Improving the floating point addition and subtraction constraints. In *CP*, volume 6308 of *LNCS*, pages 360–367. Springer, 2010.

[21] Claude Michel. Exact projection functions for floating-point number constraints. In *7th International Symposium on Artificial Intelligence and Mathematics*, 2002.

[22] Claude Michel, Michel Rueher, and Yahia Lebbah. Solving constraints over floating-point numbers. In *CP*, volume 2239 of *LNCS*, pages 524–538. Springer Verlag, 2001.

[23] Marie Pelleau, Charlotte Truchet, and Frédéric Benhamou. Octagonal domains for continuous constraints. In *Principles and Practice of Constraint Programming (CP'11)*, volume 6876 of *LNCS*, pages 706–720, 2011.

[24] Siegfried M. Rump. Verification methods : Rigorous results using floating-point arithmetic. *Acta Numerica*, 19 :287–449, 2010.

Prototyper des Moteurs de Propagation avec un DSL

Charles Prud'homme[1] **Xavier Lorca**[1] **Rémi Douence**[2] **Narendra Jussien**[1]

[1] EMNantes, INRIA TASC, CNRS LINA, FR-44307 Nantes Cedex 3, France
[2] EMNantes, INRIA ASCOLA, CNRS LINA, FR-44307 Nantes Cedex 3, France
{prenom.nom}@mines-nantes.fr

Résumé

La propagation des contraintes est au cœur des solveurs de contraintes. Deux tendances co-existent : les algorithmes de propagation orientés variables et ceux orientés propagateurs. Ces deux approches assurent le même niveau de consistance, généralement la consistance d'arc, mais leur efficacité peut varier selon l'instance traitée. Il est généralement admis qu'il n'y a pas de meilleure approche et les solveurs de contraintes modernes n'en implantent finalement qu'une seule. Dans cet article, nous souhaitons aller un peu plus loin en fournissant un langage indépendant de tout solveur qui permette de prototyper des moteurs de propagation. Nous validons notre proposition avec une implémentation de référence basée sur Choco et le langage de modélisation MiniZinc. Finalement, nous évaluons la réalisabilité de notre proposition et en illustrons la flexibilité.

Abstract

Constraint propagation is at the heart of constraint solvers. Two main trends co-exist : variable-oriented propagation engines and constraint-oriented propagation engines. Those two approaches ensure the same consistency level, generally arc-consistency, but their efficiency (computation time) can be quite different depending on the instance solved. It is usually accepted that there is no best approach in general, and modern constraint solvers implement only one. In this paper, we would like to go a step further providing a solver independent language at the modeling stage to enable the design of propagation engine prototypes. We validate our proposal with a reference implementation based on Choco and the MiniZinc modeling language.

1 Introduction

La propagation des contraintes, élément central des solveurs de programmation par contraintes, a été beaucoup étudiée ces dernières années [1, 15]. Différents algorithmes existent [9] ; les principaux sont des versions modernes d'algorithmes assurant la consistance d'arc [1, 2, 3]. Pour un algorithme donné, plusieurs orientations existent. Les plus fréquemment utilisées sont celles orientées "variables" ou "contraintes". Par exemple, IBM CPO [20], Choco [5], Minion [10] et or-tools [23] reposent sur un algorithme orienté variables, alors que Gecode [21], SICStus Prolog [4] et JaCoP [22] reposent sur un algorithme orienté contraintes. Concevoir un moteur de propagation, et plus généralement un solveur, est un processus fait de choix et compromis pour faire conjuguer adaptabilité (pour résoudre un large éventail de problèmes) et efficacité (pour les résoudre efficacement). Les stratégies de propagation de contraintes sont étroitement liées à l'implémentation même d'un solveur. Implémenter un algorithme qui soit correct, efficace, respectant l'interface spécifique du solveur ainsi que son langage de programmation, peut se révéler être un vrai défi. Il est vraisemblable que seuls les développeurs du solveur est en position de modifier cet algorithme. En conséquence, le moteur de propagation tend à devenir monolithique et relativement inaccessible dès la phase de modélisation. Donc, même s'il n'est pas raisonnable de donner accès l'algorithme de propagation de contraintes aux utilisateurs novices, il peut se révéler fort utile aux modeleurs avancés ainsi qu'aux développeurs. Les premiers y verront l'opportunité d'optimiser le processus de résolution. Les seconds pourront rapidement prototyper de nouvelles stratégies de propagation, avant la phase, complexe, de développement de la-dite stratégie.

Dans cet article, nous proposons un langage indépendant de tout solveur qui permet de configurer la propagation des contraintes dès la phase de modélisation. Notre contribution est triple. Premièrement, nous

présenterons un Langage Dédié (en anglais "*Domain Specific Language*" ou DSL) à la configuration du moteur de propagation des contraintes. Nous montrerons qu'il permet d'exprimer un large éventail de stratégies déjà existantes. Deuxièmement, nous exploiterons les propriétés de base d'un DSL pour assurer à la fois la complétude et la correction du moteur de propagation produit. Troisièmement, nous présenterons une implémentation concrète de ce langage dédié dans Choco ainsi que son intégration dans MiniZinc [19]. Enfin, nous montrerons que le surcoût induit par l'utilisation du DSL est acceptable pour le prototypage de moteurs de propagation.

Dans la suite, la Section 2 rappelle les bases de la propagation de contraintes, ainsi que l'état de l'art des améliorations apportées. La Section 3 est dédiée à la description du DSL. La Section 3.1 définit l'ensemble des techniques et services requis par le solveur sous-jacent pour accepter le langage. La Section 3.2 présente et discute des détails du langage ainsi que ses garanties. La Section 3.3 présente son intégration dans MiniZinc, alors que la Section 3.4 discute des possible limites de notre approche. Enfin, la Section 4 évalue le DSL à la fois lors de la phase d'analyse syntaxique et celle de résolution, et montre combien, sur un cas d'utilisation, le prototypage des moteurs de propagation est simple.

2 Contexte

La Programmation par Contraintes est un paradigme de programmation où les relations entre les variables sont établies par les contraintes. Un *Problème de Satisfaction de Contraintes* (PSC) est défini par le triplet $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ et consiste en un ensemble $\mathcal{V}$ de n variables, leurs domaines associés $\mathcal{D}$, et une collection $\mathcal{C}$ de m contraintes.

Domaine et Variable Le domaine $d_v \in \mathcal{D}$ associé à une variable $v \in \mathcal{V}$ définit l'ensemble fini de valeurs entières auxquelles la variable v peut être affectée. L'*affectation* $v^\star$ d'une variable v à une valeur x se traduit par la réduction de son domaine d_v à un singleton, $d_v = \{x\}$.

Constrainte Une contrainte $c \in \mathcal{C}$ est définie sur un ensemble de variables $V_c \subseteq \mathcal{V}$. Une contrainte est équipée d'une fonction booléenne définie par $f_c : V_c^\star \rightarrow \{\texttt{true},\texttt{false}\}$. Une solution de c est une affection de toutes ses variables telle que $f_c(V_c^\star) = \texttt{true}$. Une contrainte est dite *satisfiable* si il existe une solution pour c. Une solution pour un PSC est une affectation de toutes les variables de $\mathcal{V}$ telle que toutes les contraintes de $\mathcal{C}$ sont satisfaites simultanément ; un PSC est satisfiable si et seulement si il a une solution.

Réduction de domaines et propagation Une contrainte est caractérisée par son comportement face à la modification d'une de ses variables. Une contrainte est équipée d'un ou plusieurs algorithmes de filtrage, nommés *propagateurs*. Un propagateur $p \in \mathcal{P}$ est une fonction définie par $p : \mathcal{D} \rightarrow \mathcal{D}$. Il retire du domaine de ses variables les valeurs qui ne sont désormais plus cohérentes. Soit $c \in \mathcal{C}$, P_c indique l'ensemble des propagateurs de c et V_p l'ensemble des variable de p tel que pour tous $v \in V_p, v \in V_c \wedge p \in P_c$; l'ensemble des propagateurs d'un PSC est dénoté $\mathcal{P}$.

Propagation de contraintes La programmation par contraintes est basée sur la propagation de contraintes [9, 17]. Les déductions locales liées à l'application de chaque propagateur forcent à reconsidérer la satisfiabilité du PSC ; ceci est assuré par l'exécution de l'algorithme de propagation [1]. L'algorithme AC5 [3] est une variante d'un tel algorithme basée sur les événements. L'algorithme de propagation est un mécanisme qui *planifie* et *exécute* des *arcs* pour atteindre le point fixe de l'étape de propagation, et ainsi obtenir la satisfiabilité du PSC. Un arc a est une paire $\langle p^a, v^a \rangle$ qui réunit un propagateur p^a et une variable v^a, tel que $v^a \in V_{p^a}$; $\mathcal{A}$ représente l'ensemble des arcs du PSC. L'algorithme de propagation est déclenché par la modification d'un domaine (par exemple, l'application d'une décision) et se déroule en deux étapes. Premièrement un arc a est retiré d'un ensemble Q. Cet arc réunit le propagateur p^a à exécuter et la variable modifiée v^a. Deuxièmement, le propagateur p^a est exécuté, avec d_{v^a} pour paramètre. L'exécution peut résulter soit en un *échec*, par exemple un domaine devient vide, alors l'algorithme s'arrête ; soit, en la planification d'autres arcs dans Q. Ces opérations sont exécutées jusqu'à ce que Q devienne vide, ce qui signifie alors qu'un point fixe a été de nouveau atteint.

La manière dont Q est *orienté* peut avoir une incidence sur la manière dont les propagateurs doivent être implantés. L'orientation désigne le type d'objets manipulés par Q. Les alternatives aux arcs sont (a) de planifier des variables [2], (b) de planifier des propagateurs [4]. Dans le cas d'un moteur de propagation *orienté variables* (a), lorsqu'une variable est retirée de l'ensemble Q, une boucle exécute ses propagateurs un à un. Dans le cas d'un moteur de propagation *orienté propagateurs* (b), lorsqu'une variable est modifiée, une boucle planifie ses propagateurs un à un. La complexité reste inchangée ; cependant, le nombre maximum d'objets ajoutés dans Q peut varier, et l'ordre d'exécution des propagateurs également. Plus récemment, l'introduction des contraintes globales dans les solveurs de contraintes a souligné l'importance d'organiser l'exécution des propagateurs en fonction de leur complexité. Ainsi, il a été montré que la planification des propagateurs des contraintes globales devait être effectuée soigneusement. D'un côté,

dans les solveurs orientés variables, ceci est géré en ajoutant un ensemble dédié aux propagations *à gros grains* [5, 10]. Cette approche peut être améliorée en associant des *watched literals* [11] aux propagateurs. De l'autre côté, [15] ont introduit la notion de *propagateurs à états* et ont proposé une plage de priorités pour discriminer les propagateurs dans les moteurs orientés propagateurs. [13] ont introduit la notion d'*advisors* comme concept additionnel. Evidemment, ces approches sont similaires : elles ont pour but de réduire le nombre d'appels à la planification de propagateurs et d'améliorer l'efficacité du filtrage.

En parallèle de ces travaux sur les conditions de planification, l'*ordre de révision* des objets à propager a été étudié. L'ordre de révision définit comment un élément est sélectionné et retiré de l'ensemble Q. Il est généralement admis qu'une file, qui choisit l'élément le plus vieux d'abord, est un bon choix pour l'ensemble Q car il traite tous les éléments de manière équitable. Cependant, des alternatives existent. D'une part, [8] propose la sélection dynamique d'un élément en se basant sur un critère tel que la taille du domaine de la variable ou l'arité d'une propagateur. D'autre part, [15] ont proposé un moteur orienté propagateurs avec sept niveaux de priorités. Un propagateur est planifié dans une file en fonction de sa priorité dynamique [1]. Enfin, de récents progrès on été faits concernant l'accès à l'algorithme de propagation en introduisant la notion de groupe de propagateurs [16]. Néanmoins, l'implantation des groupes reste à la charge de l'utilisateur.

Enfin, contrairement aux stratégies de recherche qui sont communément disponibles sous la forme de portfolios ou de combinateurs [20], les moteurs de propagation sont plus ou moins imposés aux utilisateurs des solveurs. Cela réside, bien entendu, essentiellement dans la difficulté d'implanter un algorithme de propagation. En plus d'une connaissance de l'architecture du solveur, de solides compétences en programmation sont nécessaires pour ne pas dégrader la performance et la correction du solveur. Pour toutes ces raisons, nous proposons l'introduction d'un langage dédié au prototypage, rapide et robuste, des algorithmes de propagation.

3 Un DSL pour décrire les moteurs de propagation

Pour simplifier l'expression, ou la déclaration, d'un moteur de propagation pour l'utilisateur, nous introduisons maintenant un *langage dédié* [6] (DSL). Tout d'abord, nous donnons une courte liste des pré-requis nécessaires pour bénéficier entièrement du DSL. Ensuite, nous présentons le DSL en deux étapes : la composition des groupes et la structuration des groupes entre eux. Puis, nous rappelons ce qu'est MiniZinc [19] et décrivons comment ce langage de modélisation a été étendu avec notre DSL. Nous listons les propriétés et garanties qui accompagne notre DSL. Enfin, nous discutons des éventuels problèmes liés à notre proposition.

3.1 Accéder au propriétés du solveur

Pour entièrement bénéficier de notre DSL, des fonctionnalités sont requises. Parce qu'elles agissent sur l'efficacité des solveurs, il est très vraisemblable que ces fonctionnalités soient déjà implantées largement.

- Les propagateurs *à états* [15] : les propagateurs sont ordonnés par rapport à leur priorité. Les propagateurs *simples* (au sens de la complexité) sont exécutés avant ceux plus complexes ;
- Les groupes de propagateurs [16] : un propagateur *contrôleur* est lié à un groupe de propagateurs ;
- L'accès aux propriétés des variables et des propagateurs : par exemple, la cardinalité d'une variable, l'arité d'un propagateur ou sa priorité.

Les propriétés mentionnées dans ce dernier point doivent pouvoir être évaluées dynamiquement pendant la propagation. Une évaluation dynamique implique bien évidemment un coût et peut nécessiter des structures de données dédiées [8].

Pour plus de flexibilité et de précision, nous assumons que tous les arcs $\mathcal{A}$ sont accessibles explicitement. Ceci peut être fait en représentant explicitement chacun d'entre eux et en leur associant des watched literals [11] ou des advisors [13]. La représentation explicite des arcs d'un problème induit un surcoût mémoire : $\mathcal{O}(|\mathcal{V}| \times |\mathcal{P}|)$ objets supplémentaires doivent être créés, chacun d'entre eux maintenant deux pointeurs, l'un vers une variable, l'autre vers un propagateur. Cependant, les arcs apportent plus de flexibilité en donnant à la fois accès aux propriétés de la variable et à celles du propagateur. De plus, les arcs peuvent être regroupés autour de variables, de propagateurs ou de propriétés.

3.2 Le langage dédié

En considérant $\mathcal{A}$ en entrée, l'objectif du DSL est de décrire un ordre global sur cet ensemble. Le DSL est divisé en deux parties (Figure 1) : d'abord, l'affectation d'un arc à un groupe, puis, la structuration du moteur de propagation basée sur ces groupes. Ce processus en deux étapes permet de se concentrer d'abord sur les propriétés des arcs, puis de composer les groupes entre eux. Pour plus de clarté, chacune des parties sera présentée individuellement. La syntaxe suit une

FIGURE 1 – Le point d'entrée du DSL.

$\langle propagation_engine \rangle$: := $\langle group_decl \rangle$+ $\langle structure_decl \rangle$;

1. La priorité dynamique combine l'arité et la priorité pour fournir une évaluation fine du coût d'exécution du propagateur [15].

FIGURE 2 – Définition des groupes

```
⟨group_decl⟩   : :=  ⟨id⟩ : ⟨predicates⟩;
⟨id⟩           : :=  [[a-zA-Z]][[a-zA-Z0-9]]⋆
⟨predicates⟩   : :=  ⟨predicate⟩ | true
                |    ((⟨predicates⟩ ((&& | ||) ⟨predicates⟩)+)
                |    !⟨predicates⟩
⟨predicate⟩    : :=  in(⟨var_id⟩ | ⟨cstr_id⟩)
                |    ⟨attribute⟩ ⟨op⟩ (⟨int_const⟩ | "string" )
⟨op⟩           : :=  == | != | > | >= | < | <=
⟨attribute⟩    : :=  var[.(name|card)]
                |    cstr[.(name|arity)]
                |    prop[.(arity|priority|prioDyn)]
⟨int_const⟩    : :=  [+−][[0-9]][[0-9]]⋆
⟨var_id⟩       : :=  -⋆⟨id⟩
⟨cstr_id⟩      : :=  -⋆⟨id⟩
```

notation BNF avec les conventions suivantes : `tt` indique un symbole terminal ; ⟨*it*⟩ indique un symbole non terminal ; des crochets [*e*] définissent une option ; des doubles crochets [[a-z]] définissent un intervalle ; le symbole plus *e*+ définit une séquence de une ou plus répétitions de *e* ; le symbole étoile *e*⋆ définit une séquence de zéro ou plus répétitions de *e* ; l'ellipse *e*,... définit une séquence non vide de *e* séparés par des virgules ; l'alternance *a*|*b* indique des alternatives.

3.2.1 Déclaration des groupes

Nous présentons maintenant le langage dédié à la déclaration des groupes. La notion de groupe est définie dans [16]. La notion de groupe a un objectif simple : contrôler un ensemble de propagateurs en une fois en définissant, pour le groupe, une politique interne de planification et d'exécution. Ici, nous substituons aux propagateurs des arcs. Constituer des groupes d'arcs doit se faire dans le respect des règles d'affectation (décrites dans la Figure 2). L'utilisateur n'a accès qu'aux variables et aux contraintes lors de la modélisation. Celles-ci, et leurs propriétés, peuvent être directement référencées dans cette partie du DSL. Les propagateurs, quant à eux, ne peuvent être référencés que via leurs propriétés, puisqu'ils ne sont habituellement pas accessibles à l'étape de modélisation. Les propriétés listées ici sont présentes dans la majorité des solveurs.

Un groupe est déclaré avec l'aide d'un identifiant et d'une liste de prédicats. L'identifiant ⟨*id*⟩ est un nom unique commençant par une lettre, par exemple `G1`. Un prédicat est une fonction booléenne $\mathcal{A} \rightarrow \{\texttt{true}, \texttt{false}\}$, elle définit la condition d'appartenance à un groupe. Un prédicat est dit *en extension* s'il repose sur une variable ou une contrainte (via `var_id` et `cstr_id`). Les arcs associés à une variable ou une contrainte sont éligibles pour le groupe. Par exemple, `in(`v_1`)` définit un ensemble d'arcs A_1 tel que pour tout $a \in A_1$, $v^a = v_1$; `in(`c_1`)` définit A_2 tel que pour tout $a \in A_2$, $p^a \in P_{c_1}$. Un prédicat est dit *en intension* s'il repose sur une ou plusieurs propriétés. Tous les arcs retenus doivent alors satisfaire les propriétés. Les prédicats en intension sont construits à l'aide de trois paramètres : un attribut ⟨*attribute*⟩, un opérateur ⟨*op*⟩ et une valeur entière ou une chaîne de caractères.

FIGURE 3 – Structure d'un moteur de propagation

```
⟨structure⟩    : :=  ⟨struct⟩ | ⟨struct_reg⟩
⟨struct⟩       : :=  ⟨coll⟩ of {⟨elt⟩,...} [key ⟨comb_attr⟩]
⟨struct_reg⟩   : :=  ⟨id⟩ as ⟨coll⟩ of {⟨many⟩} [key ⟨comb_attr⟩]
⟨elt⟩          : :=  ⟨structure⟩
                |    ⟨id⟩ [key ⟨attribute⟩]
⟨many⟩         : :=  each ⟨attribute⟩ as ⟨coll⟩ [of {⟨many⟩}]
                     [key ⟨comb_attr⟩]
⟨coll⟩         : :=  queue(⟨qiter⟩)
                |    [rev] list(⟨liter⟩)
                |    [max] heap(⟨qiter⟩)
⟨qiter⟩        : :=  one | wone
⟨liter⟩        : :=  ⟨qiter⟩ | for | wfor
⟨comb_attr⟩    : :=  (⟨attr_op⟩.)⋆⟨ext_attr⟩
⟨ext_attr⟩     : :=  ⟨attribute⟩ | size
⟨attr_op⟩      : :=  any | min | max | sum
```

Un ⟨*attribute*⟩ fait référence à une propriété d'une variable (`var.name`, le nom d'une variable ; `var.card`, sa cardinalité), d'une contrainte (`cstr.name`, le nom d'une contrainte ; `cstr.arity`, sa cardinalité) ou d'un propagateur (`prop.arity`, l'arité d'un propagateur ; `prop.priority`, sa priorité statique ; `prop.prioDyn`, sa priorité dynamique). Par exemple, `prop.priority` < 4 définit un ensemble d'arcs tel que la priorité statique de chacun des arcs de cet ensemble est strictement inférieure à 4. Les prédicats peuvent être combinés avec les trois opérateurs booléens `&&`, `||` and `!`.

Le ⟨*group_decl*⟩ peut être adapté à chaque solveur en étendant ⟨*attribute*⟩, c'est à dire, en déclarant de nouvelles propriétés aux contraintes et aux variables. Lorsque des variables ou des contraintes sont pointées, ⟨*groups_decl*⟩ est dépendant du modèle, autrement il en est complètement indépendant.

Le but de ⟨*groups_decl*⟩ est de contruire des sous-ensembles d'arcs. Cette partie du DSL est évaluée de manière impérative, *i.e.*, chaque évaluation de prédicat peut déplacer un ou plusieurs arcs de $\mathcal{A}$ vers le sous-ensemble défini par le prédicat. Cependant, un arc peut répondre `true` à des prédicats de différents groupes. Pour empêcher d'avoir à discriminer les prédicats entre eux, cette partie du DSL est évaluée de haut en bas et de droite à gauche. Ainsi, un arc qui répond `true` à un prédicat devient indisponible pour les prédicats suivants.

3.2.2 Déclaration de la structure

La seconde partie du DSL (Figure 3) porte sur la description, basée sur les groupes, des structures de données. Le but ici est double : indiquer "où" stocker un arc et "comment" le sélectionner. De la déclaration de la structure résulte une structure d'arbre hiérarchisée basée sur une collection ⟨*coll*⟩.

Une ⟨*coll*⟩ est une structure de données abstraite (SDA) composée d'autres SDA et d'arcs. Une SDA définit comment les éléments vont être planifiés et retirés pour une exécution. Il en existe trois types : une `queue`, une `list` et un `heap`. Chacun d'entre eux définit également un itérateur qui décrit la manière de le traverser. Une file `queue` est une collection d'élé-

FIGURE 4 – Moteur de propagation basé sur un tas.

```
1. heap(wone) of {
2.     G1 key var.card,
3.     list(for) of {G2} key any.var.card
4. }
```

ments de la forme *first-in-first-out*. Une liste `list` est une collection d'éléments ordonnés statiquement (`rev` renverse la liste). Un tas `heap` est une collection d'éléments ordonnés dynamiquement grâce à un critère (la fonction de comparaison par défaut est $\leq$, le mot-clé `max` le transforme en $\geq$).

Les itérateurs de bases sont `one` et `wone`, définis pour toute $\langle coll \rangle$. `one` traverse un élément d'une liste $\langle coll \rangle$. `wone`, version courte pour 'while one', appelle `one` jusqu'à ce que tous les éléments aient été traversés. Une `list` définit deux itérateurs supplémentaires : `for` and `wfor`. `for` traverse un à un les éléments d'une liste dans l'ordre croissant de son index. `for` n'autorise pas de retours arrières ni d'interruption de la traversée. `wfor`, version courte pour 'while for', appelle `for` jusqu'à ce que tous les éléments de la liste aient été traversés.

Un `heap` requiert la déclaration d'un critère de comparaison (qui est optionnel pour une `list`). Le critère doit être statique pour une `list` et dynamique pour un `heap`. Comme il n'y a aucune garantie qu'une $\langle coll \rangle$ contienne exclusivement des arcs, le critère de tri doit être défini par (a) le mot-clé `key` et un $\langle attribute \rangle$ d'arc ou (b) le mot-clé `key` et une $\langle comb_attr \rangle$ pour un ensemble d'éléments. L'évaluation d'un ensemble est faite en évaluant les arcs de l'ensemble (grâce à $\langle attr_op \rangle$). Une extension de $\langle attribute \rangle$, nommée $\langle ext_attr \rangle$, est introduite dans la Figure 3. Une $\langle ext_attr \rangle$ permet d'atteindre tout attribut d'un élément d'une collection d'élément, comme sa taille `size`. Par exemple, la Figure 4 déclare un moteur de propagation basé sur un tas. Le tas haut niveau impose que les éléments qui le composent soit évaluables (comparables entre eux). Tous les arcs de `G1` sont évalués en retournant la cardinalité de leur variable (ligne 2). Le second groupe est une liste d'arcs `G2`. Comme une liste n'est pas évaluable en tant que telle, son évaluation doit être déléguée à l'un des arcs qui la composent. Un arc de `G2` est alors sélectionné arbitrairement (`any`) pour obtenir la cardinalité de sa variable (ligne 3). Le mot-clé `any` indique à la fois de déléguer l'évaluation du critère à un élément de `list` et de le choisir arbitrairement.

Maintenant, nous décrivons comment utiliser les structures de données abstraites dans une structure complète. Le point d'entrée d'une déclaration de structure est $\langle structure \rangle$. Une $\langle structure \rangle$ définit la structure du plus haut niveau qui pilotera la propagation. Elle doit être de type $\langle struct \rangle$ ou $\langle struct_reg \rangle$.

Une $\langle struct \rangle$ définit une collection $\langle coll \rangle$ composée d'éléments $\langle elt \rangle$. Un $\langle elt \rangle$ peut faire référence soit à l'identifiant d'un groupe et une instruction d'ordre ($\langle id \rangle$ [`key` $\langle attribute \rangle$]), soit une autre$\langle structure \rangle$. Une $\langle struct \rangle$ est définie en *extension*. La Figure 4 montre un `heap` composé d'arcs de `G1` et d'une liste.

Une $\langle struct_reg \rangle$ définit une structure régulière. L'expressivité impliquée par sa régularité justifie sa présence : elle permet l'expression compacte de structures imbriquées, où l'utilisation de $\langle struct \rangle$ impliquerait une expression verbeuse et sujette à l'introduction de bugs. Une $\langle struct_reg \rangle$ est définie en *intension*. Elle prend en entrée un ensemble d'arcs défini par un identifiant de groupe et elle *génère* autant de $\langle coll \rangle$ que requis par l'instruction $\langle many \rangle$, potentiellement imbriquées les uns dans les autres. $\langle many \rangle$ implique une itération sur les valeurs de l'$\langle attribute \rangle$ et regroupe les éléments avec la même valeur pour l'$\langle attribute \rangle$ dans une même $\langle coll \rangle$. La Figure 5 montre la déclaration

FIGURE 5 – Exemple de structure régulière.

```
1. G as list(wfor) of{
2.   each prop.priority as queue(for) of{
3.     each var as list(for)}
4. }
```

d'une structure régulière. La construction de la structure de données prend `G` en entrée : autant d'ensembles d'arcs qu'il y a de `prop.priority` parmi les arcs de `G` sont créés (ligne 2). Pour rappel, [15] en définissent 7. Puis, les ensembles sont traités un par un et leurs arcs sont regroupés par variables dans des listes. Toutes les listes sont ajoutées dans une `queue` qui correspond à une `priority`. Les files sont ajoutées dans la liste de haut niveau. Si un critère dynamique le nécessite, l'étape d'affectation à la bonne collection sera effectué pendant la propagation.

3.2.3 Propriétés

Un langage dédié apporte des propriétés, mais également des garanties. Nous décrivons maintenant toutes les propriétés et garanties de notre langage.

Description indépendante du solveur Un langage dédié bénéficie naturellement d'un haut niveau d'abstraction. Le fait qu'il ne repose sur l'architecture d'aucun solveur permet de l'implanter facilement dans un large éventail de solveurs de contraintes.

Expressivité Notre DSL regroupe les structures de données et les caractéristiques usuelles (de la simple file aux files de priorités). Il permet d'aller au-delà en les combinant avec les propriétés et décrire une grande variétés de moteurs de propagation de manière concise, incluant le mélange d'orientations.

Extensibilité Le DSL peut être étendu de deux manières. Premièrement, la déclaration de nouveaux attributs permet de définir de manière plus précise les groupes. Deuxièmement, l'ensemble des collections proposé peut être enrichi, tout comme les itérateurs. Cela donne l'opportunité de déclarer des propagations incomplètes, telles que décrites dans [12].

Garantie de couverture Tous les arcs d'un modèle sont représentés dans le moteur de propagation. L'expressivité de notre DSL autorise la description de moteurs de propagation dans lesquels un ou plusieurs arcs seraient absents. Il en résulte un moteur de propagation qui ignore une partie du modèle. Actuellement, nous en informons l'utilisateur et stoppons l'évaluation du DSL. Une alternative est le *collecteur d'arcs* qui réclame les arcs *libres* pour assurer la couverture.

Unicité de la propagation Un arc peut satisfaire plusieurs prédicats, et donc être éligible à différents groupes. L'évaluation vérifiée plus haut du DSL garantie que chaque arc n'est représenté qu'une seule fois dans le moteur de propagation. Donc, un événement intervenant sur une variable ne peut pas être traité plus d'une fois. Cela garantit que la complexité reste inchangée en branchant le DSL.

Conformité Les arcs liés à aucune paire (p,v) existante du modèle ne sont pas représentées dans le moteur de propagation. Grâce à cette propriété, les arcs *dénués de sens* ne peuvent être ni planifiés ni exécutés durant la propagation. Une fois de plus, cela préserve la complexité du moteur de propagation.

Complétude La complétude est assurée par une collection haut niveau définie avec un itérateur "while", tel que `wone` and `wfor`. Ainsi, quelques soient les itérateurs des sous-structures, la collection de plus haut niveau assure qu'il n'y ait plus d'événement en attente avant de terminer la propagation. Sélectionner `one` ou `for` peut ne plus assurer la complétude. Cependant, [12] explique pourquoi une propagation incomplète peut être utile.

Fiabilité Le DSL autorise la description de moteurs de propagations mal formés, bien que corrects. Par exemple : un groupe peut être vide, une définition de groupe peut être inutilisée dans la déclaration de la structure, une structure peut imbriquer des $\langle many \rangle$ basés sur le même $\langle attribut \rangle$, une structure peut ne pas contenir d'arc ou juste un seul, etc. Une solution serait d'appliquer automatiquement des règles d'optimisation et de réduction.

Complexité Les propriétés de "Conformité" et d'"Unicité de la propagation" garantissent le maintien de la complexité de l'algorithme de propagation. Cependant, la déclaration de moteurs de propagation mal formés, ou l'utilisation excessive de critères dynamiques peut avoir un impact néfaste sur les performances du moteur de propagation. Les utilisateurs doivent être en conscients durant le prototypage d'un moteur de propagation et son évaluation. Par contre, le DSL garantit que la propagation se termine.

3.3 Implémentation

Étendre un langage de modélisation est une manière simple d'implanter notre DSL : l'accès aux objets du modèle y est assuré, sans requérir un solveur concret.

FIGURE 6 – Modèle MiniZinc `magic sequence` étendu avec notre DSL .

```
1 :   include "globals.mzn";
2 :   annotation name(string: s);
3 :   annotation engine(string: s);
4 :   int: n;
5 :   array [0..n - 1] of var 0..n: x;
6 :   constraint count(x,0,x[0])::name("c0");
7 :   constraint forall (i in 1..n - 1) (count(x, i, x[i]));
8 :   solve
9 :   :: engine("
10 :  G1: in("c0");
11 :  All: true;
12 :  list(wone) of {queue(wone) of{G1},
13 :       All as queue(wone) of {each cstr as list(wfor)}};
14 :  ")
15 :  satisfy;
```

Pour cela, nous avons choisi MiniZinc [19]. Il s'agit d'un langage de modélisation simple mais expressif. Il couvre un large éventail de solveurs. Nous avons également choisi ANTLR [2] pour interpréter la grammaire. Cet outil permet la séparation claire entre l'analyseur syntaxique, (le *lexeur*), la reconnaissance syntaxique (le *parseur*) et la transposition en instructions solveur. Cela simplifie la réutilisation du lexeur et du parseur, seule la transposition doit être adaptée aux solveurs.

En pratique, nous devons ajouter la description du moteur de propagation au sein du modèle MiniZinc. Ceci est effectué à l'aide d'annotations : la première sert à intégrer la description du moteur de propagation, la seconde sert à nommer une contrainte. La Figure 6 montre l'exemple d'un modèle MiniZinc dans lequel une déclaration de moteur de propagation a été ajoutée. Les lignes 1, 4-8 and 15 détaillent le modèle du problème de `magic sequence` (prob019, [18]) de taille 5. Les annotations requises pour nommer les contraintes (ligne 2) et décrire le moteur de propagation doivent être déclarées dans l'en-tête du fichier. La première contrainte (ligne 6) est annotée avec `::name("c0")` pour la nommer. Les lignes 9-14 décrivent le moteur de propagation : deux groupes sont définis. Le premier groupe `G1` (ligne 10) contient les arcs de la contrainte nommée `c0` ; le second groupe `All` (ligne 11) contient les arcs restants, grâce à l'évaluation haut-bas gauche-droite. La structure du moteur est composé d'une liste elle-même composée de deux files (lignes 12-13). La première file stocke les arcs du groupe `G1` (ligne 12) ; la seconde est composée de listes d'arcs, réunis autour de contraintes (ligne 13). Ce moteur oblige à toujours traiter les événements de la contraintes `c0` avant les autres événements.

3.4 Discussion

MiniZinc est distribué avec un langage bas niveau appelé FlatZinc. FlatZinc est le langage cible pour les modèles contraintes dans lequel les modèles MiniZinc sont traduits, à destination des solveurs. Cependant, la

2. http ://www.antlr.org/

traduction de MiniZinc vers FlatZinc peut induire une reformulation des contraintes optionnelles grâce aux contraintes obligatoires. Pour se concentrer sur le DSL et sa mise en application, nous interdisons ce processus de reformulation. Même si la déclaration originale pouvait être transformée comme le sont les contraintes, il n'y a aucune garantie que la déclaration résultante respecte la sémantique d'origine. En effet, si on considère une déclaration retardant autant que possible la programmation d'une contrainte `AllDifferent` qui, après reformulation, soit transformée en une clique de différences : cela serait contreproductif et trompeur. De plu, l'impact de la reformulation sur l'efficacité de la propagation n'a pas encore été évaluée formellement, et représente en soi un intéressant sujet de recherche.

D'autre part, un solveur peut ne pas disposer des pré-requis listés en Section 3.1, et donc, ne pas supporter les moteurs de propagation basés sur le DSL. Une refonte du solveur est alors nécessaire pour remplir ces conditions. Une telle refonte, vraisemblablement à destination des développeurs de solveurs, est une tâche critique. Cependant, il est fort probable que les solveurs basés sur les événements sont mieux préparés aux algorithmes de filtrage orientés arcs. De plus, cette refonte apporte non seulement plus de flexibilité dans les solveurs de contraintes mais ouvre également des perspectives pour la résolution. Naturellement, notre DSL ne remet pas en cause les moteurs natifs, il vient en complément de ceux-ci. Il peut également simplifier la conception des groupes de propagateurs.

4 Evaluation expérimentale

La plupart du temps, les structures de données utilisées dans un moteur de propagation ont des complexités très faibles. Par exemple, une file ou un bitset définit les opération `add()` and `remove()` en temps constant, un tas binaire définit les opérations `add()`, `extractMin()` et `update()` en $\mathcal{O}(log(n))$. Même si l'efficacité de ces opérations est critique, l'efficacité d'une résolution est plus liée à l'ordre dans lequel les propagateurs sont exécutés. Il y aura sûrement un surcoût lié au parsing du DSL, dû à la fois au chargement des données et à la vérification des garanties. Mais le surcoût lié à l'utilisation d'un moteur interprété devra être limité aux indirections induites par les compositions de groupes et aux évaluation des critères.

La configuration. Toutes les expérimentations ont été faites sur un Macbook Pro avec un 6-core Intel Xeon cadencé à 2.93Ghz sur MacOS 10.6.8, et Java 1.6.0_35. Les résultats sont basés sur 20 exécutions (l'écart-type est toujours inférieur à 5%) et les tableaux reportent le temps de parsing en millisecondes et le temps de résolution en secondes.

Solver. L'évaluation de notre DSL et son interprétation ont été implantées en Java avec Choco [5], une bibliothèque Java pour la programmation par contraintes. Les sources ont été modifiés pour répondre aux pré-requis listés dans la Section 3.1. De plus, trois moteurs de propagation natifs (*i.e.*, non basés sur le DSL) ont été implantés dans la bibliothèque : un moteur orienté propagateurs `prop`, un autre orienté variables `var`, le dernier est un moteur orienté propagateurs, composé de sept files et évaluant la priorité des propagateurs dynamiquement `7qd` [15].

Expérimentations. La première évaluation a été faite sur la base de 39 instances extraites de 21 problèmes [3] de la distribution MiniZinc [19]. Plus de détails sur les instances sont disponibles sur demande. Ensuite, nous présenterons un cas d'utilisation, basé sur le problème du `Golomb ruler`. Il permettra de démontrer combien le prototypage de moteurs de propagation est simplifié avec le DSL.

4.1 Descriptions des moteurs

Cette section est dédiée à la description des moteurs natifs(`prop`, `var` et `7qd`) à l'aide du DSL, ainsi qu'à la comparaison du traitement des fichiers MiniZinc avec les moteurs natifs et avec les moteurs interprétés. Pour chaque moteur natif, il est garanti que sa version DSL construise le même arbre de recherche et préserve l'ordre ainsi que le nombre d'exécutions des propagateurs. Cela nous permet d'illustrer l'expressivité du DSL.

Version DSL des moteurs de propagation Tout d'abord, nous présentons comment déclarer `prop`, un moteur de propagation orienté propagateurs, à l'aide du DSL (Figure 7(a)). La structure prend tous les arcs du problème en entrée. Puis, les arcs sont regroupés par propagateurs dans des listes. Chaque liste définit une traversée incomplète (`for`) mais la file haut niveau assure la complétude de la propagation (`wone`). Ensuite, nous présentons la déclaration de `var`, un moteur de propagation orienté variables, à l'aide du DSL (Figure 7(b)). Il est très proche de la description précédente, seulement, les arcs sont regroupés par variables. Enfin, nous présentons comment déclarer `7qd` à l'aide de notre DSL, un moteur orienté propagateur, composé de sept files et évaluant la priorité des propagateurs dynamiquement (Figure 7(c)). Comme les précédents, il prend tous les arcs du problème en entrée. Ces arcs sont distribués dans les différentes files en fonction de la priorité dynamique de leurs propagateurs. Pour chaque priorité disponible, les arcs sont regroupés autours des propagateurs dans les listes. Comme

3. Les problèmes sélectionnés sont : `bacp, bibd, black-hole, CostasArrays, debruijn-binary, fastfood, fillomino, ghoulomb, golomb, jobshop, langford, latin-squares, magicseq, market_split, nonogram, p1f, pentominoes, radiation, rcpsp, slow_convergence, still_life`.

FIGURE 7 – Trois moteurs de propagation, version DSL

```
1 : All :true;
2 : All as queue(wone) of {
3 :     each prop as list(for)
4 : }
```

(a) Propagator-oriented.

```
1 : All :true;
2 : All as queue(wone) of {
3 :     each var as list(for)
4 : }
```

(b) Variable-oriented.

```
1 : All : true;
2 : All as list(wone) of {
3 :     each prop.prioDyn as queue(one) of {
4 :         each prop as list(for)
5 :     }key any.any.prop.prioDyn
6 : }
```

(c) Priority-oriented.

l'évaluation des priorités est dynamique, le choix de la file dans laquelle mettre une liste est fait grâce à l'évaluation du propagateur d'un arc de la liste. La liste de haut niveau garantit à la fois la complétude de la propagation (`wone`) et que les événements associés aux propagateurs de faible priorité sont traités avant ceux de plus forte priorité. Notez que le mot-clé `any` est doublé. Cela est nécessaire pour déléguer l'évaluation de la `priority` de la file à un élément évaluable, un arc parmi ceux présents dans ces listes. Le premier `any` délègue d'un niveau, *i.e.*, une liste, le second pointe vers un élément de cette liste, *i.e.*, un arc. Cet arc est donc le représentant de la file, et son propagateur est utilisé pour évaluer la file.

4.2 Evaluation du parsing et de la résolution

L'intérêt de la phase de prototypage est de construire et d'évaluer rapidement, mais sûrement, des moteurs de propagation. D'un côté, la phase de parsing n'est pas critique dans une démarche de prototypage, mais elle ne doit tout de même pas trop pénaliser les tests. La phase de résolution, quant à elle, est critique. Bien que les versions interprétées des moteurs de propagation ne puissent être compétitives, en terme d'efficacité, avec les approches natives, leur coût d'utilisation doit être aussi faible que possible pour valider notre approche en tant qu'outil utilisable en pratique. La Table 1 reporte le surcoût du parsing des fichiers MiniZinc sans et avec la description des moteurs de propagations. La phase de parsing inclut le chargement du fichier, l'analyse lexicographique, la reconnaissance syntaxique et la traduction en instructions du solveur. La Table 1 reporte également le surcoût de résolution des instances avec les moteurs interprétés en comparaison avec les versions natives. Dans ce cas, le temps affiché exclut le temps de parsing. Enfin, la colone "ranking" indique le pourcentage d'instances qui préserve, avec les versions interprétées, l'ordre impliqué par le temps de calcul entre les moteurs natifs. Sans surprise, la phase de parsing souffre de l'ajout du DSL, et globalement cette phase requiert 28,46% plus de temps, en moyenne. Les détails ne sont pas donnés ici, mais il y a un lien direct entre le temps de parsing et le nombre d'arcs créés : plus il y aura d'arcs, plus le parsing prendra de temps. Cela vient du besoin de représentation des arcs en mémoire et de vérification des garanties inhérentes au DSL mais également de la manière dont sont regroupés les arcs. Par exemple, dans les instances où le nombre de variables est petit comparé au nombre de propagateurs, parser la version DSL de `var` se fait plus rapidement que les deux autres. Enfin, on observe que parser `prop` et `7qd` est comparable. Regrouper les arcs autour des propagateurs est une opération commune aux deux déclarations.

TABLE 1 – Surcoûts moyens induit par l'introduction du DSL, sur 39 instances.

Moteur	Parsing		Solving		ranking
	moy.	ec.typ.	moy.	ec.typ.	
`prop`	28.56%	19.34	9.33%	7.29	
`var`	28.75%	20.15	7.01%	5.29	97.4%
`7qd`	28.07%	17.64	13.20%	7.63	

Concernant la phase de résolution, nous pouvons observer que le surcoût moyen induit par l'utilisation d'un moteur interprété est de 9,46%. De tels surcoûts proviennent vraisemblablement du manque d'optimisation dont souffre les versions interprétées. Dans ces versions, lorsqu'un arc est planifié, la liste à laquelle il appartient doit être également planifiée, si nécessaire. Ces opérations sont améliorables dans un langage de planification supportant l'optimisation directe.

Les surcoûts d'utilisation liés à `var` et `prop` sont très proches de ceux des versions natives. Ce qui n'est pas le cas de `7qd`. Dans sa version native, l'évaluation de `prioDyn` est faite de manière efficace en interrogeant directement le propagateur. Dans sa version interprétée, chaque arc doit être évalué à chaque planification. Donc, même si la version native de `7qd` est habituellement très performante, sa version interprétée est pénalisée. Sans surprise une fois de plus, les moteurs basés sur le DSL induisent un surcoût. Cependant, il est intéressant de remarquer que seule une instances sur les 39 ne préserve pas l'ordre entre les moteurs. Plus généralement, l'introduction du DSL souffre, d'une part, des évaluations multiples de critères et d'autre part, des structures de données imbriquées. Une description basée sur l'un de ces aspects sera nécessairement moins compétitive pendant la résolution. Rappelons que les moteurs de propagation dans leur version native ont fait l'objet de multiples optimisations. Bien que les version DSL introduisent des surcoûts mémoire et temporel, ils assurent toutes les propriétés naturellement. La prochaine étape d'évaluation concerne la mise en œuvre, sur un cas d'utilisation, du DSL afin de démontrer la pertinence de son utilisation.

4.3 `Golomb ruler` : un cas d'utilisation

Nous montrons maintenant, à l'aide d'un cas d'utilisation, combien il est facile de prototyper des moteurs

FIGURE 8 – Moteurs pour `Golomb ruler`.

```
1 : All : true;
2 : All as heap(wone) of {
3 :      each var as list(for) key any.var.card
4 : };
```

(a) Moteur basé sur un tas.

```
1 : All : true;
2 : All as heap(wone) of {
3 :      each var as list(for) of {
4 :           each prop.priority as list(for)
6 :           key any.prop.priority
5 : } key any.any.var.card};
```

(b) Une variante du moteur basé sur un tas.

```
1 : M : in(mark);
2 : A : true;
3 : list(wfor) of {
4 :    list(wfor) of {M key var.name},
5 :    queue(wone) of {A}
6 : };
```

(c) Moteur à deux collections.

de propagation grâce à notre langage dédié. Nous allons déclarer les moteurs de l'état de l'art et essayer deux autres moteurs ad hoc.

Le problème du `Golomb ruler` (prob006, [18]) est défini comme "un ensemble de m entiers $0 = a_1 < a_2 < \cdots < a_m$ tel que les $\frac{m(m-1)}{2}$ différences $a_j - a_i, 1 \leq i < j \leq m$ soient distinctes. Une telle règle contient m marques et est de taille a_m. L'objectif est de trouver une règle de taille minimum." [8] a montré que propager les événements en traitant d'abord ceux associés à la variable de plus petite cardinalité est une bonne stratégie pour améliorer la résolution de ce problème. Nous construisons une instance de taille $m = 10$ en utilisant MiniZinc, et instrumentons le code pour déclarer différents moteurs de propagation. En utilisant les contraintes globales disponibles dans Choco, le problème est alors modélisé grâce à 55 variables, une contrainte `allDifferent`, 11 inégalités binaires et 45 équations ternaires ; la stratégie de recherche est basée sur a, elle préserve l'ordre lexicographique et choisit la plus petite valeur de chaque variable choisie.

Tout d'abord, nous présentons les moteurs de propagation utilisés pour cette évaluation. En plus de `prop`, `var` et `7qd` présentés auparavant, nous déclarons `heap-var`, `heap-var-prio` et `2-coll`. `heap-var` (Figure 8(a)) est un moteur orienté variables basé sur un tas, comme décrit dans [8]. Les arcs sont regroupés autour des variables dans des listes, ces listes sont ensuite ajoutées dans un tas. La liste qui contient la variable de plus petite cardinalité est alors sélectionnée pour être propagée. Cette sélection est répétée tant qu'il reste des listes planifiées. Puis, nous déclarons `heap-var-prio` (Figure 8(b)), une variante de `heap-var` où les arcs, au sein de chaque liste, sont triés en fonction de la priorité de leur propagateur. L'idée ici est d'exécuter

TABLE 2 – Moteurs pour `Golomb ruler`.

Engine	$m = 10$		$m = 11$	
	`time` (sec.)	`pex` (10^6)	`time` (sec.)	`pex` (10^6)
`prop`	3.83	10.085	76.70	191.894
`var`	2.99	10.305	55.50	192.034
`7qd`	3.19	10.860	60.67	200.108
`heap-var`	2.79	10.717	53.09	196.525
`heap-var-prio`	3.31	11.242	59.84	207.973
`2-coll`	**2.69**	**9.975**	**52.48**	**185.921**

d'abord, pour une variable, les propagateurs les plus rapides. Cela devrait permettre de discriminer les propagateurs de la contrainte `AllDifferent` des autres. Enfin, nous déclarons `2-coll` (Figure 8(c)), un moteur composé de deux collections. La première collection est basée sur `M`, l'ensemble des arcs associés aux variables a. Ils sont placés dans une liste et triés par ordre lexicographique du nom de leur variable. Les événements intervenant sur la variable a_i seront traités avant ceux intervenant sur la variable a_{i+1}. Le mot-clé `wfor` garantit, avec une propagation par *balayage*, d'atteindre un point fixe local (tous les événements des arcs de `M` auront été traités). La deuxième collection est une file qui stockent les arcs restants (ceux impliqués dans les variables de différences), elle garantit un second point fixe local (`wone`). La Table 2 montre les résultats pour les différents moteurs déclarés. Le temps de résolution (`time`, en secondes) et le nombre de propagations (`per`) pour $m = 10$ et $m = 11$ y sont reportés.

Même si il n'y a pas de relation direct entre le nombre de propagations et le temps de résolution (par exemple,`7qd` demande 5.9% plus de propagations que `prop` mais prend 23.2% moins de temps), les moteurs qui exécutent moins les propagateurs sont souvent les plus rapides. Nous observons les mêmes résultats que [8] : la sélection de la variable de plus petite cardinalité permet de résoudre plus efficacement le problème. De plus, distinguer les propagateurs d'une variable grâce à la priorité, dans `heap-var-prio`, est contre-productif : cela retarde trop l'exécution des propagateurs de la contrainte `AllDifferent`. Enfin, le moteur `2-coll` est le plus efficace : son comportement est proche de `heap-var`. Par contre il ne nécessite pas d'évaluation dynamique, et passe donc mieux à l'échelle. Il est basé sur une liste et une file dont les opérations se font en temps constant. Le nombre d'exécutions de propagateurs est également le plus faible.

Ce qu'il est important que de remarquer à ce stade, c'est la facilité avec laquelle le prototypage a été effectué avec le DSL. Par trois fois, il a été possible, en quelques lignes, de définir une heuristique de propagation spécifique, reposant sur les propriétés des variables et des contraintes, et de produire un moteur de propagation sûr, interprétable et utilisable en pratique. Cela simplifie grandement le processus d'évaluation et promeut l'étude de l'ordre de révision des contraintes au sein de CSP.

5 Conclusion et Travaux Futurs

Notre première motivation, dans cet article, était de fournir un outil qui simplifie la configuration du moteur de propagation dans un solveur de contraintes. Après avoir rappelé les différents variantes de l'algorithme de propagation (orienté arcs, variables et propagateurs), ainsi que les techniques modernes pour les rendre plus efficaces, nous avons présenté un langage dédié ouvert et extensible qui permet de décrire, simplement mais efficacement, des moteurs de propagation sans connaissance spécifique d'un langage de programmation. Nous avons listé les pré-requis nécessaires pour bénéficier totalement du langage ainsi que les propriétés et garanties qu'il fournit. Nous avons discuté des faiblesses de notre approche. Enfin, nous avons évalué les surcoûts liés à l'utilisation du langage lors des phases de parsing et de résolution. Nous avons montré l'expressivité de notre proposition et la flexibilité qu'il permet d'apporter aux solveurs. En plus des moteurs communément utilisés, il permet la déclaration de schémas de propagation très précis.

Il existe également des limites à la configuration des moteurs de propagation par notre approche, traçant ainsi le chemin des travaux futurs. Tout d'abord, il est important d'empêcher la création de moteur mal formés en proposant des règles de transformation et d'optimisation. La reformulation des contraintes et son impact sur les moteurs de propagation devra également être étudiée. Notre DSL pourrait alors faire partie intégrante des standards pour la programmation par contraintes. Il convient enfin d'étudier notre approche sur des problèmes réels, et d'analyser plus finement son interaction avec la stratégie de recherche.

Références

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

[2] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inf. Sci.*, 19(3) :229–250, 1979.

[3] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *AI*, 57(2-3) :291–321, 1992.

[4] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP '97*, pages 191–206, 1997.

[5] F. Laburthe. Choco : Implementing a CP kernel. In *TRICS*, pages 71–85, 2000.

[6] A. van Deursen, P. Klint and, J. Visser Domain-specific languages : an annotated bibliography In *SIGPLAN Not.*, pages 26–36, 2000.

[7] A. López-Ortiz, CG. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI*, pages 245–250, 2003.

[8] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *CP'04*, pages 9–43, 2004.

[9] C. Bessiere. Constraint propagation. Technical report 06020, LIRMM, 2006.

[10] I. P. Gent, C. Jefferson, and I. Miguel. Minion : A fast scalable constraint solver. In *Proceedings of ECAI 2006*, pages 98–102. IOS Press, 2006.

[11] I. P. Gent and C. Jefferson, and I. Miguel. Watched literals for constraint propagation in minion. In *Procedings CP 2006*, pages 182–197, 2006.

[12] I. Katriel Expected-Case Analysis for Delayed Filtering In *Lecture Notes in Computer Science*, pages 119–125, Springer Berlin Heidelberg, 2006.

[13] M. Z. Lagerkvist and C. Schulte. Advisors for incremental propagation. In *CP 2007*, pages 409–422, 2007.

[14] I. P. Gent, I. Miguel, and P. Nightingale. Generalized arc consistency for the alldifferent constraint : An empirical survey. *Artificial Intelligence*, 172(18) :1973–2000, 2008.

[15] C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 2008.

[16] M. Z. Lagerkvist and C. Schulte. Propagator groups. In Ian Gent, editor, *CP'09*, pages 524–538, 2009.

[17] C. Schulte and G. Tack. Implementing efficient propagation control. In Christopher Jefferson, Peter Nightingale, and Guido Tack, editors, *TRICS 2010*, 2010.

[18] I. P. Gent and T. Walsh. Csplib : a benchmark library for constraints. Technical report, APES-09-1999, 1999.

[19] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck and G. Tack MiniZinc : Towards a standard CP modelling language. In *CP'07*, pages 529–543, 2007.

[20] IBM ILOG CPLEX Optimizer. http ://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/, 2009.

[21] Gecode : Generic Constraint Development Environment. http ://www.gecode.org, 2012.

[22] JaCoP : Java Constraint Solver. http ://www.jacop.eu/, 2012.

[23] or-Tools : Operations Research Tools developed at Google. http ://code.google.com/p/or-tools/, 2012.

Une adaptation simple et efficace du modèle MapReduce pour la programmation par contraintes

Jean-Charles Régin **Mohamed Rezgui** **Arnaud Malapert**

Université Nice-Sophia Antipolis
I3S UMR 6070, CNRS, France.
{Jean-Charles.REGIN,Arnaud.Malapert}@unice.fr, rezgui@i3s.unice.fr

Résumé

Nous proposons MapReduceCP, une méthode simple et efficace pour résoudre les problèmes de programmation par contraintes en parallèle. Au lieu d'utiliser l'approche classique basée sur le *work stealing*, nous adaptons le modèle MapReduce à la décomposition de l'espace de recherche. Nous découpons le problème initial en un grand nombre de sous-problèmes distincts puis nous les résolvons de façon dynamique avec les *workers* disponibles (les cœurs des machines). Les expériences avec les problèmes de satisfaction et les problèmes d'optimisation suggèrent de fixer à trente, le nombre de sous-problèmes non détectés inconsistants par *worker* afin d'obtenir un bon passage à l'échelle. Nous montrons que notre méthode est très compétitive avec l'approche *work stealing* et qu'elle est capable de résoudre certains problèmes classiques avec la capacité maximale des machines multi-cœurs. Grâce à cette méthode, l'utilisateur peut paralléliser la résolution d'un problème sans modifier le solveur ou écrire de code source parallèle.

1 Introduction

Il existe principalement deux manières de paralléliser un solveur de programmation par contraintes. D'une part, les algorithmes de filtrage (ou la propagation) sont parallélisés ou distribués. Le travail le plus remarquable sur ce sujet a été réalisé par Y. Hamadi [6]. D'autre part, le processus de recherche est parallélisé. Nous focalisons notre attention sur cette méthode. Pour une description plus complète des méthodes qui ont été testées pour l'utilisation d'un solveur CP en parallèle, le lecteur peut se référer à l'étude de Gent et al. [5].

Lorsque nous voulons utiliser k machines pour la résolution d'un problème, nous pouvons découper le problème initial en k sous-problèmes disjoints et donner un sous-problème à chaque machine. Nous appelons cette méthode : méthode de décomposition statique simple. Ensuite, nous recueillons les différents résultats afin de produire des résultats correspondant à l'ensemble du problème. L'avantage de cette méthode est sa simplicité. Malheureusement, il souffre de plusieurs inconvénients qui surviennent fréquemment dans la pratique : les temps passés à résoudre chaque sous-problème sont rarement bien équilibrés. Afin d'équilibrer les sous-problèmes qui doivent être résolus, certains travaux ont été réalisés sur la décomposition de l'arbre de recherche [9, 3, 8]. Toutefois, la taille des arbres est approximative et n'est pas strictement corrélée avec le temps de résolution. Ainsi, comme mentionné par Bordeaux et al. [1], il est assez difficile de s'assurer que chaque *worker* recevra la même quantité de travail. Par conséquent, cette méthode souffre de quelques problèmes de passage à l'échelle, car le temps de résolution est le maximum du temps de résolution de chaque *worker*. Afin de remédier à ces problèmes, une autre approche a été proposée et est actuellement plus populaire : l'idée du *work stealing*.

Le *work stealing* est assez simple : les *workers* résolvent une partie des problèmes et quand un *worker* entre en phase de famine, 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 à un autre *worker* V s'il a du travail à lui donner. Si la réponse est positive, alors le *worker* V découpe le problème qu'il résout actuellement en deux sous-problèmes et en donne un des deux au *worker* W. Si la réponse est négative, alors W demande à un autre *worker* U, jusqu'à ce qu'il arrive à avoir un travail à faire ou que tous les autres *workers* ont été considérés.

Cette méthode a été mise en œuvre dans un grand

nombre de solveurs (Comet [10] ou ILOG Solver [12] par exemple), et de plusieurs manières [14, 7, 16, 2] en fonction du fait que le travail à faire est centralisé ou non, du découpage de l'arbre de recherche, ou de la méthode de communication entre les *workers*.

L'approche *work stealing* résout en partie le problème d'équilibrage de la méthode de décomposition statique simple, principalement parce que la décomposition est dynamique. Par conséquent, il n'est pas nécessaire d'être en mesure de découper un problème en plusieurs parties équilibrées. Cependant, tous les sous-problèmes donnés à un autre *worker* ne sont pas égaux. Quand un *worker* est affamé, il doit éviter de voler trop de problèmes faciles, car dans ce cas, il devra demander un autre travail presque immédiatement. Cela se produit fréquemment à la fin de la recherche lorsqu'un grand nombre de *workers* n'ont pas de problèmes à résoudre et demandent tout le temps du travail et donc cela complexifie et ralentit la fin de la recherche en augmentant le temps de communication entre les *workers*. Ainsi, nous observons généralement que la méthode obtient un bon passage à l'échelle pour un petit nombre de *workers* mais qu'il est difficile de maintenir un gain linéaire lorsque le nombre de *workers* devient plus grand (certaines méthodes ont été développées pour tenter d'augmenter ce passage à l'échelle[15]).

Dans cet article, nous proposons une autre méthode proche de la méthode de décomposition statique simple en utilisant les principes de la méthode MapReduce.

Lorsque nous avons k *workers*, au lieu d'essayer de découper le problème en k sous-parties équivalentes, nous proposons de découper le problème dans un grand nombre de sous-problèmes, par exemple $30k$ sous-problèmes, puis de donner successivement et de façon dynamique ces sous-problèmes aux *workers* lorsque ces derniers ont besoin de travail. Au lieu d'attendre d'avoir des sous-problèmes équivalents, nous nous attendons à ce que **pour chaque *worker*, la somme des temps de résolution de ses sous-problèmes doit être équivalente**. Ainsi, l'idée **n'est pas de décomposer à priori** le problème initial en un ensemble de problèmes équivalents, mais de décomposer le problème initial en un ensemble de sous-problèmes dont le temps de résolution peut être partagé d'une manière équivalente par un ensemble de *workers*. Notez que nous ne savons pas à l'avance les sous-problèmes qui seront résolus par *worker*, car c'est déterminé de manière dynamique. **Tous les sous-problèmes sont mis dans une file d'attente et un *worker* prend un sous-problème dans cette file lorsqu'il a besoin de travailler**.

La décomposition en sous-problèmes doit être fait avec soin. Nous ne devons pas définir un sous-problème si ce sous-problème aurait été éliminé par le mécanisme de propagation du solveur dans une recherche séquentielle. Ainsi,**nous générons seulement les problèmes qui ne sont pas détéctés inconsistants par le solveur**.

Le papier est organisé comme suit. Nous rappelons d'abord quelques principes sur la méthode MapReduce et les facteurs qui peuvent influer sur la parallélisation de l'arbre de recherche. Puis, nous introduisons notre méthode, MapReduceCP, pour décomposer les problèmes initiaux. Ensuite, nous donnons des résultats expérimentaux. Enfin, nous concluons.

2 Préliminaires

MapReduce est un modèle de programmation pour le traitement de grands ensembles de données, et également le nom d'une implémentation du modèle par Google [4]. Il s'agit d'un cadre pour le traitement des problèmes massivement parallèles à travers les très grands ensembles de données utilisés par un grand nombre d'ordinateurs.

Ici, nous ne sommes pas intéressés par les détails du MapReduce, mais par ses principes. MapReduce a deux étapes : une étape *Map* et une étape *Reduce* :

• l'étape *Map* : le nœud maître prend le problème initial, le découpe en petits sous-problèmes, et les distribue aux nœuds esclaves. Un nœud esclave peut le faire à nouveau à son tour, conduisant à une structure d'arbre à niveaux multiples. Le nœud esclave traite le problème plus petit, et passe la réponse à son nœud maître.

• L'étape *Reduce* : le nœud maître recueille ensuite les réponses de tous les sous-problèmes et les combine d'une certaine manière pour former la sortie, qui correspond à la réponse au problème initial.

Dans cet article, nous représentons l'espace de recherche comme l'immense ensemble de données du modèle MapReduce.

Deux facteurs peuvent influer sur le rendement de la parallélisation de la recherche d'un solveur (pas nécessairement un solveur CP) basée sur une méthode de décomposition, comme la décomposition statique simple, ou le *work stealing*, ou encore MapReduce. Nous considérons une partition d'un problème P en un ensemble S de sous-problèmes.

– *Si la somme du temps de résolution des sous-problèmes de S est plus grande que le temps de résolution de P,* alors il sera difficile de passer à l'échelle en fonction du nombre de *workers*. Cela peut arriver dans certains cas : la stratégie de recherche utilisée est basée sur les calculs précédents. Les expériences que nous avons effectuées ne montrent pas un tel comportement avec un solveur CP. Cependant, nous les avons constaté dans certains cas, avec un solveur MIP.
– *Les temps de résolution de certains sous-problèmes de S sont courts ou bien vraiment plus courts que les autres*. Toute méthode doit être en mesure de minimiser les communications et doit équilibrer l'activité des *workers*. Les temps de résolution courts signifient que les *workers* doivent souvent demander du travail,

ce qui augmente les temps de communication. D'autre part, si seulement quelques sous-problèmes requièrent des temps plus longs alors que la plupart d'entre eux sont faciles, alors il peut être difficile d'équilibrer la charge de chaque *worker*.

3 Décomposition de problème

Dans cette partie, nous calculons S, une partition de sous-problèmes d'un problème initial P.

3.1 Principes

Nous avons vu que la décomposition du problème initial dans le même nombre de sous-problèmes que le nombre de *workers* peut causer des temps de résolution déséquilibrés pour chaque sous-problème. Ainsi, notre idée est d'augmenter fortement le nombre de sous-problèmes considérés, d'une manière similaire à celle de la méthode MapReduce. Aussi, nous définissons un nœud maître qui est en charge de la création et l'envoi des sous-problèmes, en d'autres termes de l'opération Map. Ce nœud maître va également récupérer les calculs effectués par les *workers*, c'est à dire de l'opération Reduce.

Avant d'entrer dans les détails sur la mise en œuvre, nous établissons certaines propriétés. Durant la résolution d'un problème, nous appellerons :

• *le temps d'activité d'un worker* la somme des temps de résolution d'un *worker*.

• *le temps d'inactivité d'un worker* la différence entre le temps écoulé pour résoudre le problème en entier et les temps d'activité des workers. Notre approche est principalement basée par la remarque suivante :

Remarque *Les temps d'activités de tous les workers peuvent être bien équilibrés, même si le temps de résolution de chaque sous-problème n'est pas bien équilibré*

Lorsqu'un *worker* résout plusieurs sous-problèmes, ces derniers peuvent avoir des temps de résolution différents alors que la somme des ces temps reste égale à une valeur donnée. Considérons un problème dont le temps de résolution est de 100 secondes. Ce problème peut être découpé en sous-problèmes ayant des temps de résolution non balancés : $25, 20, 15, 10, 10, 10, 5, 5$ secondes. Avec 4 *workers*, la répartition de charge pourrait être 25, $20 + 5$, $15 + 10$, $10 + 10 + 5$, ce qui la rend bien balancée.

La principale difficulté d'une décomposition statique n'est pas de définir des problèmes équivalents, mais d'éviter d'avoir des *workers* actifs pendant que d'autres ont terminÃ© leur travail. Nous n'avons pas besoin de connaître à l'avance le temps de résolution de chaque sous-problème. Nous espérons juste que les *workers* auront des temps d'activité équivalents. Pour atteindre cet objectif, il est important de décomposer les problèmes initiaux en un grand nombre de sous-problèmes pour trois raisons.

• nous augmentons ainsi nos chances d'obtenir des temps d'activités bien équilibrés pour les *workers*, car nous augmentons nos chances d'être en mesure d'obtenir une combinaison de valeurs menant à la même période d'activité pour chaque *worker*.

• lorsque l'espace de recherche tend à ne pas être équilibré, nous aurons des sous-problèmes qui prendront plus de temps à être résolus. En ayant un grand nombre de sous-problèmes, nous augmentons nos chances de découper ces sous-problèmes en plusieurs parties ayant des temps de résolution comparables et ainsi d'obtenir à la fin une répartition de charge bien équilibrée des *workers*.

• quand un sous-problème a un temps de résolution plus long que n'importe quel autre problème, nous avons une chance de voir cette différence moins importante lorsque ce sous-problème est découpé en plusieurs sous-problèmes. Dans ce cas, nous pouvons espérer réduire l'importance du sous-problème.

Considérons un problème qui nécessite 140 secondes à résoudre et que nous ayons 4 *workers*. Si nous divisons le problème en 4 sous-problèmes alors nous avons les temps de résolution suivants : $20, 80, 20, 20$. Nous aurons besoin de 80 secondes pour résoudre ces sous-problèmes en parallèle. Par conséquent, nous gagnons un facteur $140/80 = 1.75$. Maintenant, si nous séparons à nouveau chaque sous-problème en 4 sous-problèmes, nous pourrons obtenir les sous-problèmes suivants représentés par leurs temps de résolution : $(5+5+5+5)+(20, 10, 10, 40)+(2, 5, 10, 3)+(2, 2, 8, 8)$. Dans ce cas, nous pourrons avoir les assignements suivants : "worker1" : $5 + 20 + 2 + 8 = 35$; "worker2" : $5+10+2+10 = 27$; "worker3" : $5+10+5+3+2+8 = 33$ and "worker4" : $5 + 40 = 45$. Le temps écoulé est maintenant à 45 secondes et nous obtenons un facteur de gain à $140/45 = 3.1$. En découpant encore les sous-problèmes, nous réduirons le temps de résolution moyen des sous-problèmes et nous espérons couper le sous-problème prenant 40 secondes en des sous-probllèmes prenant moins de temps. Notons que nous avons seulement peu de chance d'augmenter le temps écoulé en décomposant plus le sous-problème.

Propriété 1 *Si un problème est découpé en plusieurs sous-problèmes dont le temps de résolution maximum est $tmax$, alors*

(i) le temps de résolution minimum du problème en entier est $tmax$

(ii) le temps d'inactivité maximum du worker est inférieur ou égal à $tmax$.

preuve : (i) est clair. (ii) Supposons qu'un *worker* W ait un temps d'inactivité qui est supérieur à $tmax$.

Considérons le moment où W commence à attendre après son temps d'activité. À ce moment, il n'y a plus de sous-problèmes disponibles à résoudre, autrement W aurait été actif. De plus, un autre *worker* était déjà en charge du sous-problème le plus long à résoudre ou bien ce dernier a déjà été résolu. Ainsi, la durée maximale de la résolution de cet autre *worker* est inférieure à $tmax$. □

L'approche *work stealing* a un contrôle précis et dynamique sur la façon dont l'arbre de recherche est exploré et partagé. Cependant, notre méthode peut conduire à des résultats équivalents :

Propriété 2 *Il existe une décomposition du problème initial en sous-problèmes, qui est aussi bon que celui de n'importe quelle méthode de work stealing.*

preuve : l'approche *work stealing* est juste une décomposition dynamique, et nous pouvons utiliser cette décomposition dans notre méthode. □

Un avantage intéressant de notre méthode dans la pratique, est que nous pouvons tout simplement rejouer une résolution en parallèle en sauvegardant l'ordre dans lequel les sous-problèmes ont été exécutées. Cela ne coûte presque rien et c'est très utile dans le débogage d'applications.

Dans la section suivante, nous considérons diverses façons possibles pour définir les sous-problèmes.

3.2 Génération des sous-problèmes

Même si ce n'est pas vraiment important d'avoir des sous-problèmes équivalents, il est généralement préférable d'avoir des problèmes correspondant à la même taille de l'espace de recherche. Cela ne signifie pas qu'ils auront le même temps de résolution, mais il est recommandé d'avoir de l'homogénéité dans les problèmes générés. Supposons que nous voulions découper un problème en q problèmes disjoints.

Une méthode simple peut être définie par la manière suivante :

- nous considérons tout ordre de variables x_1,...,x_n.
- nous définissons par A_k le produit cartésien $D(x_1) \times ... \times D(x_k)$.
- nous calculons la valeur k de telle sorte que $|A_{k-1}| < q \leq |A_k|$. Chaque assignement de A_k définit un sous-problème et donc A_k est la décomposition recherché.

Cette méthode fonctionne bien pour certains problèmes tels que les n-reines ou la règle de Golomb, mais elle est très mauvaise pour d'autres problèmes, parce qu'un grand nombre d'assignements de A peut se révéler être trivialement inconsistant. Considérons par exemple que x_1, x_2 et x_3 possèdent trois valeurs $\{a, b, c\}$ dans leurs domaines et qu'on applique une contrainte alldiff sur ces trois variables. Le produit cartésien de domaines de ces variables contiennent 27 tuples. Parmi eux seulement 6 ((a, b, c), (a, c, b), (b, a, c),(b, c, a),(c, a, b), (c, b, a)) qui ne sont pas inconsistants avec la contrainte *alldiff*. Ainsi, seulement $6/27 = 2/9$ des problèmes générés ne sont pas trivialement inconsistants. C'est important de noter que la plupart des problèmes inconsistants pourrait ne jamais être considérée par une résolution séquentielle. Pour certains problèmes, nous observons que plus de 99% des problèmes générés sont détéctés inconsistants en lançant le mécanisme de propagation. Ainsi, nous présentons une autre méthode qui évite de tomber dans ce cas.

Nous proposons de **générer seulement les sous-problèmes qui ne sont pas détectés inconsistants par la propagation**. Nous les appelons des sous-problèmes NDI. La génération de ces q sous-problèmes devient plus complexe car le nombre de sous-problèmes NDI peut ne pas être lié au produit catésien de certaines domaines. Un algorithme simple pourrait consister à effecturer un parcours en largeur (BFS), dans l'arbre de recherche jusqu'à que le nombre de sous-problèmes NDI désiré est atteint. Malheureusement, c'est n'est pas facile d'exécuter efficacement une BFS en général car une BFS n'est pas un algorithme incrémental comme un parcours en profondeur (DFS). Par conséquent, nous proposons d'utiliser un parcours en profondeur bornée *Depth-bounded Depth First Search* (DBDFS) en anglais, qui est une DFS qui ne visite jamais les nœuds situés à une profondeur supérieure à une valeur donnée. Chaque branche d'un arbre de recherche calculée par cette recherche définit un assignement. Nous noterons NDI_k, l'ensemble des assignements calculés pour une profondeur k. Pour générer q sous-problèmes, nous répétons la DBDFS jusqu'à que nous atteignons un niveau k de telle sorte que $|\text{NDI}_{k-1}| < q \leq |\text{NDI}_k|$. Pour plus de commodité et de simplicité, nous utilisons un ordre statique des variables.

Nous améliorons cette méthode de trois façons :

(i) nous essayons d'estimer quelques bonnes valeurs pour k afin d'éviter de très répéter l'algorithme DBDFS. Nous utilisons le produit cartésien. Par exemple, si pour une profondeur donnée u, nous générons seulement $q/1000$ sous-problèmes et si la taille des domaines des trois variables suivantes non assignées est 10, alors nous pouvons déduire que nous avons besoin d'aller au moins à la profondeur $u + 3$.

(ii) dans le but d'éviter de répéter la même DFS pour les premières variables en répétant l'algorithme DBDFS, nous sauvegardons les assignements calculés précédemment dans une contrainte de table. Plus précisément, si nous avons calculé NDI_k alors nous utilisons une contrainte de table contenant tous ces assignements lorsque nous recherchons NDI_l avec $l > k$.

(iii) Nous effectuons cette décomposition en parallèle car elle peut prendre un certain temps par rapport à l'ensemble de la résolution du problème. Nous parallélisons notre algorithme de décomposition d'une manière simple. Considérons que nous avons w *workers*. Nous recherchons w sous-problèmes NDI. Ensuite, chaque *worker* reçoit un de ces sous-problèmes et le décompose en q/w sous-problèmes NDI en utilisant notre algorithme. Le nœud maître rassemble tous les sous-problèmes calculées. Notons que l'équilibrage de charge n'est pas vraiment important car une fois qu'un *worker* a terminé la décomposition d'un sous-problème, il commence à résoudre les sous-problèmes disponibles. Il est également possible que le *worker* ne soit pas en mesure de générer q/w sous-problèmes car il peut résoudre le sous-problème donné par le nœud maître en effectuant la décomposition. Nous pourrons essayer d'éviter ce cas mais nos expériences montrent que cela arrive très rarement et étant donné que nous recherchons un certain nombre de sous-problèmes à peu près égal à q, alors ce n'est vraiment pas important. Par ailleurs, Nous verrons également que cette parallélisation n'est pas nécessaire pour les meilleurs paramètres de notre méthode.

3.3 Implémentation

MapReduceCP consiste principalement à définir les fonctions Map et Reduce de la méthode MapReduce pour les problèmes de satisfaction et d'optimisation.

3.3.1 Problèmes de satisfaction

La fonction Map function consiste à

• calculer une partition du problème initial P en un ensemble S de sous-problèmes.

• gérer l'ensemble des sous-problèmes afin de les distribuer aux *workers* lorsqu'ils demandent du travail.

Le premier point est défini par l'utilisation d'un des algorithmes décrits précédemment, alors que le deuxième est implémenté par l'utilisation d'une structure de données de type FIFO (une queue par exemple).

La fonction Reduce est assez simple : lors de la recherche d'une solution, si une solution est trouvée, alors on stoppe l'exploration de l'arbre de recherche ; lors de la recherche de toutes les solutions, les solutions trouvées par les *workers* sont collectées par le nœud maître.

3.3.2 Problèmes d'optimisation

Dans le cas des problèmes d'optimisation, nous devons gérer la meilleure valeur de l'objectif trouvée durant la résolution. Ainsi, les fonctions Map et Reduce sont légèrement modifiées.

La fonction Map consiste à

• calculer une partition du problème initial P en un ensemble S de sous-problèmes.

• gérer l'ensemble des sous-problèmes et de la meilleure valeur de la fonction objective trouvée durant la résolution. Lorsqu'un *worker* demande du travail, la fonction Map lui donne un sous-problème à résoudre. il lui donne églament la meilleure valeur de l'objectif trouvée précédemment.

Notons qu'il n'y a pas d'autres communications, ceci se présente lorsqu'un *worker* trouve une meilleure solution, les autres *workers*, qui sont en train de travailler, ne peuvent l'utiliser pour améliorer la résolution de leurs problèmes courants. De même, si l'absence de communication peut améliorer nos performances, cet aspect peut également conduire à une baisse de performance. Heureusement, nous n'observons pas ce comportement dans la pratique.

Nous pouvons voir ici un autre argument pour engendrer un grand nombre de sous-problèmes dans le cas des problèmes d'optimisation : la résolution d'un sous-problème doit être courte pour améliorer la transmission d'une meilleure valeur de l'objectif et d'éviter d'effectuer certains travaux qui auraient pu être ignorés avec une meilleure valeur objective.

La fonction Reduce gère la valeur optimale trouvée par le *worker* qui sera ensuite utilisée par la fonction Map.

3.4 Taille de la partition

Une question importante par rapport à la taille de la partition : combien de sous-problèmes devons nous générer ?

Il s'agit principalement d'une question expérimentale. Cependant, notons que, si nous voulons passer à l'échelle alors ce nombre devrait être défini en fonction du nombre de *workers* qui sont impliqués. Plus précisément, il est plus cohérent d'avoir q sous-problèmes par *worker* qu'un total de q sous-problèmes.

4 Partie Expérimentale

Machines Toutes les expériences ont été effectuées sur des machines Dell ayant quatre E7-4870 processeurs Intel, ayant chacun 10 cœurs avec 256 Go de mémoire et fonctionnant sous Scientific Linux.

En exécutant le même programme 10 fois dans chaque cœur de la machine, nous avons pu atteindre un facteur par rapport à une résolution séquentielle de 37.5. Par conséquent, c'est le gain maximum que nous pouvons espérer pour une machine.

Nos expériences peuvent être reproduites en téléchargeant le programme MapReduceCP sur Linux à partir de ce lien [13].

Solveurs Nous avons implémenté notre méthode dans deux solveurs CP : or-tools rev2555 par Google et Gecode 4.0.0 (http ://www.gecode.org/).

Protocole Expérimental Notre code exécute seulement trois opérations :

• soit il lit un fichier contenant un modèle FlatZinc ou soit le problème est directement modélisé par l'API du solveur.

• il crée les "threads" et definit une instance du solveur dans chaque "thread".

• il calcule les sous-problèmes à générer, puis les fournit aux "threads" et rassemble les résultats.

Pour chaque problème, nous allons soit chercher toutes les solutions des problèmes de satisfaction ou prouver l'optimalité des problèmes d'optimisation, ou résoudre les problèmes d'optimisation dans leur ensemble (trouver la meilleure solution et de prouver son optimalité). Nous citerons le type de résolution que nous effectuons dans chaque expérience.

Notons que les tests de performance d'une méthode parallèle est plus complexe avec un problème d'optimisation parce que la chance peut jouer un rôle. Il peut être avantage ou un inconvénient pour nous. Cependant, dans la vie réelle, les problèmes d'optimisation sont très fréquents, il est donc important de tester notre méthode sur ces derniers.

Instances sélectionnées Nous avons sélectionné un large éventail de problèmes qui sont représentatifs des types de problèmes résolus en CP. Certains viennent de la CSPLib et ont été modélisés par Hakank (http ://www.hakank.org/) et d'autres proviennent de la distribution minizinc (1.5 voir [11]). Nous avons sélectionné les instances qui ne sont ni très faciles (supérieur à 20 secondes en séquentiel), ni très longues à résoudre (inférieur à 700 secondes) avec le solveur Gecode.

1. Les exemples fournis par Hakank sont : golombruler-13 ; magicsequence-40000 ; sportsleague-10 (10 équipes, nous énumérons toutes les solutions) ; warehouses (nombre de warehouses = 10, nombre de stores = 20 et un coût fixé = 180) ; setcovering (Placer les casernes de pompiers avec 80 villes et une distance minimale fixée à 24) ; allinterval-15 (le modèle proposé par Régin et Puget de la CSPLib est utilisé).
2. Les instances Flatzinc provenant de la distribution sont : 2DLevelPacking (5-20-6), depotPlacement (att48-5 ; rat99-5), fastfood (58), openStacks (01-problem-15-15 ;01-wbp-30-15-1), sugiyama (2-g5-7-7-7-7-2), patternSetMining (k1-german-credit ; k1-segment ; k1-ionosphere), sb-sb (13-13-6-4), quasigroup7-10, non-non-fast-6, radiation-03, bacp-7, talent-scheduling-alt-film116.

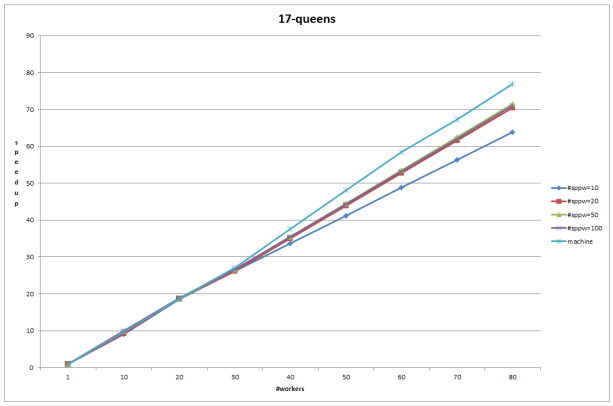


FIGURE 1 – 17-reines : performance en fonction de #sppw. Nous ne constatons plus la limite d'un facteur de gain de 29.

Tests Notons #sppw, le nombre de sous-problèmes par *worker*. Nous allons étudier les aspects suivants de notre méthode :

4.1 le facteur de gain par rapport aux autres décompositions statiques

4.2 le temps d'inactivité des *workers* selon la valeur de #sspw

4.3 la difficulté des sous-problèmes engendrés. En particulier, nous allons examiner le ratio des sous-problèmes NDI.

4.4 l'avantage de la parallélisation de la décomposition

4.5 l'influence de #sspw sur le facteur d'améliorations

4.6 l'impact du nombre des *workers* sur la meilleure valeur de #sspw

4.7 sa performance par rapport au *work stealing*

4.8 l'influence du solveur CP qui est utilisé

4.1 Comparaison avec la décomposition statique simple

Nous considérons le problème des n-dames, parce que certaines méthodes proposées [7, 1] n'ont pas été en mesure d'observer un facteur de gain par rapport au séquentiel supérieur à 29 avec une décomposition statique simple des problèmes même en utilisant 64 machines (ou cœurs). La figure 1 montre que notre méthode obtient un meilleur gain lorsque #sppw est supérieur à 20 (Notons que, pour un ou deux sous-problèmes par *worker*, nous n'obtenons pas un bon gain). La limite de ce gain (un ratio maximum à 29) décrit dans [1] disparait clairement. Nous utilisons la même strategie que dans [1]. Des résultats similaires sont obtenus pour ce problème avec d'autres valeurs de n. Nous obtenons également un facteur de gain de 32 pour le problème d'optimisation de la règle de Golomb de taille 13.

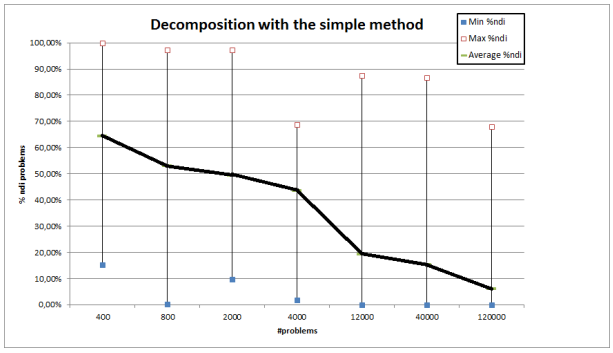


FIGURE 2 – Pourcentage des sous-problèmes NDI générés par la méthode de décomposition simple

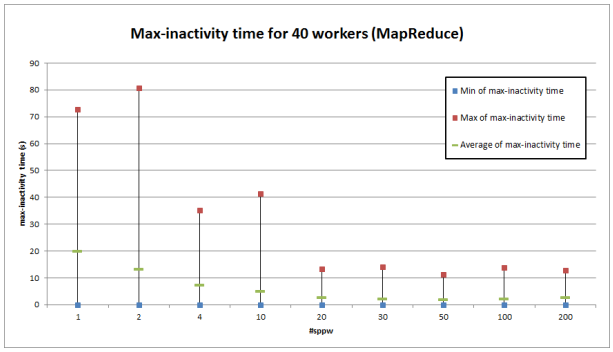


FIGURE 3 – Maximum du temps d'inactivité des *workers*.

4.2 Ratio des sous-problèmes NDI

La figure 2 montre le pourcentage des sous-problèmes NDI générés par la méthode de décomposition simple. Nous pouvons voir que ce nombre dépend fortement du nombre d'instances considérées. Pour certaines instances, le nombre est proche de 100%, tandis que pour d'autres il peut être vraiment proche de 0%, cela nous indique qu'il y a un problème de décomposition. La moyenne commence à 65% et diminue en fonction du nombre de sous-problèmes puis termine à 10%. Par conséquent, pour certaines instances (comme les n-reines ou la règle de Golomb), la décomposition simple est une bonne méthode alors que pour d'autres c'est une très mauvaise approche et ne doit en aucun cas être utilisée. C'est la raison pour laquelle nous avons développé la décomposition par DBDFS avec la méthode de propagation qui ne génère pas de problèmes qui ne sont pas NDI, donc pour le nombre de problèmes NDI générés est de 100%.

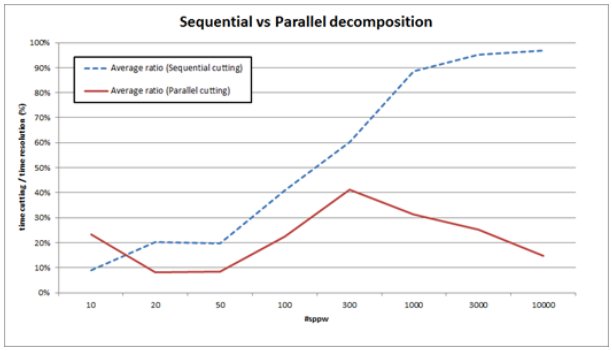


FIGURE 4 – Pourcentage du temps écoulé durant la décomposition.

4.3 Le temps d'inactivité selon la valeur de #sppw

La figure 3 montre que le temps d'inactivité des *workers* diminue lorsque le nombre de sous-problèmes par *worker* est augmenté. En outre, il montre que la répartition de charge est bonne quand nous avons plus de 10 sous-problèmes par *worker*. Il faut faire attention Ã ces résultats : on ne peut pas dériver la durée totale de la résolution seulement avec le temps d'inactivité car le temps de décomposition n'est pas considéré.

4.4 Parallélisation de la décomposition

La figure 4 compare le pourcentage de temps total nécessaire pour décomposer le problème lorsque seul le nœud maître effectue cette opération ou lorsque les *workers* sont également impliqués. Nous observons clairement que la parallélisation de la décomposition permet de gagner du temps, surtout pour un grand nombre de sous-problèmes par *worker*.

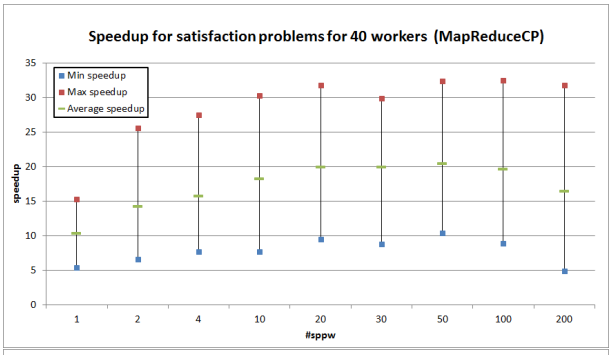
4.5 Influence du nombre de sous-problèmes à considérer

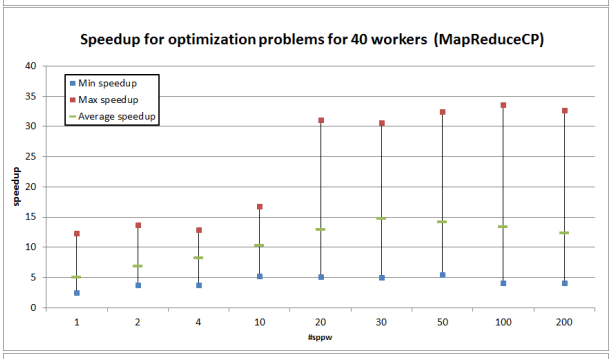
La figure 5 décrit le facteur de gain par rapport à une résolution séquentielle obtenu par la méthode MapReduceCP en fonction du nombre de sous-problèmes pour le solveur Gecode. La meilleure valeur semble être autour de 30 sous-problèmes par *worker*. En d'autres termes, nous proposons de commencer la décomposition avec $q = 30w$, où w est le nombre de *workers*.

Pour cette valeur :

- pour **80% des problèmes considérés, le facteur de gain maximum est obtenu** ;
- seuls 20% des problèmes pourraient être améliorés avec une autre valeur, mais jamais plus de 20%.

Cela montre que cette valeur ne dépend pas du problème considéré.





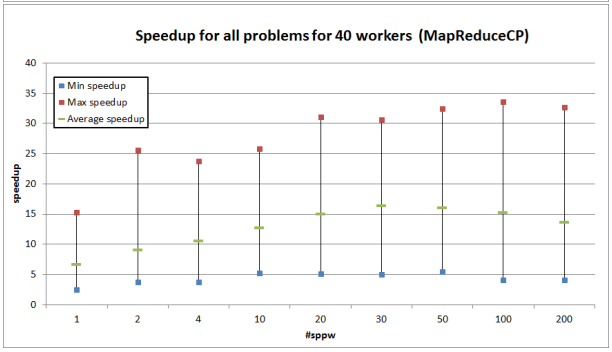


FIGURE 5 – Facteur de gain par rapport à une résolution séquentielle (speedup) en fonction du nombre de sous-problèmes pour trouver toutes les solutions des problèmes des satisfaction (au-dessus), pour trouver et prouver l'oprimalit'e des problèmes d'optimisation (au milieu), et pour tous les problèmes (en bas).

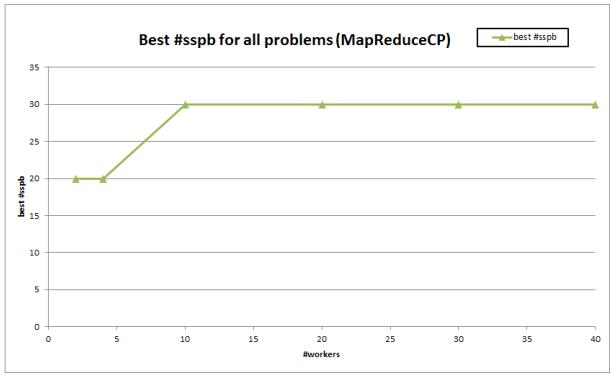


FIGURE 6 – Meilleure valeur de #sspw en fonction du nombre de *workers*.

bold for speedup winner timeout (3600 s)	Gecode 4.0.0 work stealing			MapReduceCP	
instance	seq. time (s)	parallel time (s)	speedup	parallel time (s)	speedup
allinterval_15	262,546	9,724	27,000	8,790	**29,869**
magicsequence_40000	328,194	592,648	0,554	37,334	**8,791**
sportsleague_10	172,390	7,648	22,541	6,782	**25,419**
sb_sb_13_13_6_4.fzn	135,674	9,203	14,742	7,752	**17,502**
quasigroup7_10.fzn	292,589	14,532	20,134	10,542	**27,755**
non_non_fast_6.fzn	602,222	271,254	2,220	56,843	**10,594**
golombruler_13	1355,150	54,889	24,689	44,321	**30,576**
warehouses	148,043	25,896	5,717	21,083	**7,022**
setcovering	94,431	16,117	5,859	11,089	**8,516**
2DLevelPacking_Class5_20_6.fzn	22,612	13,774	1,642	0,748	**30,230**
depot_placement_att48_5.fzn	125,154	19,083	6,558	10,17	**12,306**
depot_placement_rat99_5.fzn	21,628	6,368	3,396	2,613	**8,277**
fastfood_ff58.fzn	23,050	4,520	5,100	3,834	**6,012**
open_stacks_01_problem_15_15.fzn	102,753	6,080	16,900	5,759	**17,842**
open_stacks_01_wbp_30_15_1.fzn	185,715	15,350	12,099	11,219	**16,554**
sugiyama2_g5_7_7_7_7_2.fzn	286,537	22,751	12,594	10,766	**26,615**
pattern_set_mining_k1_german-credit.fzn	113,724	22,317	5,096	13,782	**8,252**
radiation_03.fzn	129,076	33,479	3,855	25,626	**5,037**
bacp-7.fzn	227,152	15,647	14,517	9,517	**23,868**
talent_scheduling_alt_film116.fzn	254,297	13,510	**18,823**	35,625	7,138

FIGURE 7 – Résolution de tous les problèmes avec Gecode. Le nombre de *workers* est à 40 et nous fixons #sspw=30.

La réduction de la performance en augmentant la valeur de #sspw vient du fait que le processus de décomposition résout une partie croissante du problème, et ce processus est plus lent qu'une procédure de résolution. Notons qu'avec notre méthode, seuls 10% du temps de résolution est perdu si nous utilisons une décomposition séquentielle au lieu d'une décomposition parallèle.

4.6 Nombre de workers et meilleure valeur de #sspw

Pour savoir si le nombre de sous-problèmes dépend du nombre de *workers*, nous avons cherché la meilleure valeur de #sspw pour différents nombres de *workers*. La figure 6 montre les résutats obtenus. Il apparaît clairement que ces nombres ne sont pas corrélés et que la valeur 30 semble constante.

4.7 Comparaison avec le *work stealing*

La Figure 7 présente une comparaison entre notre méthode et la méthode du *work stealing* disponible dans Gecode. Le facteur de gain moyen de la méthode *work stealing* est de 11,2 (14,5 pour les problèmes de satisfaction et 9,8 pour les problèmes d'optimisation) alors qu'avec l'approche MapReduceCP, le facteur de gain est de 16,4 (20,0 pour les problèmes de satisfaction et 14,9 pour les problèmes d'optimisation). Notre méthode est meilleure que le *work stealing* dans tous les cas sauf un.

4.8 Influence du solveur CP

Nous avons également effectué des expériences en utilisant le solver or-tools. Avec or-tools, les facteurs de gain par rapport au séquentiel de notre méthode sont augmentés. Nous présentons les résultats dans la figure 8. Nous obtenons un facteur moyen d'un gain de 14,4 pour le solveur de Gecode et 23,3 pour or-tools. Nous obtenons **un facteur de gain moyen de 16,4 pour Gecode et de 23,3 pour or-tools**.

bold for speedup winner timeout (3600 s)	Or-tools (rev2555) MapReduceCP		
instance	seq. time (s)	parallel time (s)	speedup
allinterval_15	2169,702	67,685	32,056
magicsequence_40000	timeout	timeout	timeout
sportsleague_10	timeout	timeout	timeout
sb_sb_13_13_6_4.fzn	227,566	18,149	12,539
quasigroup7_10.fzn	timeout	timeout	timeout
non_non_fast_6.fzn	2676,304	310,024	8,633
golombruler_13	16210,218	573,557	28,263
warehouses	timeout	timeout	timeout
setcovering	501,653	33,633	14,916
2DLevelPacking_Class5_20_6.fzn	56,185	3,623	15,508
depot_placement_att48_5.fzn	664,893	13,734	48,412
depot_placement_rat99_5.fzn	67,048	2,824	23,742
fastfood_ff58.fzn	452,438	25,134	18,001
open_stacks_01_problem_15_15.fzn	164,723	7,099	23,204
open_stacks_01_wbp_30_15_1.fzn	164,873	6,345	25,985
sugiyama2_g5_7_7_7_7_2.fzn	298,789	20,491	14,581
pattern_set_mining_k1_german-credit.fzn	270,742	12,818	21,122
radiation_03.fzn	416,624	23,511	17,720
bacp-7.fzn	759,733	23,752	31,986
talent_scheduling_alt_film116.fzn	575,721	15,706	36,656

FIGURE 8 – Résolution de tous les problèmes avec or-tools. Le nombre de *workers* est à 40 et nous fixons #sspw=30.

5 Remerciements

Nous tenons à remercier vivement Laurent Perron et Claude Michel pour leurs commentaires qui ont permis d'améliorer cet article.

6 Conclusion

Dans cet article, nous avons présenté MapReduceCP, une méthode simple pour résoudre les problèmes CP en parallèle, qui est basée sur le principe de la méthode MapReduce. Nous proposons de décomposer le problème initial en un ensemble de k sous-problèmes qui ne sont pas détectés inconsistants puis d'envoyer ces problèmes aux *workers* afin d'être résolu. Après quelques expérimentations, il apparaît que découper le problème initial en 30 sous-problèmes par *worker*, donne un facteur de gain moyen égal à 23 avec or-tools et $16,4$ avec Gecode tout en recherchant toutes les solutions ou alors trouver et prouver l'optimalité sur une machine ayant 40 cœurs. La méthode est compétitive avec l'approche *work stealing*.

Références

[1] Lucas Bordeaux, Youssef Hamadi, and Horst Samulowitz. Experiments with massively parallel constraint solving. In Craig Boutilier, editor, *IJCAI*, pages 443–448, 2009.

[2] 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.

[3] Gérard Cornuéjols, Miroslav Karamanov, and Yanjun Li. Early estimates of the size of branch-and-bound trees. *INFORMS Journal on Computing*, 18(1) :86–96, 2006.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : Simplified data processing on large clusters. In *OSDI*, pages 137–150. USENIX Association, 2004.

[5] Ian P. Gent, Chris Jefferson, Ian Miguel, Neil C.A. Moore, Peter Nightingale, Patrick Prosser, and Chris Unsworth. A preliminary review of literature on parallel constraint solving,. In *Proceedings PMCS-11 Workshop on Parallel Methods for Constraint Solving*, 2011.

[6] Youssef Hamadi. Optimal distributed arc-consistency. *Constraints*, 7 :367–385, 2002.

[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. IEEE Computer Society, 2004.

[8] Philip Kilby, John K. Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *AAAI*, pages 1014–1019, 2006.

[9] Donald E. Knuth. Estimating the efficiency of backtrack program. *Mathematics of Computation*, 29 :121–136, 1975.

[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. http ://www.g12.csse.unimelb.edu.au/minizinc/, 2012.

[12] 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.

[13] Jean-Charles Régin. www.constraint-programming.com/people/regin/papers.

[14] Christian Schulte. Parallel search made simple. In *University of Singapore*, pages 41–57, 2000.

[15] 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.

[16] 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.

Linear Programming and the Complexity of Finite-valued CSPs *

Johan Thapper[1] **Stanislav Živný**[2]

[1] Université Paris-Sud 11, France
[2] University of Warwick, United Kingdom
thapper@lri.fr s.zivny@warwick.ac.uk

Abstract

We consider the complexity classification project for the valued constraint satisfaction problem (VCSP) restricted to rational-valued cost functions. A year ago, classifications were known for Boolean domains and for the case when all unary cost functions are allowed. In the time that has passed since then, there has been significant progress in the area. In this paper, we describe the full complexity classification for finite-valued CSPs. Our result is a dichotomy. It states that the problem of minimising VCSP instances that use only rational-valued cost functions from a fixed set is either tractable or NP-hard, depending on the set. In the tractable cases, all instances are solved to optimality by establishing optimal soft arc consistency (OSAC), or even by the weaker basic linear programming relaxation (BLP). In the hard cases, it is possible to express a cost function that models the Max-Cut problem. Our result implies the full complexity classification for the Max-CSP which until these results seemed to be entirely out of reach.

1 Introduction

The *valued constraint satisfaction problem*, or VCSP for short, was introduced by Schiex et al. [13] as a unifying framework for studying constraint programming with soft constraints. We study the special case where the *valuation structure* is $\mathbb{Q}$ with addition as the aggregation operator, and each constraint comes with an additional *weight*. This special case is also called the *finite-valued* case, to distinguish it from the case where positive infinity is allowed as a value. From now on, whenever we talk about the VCSP, we will be referring to the finite-valued case.

*The results presented in this paper have previously been published in [14] and [15].

Definition 1 *An instance I of the valued constraint satisfaction problem is given by a set $V = \{x_1, \ldots, x_n\}$ of variables, a finite domain D, and an objective function $f_I(x_1, \ldots, x_n) = \sum_{i=1}^{q} w_i \cdot f_i(\bar{x}_i)$ where, for every $1 \leq i \leq q$, $f_i : D^{ar(f_i)} \to \mathbb{Q}$ is a* cost function, *$\bar{x}_i \in V^{ar(f_i)}$ is a tuple of variables, and $w_i \in \mathbb{Q}_{\geq 0}$ is a weight. A solution to I is a function $\sigma : V \to D$, its measure given by $\sum_{i=1}^{q} w_i \cdot f_i(\sigma(\bar{x}_i))$, where σ is applied componentwise. The goal is to find a solution of minimum measure. This measure is denoted* $\mathsf{Opt}(I)$.

A set Γ of cost functions is called a *valued constraint language*, or simply a *language*. VCSP(Γ) is the class of instances in which all functions are from Γ.

The systematic study of the complexity of VCSP(Γ) was initiated by Cohen et al. [3]. The goal is to determine, for a given Γ, whether VCSP(Γ) is solvable in polynomial-time (tractable) or NP-hard. Languages on two-element domains [3], three-element domains [7], and conservative languages [11] (i.e., languages containing all unary functions) have been completely classified with respect to exact solvability. In the special case of $\{0, 1\}$-valued languages, which correspond to Max-CSPs, languages on two-element domains [5], three-element domains [8], four-element domains [9], and conservative languages [6] have been classified.

Our contribution is a classification of *all* tractable valued languages as those admitting a binary idempotent and symmetric fractional polymorphism. We also show that this is a polynomial-time checkable condition. Our main result can be stated as follows, where BLP is a linear programming relaxation related to *optimal soft arc consistency* (see Section 3):

Theorem 1.1 *Let Γ be a finite-valued language defined on a domain D.* VCSP(Γ) *is tractable if BLP solves* VCSP(Γ). *Otherwise,* VCSP(Γ) *is NP-hard.*

Overview

In Section 2, we introduce the basic technical notions necessary for presenting our results. In Section 3, we give a characterisation of the power of a linear programming relaxation, and in Section 4, we establish our main dichotomy result. We finish with some open problems in Section 5.

2 Preliminaries

We define three important notions: the expressive power of a valued language, fractional polymorphisms, and the notion of a core valued language.

2.1 Expressive Power

The basic notion for the study of the complexity of constraint satisfaction problems is the *expressive power* and the related reduction.

Definition 2 *For a valued language Γ, we let $\langle\Gamma\rangle$ be the set of all functions $f(x_1,\dots,x_m)$ such that for some instance $I \in \mathrm{VCSP}(\Gamma)$ with objective function $f_I(x_1,\dots,x_n)$, $m \le n$, we have $f(x_1,\dots,x_m) = \min_{x_{m+1},\dots,x_n} f_I(x_1,\dots,x_m,x_{m+1},\dots,x_n)$. We say that Γ* expresses *f and we call $\langle\Gamma\rangle$ the* expressive power *of Γ.*

In other words, the expressive power $\langle\Gamma\rangle$ is the closure of Γ under addition, multiplication by nonnegative constants, and minimisation over extra variables.

Theorem 2.1 ([3, 2]) *Let Γ and Γ' be valued languages with $\Gamma' \subseteq \langle\Gamma\rangle$ with Γ' finite. Then,* $\mathrm{VCSP}(\Gamma')$ *polynomial-time reduces to* $\mathrm{VCSP}(\Gamma)$.

In particular, if Γ expresses some "NP-hard cost function" h, then, by Theorem 2.1, $\mathrm{VCSP}(\{h\})$ reduces to $\mathrm{VCSP}(\Gamma)$, and hence Γ is NP-hard.

2.2 Fractional Polymorphisms

Let Γ be a valued language defined on D. An m-ary *operation* on D is a function $g : D^m \to D$. The set of all m-ary operations on D is denoted by $\mathcal{O}_D^{(m)}$. An m-ary *fractional operation* is a function $\omega : \mathcal{O}_D^{(m)} \to \mathbb{Q}_{\ge 0}$. A (binary) fractional operation ω is called a (binary) *fractional polymorphism* [2] of Γ if $\sum_{g\in\mathcal{O}_D^{(m)}} \omega(g) = 1$ and for every function $f \in \Gamma$ and tuples $\bar{x},\bar{y} \in D^{ar(f)}$, it holds that

$$\sum_{g\in\mathcal{O}_D^{(2)}} \omega(g) f(g(\bar{x},\bar{y})) \le \frac{1}{2}\left(f(\bar{x}) + f(\bar{y})\right),$$

where $g(\bar{x},\bar{y}) = (g(x_1,y_1),\dots,g(x_{ar(f)},y_{ar(f)}))$. General m-ary fractional polymorphisms are defined analogously.

An operation g is *idempotent* if $g(x,\dots,x) = x$. A binary operation g is *symmetric* if $g(x,y) = g(y,x)$. A fractional operation is called idempotent (symmetric) if all operations in $\{g \mid \omega(g) > 0\}$ are idempotent (symmetric).

Example 1 *Let $<$ be a total order on D. A function $f : D^n \to \mathbb{Q}$ is called* submodular *(w.r.t. $<$) if, for every $\bar{x},\bar{y} \in D^n$,*

$$f(\min_<(\bar{x},\bar{y})) + f(\max_<(\bar{x},\bar{y})) \le f(\bar{x}) + f(\bar{y}).$$

Note that this is equivalent to saying that the fractional operation that assigns $\frac{1}{2}$ to each of the binary operations min *and* max *is an idempotent and symmetric fractional polymorphism of the language $\{f\}$.*

2.3 Valued Cores

The concept of a *core* of a constraint language is fundamental in the theoretical study of CSPs. Intuitively, a non-core is a language that can be *squashed* to an equivalent language over a strictly smaller domain. An analoguous concept for valued constraint languages was introduced in [7]. Here, we give a different, but equivalent, definition.

Let $S \subseteq D$. The *sub-language $\Gamma[S]$ of Γ induced by* S is the valued language defined on domain S and containing the restriction of every cost function $f \in \Gamma$ to S.

Definition 3 *A valued language Γ is a* core *if for every unary fractional polymorphism ω of Γ, all operations g with $\omega(g) > 0$ are injective.*

A valued language Γ' is a core of Γ *if Γ' is a core and for some unary fractional polymorphism ω of Γ and some operation h with $\omega(h) > 0$, we have $\Gamma' = \Gamma[h(D)]$.*

Lemma 2.2 *If Γ' is a core of Γ, then $\mathsf{Opt}(I) = \mathsf{Opt}(I')$ for all instances $I \in \mathrm{VCSP}(\Gamma)$, where I' is obtained from I by substituting each function in Γ for its restriction in Γ'.*

For a valued language Γ, let Γ_c denote the language containing all functions that can be obtained from a function in Γ by fixing a (possibly empty) subset of its variables to domain values. For a cost function f, let $\mathrm{argmin} f = \{\bar{x} \mid f(\bar{x}) = \min_{\bar{y}\in D^{ar(f)}} f(\bar{y})\}$.

Proposition 2.3 ([7]) *Let Γ be a core valued language defined on a finite domain D.*

1. *For each $a \in D$, $\langle\Gamma_c\rangle$ contains a unary function u_a such that* $\mathrm{argmin} u_a = \{a\}$.
2. *Γ is NP-hard if, and only if, Γ_c is NP-hard.*

3 The Basic LP Relaxation and OSAC

Let $I \in \mathrm{VCSP}(\Gamma)$ be given as in Definition 1 by a set of variables $V = \{x_1, \ldots, x_n\}$, and an objective function $f_I(x_1, \ldots, x_n) = \sum_{i=1}^{q} w_i \cdot f_i(\bar{x}^i)$.

For a tuple $\bar{x}$, we denote by $\{\bar{x}\}$ the set of elements in $\bar{x}$. The basic LP relaxation has variables $\lambda_{i,\sigma}$ for every $1 \le i \le q$ and $\sigma : \{\bar{x}_i\} \to D$; and variables $\mu_{x,a}$ for every $x \in V$ and $a \in D$.

$$\begin{aligned} \min \quad & \sum_{i=1}^{q} \sum_{\sigma:\{\bar{x}_i\}\to D} w_i \cdot f_i(\sigma(\bar{x}_i)) \cdot \lambda_{i,\sigma} \\ & \sum_{\substack{\sigma:\{\bar{x}_i\}\to D \\ \sigma(x)=a}} \lambda_{i,\sigma} = \mu_{x,a} \quad \forall i, x \in \{\bar{x}_i\}, a \in D \\ & \sum_{a \in D} \mu_{x,a} = 1 \qquad \forall x \in V \\ & 0 \le \lambda, \mu \le 1 \end{aligned} \tag{1}$$

Let $\mathrm{BLP}(I, \Gamma)$ denote the optimum of (1). We say that BLP *solves* $\mathrm{VCSP}(\Gamma)$ if $\mathrm{BLP}(I, \Gamma) = \mathsf{Opt}(I)$ for all instances $I \in \mathrm{VCSP}(\Gamma)$.

Theorem 3.1 ([14, 10]) *Let Γ be a valued language. Then BLP solves $\mathrm{VCSP}(\Gamma)$ if, and only if, Γ has a binary symmetric fractional polymorphism.*

LP relaxations such as (1) as well as semidefinite programming (SDP) relaxations have been popular in the study of the approximation properties of VCSPs (sometimes referred to as *generalised CSPs*) [12]. In the constraint optimisation community, a related relaxation has appeared under the name of *optimal soft arc consistency* (OSAC).

We group the terms of I with respect to their *scope*. Let $S \subseteq V$. The terms of this scope are those of the form $w_i \cdot f_i(\sigma(\bar{x}_i))$, where $\{\bar{x}_i\} = S$. For each scope S, $x \in S$, and $a \in D$, we have a variable $y_{S,x}(a)$. For each $x \in V$, we have a variable z_x.

Establishing optimal soft arc consistency amounts to solving the following linear program [4]:

$$\begin{aligned} \max \quad & \sum_{x} z_x \\ & \sum_{i:\bar{x}_i=\{x\}} w_i \cdot f_i(\sigma(x)) - z_x + \sum_{S:x\in S} y_{S,x}(\sigma(x)) \ge 0 \\ & \qquad \forall x \in V, \sigma : \{x\} \to D \\ & \sum_{i:\{\bar{x}_i\}=S} w_i \cdot f_i(\sigma(\bar{x}_i)) - \sum_{x\in S} y_{S,x}(\sigma(x)) \ge 0 \\ & \qquad \forall S \subseteq V, \sigma : S \to D \end{aligned} \tag{2}$$

By taking the dual of (2), we can compare it to the program (1). It is then clear that the programs are identical except that OSAC only allows one variable per scope and partial assignment, while BLP has distinct variables for each constraint application. This means that OSAC potentially gives a tighter lower bound on the optimal value of the VCSP.

4 A Dichotomy Theorem

Theorem 3.1 covers all previously known tractable cases of valued CSPs, as well as those previously conjectured to be tractable. It is therefore reasonable to ask whether it is also a necessary condition for tractability. We answer this question in the affirmative.

Definition 4 *We say that Γ satisfies the condition (MC) if there exist $a \neq b \in D$ such that $\langle\Gamma\rangle$ contains a binary function h with* $\operatorname{argmin} h = \{(a, b), (b, a)\}$.

The condition (MC) can be shown to imply NP-hardness via a reduction from Max-Cut [3]. Our main result implies that (MC) is the only source of intractability for valued CSPs.

Theorem 4.1 *Let D be an arbitrary finite set and let Γ be a core valued language defined on D.*

- *Either Γ has a binary idempotent and symmetric fractional polymorphism and BLP solves $\mathrm{VCSP}(\Gamma)$;*
- *or (MC) holds for Γ_c and $\mathrm{VCSP}(\Gamma)$ is NP-hard.*

We say that OSAC *solves* $\mathrm{VCSP}(\Gamma)$ if the optimum of (2) is equal to $\mathsf{Opt}(I)$ for all $I \in \mathrm{VCSP}(\Gamma)$. As a direct corollary of Theorem 4.1, we conclude that OSAC is unlikely to add any power when it comes to the exact solvability of valued CSPs.

Corollary 4.2 *Let Γ be a valued language. Assuming that $P \neq NP$, OSAC solves $\mathrm{VCSP}(\Gamma)$ if, and only if, Γ has a binary symmetric fractional polymorphism.*

5 Discussion and Open Problems

We have completely answered the question of which valued constraint languages on finite domains are solvable exactly in polynomial time. In particular, we have characterised the tractable languages as those that admit a binary idempotent and symmetric fractional polymorphism. Moreover, all tractable languages are solvable by the basic linear programming relaxation (BLP), or equivalently, by establishing optimal soft arc consistency (OSAC).

We end this section by listing a number of interesting open problems related to our results and VCSPs in general. The following *meta problems* are relevant to our classification. First, can we determine whether Γ is a core or not? If not, can we find a core of Γ. Second, can we determine whether $\mathrm{VCSP}(\Gamma)$ is tractable or NP-hard? Both of these problems are decidable.

Proposition 5.1 *Given a finite valued language* Γ*, the problem of determining whether* Γ *is a core is decidable, and a core of* Γ *can be found in polynomial time.*

Proposition 5.2 *Given a finite valued language* Γ*, the problem of determining whether* Γ *is tractable or NP-hard is decidable.*

These results rely on solving a linear program with as many inequalities as there are (unary or binary) operations on D. Hence, when D is a part of the input, these linear programs are exponentially large.

Problem 1 *Is it possible to determine, in time polynomial in the representation size of* Γ*, (1) whether* Γ *is a core ; and (2) whether* Γ *is tractable or NP-hard ?*

By Corollary 4.2, OSAC solves VCSP(Γ) if and only if BLP does, provided that P $\neq$ NP. Clearly, though, the power of OSAC should not depend on a deep complexity theoretical conjecture.

Problem 2 *Does BLP solve* VCSP(Γ) *precisely when OSAC does ?*

While fractional polymorphisms can be used to characterise tractable constraints and constraint languages, they are somewhat less useful for describing concrete constraints.

Problem 3 *Find an alternative characterisation of (some subclass of) constraints adimitting a symmetric binary fractional polymorphism.*

For example, a function $f : \{0,1\} \rightarrow \{0,1\}$ is submodular w.r.t. the order $1 < 0$ if and only if it can be represented as a *2-monotone* formula : $f(x_1, \ldots, x_k) = (x_{i_1} \wedge \cdots \wedge x_{i_s}) \vee (\neg x_{j_1} \wedge \cdots \wedge \neg x_{j_t})$ (e.g., [1]).

In practice, as a part of a general branch-and-bound procedure, establishing OSAC or solving the BLP may be prohibitively expensive operations. A weaker consistency notion, called *virtual arc consistency* (VAC) is introduced in [4]. This consistency can be established using a more efficient algorithm.

Problem 4 *Determine a condition on* Γ *that guarantees solvability of* VCSP(Γ) *using VAC (or some other weaker notion of soft arc consistency).*

Références

[1] David Cohen, Martin Cooper, Peter Jeavons, and Andrei Krokhin. Supermodular Functions and the Complexity of MAX-CSP. *Discrete Applied Mathematics*, 149(1-3) :53–72, 2005.

[2] David A. Cohen, Martin C. Cooper, and Peter G. Jeavons. An Algebraic Characterisation of Complexity for Valued Constraints. In *Proceedings of CP'06*, pages 107–121, 2006.

[3] David A. Cohen, Martin C. Cooper, Peter G. Jeavons, and Andrei A. Krokhin. The Complexity of Soft Constraint Satisfaction. *Artificial Intelligence*, 170(11) :983–1016, 2006.

[4] Martin C. Cooper, Simon de Givry, Martí Sánchez, Thomas Schiex, Matthias Zytnicki, and Tomáš Werner. Soft arc consistency revisited. *Artificial Intelligence*, 174(7–8) :449–478, 2010.

[5] N. Creignou. A dichotomy theorem for maximum generalized satisfiability problems. *Journal of Computer and System Sciences*, 51(3) :511–522, 1995.

[6] Vladimir Deineko, Peter Jonsson, Mikael Klasson, and Andrei Krokhin. The approximability of Max CSP with fixed-value constraints. *Journal of the ACM*, 55(4), 2008. Article 16.

[7] Anna Huber, Andrei Krokhin, and Robert Powell. Skew Bisubmodularity and Valued CSPs. In *Proceedings of SODA'13*, pages 1296–1305, 2013.

[8] Peter Jonsson, Mikael Klasson, and Andrei Krokhin. The Approximability of Three-valued MAX CSP. *SIAM Journal on Computing*, 35(6) :1329–1349, 2006.

[9] Peter Jonsson, Fredrik Kuivinen, and Johan Thapper. Min CSP on Four Elements : Moving Beyond Submodularity. In *Proceedings of CP'11*, pages 438–453, 2011.

[10] Vladimir Kolmogorov. The power of linear programming for valued CSPs : a constructive characterization. Technical report, July 2012. arXiv :1207.7213.

[11] Vladimir Kolmogorov and Stanislav Živný. The complexity of conservative valued CSPs. *Journal of the ACM*, 2013. To appear.

[12] Prasad Raghavendra. Optimal algorithms and inapproximability results for every CSP ? In *Proceedings of STOC'08*, pages 245–254. ACM, 2008.

[13] Thomas Schiex, Helene Fargier, and Gerard Verfaillie. Valued constraint satisfaction problems : Easy and hard problems. In *Proceedings of IJCAI'95*, pages 631–637, 1995.

[14] Johan Thapper and Stanislav Živný. The power of linear programming for valued CSPs. In *Proceedings of FOCS'12*, pages 669–678, 2012.

[15] Johan Thapper and Stanislav Živný. The complexity of finite-valued CSPs. In *Proceedings of STOC'13*, 2013. To appear.

Une étude empirique du module d'analyse de conflits dans les solveurs CDCL

Djamal Habet **Donia Toumi**

LSIS, UMR CNRS 7296
Avenue Escadrille Normandie Niemen
13397 Marseille Cedex 20, France
{Djamal.Habet, Donia.Toumi}@lsis.org

Résumé

Les solveurs SAT modernes, basés sur l'architecture CDCL, ont montré leur efficacité à résoudre des instances SAT de grande taille, particulièrement celles issues de l'industrie. Une des principales composantes de ces solveurs est l'analyse de conflits. Elle permet notamment, lors d'un conflit, de fournir un niveau de retour-arrière non chronologique et d'apprendre une clause assertive.

Le conflit analysé correspondant à la première clause falsifiée rencontrée et peu d'intérêt a été accordé au choix de cette clause à analyser. Nous proposons dans cet article, une étude empirique du comportement des solveurs CDCL en fonction de la clause à analyser, selon plusieurs critères comme le niveau de retour-arrière et la qualité de la clause apprise. Le solveur CDCL retenu dans cette étude est glucose.

Abstract

The modern SAT solvers, based on a CDCL scheme, have shown their efficiency on solving large SAT instances, in particular industrial ones. One of the main components of such solvers is the analysis of a conflict, corresponding to a falsified clause, in order to calculate a backjump level and to learn an assertive clause. The considered conflict is the first encountered one and no interest has been awarded to the choice of the falsified clause to analyze. Accordingly, we ask in this paper the question of the behavior of the CDCL solver depending on the falsified clause to analyze. To address this point, we have conducted a large experimental study on the basis of the glucose solver, by measuring its performances according to the quality of the performed backjumps, the quality of the learnt clauses and the needed time to solve an instance.

1 Introduction

Le problème de satisfiabilité (SAT) consiste à décider si une formule booléenne mise sous forme normale conjonctive est satisfiable ou non. L'une des avancées majeures de ces dernières décennies dans la résolution pratique du problème SAT réside dans l'apparition des solveurs modernes basés sur l'architecture CDCL (Conflict Driven Clause Learning) [9, 4, 8]. Ces solveurs ont montré une grande efficacité à traiter des instances, principalement industrielles et crafted, codées sur des milliers, voire des millions de variables et de clauses. Pour s'en convaincre, il suffit de consulter le classement de tels solveurs dans les dernières compétitions internationales sur SAT [1].

Contrairement aux solveurs SAT de type "look-ahead", ceux de type CDCL sont difficiles à analyser et leur comportement est difficile à prédire. En effet, ils sont le résultat d'un subtil mélange de plusieurs composantes, telles que l'apprentissage des clauses [9, 11], des heuristiques dynamiques de choix de variables de branchement (VSIDS [8]), les politiques de redémarrage [6, 7], etc. Toutefois, cette combinaison reste sensible aux changements mêmes légers.

L'une des composantes principales des solveurs CDCL est l'analyse de conflits qui est lancée dès l'apparition d'une clause falsifiée durant le processus de propagation. Cette analyse vise à expliquer l'échec en déterminant l'ensemble des littéraux responsables, et produit par conséquent un "nogood" sous forme d'une clause assertive qui est ajoutée aux clauses initiales, et retourne également un niveau de retour-arrière non chronologique. Notons que l'une des autres clés du suc-

1. www.satcompetition.org

cès des solveurs CDCL est la gestion de la base des clauses apprises afin d'éviter son augmentation exponentielle et de garder seulement les clauses jugées utiles.

Ainsi, l'analyse des conflits est lancée à la première clause falsifiée rencontrée. Mais est-ce que le comportement d'un solveur CDCL change si on continue la propagation, même si un conflit est atteint, et arrêter la propagation en fonction des caractéristiques de la clause falsifiée atteinte ? Peut être analyser la raison de la falsification des clauses initiales est plus pertinent, ce qui semble être intuitif puisque le but est de vérifier la satisfiabilité de la formule originale [2] ? Par conséquent, nous abordons dans cet article la question de la pertinence de la clause à analyser.

Dans une tentative d'une réponse à ces questions, nous avons mené plusieurs études expérimentales sur le solveur *glucose* [2] très efficace, particulièrement dans la résolution des instances industrielles. Tout d'abord, nous analysons la relation entre la clause falsifiée à analyser et la clause apprise résultante. Cette dernière est déterminante dans le processus de résolution car elle sert à calculer le niveau du retour-arrière et ses caractéristiques (taille et nombre de niveaux de décisions des littéraux qu'elle contient) déterminent sa pertinence pour la recherche. Selon les résultats obtenus, nous essayons de prédire quelle clause falsifiée est préférable pour l'analyse. Enfin, nous espérons également que ce travail apporte une contribution à la compréhension du comportement des solveurs CDCL.

Ce papier est organisé comme suit : nous donnons dans la section 2 quelques définitions et notations nécessaires pour la suite de l'article. La section 3 est consacrée à une observation empirique du module d'analyse de conflits. La section 4 tente d'exploiter les différentes constatations faites et nous concluons dans la section 5.

2 Notations et définitions

Une instance $\mathcal{F}$ du problème de satisfiabilité (SAT) est définie par la paire $\mathcal{F} = (\mathcal{X}, \mathcal{C})$, tel que $\mathcal{X} = \{x_1, x_2, \cdots x_n\}$ est un ensemble de variables booléennes (leurs valeurs appartiennent à l'ensemble $\{vrai,\ faux\}$) et $\mathcal{C} = \{c_1, c_2, \cdots c_m\}$ est un ensemble de clauses. Une clause $c_i \in \mathcal{C}$ est une disjonction finie de littéraux et un littéral est soit une variable (x_i) ou sa négation ($\neg x_i$). Une clause est aussi représentée par l'ensemble des littéraux y apparaissant. La clause vide est toujours fausse et notée par $\square$.

2. Notons que dans l'implémentation des solveurs CDCL, la propagation se fait souvent en commençant par les clauses originales de la formule SAT.

Une interpretation $\mathcal{I}$ des variables de $\mathcal{F}$ est définie par un ensemble de littéraux. $\mathcal{F}_{|l}$ dénote la formule $\mathcal{F}$ simplifiée par l'affectation du littéral l à $vrai$. De même, si $\mathcal{I} = \{l_1, \ldots, l_i\}$, $\mathcal{F}_{|\mathcal{I}}$ dénote la formule $\mathcal{F}$ simplifiée par l'affectation des littéraux $l_1, \ldots, l_i$ à $vrai$. La clause $c_j \in \mathcal{C}$ est satisfaite par l'interprétation $\mathcal{I}$ ssi elle contient au moins un littéral qui est dans $\mathcal{I}$, autrement c_j est falsifiée par $\mathcal{I}$. Un modèle de $\mathcal{F}$ est une interprétation complète qui satisfait toutes les clauses de $\mathcal{F}$. Finalement, le problème de satisfiabilité (SAT) consiste à décider si $\mathcal{F}$ admet un modèle. Si c'est le cas alors $\mathcal{F}$ est dite satisfiable, autrement $\mathcal{F}$ est dite insatisfiable.

Les solveurs CDCL (Conflict Driven Clauses Learning) sont une combinaison fine de la procédure DPLL [3] et d'une méthode basée sur le principe de résolution. Leur architecture est basée sur les notions suivantes : l'heuristique de choix de variables VSIDS (Variable State Independant, Decading Sum) [8], l'utilisation de redémarrage [6, 7] et d'une structure de données efficace (ex. Watched literals) [11] ainsi que l'apprentissage de clauses [9].

Algorithme 1: Algorithme CDCL

```
Entrée : F formule CNF, maxconflits.
Sortie : SAT(F est satisfiable) ou UNSAT (F est insatisfiable).
while true do
    // F' = (X, C ∪ L)
    L ← ∅ ; //L est l'ensemble des clauses apprises
    I ← ∅ ;
    dl ← 0 ;
    while (nbconflits < maxconflits) do
        confl ← Unitpropagation(F',I) ;
        if (confl ≠ null) then
            nbconflits + + ;
            if (dl = 0) then return UNSAT ;
            (α, bjl ) ← ConflictAnalysis(confl) ;
            L ← L ∪ {α} ;
            dl ← bjl ;
            CancelUntil(I, bjl) ;
        end
        else
            if (|I| = |V|) then return SAT ;
            dl + + ;
            l ← PickBranchLit(V,{0,1}) ;
            I ← I ∪ {l} ;
        end
    end
    //redémarrage
end
```

À chaque étape de la recherche, l'algorithme sélectionne, via la fonction *PickBranchLit()*, un littéral de décision l (via l'heuristique VSIDS). Ce littéral est ajouté à l'interprétation courante $\mathcal{I}$ ainsi que les littéraux impliqués par application de la fonction *Unitpropagation()* à $\mathcal{F}'_{|\mathcal{I}}$, où $\mathcal{F}' = (\mathcal{X}, \mathcal{C} \cup \mathcal{L})$, et $\mathcal{L}$ est l'ensemble des clauses apprises lors de la recherche. Dès qu'une clause c_i de $\mathcal{F}$ est falsifiée, la fonction *Unitpropagation()* est immédiatement interrompue et retourne en $confl$ cette clause falsifiée c_i. Dans ce cas, une analyse de conflits est appliquée à c_i, un nogood (clause

assertive) α est généré et un niveau de retour-arrière non-chronologique (backjump) bj est défini. L'insatisfiabilité de la formule $\mathcal{F}$ est prouvée si ce conflit est survenu dans la racine de l'arbre de recherche ($dl = 0$), dans ce cas, l'algorithme retourne UNSAT. Dans le cas contraire, la nouvelle clause apprise α est ajoutée à l'ensemble des clauses apprises $\mathcal{L}$ et la fonction *CancelUntil()* effectue un retour-arrière au niveau de décision bj. Un redémarrage est effectué chaque fois que le nombre maximal de conflits autorisés est atteint. Si toutes les variables sont affectées et que toutes les clauses sont satisfaites, l'algorithme retourne SAT, ce qui prouve que la formule $\mathcal{F}$ est satisfiable. L'algorithme réitère ce processus jusqu'à ce que la satisfiabilité ou l'insatisfiabilité de la formule soit prouvée.

Le mécanisme de l'analyse de conflits et d'apprentissage de clauses exigent la mise en oeuvre d'une stratégie de gestion des clauses apprises afin d'identifier et de supprimer les clauses supposées non utiles pour la recherche. Autrement, l'ensemble $\mathcal{L}$ peut augmenter de manière exponentielle. Dans ce cadre, différentes approches de nettoyage de la base de données des clauses apprises ont été étudiés. En particulier dans [2], les auteurs proposent une estimation de la qualité des clauses apprises basée sur une mesure nommée LBD (Literals Blocks Distance). Cette mesure correspond au nombre de niveaux de décision différents apparaissant dans la clause apprise. Plus le LBD d'une clause est faible plus cette clause est jugée utile et les clauses ayant un LBD élevé peuvent être supprimées.

3 Etude expérimentale de l'analyse de conflits

Comme déjà expliqué dans la section précédente, un solveur CDCL effectue l'analyse de conflits sur la base de la première clause falsifiée (que nous noterons par c_f) atteinte durant la phase de propagation. Classiquement, cette analyse se fait selon "the first UIP" [11] en appliquant une succession de résolutions entre les clauses concernées par le conflit. La clause c_f est la première à être utilisée dans cette séquence.

De plus, pour des raisons combinatoires, garder toutes les clauses apprises lors de la recherche s'avère non utile. Par conséquent, certaines clauses apprises sont conservées, et d'autre sont supprimées, selon certains paramètres (activités des clauses, valeurs de leurs LBD, etc.).

Cependant, qu'en est-il de la pertinence des clauses apprises en ce qui concerne l'analyse des conflits ? Est-il pertinent de réaliser l'analyse sur la base de la première clause falsifiée atteinte ? Y a-t-il une différence dans le comportement d'un solveur CDCL si nous préférons l'analyse sur la base d'une clause falsifiée particulière ? A notre connaissance, il n'y a pas de travail dans l'état de l'art des solveurs CDCL qui traite d'une telle étude. Pour tenter de répondre aux questions ci-dessus, nous avons procédé à une large étude expérimentale sur des instances SAT issues de la compétition SAT 2011.

Le solveur de base utilisé dans nos expériences est *glucose 2.1* [2] avec un prétraitement par SatElite [5]. *glucose* est un solveur basé sur minisat [4] avec une gestion plus particulière de la base de clauses apprises basée sur les valeurs de LBD des clauses apprises. Celles avec un petit LBD (≤ 2) sont considérées comme plus utiles que les autres. Par conséquent, une partie des clauses apprises dont la taille et la valeur de LBD sont supérieures à 2 sont supprimées (voir [2] pour plus de détails). Comme n'importe quel solveur CDCL, *glucose* propage les littéraux en file d'attente (correspondant aux propagations unitaires) et arrête la propagation à la première clause falsifiée.

3.1 L'impact du choix de la clause à analyser

Dans cette section, nous étudions empiriquement l'effet de choix de la clause à analyser sur les caractéristiques de la clause assertive générée selon trois critères : sa taille, la valeur de son LBD et enfin le niveau de retour en arrière qu'elle produit. En effet, on préjuge que plus le LBD et la taille d'une clause apprise sont faibles, plus cette clause pourrait être utile pour la recherche. On sait également que plus le niveau du backjump est plus haut dans l'arbre de recherche (petites valeurs de niveau du backjump), plus il est intéressant pour la recherche [1].

Comme décrit précédemment en Algorithme 1, lors d'un conflit, la fonction *UnitPropagation* est interrompue, et retourne comme conflit la première clause falsifiée c_f par l'interprétation courante $\mathcal{I}$. Or, cette même interprétation peut falsifier d'autres clauses. Cependant ces dernières sont ignorées. Ceci est dû principalement à une question de performance.

Dans cette section, nous modifions la fonction *UnitPropagation* afin qu'elle tienne compte, dans le cas d'échecs, de toutes les propagations possibles engendrées par l'ajout de la dernière variable de décision à l'interprétation courante $\mathcal{I}$ et qu'elle détermine l'ensemble de toutes les clauses pouvant être falsifiées, que nous notons par $\mathcal{C}_f$ (donc à chaque conflit k, nous avons un ensemble de clauses falsifiée $\mathcal{C}_f(k)$). Afin de voir l'effet de la clause falsifiée à analyser sur la clause apprise générée, nous appliquons, au conflit k, la procédure *ConflictAnalysis*() à chaque clause falsifiée $cf_i \in \mathcal{C}_f(k)$, et nous enregistrons le niveau backjump bj_i produit par cette analyse, ainsi que la valeur du LBD (lbd_i) et la taille $size_i$ de la clause apprise générée par cette analyse. Le cas $i = 0$ correspond au

premier conflit atteint et classiquement analysé par *glucose*.

3.1.1 Étude 1

D'abord, voyons si, à un conflit k donné, nous pouvons améliorer les caractéristiques de la clause apprise (à savoir sa taille et son LBD ainsi que le niveau du backjump qu'elle produit), en variant la clause c_f à analyser. Ainsi, à chaque conflit k et pour toutes les clauses falsifiées $cf_j,\ j = \{1 \dots |\mathcal{C}_f|\}$, on cherche s'il existe un niveau backjump $bj_i,\ i \in \{1 \dots |\mathcal{C}_f|\}$ tel que $bj_i < bj_0$. Si au conflit k, bj_i existe, on incrémente le compteur $compteur_{bj}$ (intialisé à zéro au début de la recherche), qui compte le nombre de fois où bj_0 est amélioré tout au long de la recherche. A la fin de la recherche, nous calculons $A^+_{bj} = 100 \times (compteur_{bj}/\ \#conflits)\%$, le taux d'amélioration du backjump level bj_0 sur tous les conflits $\#conflits$ rencontrés depuis le début de la recherche. Nous procédons de la même façon pour calculer A^+_{lbd} (le taux d'amélioration de LBD de la clause apprise) et A^+_{size} (le taux d'amélioration de taille de la clause apprise). Nous lançons le solveur *glucose* sur 300 instances issues des dernières compétitions SAT. La recherche est limitée par 10^6 conflits ($\#conflits \leq 10^6$).

La table 1 donne les résultats obtenus par cette première expérimentation sur une sélection d'instances. $\#inst.$ présente le nombre d'instances d'une série. Pour les instances d'une série donnée, b^+, l^+ et s^+ donnent respectivement les moyenne des taux d'amélioration de A^+_{bj}, A^+_{lbd} et A^+_{size}.

Ce tableau montre clairement que b^+ est généralement faible et inférieur à 10% pour le cas des instances crafted, bien qu'il soit légèrement plus élevé pour quelques instances industrielles mais sans dépasser le 19.50%. En revanche, les valeurs pour s^+ et l^+ sont plus dispersées : l^+ est entre 13.56% et 45.40% et le s^+ est entre 16.67% et 78.37%. s^+ donne des valeurs importantes pour certaines types d'instances (comme les séries anton, jarvisalo, kullman, spence/sat, leberre).

3.1.2 Étude 2

Dans l'étude précédente, nous voulions seulement savoir si une amélioration des caractéristiques (bj, $size$, lbd) de la clause apprise, en fonction de la clause falsifiée donnée en entrée, est possible. Nous n'avons pas cherché la meilleure amélioration possible.

Pour le niveau de backjump, cette amélioration pourrait être donnée par une comparaison, au conflit k, à $bj_{min}(k) = min\{bj_i(k)\},\ i = \{1 \dots |\mathcal{C}_f(k)|\}$. Nous rappelons que $bj_i(k)$ est le niveau du backjump produit par l'analyse, au conflit k, de la clause falsifiée $cf_i \in \mathcal{C}_f(k)$. Dans ce qui suit nous voulons savoir si ce bj_{min} est souvent atteint par l'analyse de la première clause falsifiée rencontrée ou non ? nous voulons savoir notamment si ce bj_{min} est atteint souvent par la majorité des autres clauses falsifiées ? La réponse à ces questions fait l'objet de la deuxième étude.

Séries	$\#inst.$	b^+	l^+	s^+
Instances industrielles				
fuhs/AProVE11	10	9,90%	16,60%	22,10%
fuhs/bottom	15	8,60%	31,13%	48,07%
fuhs/top	9	16,44%	38,56%	54,00%
jarvisalo	47	19,50%	40,50%	65,83%
kullmann/128	4	6,50%	20,75%	23,50%
kullmann/32	5	2,40%	22,20%	42,00%
kullmann/64	4	4,25%	30,00%	39,25%
leberre/	17	16,89%	38,44%	60,44%
manthey/	9	7,22%	30,00%	58,11%
Instances crafted				
anton	28	9,74%	43,58%	78,37%
kullmann/G*	10	6,88%	40,75%	58,13%
kullman/V*	26	6,92%	34,00%	65,85%
mosoi/sat	24	4,29%	24,14%	20,57%
mosoi/unsat	6	2,83%	17,67%	16,83%
skvortsov/automata	12	6,67%	23,08%	44,08%
skvortsov/battleship	24	8,08%	40,50%	54,58%
spence/sat	10	8,00%	45,40%	68,80%
spence/unsat	9	1,22%	13,56%	16,67%

TABLE 1 – Taux des améliorations obtenues (par l'analyse d'autres clauses falsifiées au même conflit) des caractéristiques de la première clause apprise (à savoir, bj_0, lbd_0, $size_0$).

Dans cette étude nous lançons *glucose* sur les mêmes séries d'instances présentes dans Table 1. et nous incrémentons, à chaque conflit k, le compteur $compt_{bj0}$ par 1 si $bj_0(k) = bj_{min}(k)$ et le compteur $compt'_{bj}$ par le nombre de clauses cf_i ayant un $bj_i(k) = bj_{min}(k),\ i \in \{1 \dots |\mathcal{C}_f(k)|\}$.

A la fin de la recherche, nous déterminons $E^+_{bj0} = 100 \times (compt_{bj0}/\#conf)\%$, le taux de réussite de cf_0 à produire le bj_{min} et $E^+_{bj} = 100 \times (compt'_{bj}/\#conf)\%$ le taux de réussite des autres clauses falsifiées, au même conflit k, à produire le bj_{min}. De la même manière nous déterminons $E^+_{lbd_0}$, le taux de réussite de cf_0 à produire le lbd_{min} et E^+_{lbd}, le taux de réussite des autres clauses falsifiées à produire le lbd_{min} et également $E^+_{size_0}$, le taux de réussite de cf_0 à produire le $size_{min}$ et E^+_{size}, le taux de réussite des autres clauses falsifiées à produire $size_{min}$.

Les résultats sont présentés dans les tables 2 et 3. $\#inst$ présente le nombre d'instances de la même série, bj^+_0 et bj^+ donnent respectivement la moyenne, sur les instances de la même série, de $E^+_{bj_0}$ et E^+_{bj}. Alors que, lbd^+_0 et lbd^+ donnent respectivement la moyenne, sur les instances de la même série, de E^+_{lbd0} et E^+_{lbd}. Enfin, $size^+_0$ et $size^+$ donnent respectivement la moyenne, sur les instances de la même série, de $E^+_{size_0}$ et E^+_{size}.

Table 2 montre que les valeurs de bj_0^+ sont comprises entre 65% et 90%, ce qui montre que bj_{min} est souvent atteint par l'analyse de cf_0. lbd_0^+ est compris entre 53% et 70% pour les instances industrielles, alors qu'il se situe entre 44% et 57% pour les instances crafted, à l'exception de deux catégories d'instances (Skvortsov/automata 70% et Kullman/V* 29%). Les valeurs pour $size_0^+$ varient généralement entre 33% et 55% avec deux exceptions de 19, 67% pour la série anton et 62% pour la série Kullman/*/128.

Séries	*#inst.*	bj_0^+	lbd_0^+	$size_0^+$
Instances Industrielles				
fuhs/AProVE11	10	65,60%	59,10%	53,80%
fuhs/bottom*	15	82,73%	60,73%	44,13%
fuhs/top*	9	75,44%	53,78%	38,67%
jarvisalo	47	72,10%	53,60%	33,20%
kullmann/128/	4	80,25%	66,00%	62,75%
kullmann/32	5	89,80%	70,40%	51,40%
kullmann/64	4	90,75%	65,00%	55,75%
leberre/	17	76,33%	55,00%	33,33%
manthey/	9	88,00%	65,22%	37,56%
Instances Crafted				
anton	28	88,14%	54,19%	19,67%
kullmann/G*	10	90,00%	56,50%	39,25%
kullman/V*	26	81,92%	29,73%	33,92%
mosoi/sat/	24	66,50%	52,00%	55,25%
mosoi/unsat/	6	65,40%	44,93%	46,13%
skvortsov/automata	12	86,75%	70,92%	50,42%
skvortsov/battleship	24	69,05%	45,05%	48,50%
spence/sat	10	65,40%	46,80%	47,70%
spence/unsat	9	68,00%	57,00%	53,78%

TABLE 2 – Taux de réussite de la première analyse à produire une clause apprise la plus pertinente en terme du niveau du backjump qu'elle définit, son LBD et sa taille.

Table 3 montre que les valeurs de bj^+ sont souvent élevées avec une moyenne de 75, 47% ce qui prouve que le bj_{min} est aussi souvent atteint par l'analyse des autres clauses falsifiées du même conflit. lbd^+ est compris entre 24% et 47% pour les instances industrielles, et entre 20% et 53% pour les instances crafted. Les valeurs pour $size^+$ se situent généralement entre 20% (série kullman/32) et 49% à l'exception de quelques catégories d'instances où le $size^+$ possède des valeurs très basses (comme les séries anton, spence/sat et kullman/64).

3.1.3 Étude 3

Pour raffiner l'étude 2, nous allons nous concentrer sur les cas où les valeurs de lbd_0^+ et $size_0^+$ sont élevées. Nous sélectionnons 20 instances industrielles et 53 instances crafted. Nous lançons *glucose* sur ces instances. Pour chaque instance et à chaque conflit k, nous récupérons les résultats (bj_i, lbd_i et $size_i$, $i = 1 \cdots |\mathcal{C}_f|$) retournés par la fonction *ConflictAnalysis()* sur l'ensemble des clauses $cf_i \in \mathcal{C}_f(k)$. Nous calculons, au conflit k, en ce qui concerne le niveau du backjump, le backjump minimum $bj_{min}(k) = min(bj_i)$, $i = 1 \cdots |\mathcal{C}_f|$, le backjump maximum $bj_{max}(k) = max\{bj_i\}$, la moyenne des backjumps $bj_{moy}(k) = moyennne\{bj_i\}$ et enfin l'écart type entre les backjumps $bj_\sigma(k) = écart_type\{bj_i\}$, $i = 1 \cdots |\mathcal{C}_f|$.

Séries	*#inst.*	bj^+	lbd^+	$size^+$
Instances Industrielles				
fuhs/AProVE11	10	62,90%	47,40%	38,90%
fuhs/bottom	15	77,40%	39,87%	25,33%
fuhs/top	9	71,11%	33,11%	20,89%
jarvisalo	46	64,88%	36,94%	22,94%
kullmann/128	4	76,75%	38,25%	34,75%
kullmann/32	5	87,00%	43,20%	34,80%
kullmann/64	4	86,25%	24,50%	15,75%
leberre/	17	67,44%	33,22%	20,78%
manthey/	9	81,33%	44,00%	21,89%
Instances Crafted				
anton	28	80,28%	32,48%	8,16%
kullmann/G*	10	80,00%	20,00%	20,33%
kullman/V*	26	81,92%	29,73%	33,92%
mosoi/sat/	24	65,08%	45,83%	47,67%
mosoi/unsat/	6	64,47%	39,87%	40,13%
skvortsov/automata	12	83,67%	47,08%	38,92%
skvortsov/battleship	24	77,54%	33,08%	22,21%
spence/sat	10	83,00%	24,40%	8,00%
spence/unsat	9	67,44%	53,22%	49,33%

TABLE 3 – Taux de réussite de l'analyse, sur l'ensemble des clauses falsifiées déterminées à un conflit donné, à produire la clause apprise la plus pertinente en terme du niveau du backjump qu'elle définit, son LBD et sa taille.

Nous procédons de la même manière pour les valeurs de LBD et les tailles des clauses apprises, après analyse, et nous déterminons, au conflit k, le LBD minimum $lbd_{min}(k)$ ainsi que la taille minimale $size_{min}(k)$, le LBD maximum $lbd_{max}(k)$ et la taille maximale $size_{max}(k)$, la moyenne des LBD $lbd_{moy}(k)$ et la moyenne des tailles $size_{moy}(k)$ et enfin l'écart type entre les LBD $lbd_\sigma(k)$ et l'écart type entre les tailles $size_\sigma(k)$).

À la fin de la recherche, soit n le nombre de conflits atteints, nous calculons les paramètres suivants, en ce qui concerne le niveau du backjump :

- $\delta 1_{bj} = \frac{1}{n} \times \sum_{k=1\cdots n} bj_0(k) - bj_{min}(k)$
- $\delta 2_{bj} = \frac{1}{n} \times \sum_{k=1\cdots n} (bj_0(k) - bj_{max}(k))$
- $\delta 3_{bj} = \frac{1}{n} \times \sum_{k=1\cdots n} |bj_0(k) - bj_{moy}(k)|$
- $bj_\sigma = \frac{1}{n} \times \sum_{k=1\cdots n} (bj_\sigma(k))$

De même, nous déterminons en ce qui concerne le LBD de la clause apprise :

- $\delta 1_{lbd} = \frac{1}{n} \times \sum_{k=1\cdots n} lbd_0(k) - lbd_{min}(k)$
- $\delta 2_{lbd} = \frac{1}{n} \times \sum_{k=1\cdots n} (lbd_0(k) - lbd_{max}(k))$
- $\delta 3_{lbd} = \frac{1}{n} \times \sum_{k=1\cdots n} |lbd_0(k) - lbd_{moy}(k)|$
- $lbd_\sigma = \frac{1}{n} \times \sum_{k=1\cdots n} (lbd_\sigma(k))$

Enfin, nous déterminons en ce qui concerne la taille de la clause apprise :

- $\delta 1_{size}=\frac{1}{n} \times \sum_{k=1\cdots n} size_0(k) - size_{min}(k)$
- $\delta 2_{size}=\frac{1}{n} \times \sum_{k=1\cdots n} (size_0(k) - size_{max}(k))$
- $\delta 3_{size}=\frac{1}{n} \times \sum_{k=1\cdots n} |size_0(k) - size_{moy}(k)|$
- $size_\sigma = \frac{1}{n} \times \sum_{k=1\cdots n} (size_\sigma(k))$

Nous représentons les résultats obtenues dans Table 4. $\delta 1^m_{bj}$ (respectivement $\delta 2^m_{bj}$, $\delta 3^m_{bj}$, bj^m_σ, $\delta 1^m_{lbd}$, $\delta 2^m_{lbd}$, $\delta 3^m_{lbd}$, lbd^m_σ, $\delta 1^m_{size}$, $\delta 2^m_{size}$, $\delta 3^m_{size}$, $size^m_\sigma$) indique la moyenne, sur les instances de la même série, de $\delta 1_{bj}$ (respectivement $\delta 2_{bj}$, $\delta 3_{bj}$, bj_σ, $\delta 1_{lbd}$, $\delta 2_{lbd}$, $\delta 3_{lbd}$, lbd_σ, $\delta 1_{size}$, $\delta 2_{size}$, $\delta 3_{size}$, $size_\sigma$).

À un conflit donné k, ces différentes mesures donnent une estimation de la dispersion des différents valeurs de backjump (resp. LBD et taille de la clause apprise) retournées par l'analyse sur toutes les clauses falsifiées.

Les faibles valeurs de $\delta 1_{bj}, \delta 2_{bj}, \delta 3_{bj}$ et bj_σ prouvent que les valeurs des différents niveaux du backjump bj_i, $i = 1\cdots|\mathcal{C}_f|\}$, produits par les différentes clauses falsifiées, au même conflit, sont rapprochées. Aussi, les valeurs pour $\delta 1_{lbd}, \delta 2_{lbd}, \delta 3_{lbd}$ et lbd_σ, sont un peu plus élevées. En revanche, les valeurs pour $\delta 1_{size}, \delta 2_{size}, \delta 3_{size}$ et $size_\sigma$ sont généralement très importantes.

3.1.4 Bilan partiel

Ces résultats empiriques peuvent être interprétés comme suit : tout d'abord, réaliser l'analyse sur la première clause falsifiée est, en général, suffisant pour atteindre le niveau backjump pertinent (cela reste valable pour les instances k-SAT aléatoires, testées également mais non rapportées dans ce papier). En effet, le niveau backjump fourni par la première clause falsifiée est rarement améliorée. Ce résultat est intéressant et pourrait contribuer à expliquer l'efficacité des solveurs CDCL. Deuxièmement, le choix de la clause conflit cf_i peut améliorer dans certains cas le LBD de la clause apprise. Enfin, la taille de la clause apprise varie en fonction de la clause cf_i et peut être amélioré dans la plupart des cas.

Cela nous laisse poser la question suivante : quelle clause falsifiée est la plus intéressante à analyser ? Nous tentons de répondre à cette question dans la section 3.2.

3.2 Identifier la bonne clause à analyser

À ce stade, nous avons montré que le choix de la clause falsifiée à analyser n'a que peu d'effet sur le niveau de backjump, mais peut améliorer le LBD et la taille de la clause apprise. Dans cette section, notre objectif est de tenter de voir s'il est possible de prédire quelle clause falsifiée est la plus appropriée à analyser. Exactement, nous voulons savoir si certaines catégories de clauses sont plus intéressantes à analyser que d'autres.

Nous menons cette étude comme suit : nous lançons *glucose* sur les instances sélectionnées. Nous limitons la recherche à 10^6 conflits. Pour une instance donnée, chaque fois qu'un conflit est atteint, on détermine $\mathcal{C}_f$, bj_{min}, lbd_{min}, $size_{min}$ comme déjà expliqué dans la section 3.1. Ensuite, nous classons les clauses de $\mathcal{C}_f$ comme suit ($lbd(cf)$ est le LBD de la clause cf et $size(cf)$ sa taille) :

- $\mathcal{C}_{f_I}$: l'ensemble des clauses originales falsifiées = $\{ cf_i \in \mathcal{C}_f$ et $cf_i \in \mathcal{C}\}$
- $\mathcal{C}_{f_{L2}}$: l'ensemble des clauses apprises falsifiées dont le LBD est inférieur ou égal à deux = $\{ cf_i \in \mathcal{C}_f$ et $cf_i \in \mathcal{L}$ et $lbd(cf_i) <= 2\}$
- $\mathcal{C}_{f_{L>2}}$: l'ensemble des clauses apprises falsifiées dont le LBD est supérieur strictement à deux = $\{ cf_i \in \mathcal{C}_f$ et $cf_i \in \mathcal{L}$ et $lbd(cf_i) > 2\}$
- $\mathcal{C}_{f_{S2}}$: l'ensemble des clauses falsifiées dont la taille est égal à deux = $\{ cf_i \in \mathcal{C}_f$ et $size(cf_i) = 2\}$
- $\mathcal{C}_{f_{S3}}$: l'ensemble des clauses falsifiées dont la taille est égal à trois = $\{ cf_i \in \mathcal{C}_f$ et $size(cf_i) = 3\}$
- $\mathcal{C}_{f_{S>3}}$: l'ensemble des clauses falsifiées dont la taille est supérieur strictement à trois = $\{ cf_i \in \mathcal{C}_f$ et $size(cf_i) > 3\}$

Nous voulons par ce classement comparer les résultats obtenus par l'analyse sur les clauses originales $\mathcal{C}_{f_I}$ à ceux obtenus par l'analyse sur les clauses apprises. De même, comparer les résultats obtenus par l'analyse sur les clauses de petite taille $\mathcal{C}_{f_{S2}}$ et $\mathcal{C}_{f_{S3}}$ à ceux obtenus par l'analyse sur les clauses de grande taille $\mathcal{C}_{f_{S>3}}$. Également, comparer les résultats obtenus par l'analyse sur les clauses apprises de LBD inférieur ou égal à deux (vu l'intérêt de ces dernières [2]) $\mathcal{C}_{f_{L2}}$ à ceux obtenus par l'analyse sur les clauses apprises dont le LBD est élevé $\mathcal{C}_{f_{L>2}}$.

Pour chaque classe $\mathcal{C}_{f_i}$, $i \in \{I, L2, L > 2, L2, L3, S > 3\}$, et à chaque conflit k nous mesurons :

- $r_{bj}(\mathcal{C}_{f_i})$: nombre de clauses de la classe $\mathcal{C}_{f_i}$ qui ont produit une clause apprise avec un backjump égal à bj_{min}, divisé par le nombre des clauses falsifiées de la même classe.
- $r_{lbd}(\mathcal{C}_{f_i})$ nombre de clauses de la même classe qui ont produit une clause apprise avec un LBD égal à lbd_{min}, divisé par le nombre des clauses falsifiées de la même classe.
- $r_{size}(\mathcal{C}_{f_i})$ nombre de clauses de la même classe qui ont produit une clause apprise avec une taille égal à $size_{min}$, divisé par le nombre des clauses falsifiées de la même classe.

Séries	*#inst.*	$\delta 1^m_{bj}$	$\delta 2^m_{bj}$	bj^m_σ	$\delta 3^m_{bj}$	$\delta 1^m_{lbd}$	$\delta 2^m_{lbd}$	lbd^m_σ	$\delta 3^m_{lbd}$	$\delta 1^m_{size}$	$\delta 2^m_{size}$	$size^m_\sigma$	$\delta 3^m_{size}$
Instances Industrielles													
fuhs/bottom*	15	0,71	-0,51	0,43	1,82	1,15	-3,41	1,62	2,07	3,14	-11,72	5,15	4,92
fuhs/top*	9	1,76	-1,32	1,03	2,82	1,45	-4,57	2,13	2,4	3,24	-12,35	5,15	4,76
javisalo	4	2,27	-1,41	1,27	3,85	1,73	-5,75	2,45	2,46	17,01	-90,50	33,28	18,71
leberre	7	1,46	-1,31	0,84	1,38	2,71	-7,34	3,21	2,90	11,45	-48,85	18,50	12,96
Instances Crafted													
anton	28	0,27	-0,34	0,14	0,24	6,42	-10,24	3,65	3,60	39,52	-260,08	67,09	40,72
kullman/G*	10	0,19	-0,25	0,07	0,09	6,83	-79,3	16,09	5,41	10,64	-105,01	21,69	4,41
kullman/V*	26	0,11	-0,21	0,08	0,13	0,98	-3,71	1,04	0,45	8,17	-11,87	5,06	7,13
skvortsov	8	0,22	-0,17	0,11	0,36	6,37	-12,68	5,45	5,32	12,43	-27,64	10,74	8,66
spence	7	0,15	-0,16	0,09	0,22	2,23	-5,54	2,11	1,36	5,54	-17,24	6,159	2,92

TABLE 4 – Mesures de la dispersion des différents valeurs de backjump (resp. LBD et taille de la clause apprise) retournées par l'analyse sur toutes les clauses falsifiées à un conflit donné k.

Par exemple, si au conflit numéro 500, le $bj_{min} = 9$ et si le nombre de clauses originales falsifiées est $|\mathcal{C}_{f_I}| = 5$, et si après l'analyse de ces 5 clauses falsifiées, seulement deux ont donné un niveau du backjump = 9 et les autres ont donné un niveau du backjump > 9, alors $r_{bj}(\mathcal{C}_{f_I})$ au 500ème conflit est égal à 2/5. A la fin de la recherche, soit n le nombre de conflits atteint, nous considérons :

- $B^+(\mathcal{C}_{f_i}) = \frac{1}{n} \times \sum_{k=0\cdots n} (r_{bj}(\mathcal{C}_{f_i}))$
- $L^+(\mathcal{C}_{f_i}) = \frac{1}{n} \times \sum_{k=0\cdots n} (r_{lbd}(\mathcal{C}_{f_i}))$
- $S^+(\mathcal{C}_{f_i}) = \frac{1}{n} \times \sum_{k=0\cdots n} (r_{size}(\mathcal{C}_{f_i}))$

Ces trois paramètres donnent une estimation sur la bonne catégorie de clauses à prédire pour l'analyser. Plus le $r_{size}(\mathcal{C}_{f_i})$ (par exemple) pour une classe donnée est élevé, plus cette classe est intéressante. En effet, cela prouve que la majorité des clauses de cette classe produit probablement une clause apprise de taille le plus petit possible. Il en est de même pour $r_{bj}(\mathcal{C}_{f_i})$ et $r_{lbd}(\mathcal{C}_{f_i})$.

Les résultats sont récapitulés dans Table 5. Le symbole '-' indique que pour une instance donnée, la classe $\mathcal{C}_{f_i} = \emptyset,\ i \in \{I, L2, L > 2, L3, L > 3\}$. Il est clair que $\forall \mathcal{C}_{f_i},\ i \in \{I, L2, L > 2, L3, L > 3\}$, le bj_{min} est atteint souvent par la majorité des clauses de la même classe $\mathcal{C}_{f_i}$. Nous confirmons encore nos résultats trouvés dans les études précédentes : le choix de la clause falsifiée à analyser n'influe pas sur le niveau du backjump. Les résultats concernant le LBD de la clause apprise sont plus contrastées et diffèrent suivant la catégorie des instances (industrielles ou crafted).

Cependant, concernant la taille des clauses apprises et pour les séries des instances application, les clauses originales ($\mathcal{C}_{f_I}$) sont les plus utiles à produire une clause apprise avec une taille égal à $size_{min}$. Puis (généralement et à l'exception des séries fuhs/AProVE11 et Kullman), on trouve les clauses binaires ($\mathcal{C}_{f_{S2}}$), puis les clauses ternaires ($\mathcal{C}_{f_3}$) (sauf pour les séries Kullman et leberre) et enfin les clauses de taille supérieure à trois ($\mathcal{C}_{f_{S>3}}$). Pour le cas des instances crafted, les résultats sont mitigés et diffèrent suivant les séries.

4 Exploitation des résultats

Dans cette section nous voulons profiter des résultats obtenus dans la section 3.2 où nous avons classé les clauses suivant leur originalité, leur taille et également leur LBD (pour les clauses apprises). Nous proposons alors une approche qui permet de préférer, dans le cas d'échecs, l'analyse sur des clauses qui vérifient certaines caractéristiques. Nous expliquons dans ce qui suit notre approche.

Dans le reste de cet article, les solveurs sont testés sous Linux et sont lancés sur des lames de calcul avec 2 processeurs Intel Xeon 2.4 Ghz et 24 GO de mémoire vive. Le temps limite est fixé à 1800 secondes.

4.1 Critères d'arrêt du processus de propagation lors d'un conflit

Afin d'étudier l'effet de choix de la clause falsifiée à analyser sur les performances d'un solveur CDCL, nous considérons cinq variantes de *glucose* qui diffèrent par les critères d'arrêt de processus de propagation lors d'un conflit.

- *glucose_o* : arrêter la propagation à la première clause falsifiée originale, $c_f \in \mathcal{C}$. L'objectif est de lancer l'analyse lorsqu'une clause de la formule originale est falsifiée.
- *glucose_a* : arrêter la propagation à la première clause falsifiée apprise $c_f \in \mathcal{L}$. L'objectif est de lancer l'analyse lorsqu'une clause apprise est falsifiée.
- *glucose_l2* : arrêter la propagation à la première clause falsifiée apprise dont le LBD est inférieur ou égal à deux, $c_f \in \mathcal{L}$ et $lbd(c_f) \leq 2$.
- *glucose_s3* : arrêter la propagation à la première clause falsifiée de taille égale à trois, $size(c_f) \leq 3$.
- *glucose_l2s3* : arrêter la propagation à la première clause falsifiée apprise dont le LBD est égal à deux ($c_f \in \mathcal{L}$ et $lbd(c_f) \leq 2$) ou arrêter la propagation

Séries	*#inst.*	L^+_{L2}	$L^+_{L>2}$	L^+_I	L^+_{S2}	L^+_{S3}	$L^+_{S>3}$	B^+_{L2}	$B^+_{L>2}$	B^+_I	B^+_{S2}	B^+_{S3}	$B^+_{S>3}$	S^+_{L2}	$S^+_{L>2}$	S^+_I	S^+_{S2}	S^+_{S3}	$S^+_{S>3}$
Instances Industrielles																			
fuhs/AProVE11	10	69,61	63,23	75,08	63,02	72	65,05	85,34	89,4	88,88	81,47	86,62	84,68	58,98	54,35	67,72	52,78	63,91	56,29
fuhs/bottom*	15	43,21	42,48	55,50	53,31	48,96	44,83	81	81,33	85,38	83,96	83,29	82,09	26,24	28,54	41,92	39,36	34,33	29,69
fuhs/top*	9	36,36	35,52	52,34	48,57	41,47	41,12	71,79	72,34	80,34	77,54	75,79	75,50	22,99	23,89	41,48	37,14	31,19	28,81
jarvisalo	46	27,14	25,30	36,92	30,99	29,11	31,74	63,42	63,36	68,45	64,89	63,25	66,57	12,46	12,14	22,95	17,81	16,04	17,55
kullmann	13	43,07	43,66	47,24	35,15	39,74	46,03	87,48	88,07	89,79	85,59	87,35	89,567	35,06	35,49	41,55	29,36	33,69	39,31
leberre	17	33,02	32,09	40,17	36,56	35,47	33,76	67,47	68,94	71,57	68,82	67,29	69,10	20,07	19,93	28,48	25,73	24,80	21,53
manthey	9	45,96	45,80	48,35	48,61	46,83	46,24	83,39	85,10	85,16	85,17	84,42	84,42	24,88	26,54	27,72	26,83	26,27	25,37
Instances Crafted																			
anton	28	37,42	38,79	35,70	36,27	59,71	36,02	81,67	83,17	81,13	81,31	91,05	81,57	13,07	16,35	10,8	11,85	30,68	11,58
kullman/G*	10	22,44	23,31	23,40	-	26,51	23,10	83,82	84,87	84,06	-	83,29	84,06	21,49	21,48	22,15	-	23,84	22,06
kullman/V_3k*	10	26,02	32,34	28,04	-	28,07	28,27	83,51	85,80	84,05	-	84,07	84,18	31,64	29,56	30,30	-	29,47	30,23
kullman/V_pd_3k	16	39,55	39,92	32,36	37,30	32,16	32,7	84,79	85,30	82,70	83,60	82,70	82,85	35,51	36,50	38,20	33,04	36,76	38,15
mosoi/sat	24	-	55,99	75,39	58,17	-	74,07	-	91,93	95,12	85,91	-	95,15	-	55,55	77,26	54,39	-	76,12
mosoi/unsat	6	-	49,78	80,14	41,10	-	77,67	-	93,17	95,19	89,69	-	95,35	-	49,55	80,69	39,88	-	78,30
skvortsov	36	53,49	45,36	46,67	47,12	53,14	43,20	87,39	87,90	87,90	87,77	87,58	87,60	40,241	34,32	38,16	38,66	40,03	34,35
spence	19	48,36	46,20	58,82	28,52	86,41	48,15	90,12	90,21	91,59	85,45	97,13	90,45	36,96	36,83	49,71	13,77	81,96	38,04

TABLE 5 – Estimation des bonnes classes ($\mathcal{C}_{f_i},\ i \in \{I, L2, L > 2, L3, L > 3\}$) de clauses à prédire pour les analyser par rapport à leur utilité à produire une clause apprise la plus pertinente (en terme du niveau du backjump, LBD et taille de cette clause apprise)

à la première clause falsifiée dont la taille est égal à trois ($size(c_f) \leq 3$).

Le choix de ces variantes se fait selon les résultats obtenus dans la section 3.2. La Figure 1 donne les résultats obtenus par tous les solveurs lors de la résolution des instances sélectionnées industriels et crafted (qui étaient utilisées dans Table 1). Il est clair que le choix de la clause falsifiée à analyser modifie le comportement de solveur *glucose*. Voyons maintenant en détails les résultats par catégorie.

Pour les instances de type industriel (Figure 1 à gauche), *glucose* demeure le plus compétitif. *glucose_o* est meilleur que les autres solveurs. *glucose_s3* a des performances proches de *glucose_l2s3*. *glucose_l2* et *glucose_a* sont moins compétitifs. Comme nous pouvons le voir, les résultats ne sont pas surprenants et confortent nos résultats trouvés dans la section 3. Pour les instances crafted (Figure 1 à droite), le premier contraste est que *glucose_a* est le plus mauvais. *glucose_o* possède des performances proches de *glucose_l2s3*. Il semble que réaliser l'analyse sur la première clause originale falsifiée est largement meilleure que celle sur une clause apprise. De Plus, *glucose_l2s3* est légèrement meilleur que les autres variantes (excepté *glucose*).

Après une évaluation de diverses variantes de *glucose*, nous montrons que le choix de la clause sur laquelle on lance la procédure d'analyse du conflit modifie le comportement du solveur CDCL, en modifiant les résultats des solveurs d'origines selon des critères d'arrêt et la catégorie de l'instance. Mais en général, *glucose* reste plus compétitif. Pourquoi ? Est-ce que cela est dû aux sur-coûts imposés par le calcul des propagations supplémentaires ? Ou il est dû aux caractéristiques selon laquelle nous choisissons la clause c_f ? Nous répondons à cette question dans la section suivante.

4.2 Imposer des préférences dans le processus de propagation

Dans cette section, notre objectif est d'imposer des préférences sur les clauses qui apparaissent dans le graphe d'implication et notamment sur les clauses à analyser en cas d'échecs.

Durant la propagation d'un littéral donné l, un solveur CDCL vérifie dans toutes les clauses où le littéral $\neg l$ apparait comme 'watched littéral', si l'affectation du littéral l à vrai ($\neg l$ à faux) cause la falsification ou la propagation d'autres clauses. Soit $\mathcal{W}$ l'ensemble de toutes les clauses où $\neg l$ apparait comme 'watched littéral'. À la même manière de ce qui est déjà fait dans la section 3.2, nous classons les clauses de cet ensemble $\mathcal{W}$ suivant leur originalité, leur taille et leur LBD.

- $\mathcal{W}_I$: l'ensemble des clauses originales = $\{\ c_i \in \mathcal{W}$ et $c_i \in \mathcal{C}\}$
- $\mathcal{W}_A$: l'ensemble des clauses apprises = $\{\ c_i \in \mathcal{W}$ et $c_i \in \mathcal{L}\}$
- $\mathcal{W}_{L2}$: l'ensemble des clauses apprises dont le LBD est inférieur ou égal à deux = $\{\ c_i \in \mathcal{W})$ et $c_i \in \mathcal{L}$ et $lbd(c_i) <= 2\}$
- $\mathcal{W}_{L>2}$: l'ensemble des clauses apprises dont le LBD est supérieur strictement à deux = $\{\ c_i \in \mathcal{W}$ et $c_i \in \mathcal{L}$ et $lbd(c_i) > 2\}$
- $\mathcal{W}_{S2}$: l'ensemble des clauses dont la taille est égal à deux = $\{\ c_i \in \mathcal{W}$ et $size(c_i) = 2\}$

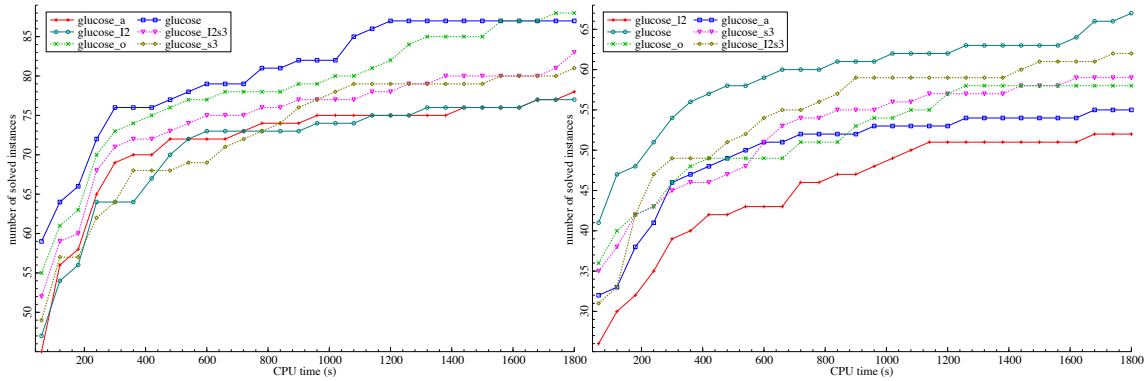


FIGURE 1 – Comparaison des variantes de *glucose* qui diffèrent par les critères d'arrêt de processus de propagation lors d'un conflit.

- $\mathcal{W}_{S3}$: l'ensemble des clauses dont la taille est égal à trois = $\{ c_i \in \mathcal{W}$ et $size(c_i) = 3\}$
- $\mathcal{W}_{S>3}$: l'ensemble des clauses dont la taille est supérieur strictement à trois = $\{ c_i \in \mathcal{W}$ et $size(c_i) > 3\}$

Dans cette section, nous proposons d'imposer un certain ordre sur le traitement de cet ensemble $\mathcal{W}$ afin de propager tout d'abord les clauses que nous jugeons plus utiles. Par exemple, propager d'abord les clauses originales $\mathcal{W}_I$ (tant que c'est possible), puis les clauses apprises $\mathcal{W}_A$. De cette manière, nous favorisons, d'une part, l'apparition dans le graphe d'implication des clauses jugées plus utiles. D'autre part, lors d'un conflit causé par l'affectation du littéral l, nous vérifions (d'abord et sans sur-coûts supplémentaires) si cette affectation cause la falsification de clauses 'jugées plus utiles' afin de lancer l'analyse de conflits sur de telles clauses. Nous considérons sept variantes du glucose :

- *glucose_o* : propager d'abord les clauses originales $\mathcal{W}_I$.
- *glucose_a* : propager d'abord les clauses apprises $\mathcal{W}_A$ puis les originales $\mathcal{W}_I$.
- *glucose_al* : si un conflit survient, continuer toutes les propagations possibles et calculer l'ensemble de toutes les clauses falsifiées $\mathcal{C}_f$ puis choisir la clause à analyser c_f aléatoirement parmi cet ensemble.
- *glucose_l2* : tant qu'un conflit n'est pas atteint, propager d'abord les clauses apprises dont le LBD est inférieur ou égal à deux ($\mathcal{W}_{L2}$), puis propager le reste des clauses de $\mathcal{W}$.
- *glucose_s3* : tant qu'un conflit n'est pas atteint, propager d'abord les clauses dont la taille est égal à trois ($\mathcal{W}_{S3}$) puis le reste des clauses de $\mathcal{W}$.
- *glucose_l2s3* : tant qu'un conflit n'est pas atteint, propager d'abord les clauses dont le LBD est inférieur ou égal à deux ($\mathcal{W}_{L2}$) puis les clauses dont la taille est égal à trois ($\mathcal{W}_{S3}$). Enfin, propager le reste des clauses de $\mathcal{W}$.
- *glucose_s3l2* : tant qu'un conflit n'est pas atteint, propager d'abord les clauses dont la taille est égal à trois ($\mathcal{W}_{S3}$) puis les clauses dont le LBD est inférieur ou égal à deux ($\mathcal{W}_{L2}$). Enfin, propager le reste des clauses.

Notons que *glucose* impose, dans $\mathcal{W}$, la propagation des clauses binaires $\mathcal{W}_{S2}$, ce que nous gardons encore dans toutes ces versions. Ainsi la classe des clauses binaires $\mathcal{W}_{S2}$ est traitée toujours avant toute autre classe.

La Figure 2 donne les résultats obtenus par tous les solveurs lors de la résolution des instances sélectionnées industriels et crafted. Pour la catégorie des instances de type industriel (Figure 2 à gauche), on peut remarquer que *glucose* et *glucose_l2s3* ont approximativement les mêmes performances. *glucose_s3l2* est le plus mauvais. Les résultats des autres solveurs sont proches. Pour la catégorie des instances crafted, les résultats sont donnés dans Figure 2 à droite. Ces résultats montrent que *glucose_l2s3* améliore quelque peu *glucose*. Pour les autres solveurs, les courbes sont similaires et les résultats sont proches, sauf pour *glucose_s3l2*.

Ces résultats montrent que préférer l'analyse et l'apparition dans le graphe d'implications, de certaines clauses modifie et, dans certain cas (*glucose_l2s3*), améliore légèrement les performances de *glucose*.

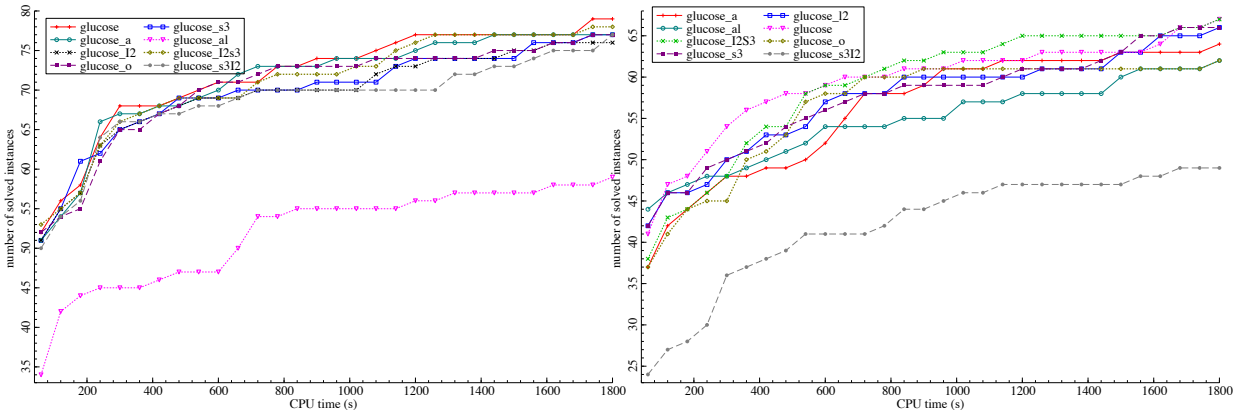


FIGURE 2 – Comparaison des variantes de *glucose* qui diffèrent par l'ordre de traitement des propagations dans l'ensemble $\mathcal{W}$.

5 Conclusion

L'un des objectifs de ce travail est de tenter d'apporter une contribution à la compréhension du comportement des solveurs CDCL. Nous avons concentré cette étude sur leur module d'analyse de conflits, notamment sur le choix de la clause falsifiée à analyser. Les observations empiriques confirment que celui-ci modifie sensiblement le comportement du solveur considéré dans cet article, *glucose 2.1*. Par ailleurs, les expérimentations ont montré que le niveau de retour-arrière non-chronologique n'est pas très perturbé par le choix de la clause à analyser. Pour les autres critères observés (LBD et taille des clauses produites après analyse), il est difficile de tirer une conclusion globale, à part de constater qu'effectivement ils varient significativement suivant les caractéristiques de la clause falsifiée. L'exploitation faite des différentes constations reste préliminaire et son amélioration constitue l'une des perspectives envisageables. Enfin, ce travail vient aussi confirmer la difficulté pratique à analyser les solveurs CDCL.

Références

[1] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour et Lakhdar Sais. A generalized framework for conflict analysis. *Proceedings of Eleventh International Conference on Theory and Applications of Satisfiability Testing*, pages 21–27, 2008.

[2] Gilles Audemard et Laurent Simon. Predicting learnt clauses quality in modern sat solvers. *Proceedings of the 21st international joint conference on Artificial intelligence*, pages 399–404, 2009.

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

[4] N. Eén et N. Sörensson. An extensible sat-solver. *Proceedings of 6th International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[5] Niklas Eén et Armin Biere. Effective preprocessing in sat through variable and clause elimination. *Proceedings of 8th International Conference on Theory and Applications of Satisfiability Testing*, pages 61–75, 2005.

[6] Carla P. Gomes, Bart Selman et Henry A. Kautz. Boosting combinatorial search through randomization. *Proceedings of the fifteenth national/tenth conference on Artificial intelligence/Innovative applications of artificial intelligence*, pages 431–437, 1998.

[7] Jinbo Huang. The effect of restarts on the efficiency of clause learning. *Proceedings of the 20th international joint conference on Artificial intelligence*, pages 2318–2323, 2007.

[8] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. *Annual ACM IEEE Design Automation Conference*, pages 530–535, 2001.

[9] João P. Marques Silva and Karem A. Sakallah. Grasp : A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5) : 506–521, 1999.

[10] Hantao Zhang. Sato : An efficient propositional prover. *Proceedings of the 14th International Conference on Automated Deduction*, pages 272–275, 1997.

[11] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz et Malik. Efficient conflict driven learning in a boolean satisfiability solver. *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285.

Sélection adaptative d'opérateurs en recherche locale et Progressive Party Problem

Nadarajen Veerapen[1] **Youssef Hamadi**[2,3] **Frédéric Saubion**[1]

[1] LUNAM Université, Université d'Angers, LERIA, Angers, France
[2] Microsoft Research, Cambridge, United Kingdom
[3] LIX Ecole Polytechnique, F91128 Palaiseau, France
prenom.nom@univ-angers.fr youssefh@microsoft.com

Résumé

Cet article[1] présente l'utilisation de la sélection adaptative d'opérateurs dans le contexte de la recherche locale afin de résoudre un problème de satisfaction de contraintes, plus précisément le Progressive Party Problem. Les opérateurs sont sélectionnés en fonction de leur utilité. Celle-ci est calculée à partir de la qualité des solutions obtenues et de la distance de la solution candidate par rapport à la trajectoire de recherche. Nous montrons que l'utilisation de plusieurs opérateurs indépendants du problème permet d'atteindre des taux de résolutions comparables à ceux d'un algorithme ayant été adapté au problème avec toutefois des temps de résolution plus longs.

1 Introduction

Il existe de nos jours un grand nombre d'outils et de techniques pour résoudre une large gamme de problèmes d'optimisation et de satisfaction de contraintes. Les métaheuristiques ont largement été utilisées afin de résoudre de tels problèmes. Toutefois, elles sont souvent réglées pour un seul problème ou une unique classe de problèmes et peuvent être difficiles à adapter et à régler pour un nouveau problème, en particulier si l'utilisateur ne possède pas un savoir approfondi du problème ainsi que de la méthode de résolution. La tentative de concevoir des métaheuristiques de plus en plus efficaces résulte en des systèmes très complexes requérant de nombreux paramètres et un réglage pointu.

De base, les algorithmes de recherche locale (LS) appliquent des opérateurs à une configuration du problème afin de trouver une solution. Les opérateurs réalisent des mouvements d'une configuration actuelle vers un de ses voisins, construisant progressivement une trajectoire de recherche. La caractéristique clé des algorithmes LS est donc la gestion de la trajectoire de recherche, qui doit être capable de suffisamment explorer l'espace de recherche mais aussi d'exploiter les zones prometteuses afin d'atteindre des solutions optimales. Les paramètres des algorithmes LS sont utilisés afin d'ajuster le compromis entre exploration et exploitation (aussi appelés diversification et intensification). La plupart du temps, les paramètres sont difficiles à régler. Selon les taxonomies classiques, le réglage de paramètres consiste à apprendre les bonnes valeurs des paramètres d'après des expériences préliminaires tandis que le contrôle de paramètres se réfère à l'ajustement en ligne de ces valeurs.

Récemment, des stratégies de contrôle de haut niveau ont été proposées avec l'objectif de rendre les techniques d'optimisation plus simples pour l'utilisateur [3]. Parmi un nombre croissant de publications, ces approches ont été présentées dans [10]. Les stratégies de sélection d'opérateurs ont été principalement étudiées récemment dans le contexte des algorithmes génétiques [6, 5]. En reprenant certaines des idées de ces publications afin de les appliquer à la LS, nous avons présenté dans [20, 19] un cadre général afin de contrôler dynamiquement le processus de recherche d'un algorithme LS visant les problèmes d'optimisation pouvant être représentés par des permutations.

Cet article a trois objectifs principaux :

- intégrer à notre méthode de résolution la possibilité de résoudre des problèmes de satisfaction de contraintes ;
- prendre en compte des fonctions objectif plus complexes, c.-à-d. des fonctions de pénalité, tout en simplifiant la tache de l'utilisateur en agrégeant

1. Cet article a été accepté en version anglaise à CEC 2013.

ces fonctions de façon simple ;
– essayer de concurrencer les algorithmes LS existants.

Dans cet article nous prolongeons notre framework afin de gérer les problèmes de satisfaction de contraintes (CSP) en variables entières. Les frameworks de programmation par contraintes permettent à l'utilisateur de bénéficier de contraintes puissantes prédéfinies qui sont gérées par un solveur de type boîte noire. La tâche de l'utilisateur est donc centrée sur le modèle, le processus étant très déclaratif. Les solveurs classiques de programmation par contraintes sont basés sur des techniques de propagation de contraintes et sur l'exploration arborescente de l'espace de recherche. Des techniques spécifiques de propagation ont été ajoutées progressivement afin de gérer des contraintes plus complexes, les contraintes globales. Néanmoins, les solveurs complets peuvent atteindre leurs limites face à de grandes ou de très difficiles instances. Puisque les algorithmes LS se sont montrés efficaces pour résoudre les problèmes d'optimisation combinatoire, des frameworks basés sur les contraintes ont donc été proposés afin d'améliorer la conception d'algorithmes LS en programmation par contraintes. En recherche locale basée sur les contraintes (CBLS), l'espace de recherche correspond au produit cartésien des valeurs des domaines des variables – les configurations sont en effet des affectations de valeurs aux variables – et des fonctions de pénalité sont requises afin de calculer le degré de violation de chaque contrainte. Les techniques de consistance locale peuvent être utilisées pendant l'exploration du voisinage d'une affectation donnée. La combinaison de toutes les fonctions de pénalité donne la fonction objectif qui devra être minimisée par la procédure LS.

2 Contexte technique

Bien entendu, la programmation par contrainte (CP) est le paradigme *de facto* pour résoudre les CSP lorsque l'on parle des méthodes complètes. Dans cet article, nous utilisons Comet [18] qui est un langage de programmation CBLS permettant la manipulation des contraintes et de leurs violations de façon transparente puisque les mécanismes internes, comme la mise à jour des invariants, sont cachés. L'utilisateur garde toutefois la possibilité de modifier le comportement de ces mécanismes s'il le souhaite. Au delà de la CP et de la CBLS, la gestion des contraintes a également été étudiée dans le contexte des algorithmes évolutionnaires [4].

2.1 Problèmes de satisfaction de contraintes et recherche locale

Dans les CSP, l'objectif est de trouver une solution satisfaisant toutes les contraintes du problème. Il existe aussi le cas où certaines contraintes, dites dures, ne doivent pas être violées et où d'autres contraintes, dites faibles, doivent être satisfaites seulement si cela est possible. Nous ne nous intéressons ici qu'au premier cas.

Definition 1 *Un* problème de satisfaction de contraintes (CSP) *est défini par un triplet* $(X, \mathcal{D}, \mathcal{C})$ *où* $X = \{x_1, \ldots, x_m\}$ *est un ensemble fini de variables,* $\mathcal{D} = \{\mathcal{D}_{x_1}, \ldots, \mathcal{D}_{x_m}\}$ *est l'ensemble des domaines associés et* $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_p\}$ *est un ensemble de p contraintes.*

Une solution d'un CSP est une affectation aux variables des valeurs de leur domaine satisfaisant toutes les contraintes. Typiquement, cette affectation est représentée par un tuple des valeurs affectées. L'espace de recherche $\mathcal{S}$ est donc le produit cartésien des domaines $\mathcal{S} = \Pi_{i=1}^{m} \mathcal{D}_{x_i}$.

Afin de gérer les contraintes dans un contexte LS, c.-à-d. comme un problème d'optimisation, il est nécessaire de définir des fonctions de violation de contraintes, ou de pénalité, pour chaque contrainte du problème correspondants à une fonction *eval* à minimiser. Ces fonctions expriment le degré de violation de chaque contrainte. En les combinant, il est possible d'obtenir une fonction pouvant être utilisée comme fonction objectif de la procédure LS. En effet, une fois que la fonction de pénalité du CSP a été déterminée, il s'agit d'un problème d'optimisation. La définition des fonctions de pénalité pour chaque contrainte n'est pas une chose triviale. [2]

Dans sa forme la plus simple, une fonction de pénalité d'une contrainte donnée peut simplement être sa valeur de vérité, c.-à-d. *vraie* ou *fausse*, correspondant aux valeurs 0 et 1 respectivement. En général, il est plus intéressant de considérer des fonctions à la granularité plus fine en tenant, par exemple, compte des valeurs des variables et de leur sémantique. Ainsi pour la contrainte globale `alldifferent`, qui est vraie lorsque toutes les variables sont différentes, une contrainte de pénalité pourrait compter le nombre d'inégalités entre paires de variables ou compter le nombre de valeurs apparaissant plus d'une fois.

Une granularité fine autorise une procédure de recherche plus ciblée. Prenons l'exemple le plus extrême où la fonction de pénalité d'un problème retourne systématique 1 lorsqu'au moins une contrainte est vio-

2. Notons que sans fonctions de pénalité, l'espace de recherche se résumerait uniquement aux solutions réalisables.

lée. L'espace de recherche ressemblerait vraisemblablement à un large plateau ponctué de petites dépressions là où une solution est réalisable. Cette topographie est extrêmement défavorable aux algorithmes de LS puisqu'il n'y a pas de gradient pour guider la recherche vers un optimum local.

Lorsque l'on considère un système de contraintes $\mathcal{C} = \{\mathcal{C}_1, \ldots, \mathcal{C}_p\}$, les fonctions de pénalité de chaque contrainte du systèmes doivent être combinées afin d'obtenir une fonction de pénalité globale pour tout le système. En général, cette combinaison peut être la somme, ou bien la valeur maximum, des fonctions de pénalité. De plus, on peut également souhaiter que toutes les contraintes n'aient pas la même importance. Une fonction de pénalité peut donc être exprimée par

$$eval_{\mathcal{C}}(s) = \sum_{c_i \in \mathcal{C}} \alpha_i v_{c_i}(s) \quad (1)$$

où α_i est un poids et $v_{\mathcal{C}_i}(s)$ est la fonction de pénalité de $\mathcal{C}_i$ pour l'affectation s.

La LS applique des opérateurs de mouvement dans le but d'atteindre une solution optimale dictée par une fonction objectif. Étant donnée une configuration dans $\mathcal{S}$, un opérateur génère une nouvelle configuration. Simplement, un opérateur est une combinaison d'un voisinage et d'une fonction de sélection. Le voisinage est un ensemble de configurations pouvant être atteintes depuis une configuration donnée. La fonction de sélection choisit ensuite une configuration parmi le voisinage selon un certain critère.

Dans sa forme la plus simple, une métaheuristique LS applique un opérateur en boucle comme illustré dans l'Algorithme 1 où la méthode `DoSomething()` est spécifique à la métaheuristique utilisée, par exemple une perturbation ou l'application de prohibitions.

```
s ← initial configuration;
s* ← s;
while end condition not met do
    s ← op(s);
    if eval_C(s) < eval_C(s*) then s* ← s;
    DoSomething();
end
return s*
```

Algorithm 1: Squelette algorithmique d'une métaheuristique LS (pour la minimisation).

Definition 2 *Une* relation de voisinage *est une relation binaire irréflexive $\mathcal{N} \subseteq \mathcal{S}^2$ sur l'espace de recherche. Généralement, elle est symétrique.*

Definition 3 *Un* sélecteur *est une fonction qui permet une sélection dans un voisinage. Celle-ci peut être guidée par l'ordre induit par $eval_{\mathcal{C}}$ et est définie par $\sigma : \mathcal{S} \times 2^{\mathcal{S}^2} \mapsto \mathcal{S}$ (dans ce cas la sélection ne retourne qu'un voisin et $2^{\mathcal{S}^2}$ est l'ensemble des parties de $\mathcal{S}^2$), telle que $(s, \sigma(s, \mathcal{N})) \in \mathcal{N}^=$ (la clôture réflexive de $\mathcal{S}$ afin d'inclure l'identité).*

Definition 4 *Un* opérateur *est défini par une paire $(\mathcal{N}, \sigma)$.*

2.2 Gestion d'opérateurs pour la recherche locale

Un bon algorithme de LS est capable d'explorer efficacement l'espace de recherche afin de trouver une solution optimale. Ceci implique de trouver un compromis entre intensification et diversification. Nous essayons d'atteindre cet objectif en contrôlant les opérateurs qui dirigent la solution dans l'espace de recherche.

Un opérateur est choisi parmi un ensemble d'opérateurs disponibles et appliqué à chaque itération. La probabilité qu'un opérateur soit sélectionné est déterminée par ses performances passées en termes de qualité et de diversité. Nous utilisons le terme *utilité* afin de représenter le bénéfice attendu de l'application d'un opérateur donné à un état donné de la recherche. Notons que plusieurs méthodes de sélection d'opérateurs ne se basent que sur la qualité, toutefois [12, 11, 19] ont démontré que le fait de considérer aussi la diversité permet de mieux gérer le compromis intensification-diversification.

La fonction objectif est utilisée afin de mesurer la qualité, et le changement relatif de qualité est pris en compte. Plus précisément, soit le chemin $P_{0,n} = (s_0, \cdots, s_n)$, la qualité de ce chemin correspond à la moyenne du changement de qualité des W dernières configurations $s_{n+1-W}, \cdots, s_n$. W est la taille de la fenêtre glissante nous permettant d'observer la progression de la recherche. Mesurer la diversité est moins évident.

La diversité est un concept commun des algorithmes évolutionnaires mais plus difficile à appréhender en LS. Nous considérons un ensemble de solutions composé des dernières solutions du chemin de recherche (dans une fenêtre glissante). Nous mesurons ensuite la différence entre la solution candidate courante $c = op(s)$ et ces solutions. Cette distance est calculée en fonction des paires variable-valeur : si le nombre d'occurrences des paires variable-valeur de la solution candidate est faible alors la distance sera grande. Ceci est formalisé dans l'équation suivante. Soient n le nombre de variables et $P_{i,j}$ le chemin commençant à l'itération i et se terminant à j, $i \leq j$, alors

$$d(c, P_{i,j}) = \frac{1}{n} \times \sum_{k_c=1}^{n} \left(1 - \frac{occ(P_{i,j}, (k_c, x_{k_c}))}{|P_{i,j}|}\right) \quad (2)$$

où $occ(P_{i,j},(a,b))$ correspond au nombre de fois où la paire variable-valeur (a,b) apparait dans $P_{i,j}$. Par exemple pour $(2,1,3)$, les paires seront $(1,2)$, $(2,1)$ et $(3,3)$. A nouveau, la taille de la fenêtre glissante est W.

Un opérateur est représenté par une paire (q,d) dans l'espace qualité-diversité correspondant à son utilité, c.-à-d. la variation en terme de qualité et de diversité ayant été induite dans le chemin de recherche.

Nous décrivons brièvement les techniques de gestion d'opérateurs que nous testerons sur le Progressive Party Problem dans cet article.

2.3 Approche par Pareto dominance

Dans [20] nous proposons de combiner les moyennes de la qualité et de la diversité en utilisant le nombre d'opérateurs Pareto-dominés comme mesure d'utilité. Une sélection proportionnelle à l'utilité est ensuite effectuée. Nous utiliserons PP pour désigner cette approche.

Soient u et v deux vecteurs de même cardinalité p. Dans un contexte de maximisation, u *domine* v si $u_k \geq v_k, \forall k \in \{1,\ldots,p\}$ avec au moins une inégalité stricte. Ceci est communément appelé la Pareto dominance et dénotée par $u \succ v$.

L'utilité basée sur la Pareto dominance, U_P, d'un opérateur o parmi un ensemble O est alors définie par

$$U_P(o) = |\{o' | o' \in O, o \succ o'\}| + \epsilon \quad (3)$$

où ϵ est une valeur positive interdisant une valeur nulle. Dans la cas présent, $p = 2$ car il y a deux objectifs, la qualité et la diversité.

Nous maintenons une estimation de l'utilité de chaque opérateur $U_{Po,t}$ à chaque itération t. La probabilité proportionnelle de sélection est alors

$$p_{o,t} = \frac{U_{Po,t}}{\Sigma_{k=1}^{|O|} U_{Pk,t}} \quad (4)$$

où $p_{o,t}$ la probabilité de sélection de l'opérateur o à l'itération t. Notons que généralement une probabilité minimale p_{min} est utilisée afin d'interdire une probabilité nulle. Ceci n'est pas nécessaire étant donné ϵ le paramètre de U_P.

Une alternative à PP est PAP qui utilise la même mesure d'utilité mais remplace la sélection proportionnelle par un mécanisme de poursuite adaptative [17]. Celui-ci est basé sur une stratégie du gagnant remporte tout : la probabilité de sélection du meilleur opérateur courant est grandement augmentée tandis que la probabilité des autres opérateurs est grandement diminuée :

$$p_{o,t} = \begin{cases} p_{o,t-1} + \beta(p_{max} - p_{o,t-1}) & \text{si } o = o_t^* \\ p_{o,t-1} + \beta(p_{min} - p_{o,t-1}) & \text{sinon} \end{cases} \quad (5)$$

où

$o_t^* = \text{argmax}_{o=1\ldots k}\{U_{Po,t}\}$ et $p_{max} = 1 - (k-1)p_{min}$

2.4 Approche adaptative

Dans [19] nous proposons un mécanisme adaptatif afin de faire varier le compromis entre les deux critères, la qualité et la diversité.

Soient o_1 et o_2 deux opérateurs, ayant chacun une paire qualité-distance correspondante (q_1,d_1) et (q_2,d_2), le déplacement rectilinéaire pondéré de o_1 à o_2 est alors défini par

$$d_\alpha(o_1,o_2) = \alpha(d_1 - d_2) + (1-\alpha)(q_1 - q_2) \quad (6)$$

À partir de cette métrique, l'utilité $U_\alpha^{\Sigma+}$, correspondant à la somme des déplacements positifs vers tous les autres opérateurs, est définie par

$$U_\alpha^{\Sigma+}(o) = \sum_{i=1}^{n} max(0, d_\alpha(o,o_i)) \quad (7)$$

Une stratégie de diversité « correcte » (CD) est utilisée afin de faire varier le paramètre α. Cette stratégie a été introduite pour les algorithmes évolutionnaires dans [13]. Dans la stratégie CD, l'évaluation de la diversité de la population est basée sur la qualité des configurations de la population. Quand plus de T_{max} configurations possèdent la même qualité, on suppose que la population est trop homogène. La diversité commandée est donc incrémentée d'un pas s_{inc}. De façon équivalente, quand moins de T_{min} solutions ont la même qualité, la diversité commandée est décrémentée d'un pas s_{dec} car l'on suppose que l'exploration est faible puisque la population n'est pas en mesure de se maintenir autour d'optima de bonne qualité.

Nous dénoterons l'approche utilisant CD avec $U_\alpha^{\Sigma+}$ et la sélection proportionnelle par CDP.

3 Progressive Party Problem

Le Progressive Party Problem (PPP) [16] est un CSP d'emploi du temps. Le PPP est un problème régulièrement utilisé dans des publications présentant des systèmes CBLS [7, 18, 14]. Le contexte apparemment frivole du PPP est un événement mondain pendant une régate. Certains bateaux sont désignés comme hôtes et les équipages des autres bateaux doivent leur rendre visite pendant un certain de nombre de périodes de temps. En sus des contraintes de capacité, un équipage ne peut rendre visite plus d'une fois à un bateau hôte et deux équipages ne peuvent se rencontrer plus d'une fois. Différentes instances du PPP sont créées en sélectionnant différents hôtes et en changeant le

nombre de périodes. [3] Afin que chaque équipage puisse rencontrer un maximum d'autres équipages plusieurs contraintes doivent être observées.

Ce CSP peut être modélisé par $B = (b_{i,j})$ une matrice entière de taille $|G| \times |P|$ où G est l'ensemble des équipages, P est l'ensemble des périodes, $b_{g,p}$ est le bateau hôte sur lequel se trouve l'équipage du bateau g à la période p, le vecteur $E = \{e_1, \ldots, e_{|G|}\}$ représente la taille de chaque équipage invité et le vecteur $C = \{c_1, \ldots, c_{|H|}\}$, où H est l'ensemble des bateaux hôtes, représente la capacité d'accueil de chaque bateau. Dans un modèle basé sur les contraintes globales, le problème est alors donné par :

$$\forall g \in G, \quad \texttt{alldifferent}(\{b_{g,p} | p \in P\}) \tag{8a}$$

$$\forall p \in P, \quad \texttt{multiknapsack}(\{b_{g,p} | g \in G\}, E, C) \tag{8b}$$

$$\forall i, j \in \{1, \ldots, I\}, j > i, \quad \texttt{atmost}(1, \{b_{i,p} = b_{j,p} | p \in P\}) \tag{8c}$$

La contrainte $\texttt{alldifferent}(\{b_{g,p} | p \in P\})$ vérifie qu'un équipage invité ne rende pas deux fois visite au même hôte. La contrainte $\texttt{multiknapsack}(\{b_{g,p} | g \in G\}, E, C)$ vérifie que, pour chaque période, le nombre d'invités à bord correspond à la capacité du bateau. Finalement, la contrainte $\texttt{atmost}(1, \{b_{i,p} = b_{j,p} | p \in P\})$ spécifie que deux équipages ne peuvent se rencontrer plus de deux fois.

Le PPP fut introduit [16] pour démontrer la puissance de la programmation par contrainte par rapport à la programmation linéaire en nombres entiers. Plus tard le PPP fut également utilisé afin d'illustrer la performance accrue des approches CBLS tabou [7, 18]. Dans [15] des approches CP classiques sont comparées avec [18] qui était – et, que nous sachions, demeure – la meilleure approche CBLS. Nous utiliserons donc [18] à titre de comparaison.

Un aspect intéressant du PPP est qu'il est composé de différents types de contraintes globales et, donc, de fonctions de pénalités potentiellement différentes. Ces fonctions doivent être agrégées en une seule fonction, ce qui est fait en utilisant des pondérations. Ceci nous permet de tester la sélection d'opérateur avec ou sans pondération.

Toutes les méthodes testées utilisent le même modèle (tel que donné dans l'Expression 8). Nous utiliserons toutefois deux versions du modèle selon le contexte : un modèle avec des pondérations et l'autre sans. Lorsque les pondérations sont utilisées, les contraintes `alldifferent` (Expression 8a) et `multiknapsack` (Expression 8b) ont un poids de 2.

3. Plus de détails à propos du PPP sont disponibles sur sa page CSPLIB (prob013) [8].

Notons que ces poids sont apparus pour la première fois dans [7]. Les auteurs choisirent ces valeurs de façon empirique. Ces poids ont été réutilisés, de façon identique, dans les travaux de Van Hentenryck et Michel, en particulier dans [18].

Puisque nous réutilisons les fonctions de pénalité par défaut de COMET, le problème se présente de la façon suivante :

$$\begin{aligned} \min \quad & 2 \sum_{\substack{g \in G \\ h \in H}} \max(\#(h, \{b_{g,p} | p \in P\}) - 1, 0) \\ & + 2 \sum_{\substack{p \in P \\ h \in H}} \max\left(\sum_{i \in \rho(h, \{b_{g,p} | g \in G\})} C_i - Q_h, 0 \right) \\ & + \sum_{\substack{i,j \in G \\ j > i}} \max(\#(v, \{b_{i,p} = b_{j,p} | p \in P\}) - 1, 0) \end{aligned} \tag{9}$$

où $\#(v, a)$ est le nombre d'occurrences de v dans a et $\rho(v, a)$ est l'ensemble des indices de a dont les valeurs sont égales à v. Les vecteurs C et Q contiennent les données de la taille des équipages et de la capacité des bateaux respectivement.

4 Recherche tabou

L'algorithme suivant [18] a été conçu pour illustrer la puissance du langage de programmation COMET. Cet algorithme est donc idéalement adapté à être utilisé comme comparaison avec d'autres méthodes codées en COMET, dont la nôtre.

L'algorithme est basé sur une heuristique de conflit minimum qui sélectionne la variable de décision étant impliquée dans le plus de violations et ensuite cherche la valeur minimisant ces violations. La longueur de la liste tabou [9] est variable et change en fonction de l'évolution des violations d'une itération à l'autre (telle que dans la recherche tabou réactive [1]). Les paires variable-valeur sont rendues taboues pendant un certain nombre d'itérations. Si les nouvelles violations sont inférieures à la précédente itération alors la durée tabou est décrémentée. Inversement, elle est incrémentée quand il n'y a pas d'amélioration. Un critère d'aspiration classique est utilisé : un mouvement est accepté s'il est tabou et qu'il produit une nouvelle meilleure solution.

Cette recherche tabou est améliorée grâce à un composant d'intensification qui retourne à la meilleure solution obtenue si aucune amélioration n'a été obtenue pendant un certain nombre d'itérations. Ceci permet une exploration plus intensive des régions prometteuses de l'espace de recherche tout en évitant de

« tourner en rond » ou simplement en évitant de stagner dans des régions moins intéressantes. Cet algorithme possède également un composant de redémarrage. Si aucune solution réalisable n'a été trouvée pendant un certain nombre d'itérations, alors le processus de recherche redémarre à partir d'une solution complètement nouvelle dans l'espoir que cela permettra au processus de s'échapper de la région de recherche dans laquelle il était prisonnier.

Au final, cet algorithme est plus qu'une simple recherche tabou puisqu'il comprend différents mécanismes afin d'améliorer la recherche tabou classique. De plus, l'algorithme utilise plusieurs paramètres : pour la durée tabou maximum et minimum, pour déclencher l'intensification et le redémarrage. On peut également considérer que les pondérations du modèle font partie de la configuration des paramètres.

5 Expériences

5.1 Opérateurs utilisés

Comme nous l'avons mentionné précédemment, les opérateurs sont composés d'un voisinage et d'un sélecteur. Nous décrivons ces briques dans cette section.

5.1.1 Voisinages

Soient D un domaine discret fini et $x = (x_1, \ldots, x_n) \in X$ une solution du problème avec $x_i \in D$, alors $\mathcal{N}_D$ est la relation de voisinage où chaque solution diffère de l'autre d'une valeur de ses variables $x_i : (x, x') \in \mathcal{N}_D$ ssi $\exists! i \in \{1, \ldots, n\}$ tel que $x_i \neq x'_i$.

Soit f_k^{mv} la fonction qui retourne les k indices des k variables qui sont impliquées dans les violations. La relation de voisinage $\mathcal{N}_D^{f_k^{mv}}$ restreint alors $\mathcal{N}_D$ aux voisins ayant les violations les plus élevées.

5.1.2 Sélecteurs

Nous utilisons le sélecteur des k-meilleures amélioration σ_{BIk} défini tel que $\sigma(s, \mathcal{N})$ soit un élément sélectionné uniformément $s' \in K$, avec K l'ensemble de k-meilleurs éléments selon un ordre $<$, tels que $(s, s') \in \mathcal{N}$.

Le sélecteur σ_{BIk} est ensuite modifié en introduisant une fonctionnalité tabou. Ainsi σ_{BIk}^{Tl} sera le sélecteur σ_{BIk} associé à une liste tabou. Une solution est taboue si elle contient une affection interdite dans la liste tabou. Le nombre d'itérations pendant lesquelles une affectation est taboue est donné par l. Si les opérateurs utilisent la fonctionnalité tabou, la liste tabou est partagée.

5.1.3 Opérateurs

En combinant la relation de voisinage $\mathcal{N}_D^{f_k^{mv}}$ avec les valeurs $k \in \{1, 5\}$, le sélecteur σ_{BIk}^{Tl} avec les valeurs $k \in \{1, 5\}$ et $l \in \{7, 8, 9\}$ nous générons combinatoirement 12 opérateurs. Les valeurs des itérations tabou l sont choisies dans l'intervalle de la recherche tabou Comet (entre 2 et 10, inclus).

Étant donné que notre implémentation nous le permet facilement, nous avons initialement essayé de générer des opérateurs ne considérant que les variables impliquées dans une unique contrainte. Ceci génère au moins un opérateur par contrainte (approximativement 440), et combinatoirement beaucoup plus si l'on tient compte des autres choix lors de la génération des opérateurs, par exemple le type de sélecteur.

Dans ce cas, l'approche « une contrainte – un opérateur » ne fonctionne pas correctement car la recherche ne converge pas. C'est un résultat intéressant qui devra faire l'objet de recherches futures. Nous utiliserons donc des opérateurs qui considèrent toutes les variables du problème.

5.2 Protocole expérimental

Toutes les configurations, c.-à-d. les instances du problème, sont composées de 13 hôtes, 29 équipages invités et p périodes. La configuration h-p représente un groupe h d'hôtes choisis parmi les bateaux et le nombre de périodes p. Le modèle est alors constitué de 29 contraintes `alldifferent`, p contraintes `multi-knapsack` et $29^2/2$ contraintes `atmost`. Une instance de 10 périodes a 390 variables de décision, chacune avec un domaine de taille 13.

Initialement nous comparons la sélection uniforme (Uni), la sélection proportionnelle associée à l'utilité Pareto (PP) et la recherche tabou (TS). Nous modifions chaque algorithme afin d'analyser l'importance de la pondération des fonctions de pénalités (au sein du modèle) ainsi que l'impact du composant d'intensification. Ce composant est également intégré au mécanisme de contrôle.

Pareto dominance Pour PP, $\epsilon = 0.1$.

Recherche tabou Nous conservons les paramètres par défaut. La durée tabou peut varier de 2 à 10, la recherche retourne à la meilleure solution trouvée s'il n'y a pas d'améliorations pendant 2 000 itérations (intensification) et redémarre s'il n'y a pas d'améliorations pendant 100 000 itérations.

Intensification Quand PP est utilisée avec l'intensification, le même nombre d'itérations est utilisé (2 000).

TABLE 1 – Configuration de F-Race pour CDP.

Paramètre	Domaine	Gagnant
T_{max}	$\{0.2, 0.3, 0.4, 0.5\}$	0.2
T_{min}	$\{0.15, 0.2, 0.25, 0.3\}$	0.25
s_{inc}	$\{0.0001, 0.001, 0.01\}$	0.0001
s_{dec}	$\{0.01, 0.1, 0.2\}$	0.1

Poursuite adaptative Après les expériences initiales avec PP, nous ajoutons des comparaisons avec la poursuite adaptative associée à l'utilité Pareto (PAP) et la stratégie adaptative associée à la somme des déplacements rectilinéaires positifs avec une sélection proportionnelle (CDP). PAP et CDP sont réglées avec F-Race [2], un algorithme de réglage hors-ligne.

Le gagnant de la procédure F-Race pour PAP est ($\beta = 0.9$, $p_{min} = 0.01$). Les combinaisons de paramètres testées sont générées à partir des domaines $\{0.1, 0.3, 0.6, 0.9\}$ pour β, et $\{0, 0.001, 0.01, 0.1, 0.1\}$ pour p_{min}.

Stratégie adaptative Les paramètres de CDP sont testés avec les valeurs du Tableau 1. Ceci donne 144 configurations possibles. La 3e colonne indique la configuration gagnante.

Notons que puisqu'un grand nombre d'instances sont facilement résolues, le critère d'optimisation de F-Race est le temps d'exécution et non le taux de réussite.

Dans tous les cas, le nombre maximum d'itérations par exécution est fixé à 1 000 000 et chaque algorithme est lancé 100 fois. Toutes les fenêtres glissantes sont de taille 100.

Les programmes sont codés en COMET 2.1.1. Les expériences sont lancées sur un processeur Xeon à 2,8 GHz. Les valeurs présentées correspondent à la moyenne des 100 exécutions indépendantes.

5.3 Résultats et discussion pour PP

Le Tableau 2 présente le pourcentage de configurations résolues avec succès en fonction de l'utilisation des pondérations et du composant d'intensification.

5.3.1 Taux de résolution

Le Tableau 2 indique que la majorité des configurations sont facilement résolues avec un taux de succès de 100 %, et ce même pour la sélection uniforme. Les configurations 1-10, 2-9, 3-9, 4-9, 5-7 et 6-7 sont les plus difficiles.

Sans pondération des contraintes Considérons d'abord le modèle sans pondération, c.-à-d. où toutes les contraintes ont la même importance. Sans intensification, Uni produit de meilleurs résultats que TS sur les configurations 4-8 et 5-7. L'inverse est vrai pour 1-10. Toutefois PP obtient les meilleurs résultats de ce groupe.

Le groupe sans pondération mais avec intensification peut représenter la situation où l'utilisateur n'a pas d'expérience du problème et ne pondère donc pas les contraintes. Dans ce cas, nous observons que PP obtient à nouveau les meilleurs résultats et que Uni dépasse légèrement TS.

Avec pondération Pour le groupe sans pondération mais avec l'intensification, on remarque les même différences entre les méthodes. Toutefois il semble que Uni soit la méthode qui tire le plus de bénéfices de l'introduction des poids.

Finalement, dans le dernier groupe avec la pondération et l'intensification, nous obtenons les meilleures performances de chaque méthode. Notons toutefois qu'un ensemble de bons opérateurs à lui tout seul n'obtient pas les meilleurs résultats. Ainsi, le fait de donner la priorité à certains opérateurs confère un certain avantage comme le montrent les résultats de PP. Il n'y a pas de différence statistique entre PP et TS. PP avec intensification est donc capable d'atteindre un taux de résolution équivalent à TS. Il est utile de noter que les mécanismes de diversification des deux méthodes sont différents. TS applique le tabou et le redémarrage alors que PP n'utilise que le tabou dans ces opérateurs et une sélection uniforme parmi les k variables avec les violations maximum et parmi les k meilleures solutions. Ce mécanisme de diversification est basé sur des perturbations très modérées tandis qu'un redémarrage est beaucoup plus drastique.

5.3.2 Analyse d'une exécution

La Figure 1 représente les courbes empilées des probabilités de sélection des opérateurs en fonction de l'itération pour une exécution de la configuration 6-7.

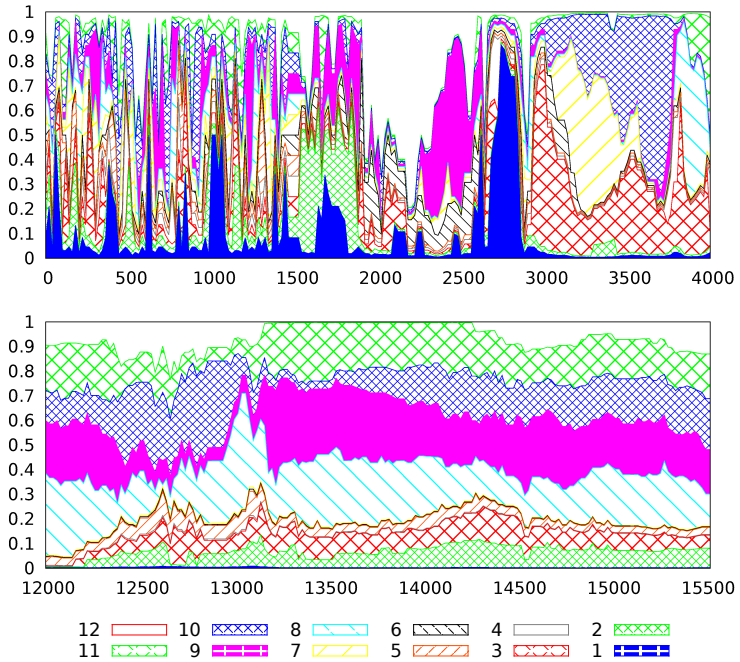
Pendant les 2 000 premières itérations les changements sont relativement rapides et tous les opérateurs ont la possibilité d'être sélectionnés. Cela s'explique, en partie, par le fait qu'au début tous les opérateurs peuvent facilement améliorer la solution précédente.

Pendant la première partie de la recherche (moins de 8 000 itérations) l'opérateur 1 – qui sélectionne la variable la plus violée et qui sélectionne le meilleur voisin avec la durée tabou la plus petite – a une probabilité non négligeable de sélection, et est même parfois l'opérateur avec la probabilité la plus grande. Par la suite, cette probabilité est presque nulle puisqu'il n'y a pas suffisamment de diversification – pour rappel la diver-

TABLE 2 – Pourcentage de succès en fonction de l'utilisation de la pondération et du composant d'intensification pour Uni, PP et TS. Les meilleures valeurs sont en gras.

	Taux de succès											
	Sans pondération						Avec pondération					
	Sans intens.			Avec intens.			Sans intens.			Avec intens.		
Config	Uni	PP	TS	Uni	PP	TS	Uni	PP	TS	Uni	PP	TS
1-6	100	100	100	100	100	100	100	100	100	100	100	100
1-7	100	100	100	100	100	100	100	100	100	100	100	100
1-8	100	100	100	100	100	100	100	100	100	100	100	100
1-9	100	100	100	100	100	100	100	100	100	100	100	100
1-10	12	**95**	93	**100**	99	**100**	69	**100**	97	100	100	100
2-6	100	100	100	100	100	100	100	100	100	100	100	100
2-7	100	100	100	100	100	100	100	100	100	100	100	100
2-8	100	100	100	100	100	100	100	100	100	100	100	100
2-9	0	**12**	0	45	**85**	51	2	**18**	3	89	97	**99**
3-6	100	100	100	100	100	100	100	100	100	100	100	100
3-7	100	100	100	100	100	100	100	100	100	100	100	100
3-8	100	100	100	100	100	100	100	100	100	100	100	100
3-9	0	**4**	0	49	**72**	36	0	**14**	2	88	98	**100**
4-6	100	100	100	100	100	100	100	100	100	100	100	100
4-7	100	100	100	100	100	100	100	100	100	100	100	100
4-8	84	**100**	68	100	100	100	100	100	100	100	100	100
4-9	0	**1**	0	15	**48**	10	0	**7**	0	76	**96**	93
5-6	100	100	100	100	100	100	100	100	100	100	100	100
5-7	48	**100**	1	74	**99**	65	**100**	**100**	22	98	**100**	**100**
6-6	**100**	**100**	99	100	100	100	100	100	100	100	100	100
6-7	4	**63**	0	43	**78**	7	72	**100**	0	86	**96**	90

FIGURE 1 – Courbes empilées des probabilités de sélection des opérateurs en fonctions des itérations pour une exécution de la configuration 6-7. Le premier graphique correspond aux 4 000 premières itérations. Le second correspond à la fin de la recherche.



sification se produit à travers le mécanisme tabou et en choisissant parmi les meilleurs voisins.

Les probabilités de sélection à la fin de la recherche sont monopolisées par les opérateurs 8-12 qui produisent plus de diversification à travers les mécanismes précédemment mentionnés.

5.4 Résultats et discussion pour les stratégies adaptatives

Maintenant que nous avons établi qu'il était intéressant d'utiliser plusieurs opérateurs sélectionnés de façon proportionnelle, examinons des mécanismes plus avancés présentés précédemment : PAP et CDP. Le Tableau 3 présente les résultats obtenus par ces méthodes et reprend les résultats de PP et TS uniquement pour le modèle pondéré avec le composant d'intensification.

Remarquons d'abord que, lorsque l'on considère le taux de résolution, CDP semble améliorer les résultats de PP, toutefois cet amélioration n'est pas vraiment significatif. PAP est à la traine ; il obtient les mêmes performances que la sélection uniforme. La stratégie du gagnant remporte tout ne fonctionne pas ici.

En ce qui concerne le nombre moyen d'itérations, PP et CDP obtiennent des résultats assez similaires. La différence plus prononcée sur les configurations les plus difficiles provient principalement du fait qu'il est très pénalisant de ne pas trouver une solution dans le nombre imparti d'itérations. Ceci augmente mécaniquement le nombre moyen d'itérations. Dans les cas où une solution est toujours trouvée, les différences sont moins visibles.

Examinons les temps d'exécution. Ici TS est clairement plus rapide : à peu près un ordre de magnitude. Les autres méthodes reposent sur l'architecture de notre solveur qui est une surcouche de Comet : la gestion des fenêtres glissantes, les calculs d'impact et d'utilité de chaque opérateur, et les différent mécanisme de communication entre les différents opérateurs y contribuent partiellement. Notons que CDP est plus rapide que PP. CDP a le bénéfice du réglage F-Race basé sur l'optimisation du temps d'exécution.

Nous avons vu, pour le PPP, que la sélection d'opérateurs peut atteindre un niveau de performance équivalent à la meilleure méthode en terme de taux de réussite. Comme on pouvait le prévoir, les temps d'exécutions sont toutefois supérieurs.

6 Conclusion

Dans cet article nous avons montré que la sélection adaptative d'opérateurs pouvait être utilisée afin de résoudre le PPP avec succès en atteignant des résultats comparables à un algorithme de recherche tabou adapté au problème. Toutefois les temps d'exécution de notre approche sont supérieurs. Dans le prolongement de ce travail, nous souhaitons automatiser le processus de génération des opérateurs en prenant en compte les propriétés du modèle.

Note. Ces travaux sont financés par Microsoft Research à travers le PhD Scholarship Programme.

Références

[1] R. Battiti, M. Brunato, and F. Mascia. *Reactive Search and Intelligent Optimization*, volume 45 of *Operations research/Computer Science Interfaces*. Springer Verlag, 2008.

[2] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 11–18. Morgan Kaufmann Publishers Inc., 2002.

[3] E. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics : An emerging direction in modern search technology. In F. Glover and G.A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 457–474. Springer, 2003.

[4] C.A. Coello Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms : a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11-12) :1245–1287, 2002.

[5] G. di Tollo, F. Lardeux, J. Maturana, and F. Saubion. From adaptive to more dynamic control in evolutionary algorithms. In P. Merz and J.-K. Hao, editors, *Evolutionary Computation in Combinatorial Optimization*, volume 6622 of *LNCS*, pages 130–141. Springer Berlin / Heidelberg, 2011.

[6] Á. Fialho. *Adaptive Operator Selection for Optimization*. PhD thesis, Université Paris-Sud 11, Orsay, France, December 2010.

[7] Ph. Galinier and J.-K. Hao. A general approach for constraint solving by local search. *Journal of Mathematical Modelling and Algorithms*, 3 :73–88, 2004.

[8] I.P. Gent and T. Walsh. Csplib : a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from http ://www.csplib.org/.

[9] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

TABLE 3 – Taux de succès, nombre moyen d'itération et moyenne des temps d'exécution en secondes pour PAP, PP, CDP et TS. Seul le modèle avec pondération, avec le composant d'intensification, est considéré ici. Les meilleures valeurs sont en gras.

	Résolution				Itérations				Temps (secondes)			
Config	PAP	PP	CDP	TS	PAP	PP	CDP	TS	PAP	PP	CDP	TS
1-6	100	100	100	100	215	**204**	209	226	0.39	0.36	0.24	**0.13**
1-7	100	100	100	100	328	**306**	321	376	0.55	0.51	0.37	**0.20**
1-8	100	100	100	100	757	710	**681**	750	1.11	1.07	0.66	**0.25**
1-9	100	100	100	100	2847	2300	2104	**1998**	3.68	3.19	1.74	**0.44**
1-10	100	100	100	100	48124	28679	37889	**19338**	60.29	38.70	28.33	**2.15**
2-6	100	100	100	100	807	**720**	772	761	1.00	0.92	0.57	**0.18**
2-7	100	100	100	100	1853	1486	**1343**	2189	2.16	1.87	1.01	**0.36**
2-8	100	100	100	100	10752	7103	**5384**	15199	12.16	8.74	3.75	**1.50**
2-9	87	97	**99**	**99**	282528	123307	**101930**	218156	329.74	154.96	70.96	**19.70**
3-6	100	100	100	100	710	**664**	754	769	0.89	0.87	0.55	**0.18**
3-7	100	100	100	100	1813	**1598**	1580	2357	2.15	2.01	1.14	**0.36**
3-8	100	100	100	100	11940	7271	**6593**	16433	13.58	8.94	4.56	**1.59**
3-9	81	98	95	**100**	423701	**175256**	194428	280598	498.38	220.23	134.30	**25.27**
4-6	100	100	100	100	1492	1397	2200	**976**	1.69	1.66	1.35	**0.20**
4-7	100	100	100	100	3019	**2587**	2991	3469	3.48	3.13	1.98	**0.44**
4-8	100	100	100	100	20922	**11443**	11796	20962	24.02	14.01	7.89	**1.97**
4-9	75	96	**98**	93	487386	215666	**156460**	431750	570.36	270.38	107.63	**39.04**
5-6	100	100	100	100	5261	**3703**	4802	9526	5.51	4.19	2.81	**0.82**
5-7	96	**100**	**100**	**100**	144820	36924	**34298**	141654	150.28	42.06	21.09	**11.14**
6-6	100	100	100	100	8686	**6157**	8362	21678	8.98	6.74	4.80	**1.69**
6-7	84	96	**98**	90	300979	143494	**116907**	362559	302.79	157.72	70.83	**28.07**

[10] Y. Hamadi, É. Monfroy, and F. Saubion. *Autonomous Search*. Springer Berlin Heidelberg, 2012.

[11] J. Maturana, A. Fialho, F. Saubion, M. Schoenauer, and M. Sebag. Compass and dynamic multi-armed bandits for adaptive operator selection. In *Proc. of IEEE Congress on Evolutionary Computation CEC*, 2009.

[12] J. Maturana, F. Lardeux, and F. Saubion. Autonomous operator management for evolutionary algorithms. *Journal of Heuristics*, 2010.

[13] J. Maturana and F. Saubion. On the design of adaptive control strategies for evolutionary algorithms. In N. Monmarché, El-G. Talbi, P. Collet, M. Schoenauer, and E. Lutton, editors, *EA 2007*, volume 4926, pages 303–315. Springer, Heidelberg, 2007.

[14] M. Ågren, P. Flener, and J. Pearson. Revisiting constraint-directed search. *Information and Computation*, 207(3) :438–457, March 2009.

[15] H. Simonis. Progress on the progressive party problem. In W.-J. van Hoeve and J. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5547 of *LNCS*, pages 328–329. Springer Berlin / Heidelberg, 2009.

[16] B.M. Smith, S.C. Brailsford, P.M. Hubbard, and H.P. Williams. The progressive party problem : Integer linear programming and constraint programming compared. *Constraints*, 1(1) :119–138, 1996.

[17] D. Thierens. An adaptive pursuit strategy for allocating operator probabilities. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1539–1546, New York, NY, USA, 2005. ACM.

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

[19] N. Veerapen, J. Maturana, and F. Saubion. An exploration-exploitation compromise-based adaptive operator selection for local search. In *Proceedings of the 14th international conference on Genetic and evolutionary computation*, GECCO '12, pages 1277–1284, New York, NY, USA, 2012. ACM.

[20] N. Veerapen and F. Saubion. Pareto autonomous local search. In C.A. Coello Coello, editor, *Proceedings of the 5th International Conference on Learning and Intelligent Optimization (LION5)*, volume 6683 of *LNCS*, pages 392–406. Springer, Heidelberg, 2011.

Consistance d'arc par MDD-réduction

Julien Vion

Université de Valenciennes et du Hainaut Cambrésis, LAMIH CNRS UMR 8201
julien.vion@univ-valenciennes.fr

Résumé

Dans cet article, nous présentons *MDDR*, un nouvel algorithme de révision de contraintes définies sous forme de diagramme de décision multi-valué (MDD). En réalisant des copies partielles à la volée des MDD modifiés au cours de la recherche, *MDDR* parvient à une meilleure incrémentalité que *MDDC* proposé par Cheng & Yap en 2010 [2]. La meilleure incrémentalité, couplée à la nouvelle possibilité d'exploiter les informations sur les variables modifiées permet d'éviter de parcourir systématiquement toute la structure de données à chaque filtrage, et ce malgré l'utilisation d'une file de propagation à gros grain. Outre ses performances sur les contraintes définies sous forme de MDD, *MDDR* est presque aussi performant que l'état de l'art *STR2* de Ullmann et Lecoutre [12, 7] sur les contraintes *table* non structurées, définies sous forme de liste d'affectations autorisées.

1 Introduction

Le problème de satisfaction de contraintes (CSP pour *Constraint Satisfaction Problem*) est l'un des problèmes NP-complets de référence, à la fois pour sa relative clarté théorique et sa souplesse pour la modélisation des problèmes de décision combinatoires. Ceux-ci peuvent très naturellement être décomposés en variables et contraintes : le problème consiste alors à déterminer s'il existe une affectation des variables qui satisfasse toutes les contraintes. La résolution des CSP discrets se fait généralement par exploration arborescente systématique, et filtrage par consistance d'arc (AC pour *Arc Consistency*) à chaque nœud. La consistance d'arc consiste simplement à supprimer toutes les valeurs dont on peut montrer qu'il n'existe pas de solution faisant apparaitre cette valeur et satisfaisant une contrainte donnée. Le filtrage par consistance d'arc est NP-difficile dans le cas général, mais il existe de nombreux cas particuliers de contraintes qu'il est possible de filtrer en temps et en espace polynomiaux (les plus évidentes étant les contraintes portant sur un nombre borné de variables).

La contrainte *table* bénéficie d'un intérêt particulier : il s'agit d'une contrainte définie par une liste exhaustive de k-uplets (une *relation*) qui vont satisfaire (on parle de k-uplets *autorisés*), ou au contraire invalider (on parle de *nogoods*) la contrainte (dite respectivement *positive* ou *négative*). Ce type de contrainte trouve sa place dans la modélisation de nombreux problèmes de configuration ou de bases de données. De plus, des représentations compressées de ces tables permettent de modéliser efficacement certaines contraintes structurées [2].

Exemple 1. *La contrainte C, portant sur les variables X, Y et Z, autorise six affectations de ces trois variables :*

$$\{ \quad \langle X = b, Y = c, Z = b \rangle , \quad \langle X = a, Y = b, Z = a \rangle , \\ \langle X = a, Y = a, Z = a \rangle , \quad \langle X = c, Y = a, Z = c \rangle , \\ \langle X = c, Y = a, Z = a \rangle , \quad \langle X = a, Y = a, Z = c \rangle \}$$

Pour établir la consistance d'arc sur une telle contrainte, il faut montrer que chaque valeur de chaque variable dispose d'un *support*, c'est-à-dire d'un k-uplet *autorisé* et *valide* (i.e., dont toutes les valeurs sont présentes dans les domaines des contraintes) où la valeur apparait. Dans l'exemple 1, si le domaine de chaque variable est égal à $\{a, b, c\}$, la valeur b du domaine de la variable X est supportée par le premier k-uplet $\langle X = b, Y = c, Z = b \rangle$. Si c est supprimée du domaine de Y, ce k-uplet n'est plus valide, et la valeur $X = b$ n'est plus supportée : elle peut alors être filtrée.

Il existe deux stratégies principales pour trouver un support à chaque valeur [9] : soit on itère sur les k-uplets *valides* (c'est-à-dire appartenant au produit cartésien du domaine des variables), en vérifiant à chaque itération si le k-uplet est autorisé par la contrainte, soit on itère sur les k-uplets *autorisés* par la contrainte,

en vérifiant à chaque itération si le k-uplet est valide. La première approche est généralement retenue pour les contraintes lâches et/ou d'arité faible, mais elle ne peut être efficacement appliquée à des contraintes très fortes : pour une contrainte autorisant λ k-uplets sur un alphabet de d valeurs, sa complexité temporelle dans le pire des cas est alors en $O(k(d^k - \lambda))$. Dans ce cas, on préfère la seconde option – en $O(k\lambda)$. Des algorithmes relativement efficaces ont été proposés pour cette approche : GAC-schema combiné à une représentation indexée ou compressée de la table de k-uplets [5, 10, 6] ; ou encore des algorithmes spécifiques comme *STR* [12, 7, 8], *AC5TC* [11] ou *MDDC* [2].

La représentation des tables par un diagramme de décision multi-valué (MDD) est l'une des approches les plus prometteuses, d'une part en termes d'espace (les MDD peuvent représenter un nombre exponentiel de k-uplets en espace polynomial), mais aussi de temps et de modélisation [2]. Dans cet article, nous proposons *MDDR*, un nouvel algorithme permettant d'établir la consistance d'arc sur une telle contrainte.

2 Définitions et notations

Définition 1 (Réseau de contraintes, variable, domaine, contrainte, affectation)**.** Un *réseau de contraintes* $\mathcal{N}$ est un couple $(\mathscr{X}, \mathscr{C})$ constitué :

- d'un ensemble de $\boldsymbol{n}$ *variables* $\mathscr{X}$; un *domaine* $\mathrm{dom}(X)$ est associé à chaque variable $X \in \mathscr{X}$ et définit l'ensemble fini d'au plus $\boldsymbol{d}$ valeurs auxquelles X peut être affectée, et
- d'un ensemble de $\boldsymbol{e}$ *contraintes* $\mathscr{C}$; chaque contrainte $C \in \mathscr{C}$ implique au plus $\boldsymbol{k}$ variables $\mathrm{vars}(C) \subseteq \mathscr{X}$; la contrainte définit au plus $\boldsymbol{\lambda}$ *affectations* autorisées $\mathrm{rel}(C)$ de ces variables.

La lâcheté d'une contrainte $\boldsymbol{l}$ (*looseness* ou *taux de remplissage*), comprise entre 0 et 100 %, est telle que $\lambda = ld^k$. Sa dureté t (*tightness*) est alors telle que $t + l = 100\,\%$.

Le problème de satisfaction de contraintes (CSP) consiste à décider si une solution au CN (c'est-à-dire une affectation de toutes les variables satisfaisant toutes les contraintes du CN) existe. Si toutes les contraintes peuvent être vérifiées en temps et en espace polynomiaux, le CSP est NP-complet.

Définition 2 (Consistance d'arc, support)**.** Soit C une contrainte et $X \in \mathrm{vars}(C)$. La valeur $v \in \mathrm{dom}(X)$ est arc-consistante (AC) pour C ssi il existe une affectation de $\mathrm{vars}(C)$, autorisée par C, qui instancie X à v (une telle affectation est appelée un *support* de v pour C). C est AC ssi $\forall X \in \mathrm{vars}(C)$, $\forall v \in \mathrm{dom}(X)$, v est AC pour C.

Fonction AC($\mathcal{N} = (\mathscr{X}, \mathscr{C})$, δ)

```
1  (Q, modif) ← initQueue(δ)
2  tant que Q ≠ ∅ faire
3      Choisir et retirer C de Q
4      Δ ← C.revise(modif(C))
5      si Δ = ⊥ alors retourner ⊥
6      sinon
7          modif(C) ← ∅
8          pour chaque Y ∈ Δ,
           C' ∈ 𝒞 − C | Y ∈ vars(C') faire
9              modif(C') ← modif(C') ∪ {Y}
10     Q ← Q ∪ {C' ∈ 𝒞 − C | vars(C') ∩ Δ ≠ ∅}
11 retourner 𝒩
```

Notre définition de la consistance d'arc est généralisée aux contraintes d'arité quelconque. Appliquer la consistance d'arc sur une contrainte C consiste à supprimer toutes les valeurs du domaine des variables $\mathrm{vars}(C)$ qui ne sont pas AC pour C (on appelle cette procédure une *révision* de contrainte). Appliquer la consistance d'arc sur un CN consiste à réviser toutes les contraintes non AC jusqu'au point fixe.

Dans la suite de cet article, on considèrera à fin de simplification que toutes les contraintes sont définies sous la forme d'une *relation*, c'est-à-dire une liste de k-uplets *autorisés*. Il reste toujours possible d'utiliser plusieurs types de contraintes dans le même réseau, en associant à chacune un algorithme de filtrage différent.

Les structures de données sont définies à partir d'ensembles, tableaux (ou toute autre structure indexée à partir de 1) et de séquences (généralement implantées par des listes chainées). Pour un tableau T, $T[i]$ représente le i^{e} élément du tableau. Pour une séquence $S = \langle 1, 2, 3\rangle$, on définit l'opérateur d'ajout :: tel que $0 :: S = \langle 0, 1, 2, 3\rangle$.

3 Algorithmes de base

On considère que le CSP est résolu par une recherche arborescente systématique, en profondeur d'abord, avec maintien de la consistance d'arc à chaque nœud (MAC). La fonction `AC` établit la consistance d'arc à l'aide d'une file de propagation à gros grain, contenant l'ensemble des contraintes à réviser, par un appel à la fonction `revise` qui lui est rattachée (ligne 4). Celle-ci renvoie la liste des variables ainsi modifiées (ou la valeur spéciale $\bot$ si une inconsistance, i.e., un domaine ou une relation vide, a été détectée), et la file de propagation est alors mise à jour en conséquence : il faudra à nouveau réviser les autres contraintes impliquant les variables modifiées (ligne 10). De plus, l'algorithme

Fonction seekSupports(C)

```
1  ∀X ∈ vars(C), supp(X) ← ∅
2  seek ← vars(C)
3  pour chaque τ ∈ rel(C) faire
4      si ∀X ∈ vars(C), τ[X] ∈ dom(X) alors
5          pour chaque X ∈ seek faire
6              supp(X) ← supp(X) ∪ τ[X]
7              si supp(X) = dom(X) alors
8                  seek ← seek − X
9                  si seek = ∅ alors retourner supp
10 retourner supp
```

maintient pour chaque contrainte la liste *modif* des variables modifiées depuis la dernière révision de celle-ci (lignes 7 à 9). La fonction `initQueue` initialise la file de propagation en fonction des décisions logiques utilisées pour la recherche systématique (typiquement, une valeur affectée à une variable), d'une manière similaire aux lignes 7 à 10 de l'algorithme.

Un algorithme simple pour réviser une contrainte définie comme une liste de k-uplets autorisés consiste à appliquer la fonction `seekSupports`. La fonction renvoie la liste des valeurs ayant un support, on peut alors supprimer toutes les autres valeurs des domaines des variables. L'algorithme parcourt la liste des k-uplets autorisés (ligne 3) et vérifie leur validité (ligne 4) : pour chaque k-uplet τ, $\tau[X]$ est la valeur du k-uplet correspondant à la variable X. Enfin, on complète la liste des valeurs supportées (lignes 5 et 6). Si toutes les valeurs de toutes les variables sont déjà supportées, on peut interrompre l'algorithme (lignes 7 à 9).

La fonction `seekSupports` admet une complexité temporelle en $O(k\lambda)$. Elle peut être appelée par `AC` jusqu'à kd fois par contrainte tout au long d'une branche de l'arbre de recherche : à chaque fois qu'une valeur est supprimée du domaine d'une variable impliquée par la contrainte. Cette version ne présente aucune propriété d'incrémentalité : la fonction `AC` admet alors une complexité de $O(ek^2d\lambda)$ pour une branche de l'arbre de recherche.

4 L'algorithme *STR(2)* (*Simple Table Reduction*)

Initialement proposé en 2007 par J.R. Ullmann [12], puis amélioré par C. Lecoutre [7] (sous le nom *STR2*), l'algorithme *STR* améliore la fonction `seekSupports` en supprimant préalablement de rel(C) tous les k-uplets non valides : cela revient à effectuer l'opération de la ligne 4 préalablement à rechercher les supports de chaque valeur, et à maintenir cette version filtrée

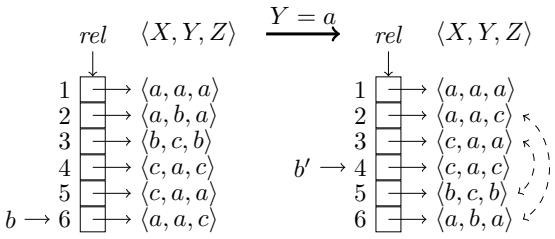


FIGURE 1 – Filtrage d'une relation (rel, b) par *STR*

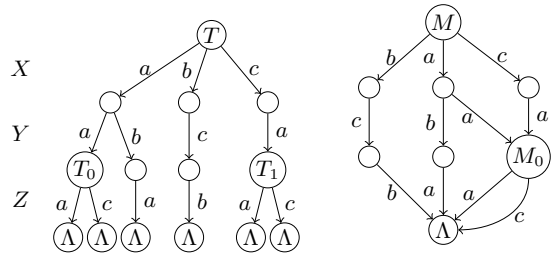
Fonction filterSTR((rel, b), *mod*)

```
1  b′ ← b ; i ← 1
2  tant que i ≤ b′ faire
3      si ∀X ∈ mod, rel[i][X] ∈ dom(X) alors
4          i ← i + 1
5      sinon
6          permuter rel[i] et rel[b′]
7          b′ ← b′ − 1
8  retourner (rel, b′)
```

de la relation entre deux appels à l'algorithme. Cette stratégie permet de ne contrôler la validé des k-uplets qu'au niveau des variables modifiées (c'est le principal apport de *STR2*). De plus, un k-uplet détecté invalide une fois ne sera plus considéré dans la même branche de l'arbre de recherche.

Cependant, maintenir une version filtrée de rel(C) au cours de la recherche n'est pas trivial : il faut être capable de restaurer rel(C) à une version antérieure lorsque l'on change de branche dans l'arbre de recherche (typiquement, après un retour-arrière). *STR* propose une astuce d'implantation permettant de sauvegarder et restaurer une liste de k-uplets en temps constant tout en n'empilant qu'un simple nombre entier à chaque niveau de l'arbre de recherche. Cf figure 1 et fonction `filterSTR` : *mod* contient la liste des variables modifiées depuis le dernier appel à la fonction. La relation est représentée par un couple (rel, b) : *rel* est un tableau de k-uplets. b est un nombre entier représentant l'index du dernier k-uplet valide (initialement, $b = \lambda$). Ainsi, les lignes 3 et 4 de `filterSTR` remplacent la ligne 4 de `seekSupports`. Pour supprimer un k-uplet de la structure, on le permute avec $rel[b]$, puis on décrémente b (lignes 7 et 8 de la fonction `filterSTR`). Il suffit de sauvegarder b pour être capable de restaurer la structure à un état antérieur.

Dans l'exemple de la figure 1, les k-uplets 2 et 3 ne sont plus valides après l'affectation $Y = a$: ils sont respectivement permutés avec les k-uplets 6 et 5, et b est décrémenté deux fois. Sa nouvelle valeur b' indique que ces k-uplets ne doivent plus être pris en compte.



(a) Trie (b) MDD

FIGURE 2 – Un Trie T et un MDD M représentant la relation de l'exemple 1. T_0 et T_1 sont identiques : ils sont fusionnés en un seul nœud M_0 dans le MDD correspondant.

Si b est restauré à son ancienne valeur 6, les k-uplets supprimés sont immédiatement disponibles à nouveau.

Comme chaque variable ne peut être modifiée que d fois, la complexité amortie du filtrage de la relation par STR2 tout au long d'une branche d'un arbre de recherche est en $O(kd\lambda)$ (i.e., un facteur k par rapport à STR). Cependant, une telle incrémentalité théorique n'a pas été observée pour la fonction `seekSupports` (bien qu'en pratique on constate que le temps passé à filtrer est généralement bien plus important que le temps passé à chercher les supports, grâce notamment aux optimisations apportées par la structure *supp*). La complexité globale d'`AC` reste donc inchangée.

La dernière amélioration de l'algorithme, *STR3* [8], atteint l'optimal $O(e(k\lambda + m))$ le long d'une branche de l'arbre de recherche de profondeur m [1] en indexant les k-uplets (chaque valeur des variables pointe sur les k-uplets où elle apparait) et en générant des listes de dépendances. *STR3* est considérablement plus compliqué que les précédentes versions de l'algorithme et nécessite l'utilisation d'une file de propagation à grain fin (i.e., maintenant la liste des valeurs supprimées entre deux révisions). Enfin, Mairy *et al.* ont récemment proposé une famille d'algorithmes, dont certains théoriquement optimaux, pour la propagation de contraintes *table* : *AC5TC* [11]. Cependant, ces algorithmes nécessitent une file de propagation à grain fin et ne permettent pas de traiter efficacement les contraintes sans structure, ou au contraire certains types de contraintes très structurées (e.g., sous forme de MDD).

5 Tries et MDD

Le Trie (pour *Re*trie*val*) est une structure de données arborescente utilisée pour représenter un ensemble R de k-uplets de taille variable (le plus souvent des chaines de caractères) en un espace limité, et vérifier l'appartenance d'une chaine en $O(k)$ – avec un comportement théorique meilleur qu'une table de hash. Pour notre application (tous les k-uplets sont de même taille non-nulle), on peut le définir récursivement comme une application $T : i \rightarrow T'$ telle que T' est un Trie qui représente l'ensemble des $(k-1)$-uplets $\{\tau, \ldots\}$ tels que $(i :: \tau) \in R$. T' peut être une feuille (notée Λ), c'est-à-dire représenter l'ensemble $\{\langle\rangle\}$.

T forme un arbre de ν_{trie} arcs, comme sur la figure 2a. Un nœud donné représente tous les k-uplets commençant par les mêmes valeurs (i.e., le même préfixe) : le Trie réalise une *compression* de la relation. Chaque feuille du Trie représente un k-uplet de la relation d'origine. Dans le pire des cas, aucun préfixe commun ne peut être identifié (i.e., toutes les premières valeurs de chaque k-uplet sont différentes), et le Trie contient alors $k\lambda$ nœuds (notez que cela nécessite $d \geq \lambda$). On peut également montrer que $\nu_{\text{trie}} \in O(d^k)$, ce qui peut être intéressant – par rapport à une table en $O(kd^k)$ – si d et $k \ll \lambda$ (ce qui implique généralement un taux de remplissage l important).

Un MDD est une généralisation du Trie, pour lequel on autorise que des sous-arbres identiques soient regroupés. La représentation d'un MDD est donc un DAG de ν_{mdd} arcs (cf figure 2b). Pour une même relation, on a donc $\nu_{\text{mdd}} \leq \nu_{\text{trie}} \leq k\lambda$. Pour construire le MDD à partir d'une relation quelconque, on commence par construire le Trie correspondant, puis on parcourt celui-ci en indexant chaque nœud ; le calcul de l'index se fait généralement en $O(d)$. Quand on rencontre un nœud tel qu'un nœud équivalent a déjà été indexé, il peut être supprimé et remplacé par celui-ci (fonction `mddReduce` de [2]). L'opération totale se fait en $O(k\lambda + d\nu_{\text{trie}})$. Le MDD d'une relation structurée peut souvent être construit directement par une fonction ad-hoc, mais le principe de l'indexation reste souvent présent. On atteint alors généralement une complexité en $O(\nu_{\text{trie}})$.

Différents ordonnancements des variables peuvent conduire à des Tries et MDD de taille différente. Trouver un ordonnancement de variables conduisant au Trie ou MDD de taille minimale est un problème NP-difficile [1] qui ne sera pas traité dans cet article.

L'utilisation de Tries et MDD pour représenter des contraintes *table* a déjà été envisagée, notamment dans les travaux de Gent *et al.* en 2007 (pour les Tries [5]) et de Cheng & Yap en 2010 (pour les MDD [2]). L'algorithme de Gent *et al.* recherche un support pour chaque valeur en parcourant successivement le Trie. Par exemple, on peut rechercher un support pour $X = b$ alors que $Y = a$ dans le Trie T de la figure 2a : partant de la racine T, on suit l'arc b (puisque $X = b$).

1. On remarquera que $m \leq nd$ dans la mesure où une hypothèse va supprimer au moins une valeur pour être exploitable.

L'arc suivant est invalide, puisque $Y \neq c$. Il n'y a pas d'autre arc disponible à ce niveau : $X = b$ n'a pas de support et peut être supprimée. Rechercher un support pour une valeur d'une variable située en bas de l'arbre nécessite de parcourir beaucoup plus de nœuds. Gent *et al.* proposent de générer plusieurs copies du même Trie : une pour chaque variable, le Trie étant réordonné pour que cette variable soit en haut de l'arbre. Cela ne tient malheureusement pas compte du fait qu'il est possible qu'une variable située en bas de l'arbre ait aussi pu être réduite à une seule valeur. Par exemple, si $Z = b$, il faut parcourir tout le sous-arbre de gauche pour montrer que $X = a$ n'a pas de support.

En plus d'utiliser des MDD, dont le nombre de nœuds est moindre que les Tries (et sont donc plus économes en mémoire et en temps), Cheng & Yap proposent dans leur algorithme *MDDC* un système permettant de sauvegarder, d'une exécution à l'autre de l'algorithme, quelles sous-parties du DAG ont été démontrées valides ou invalides. Ainsi, lorsque l'on rencontre plusieurs fois le même nœud au cours de la recherche, ou même au cours de deux recherches successives, on peut récupérer le résultat précédent immédiatement. Deux mécanismes sont utilisés pour sauvegarder les résultats d'une recherche : un *timestamp* associé à chaque nœud assure de ne pas parcourir celui-ci deux fois au cours de la même exécution, et un *sparse set* enregistre les nœuds invalides d'une exécution à l'autre de l'algorithme. Les *sparse sets* peuvent être « sauvegardés » et « restaurés » en temps constant (la technique est similaire à *STR*) en cas de retour-arrière.

Dans la suite de cet article, nous nous intéresserons plus particulièrement aux MDD. Nous proposons un nouvel algorithme, inspiré de *STR2* mais basé sur des MDD. Notre algorithme développe plus efficacement les possibilités d'incrémentalité exploitables sur une telle structure.

6 MDDR (MDD-Reduction)

Notre proposition s'inspire des structures de données immuables (*immutables*) utilisées dans les langages de programmation fonctionnels comme ML, Haskell ou Scala. Ces langages découragent ou interdisent la modification des données au cours de l'exécution des programmes : quand on ajoute un élément à une structure de données, on en crée une nouvelle, « copie » de la précédente à laquelle on ajoute l'élément. Quand la structure prend la forme d'un graphe acyclique, il est généralement possible de réutiliser une grande partie du graphe original en pointant vers des parties du graphe non impactées par la modification. De ce fait, la complexité temporelle des opérations sur les structures immuables est généralement la même

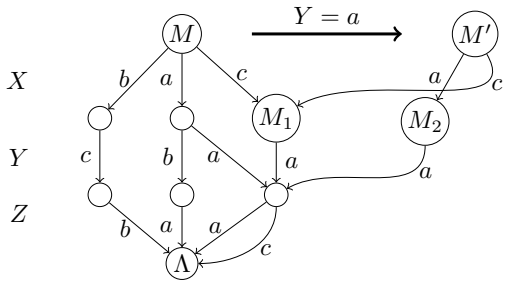


FIGURE 3 – Filtrage d'un MDD immuable.

que sur des structures modifiables. Les structures immuables simplifient considérablement la conception et le débogage des algorithmes, mais elles permettent surtout de s'affranchir de toute gestion de la synchronisation lors de la parallélisation.

Une version légèrement dégradée de *STR(2)* peut être implantée en utilisant des structures de données immuables. Il suffit de représenter la relation sous la forme d'une liste simplement chainée de k-uplets. Lorsque l'on filtre la relation, on génère une autre liste ne comprenant que les k-uplets valides, et on conserve une référence sur la liste d'origine pour réaliser le retour-arrière en temps constant. Il n'y a pas de surcout de complexité temporelle (on ajoute cependant l'allocation d'une cellule de liste chainée à chaque fois que l'on vérifie la validité d'un k-uplet, et la libération de la mémoire en cas de retour-arrière – transparente pour le développeur si un ramasse-miettes est utilisé). En terme d'espace, on ajoute une liste chainée de taille $O(\lambda)$ à chaque niveau de l'arbre de recherche (il n'est pas nécessaire de dupliquer les k-uplets eux-mêmes, qui sont déjà immuables). Coder la phase de filtrage de *STR(2)* peut ainsi se faire en une seule ligne dans un langage fonctionnel.

L'idée centrale de notre contribution est d'appliquer un principe similaire à cette version immuable de *STR* aux MDD : la phase de filtrage au cœur du principe de *STR* est appliquée à un MDD. Une fonction (`filterMDD`) renvoie une « copie » filtrée du MDD, tout en conservant telles quelles les parties non modifiées du MDD, comme illustré sur la figure 3 : obtenir une version du MDD d'origine tel que $Y = a$ ne nécessite que de créer les deux nœuds M' et M_2. Par rapport à *STR(2)*, on bénéficie de toute la puissance de la compression apportée par les MDD. Ainsi, réduire le MDD après une modification de la première variable X ne nécessite que de parcourir et générer un seul nouveau nœud. Le gain potentiel par rapport à *STR* est de $O(\lambda)$ opérations ; et ce sans aucun surcout théorique en termes de complexité temporelle (avec cependant une restriction par rapport à *STR2*, discu-

Fonction filterMDD($M, ts, mod, \mathcal{X}$)

```
1  si M est vide alors retourner M
2  si mod = ⟨⟩ alors retourner M
3  si ts ≠ Ts(M) alors
4      Ts(M) ← ts
5      M' ← nouveau tableau de d MDD vides
6      X_m :: mod' ← mod
7      X :: X' ← X
8      si X_m = X alors
           /* Var modifiée au niveau courant */
9          pour chaque M[i] non vide | i ∈ dom(X)
           faire
10             M'[i] ← filterMDD(M[i], ts, mod', X')
11     sinon
12         pour chaque M[i] non vide faire
13             M'[i] ← filterMDD(M[i], ts, mod, X')
14     Last(M) ← si M = M' alors M sinon M'
15 retourner Last(M)
```

tée ci-après). Le fait que le MDD d'origine ne soit pas modifié permet de garder une trace de celui-ci et de réaliser un retour-arrière en temps constant.

Dans un premier temps, nous avons implanté un Trie strictement immuable en Scala. Un Trie est représenté par un tableau de d éléments, chacun contenant le sous-Trie (éventuellement vide) préfixé par la valeur de l'arc correspondant à l'index du tableau. Une fois le Trie construit, il est réduit en MDD à l'aide de la fonction `mddReduce` de Cheng & Yap [2] en $O(d\nu_{\text{trie}})$.

Le cœur de notre contribution est la fonction `filterMDD` ci-dessus. Elle prend en paramètre un MDD M, un *timestamp* ts, i.e., un nombre entier différent à chaque appel à l'algorithme, une séquence mod contenant la liste des variables modifiées depuis le dernier appel à l'algorithme (générée par la fonction `AC`), et la séquence $\mathcal{X}$ des variables impliquées par la contrainte, dans l'ordre qui a servi à générer le MDD. Les variables doivent apparaitre dans le même ordre dans mod et $\mathcal{X}$.

Les deux premières conditions d'arrêt de la fonction récursive sont triviales : le MDD est inchangé s'il est vide ou si aucune variable n'a été modifiée (lignes 1 et 2). Le *timestamp* sert à ne parcourir chaque nœud qu'une seule fois par appel à l'algorithme. Il est comparé à une valeur $Ts(M)$ associée à chaque nœud. Quand le nœud est traité, on réaffecte $Ts(M)$ à la nouvelle valeur de ts : si le même *timestamp* est à nouveau rencontré, il s'agit du même nœud que précédemment et on peut renvoyer le même résultat $Last(M)$ qu'à la précédente exécution (lignes 3, 4, 14 et 15).

On récupère le premier élément X_m et X de chacune

Fonction seekSuppMDD(M, ts, $supp$, $\mathcal{X}$, $seek$)

```
1  si ts ≠ Ts(M) alors
2      Ts(M) ← ts
3      X :: X' ← X
4      pour chaque M[i] non vide faire
5          si X ∩ seek = ∅ alors break
6          supp[X] ← supp[X] ∪ {i}
7          si supp[X] = dom(X) alors
8              seek ← seek − X
9          (supp, seek) ←
           seekSuppMDD(M[i], ts, supp, X', seek)
10 retourner (supp, seek)
```

des séquences mod et $\mathcal{X}$ (lignes 5 et 6) : si $X_m = X$, c'est que la variable correspondant au niveau courant du MDD a été modifiée : il faudra contrôler la validité de chacune des valeurs au niveau courant. Les appels récursifs se feront avec mod' (i.e., mod amputé de cette variable). Une fois toutes les variables modifiées traitées, il devient inutile de parcourir les nœuds situés plus bas dans le graphe (test de la ligne 2). Notez que ce test permet aussi de contrôler si l'on est arrivé en bas du graphe (i.e., sur une feuille) : à ce moment, toutes les variables ont forcément été traitées.

À chaque fois, on reconstitue un nouveau nœud M' (lignes 5 et boucle des lignes 9-10, ou 12-13) créé pour l'occasion. Si le nouveau nœud M' est identique à l'ancien M, il peut être libéré et M utilisé à la place. Il est possible qu'après un appel à la fonction `filterMDD`, d'autres nœuds se révèlent identiques. Par exemple sur la figure 3, les nœuds M_1 et M_2 sont identiques après filtrage et pourraient être fusionnés. Cependant, le cas semble suffisamment rare pour que chercher à réduire les MDD après chaque filtrage soit trop couteux, ce qui a été confirmé par des tests préliminaires.

La fonction `seekSuppMDD` recherche ensuite un support pour chaque valeur en parcourant le MDD : cette partie de l'algorithme est très proche de *MDDC*, à ceci près que grâce à la phase de filtrage préalable, on sait que toutes les valeurs apparaissant dans le MDD sont supportées. Les paramètres sont le MDD M, un *timestamp* ts (différent du *timestamp* de `filterMDD`), l'ensemble des supports trouvés $supp$ (initialement vide), la liste des variables impliquées par la contrainte dans l'ordre ayant servi à générer le MDD $\mathcal{X}$ et enfin la liste des variables pour lesquelles des valeurs n'ont pas de support trouvé $supp$ (initialement, toutes les variables impliquées par la contrainte). Ici encore, on utilise les *timestamps* pour ne pas traiter deux fois le même nœud (lignes 1 et 2) ; l'ensemble $seek$ permet d'arrêter le parcours si un support pour chaque va-

leur des variables situées plus bas dans le MDD a déjà été trouvé (similaire au *early cutoff optimization* de *MDDC*, mais à grain plus fin, lignes 5, 7 et 8). À la fin de l'exécution de l'algorithme, l'ensemble *seek* ne contient plus que les variables pour lesquelles des valeurs n'ont pas de support : seules ces variables doivent être filtrées.

Pour optimiser l'exécution des boucles des lignes 9 et 12 de `filterMDD` ou de la ligne 4 de `seekSuppMDD`, notre implantation maintient la liste des indices pour lesquels $M[i]$ n'est pas vide, en utilisant une technique proche de STR. Cette optimisation empêche de partager des nœuds entre plusieurs MDD, par exemple si plusieurs contraintes partagent la même relation ; l'implantation n'est plus non plus strictement immuable. D'autre part, une conception orientée objets nous permet de définir une implantation spécifique et optimisée des MDD vide et feuille (objets singleton), ainsi que des MDD à un seul fils ou à deux fils (i.e., nœuds unaires et binaires).

7 Complexités et discussion

Observation 1. *La complexité temporelle dans le pire des cas de* `filterMDD` *est en* $O(\nu_{\text{mdd}})$.

Au cours de la procédure de filtrage, chaque arc du MDD est considéré au maximum une fois : la complexité de l'opération de filtrage est donc en $O(\nu_{\text{mdd}})$. On remarque qu'il n'est pas possible d'ignorer complètement les niveaux intermédiaires auxquels les variables n'ont pas été modifiées : si des variables ont été modifiées plus bas dans l'arbre, il faut quand même traverser ces nœuds (c'est ce que fait la boucle de la ligne 12). Les propriétés d'incrémentalité de *MDDR* ne sont donc pas aussi fines que celles de *STR2*. Cependant, les *vérifications de domaine* ne sont faites qu'aux niveaux où les variables ont été modifiées. Comme chaque variable ne peut être modifiée que d fois, chaque arc ne va être contrôlé qu'au maximum d fois tout au long d'une branche de l'arbre de recherche.

Observation 2. *La complexité temporelle dans le pire des cas de* `seekSuppMDD` *est en* $O(\nu_{\text{mdd}})$.

Le test à la ligne 5 de la fonction `seekSuppMDD` peut être réalisé en temps amorti $O(k+\nu_{\text{mdd}})$ au cours d'une exécution de l'algorithme en maintenant la position de la dernière variable présente dans *seek* (dans l'ordre défini par la contrainte). Les autres opérations considèrent chaque arc au plus une fois, et on a $k \leq \nu_{\text{mdd}}$ si $\lambda > 0$ ($\lambda = 0$ implique un CN trivialement inconsistant), d'où l'observation.

Observation 3. *La complexité temporelle dans le pire des cas de* `AC` *tout au long d'une branche de l'arbre de recherche si toutes les contraintes sont filtrées par* MDDR *est en* $O(ekd\nu_{\text{mdd}})$.

Aucune propriété théorique d'incrémentalité n'a été observée sur `filterMDD` ni `seekSuppMDD`. Dans le pire des cas, chaque appel à l'algorithme concerne la modification de la variable située le plus bas dans le MDD, ce qui nécessite de parcourir tous les arcs du graphe. Cependant, le nombre de *vérifications de domaine* effectuées par *MDDR* sera en $O(ed\nu_{\text{mdd}})$ tout au long d'une branche de l'arbre de recherche. Après avoir filtré la relation et recherché les supports, il faut également supprimer les valeurs n'ayant pas de supports. Il est évident que ce cout est $O(nd)$ amorti sur toute la branche de l'arbre de recherche, chaque valeur ne pouvant être supprimée qu'une fois. Si chaque variable est impliquée par au moins une contrainte, on a $n \leq ek$: cet aspect est négligeable.

À titre de comparaison, la complexité temporelle dans le pire des cas de *STR2* est tout au long d'une branche de l'arbre de recherche de $O(ek^2d\lambda)$ ($O(ekd\lambda)$ pour les vérifications de domaine de la phase de filtrage), celle de *MDDC* est $O(ekd\nu_{\text{mdd}})$ (tant en termes d'opérations que de vérifications de domaine).

Observation 4. *Si toutes les contraintes sont filtrées par* MDDR*, la complexité spatiale dans le pire des cas de* `AC` *tout au long d'une branche de l'arbre de recherche est* $O(ekd\nu_{\text{mdd}})$.

Il est intéressant de noter que chaque arc supprimé ne nécessite de générer qu'un nœud à chaque niveau du MDD, soit $O(kd)$ arcs. Au total, au maximum $O(kd\nu_{\text{mdd}})$ arcs (par contrainte) peuvent ainsi être générés tout au long de la recherche. À titre de comparaison, les complexités spatiales de *STR2* et *MDDC* sont respectivement $O(ek\lambda)$ et $O(e\nu_{\text{mdd}})$.

L'algorithme *MDDC* de Cheng & Yap met en place une incrémentalité en enregistrant les nœuds valides et invalides : un nœud est valide si au moins un arc partant de ce nœud conduit à un nœud valide. L'information concernant les nœuds invalides peut être conservée entre deux appels successifs. *MDDR* va plus loin : en enregistrant une copie du MDD, *MDDR* est capable d'éliminer chaque *arc* non valide : l'incrémentalité obtenue est bien meilleure, et ce sans surcout en termes de complexité temporelle. De plus, *MDDR* permet d'exploiter plus finement les informations sur les variables modifiées fournies par l'algorithme de propagation `AC` (i.e., structure *mod* de `filterMDD`, à la manière de l'optimisation effectuée entre *STR* et *STR2*). *MDDC* tel que décrit par Cheng & Yap n'exploite aucunement les informations concernant les variables modifiées. Il est possible de modifier l'algorithme pour

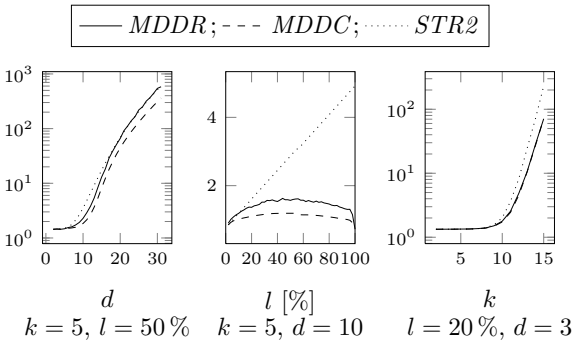


FIGURE 4 – Mémoire utilisée (en Mio) au 20e niveau d'un arbre de recherche simulé (médianes sur 50 exécutions).

les prendre en compte, mais les structures de données de *MDDC* ne permettent pas de le faire efficacement, puisque il faut vérifier à nouveau au moment de la recherche des supports si les arcs sont effectivement valides. Dans le cas où seule la variable tout en haut de l'arbre a été modifiée, *MDDR* permet de ne vérifier que $O(kd)$ valeurs alors que *MDDC* va potentiellement parcourir tout le graphe.

Finalement, *MDDR* nécessite d'instancier un tableau de taille d à chaque parcours de nœud. Cela n'a pas d'impact sur la complexité temporelle, mais les allocations de mémoire peuvent avoir un cout non négligeable en pratique. L'impact sur la complexité spatiale n'est pas non plus neutre. Comme nous le verrons expérimentalement, si *MDDR* est presque toujours plus rapide que *MDDC*, il est surtout intéressant par rapport à *STR2* quand $\nu_{\text{mdd}} \ll k\lambda$, ce qui implique des contraintes structurées et/ou un taux de remplissage des relations (l) important.

8 Expérimentations

Les algorithmes ont été implantés en langage Scala 2.10 sur notre solveur CSP4J 2 (le code source est disponible sur le site du projet), exécuté sur une machine virtuelle Java 7.0u9 (OpenJDK 64 bit Server) avec 2 GiB de mémoire de tas allouée. Le système d'exploitation est basé sur un noyau Linux 3.8.3-x86_64 fonctionnant sur un processeur Intel Core i5-2500S CPU @ 2,7 GHz, disposant de 8 GiB RAM. Les algorithmes de révision testés pour les contraintes d'arité supérieure à 2 définies en extension sont *STR2*, *MDDC* et *MDDR*. La stratégie de recherche est un arbre binaire basé sur une affectation des variables par $^{dom}/_{wdeg}$ et un ordre lexicographique décroissant pour les valeurs, avec redémarrages périodiques. La file de propagation des contraintes est ordonnée en FIFO. Notre implantation garantit que les arbres de recherche sont identiques quel que soit l'algorithme de révision utilisé.

Dans un premier temps, nous avons expérimenté les algorithmes sur des MDD générés aléatoirement (suivant une technique proposée dans [2]) d'après six paramètres (n, d, k, e, l, q) : respectivement le nombre de variables n, de valeurs d, l'arité des contraintes k, leur nombre e, leur taux de remplissage l (en termes de probabilité), et enfin un « taux de structure » q, qui définit la probabilité qu'a un nœud du MDD d'être identique à un autre. Pour construire ces MDD, on parcourt un Trie « complet » de profondeur k sur un alphabet de d valeurs. Chaque feuille est conservée avec une probabilité l, ou supprimée. Enfin, chaque nœud est remplacé avec une probabilité q par un autre nœud déjà généré.

La figure 4 montre l'évolution de l'utilisation de la mémoire pour stocker une contrainte quand l'un des paramètres d, l ou k augmente (on a gardé $q = 0$ ici). Pour montrer l'impact de la création de nouveaux nœuds de MDD au cours de la recherche avec *MDDR*, une branche d'arbre de recherche de profondeur 20 est simulée en retirant au hasard 5 % des valeurs et en effectuant une révision à chaque nœud (courbes du bas sur la figure). On remarque qu'avec les MDD, la mémoire atteint rapidement un maximum (en $O(d^k)$ avec une constante relativement faible) quand l augmente, et décroit quand l approche les 100 % (une relation « complète » se représente en dk arcs avec un MDD), là où une représentation en table voit sa taille augmenter linéairement. Cette courbe laisse supposer que les algorithmes présentés restent performants même avec des taux de remplissage très importants (i.e., contraintes définies sous forme de k-uplets interdits). On constate que *MDDR* utilise plus de mémoire que *MDDC* au cours de la recherche, mais cela reste raisonnable : la mémoire utilisée reste inférieure à *STR2* avec ces paramètres. Enfin, la courbe la plus adroite laisse supposer que le paramètre k n'a en réalité pas d'impact sur la complexité spatiale de *MDDR*, et permet de conjecturer que sa complexité spatiale en moyenne est en $O(ed\nu_{\text{mdd}})$.

Nous avons souhaité observer l'évolution des performances en fonction du taux de remplissage (auquel le nombre de k-uplets λ ou le nombre de nœuds des MDD ν_{mdd} sont directement liés) sur un problème de base tel que $(d, k) = (5, 5)$, puis en augmentant séparément d et k à 8. Le nombre de variables n a été ajusté en conséquence pour que les problèmes restent de difficulté moyenne (respectivement 20, 13 et 13). Enfin, le rapport entre le nombre de contraintes et le taux de remplissage est ajusté de sorte que les problèmes soient proches de la transition de phase d'après les théorèmes du modèle RD [13]. Pour une meilleure

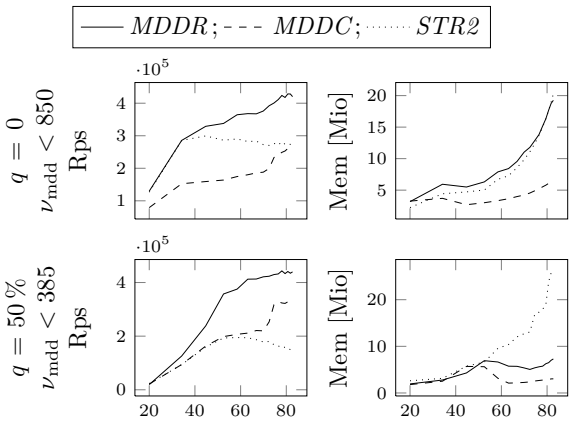
précision, nous avons fixé le nombre de contraintes et calculé le taux de remplissage : $l = \exp(-e^{-1} n \ln d)$. Rappelons que pour que ces théorèmes s'appliquent, il faut que $l \geq k^{-1}$, ce qui fixe la limite basse de nos tests. Enfin, nous avons testé des problèmes sans aucune structure ($q = 0$) ou au contraire assez structurés ($q = 50\,\%$). Des valeurs indicatives de e, λ et ν_{mdd} sont indiqués sur les graphes.

Les résultats sont présentés sur la figure 5. Rappelons que répartir les k-uplets interdits sur un plus grand nombre de contraintes réduit la capacité de filtrage de celles-ci, ce qui augmente la taille des arbres de recherche de manière exponentielle. Nous avons donc préféré évaluer la performance des algorithmes à partir du nombre de révisions par seconde plutôt que par le temps total de résolution : les courbes résultantes sont bien plus lisibles. Rappelons que l'arbre de recherche ainsi que la séquence de révisions sont strictement identiques entre les trois versions présentées.

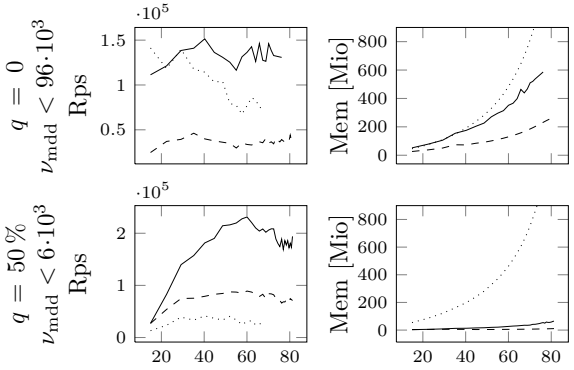
On constate que sur ces problèmes aléatoires, *MDDR* est toujours plus rapide que *MDDC*. Il est aussi plus rapide que *STR2* à l'exception des problèmes ayant une arité forte et un taux de remplissage faible : c'est dans ces conditions que la limitation des possibilités d'incrémentalité de *MDDR* par rapport à *STR2* a le plus d'impact. Sur des problèmes structurés, *STR2* est bien sûr de très loin le moins efficace. *MDDC* reste toujours le moins gourmand en espace mémoire, grâce à ses structures de données minimalistes.

Des tests préliminaires sur des problèmes plus structurés confirment les bons résultats de *MDDR*. Les contraintes de *séquence* (ici avec *cardinalité*) que l'on peut rencontrer dans les problèmes de *Car Sequencing* sont de forme pseudo-booléenne et peuvent être facilement modélisées sous forme de BDD (i.e., MDD à nœuds binaires) [3]. Les contraintes sont de très forte arité (jusque 400 sur les données de la *CSPLib* [4]) ; il n'est pas envisageable de les résoudre avec des tables, mais leur représentation sous forme de BDD est très compacte (cf table 1). Les problèmes de *Knapsack* sont formés de deux grandes contraintes linéaires ($\sum_{i=1}^{k} c_i x_i = 0$) et ont un comportement analogue.

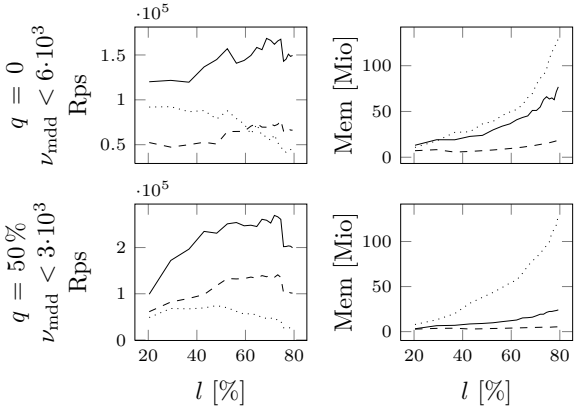
Les problèmes de voyageur de commerce (*TSP*) ou de mots-croisés (*CrossWord*) illustrent une limite de *MDDR* (et *MDDC*) par rapport à *STR2* : quand le MDD ne compresse pas efficacement la liste des k-uplets (on a ici $\nu_{\text{mdd}} > \lambda$), *STR2* se révèle la meilleure implémentation de l'algorithme de filtrage. Les performances de *MDDR* restent cependant raisonnables.



(a) $(n, d, k) = (20, 5, 5)$; $e = 20 \ldots 170$; $\lambda = 625 \ldots 3 \cdot 10^3$



(b) $(n, d, k) = (13, 5, 8)$; $e = 11 \ldots 101$; $\lambda \approx 58 \cdot 10^3 \ldots 318 \cdot 10^3$



(c) $(n, d, k) = (13, 8, 5)$; $e = 17 \ldots 117$; $\lambda \approx 7 \cdot 10^3 \ldots 26 \cdot 10^3$

FIGURE 5 – Vitesse de propagation (révisions par seconde) et mémoire utilisée pour résoudre des problèmes aléatoires. Chaque point est une médiane de 25 problèmes générés et résolus.

					MDDR		*MDDC*		*STR2*	
	d	k	λ	ν_{mdd}	trps	mem	trps	mem	trps	mem
carseq-100	23	57	10^{22}	2 132	**184**	8	101	**3**	–	out
carseq-200	24	80	10^{39}	6 151	**190**	42	64	**5**	–	out
carseq-300	24	97	10^{52}	7 302	**162**	71	65	**8**	–	out
carseq-400	24	111	10^{91}	16 772	**122**	203	35	**12**	–	out
knapsack	13	201	10^{48}	474 697	**5**	98	1	**12**	–	out
renault	20	9	17 733	876	**37**	**7**	31	**7**	24	15
crossword	26	15	36 192	39 557	10	20	6	8	**19**	**5**
tsp	10^3	3	26 872	27 389	1 002	16	211	**6**	**1 054**	7

TABLE 1 – Résultats sur problèmes structurés : pour chaque classe d'instances, métriques principales (moyenne+ 2,3 × écart-type), la vitesse de recherche en milliers de révisions par seconde, et la mémoire requise (en Mio, médianes).

9 Conclusion

Dans cet article, nous avons présenté *MDDR*, un algorithme de révision de contraintes définies sous forme de MDD utilisant une file de propagation à gros grain. Cette structure permet de modéliser efficacement des contraintes définies sous forme d'une liste exhaustive de k-uplets autorisés, ainsi que certaines contraintes structurées (e.g., contraintes pseudo-booléennes). Ce nouvel algorithme est plus rapide en pratique que l'algorithme *MDDC* proposé par Cheng & Yap en 2010 [2], au prix d'un surcout raisonnable en termes d'espace. Sur certains types de contraintes (i.e., sans structure et à taux de remplissage faible), les MDD ne compressent pas efficacement les tables. Les performances de *MDDR* restent cependant comparables à *STR2*, et il est possible de chercher à améliorer la compression en réordonnant les variables.

Nous avons également montré qu'il était possible d'implémenter des algorithmes de révision efficaces en utilisant exclusivement des structures de données immuables. Ces structures de données facilitent considérablement l'implantation d'algorithmes parallélisés. Il s'agit d'une perpective intéressante de ce travail.

Références

[1] B. BOLLIG et I. WEGENER. « Improving the variable ordering of OBDDs is NP-complete ». Dans : *IEEE Transactions on Computers* 45.9 (1996), p. 993–1002.

[2] K.C.K. CHENG et R.H.C. YAP. « An MDD-based GAC algorithm for positive and negative table constraints and some global constraints ». Dans : *Constraints* 15.2 (2010), p. 265–304.

[3] N. EÉN et N. SÖRENSSON. « Translating pseudo-boolean constraints into SAT ». Dans : *JSAT* 2 (2006), p. 1–26.

[4] I.P. GENT et al. *A problem library for constraints*. `http://csplib.org/`. 1996–2012.

[5] I.P. GENT et al. « Data structures for generalised arc consistency for extensional constraints ». Dans : *Proc. AAAI'2007*. 2007, p. 191–197.

[6] G. KATSIRELOS et T. WALSH. « A Compression Algorithm for Large Arity Extensional Constraints ». Dans : *Proc. CP'2007*. 2007, p. 379–393.

[7] C. LECOUTRE. « STR2 : Optimized Simple Tabular Reduction for Table Constraints ». Dans : *Constraints* 16.4 (2011), p. 341–371.

[8] C. LECOUTRE, C. LIKITVIVATANAVONG et R.H.C. YAP. « A path-optimal GAC algorithm for table constraints ». Dans : *Proc. ECAI'2012*. 2012, p. 510–515.

[9] C. LECOUTRE et R. SZYMANEK. « GAC for Positive Table Constraints ». Dans : *Proc. CP'06*. Nantes, France, 2006, p. 284–298.

[10] O. LHOMME et J.-C. RÉGIN. « A fast arc consistency algorithm for n-ary constraints ». Dans : *Proc. AAAI'2005*. 2005, p. 405–410.

[11] J.-B. MAIRY, P. van HENTENRYCK et Y. DEVILLE. « An Optimal Filtering Algorithm for Table Constraints ». Dans : *Proc. CP'2012*. 2012.

[12] J.R. ULLMANN. « Partition search for non-binary constraint satisfaction ». Dans : *Information Science* 177 (2007), p. 3639–3678.

[13] K. XU et al. « A simple model to generate hard satisfiable instances ». Dans : *Artificial Intelligence* 171 (2007), p. 514–534.

Assemblage de vins sous contraintes

Philippe Vismara[1,2] **Remi Coletta**[1] **Gilles Trombettoni**[1]

[1] LIRMM, UMR5506 Université Montpellier II - CNRS, Montpellier, France
[2] MISTEA, UMR729 Montpellier SupAgro - INRA, Montpellier, France
{Philippe.Vismara,Remi.Coletta,Gilles.Trombettoni}@lirmm.fr

Résumé

En œnologie, l'assemblage consiste à mélanger plusieurs cuvées afin d'obtenir un vin cible. Des progrès dans l'analyse des arômes permettent aujourd'hui de mesurer un ensemble de composés chimiques influençant le goût d'un vin. Il devient ainsi possible de concevoir un outil d'aide à la décision pour le problème suivant : étant donné un ensemble de vins cibles à produire, quelles quantités doit-on prélever dans chaque cuvée afin de produire des vins qui respectent des contraintes de concentration en arômes, de volumes, de degré alcoolique et de prix. L'article décrit la modélisation de ce problème sous forme d'une optimisation Min-Max (pour minimiser les écarts obtenus avec les concentrations souhaitées pour chaque critère aromatique) sous contraintes numériques linéaires et quadratiques, et son traitement efficace avec le *branch and bound* à intervalles de `Ibex`.

Abstract

Assemblage consists in blending base wines in order to create a target wine. Recent advances in aroma analysis allow us to measure chemical compounds impacting the taste of wines. This chemical analysis makes it possible to design a decision tool for the following problem : given a set of target wines, determine which volumes must be extracted in each cuvée to produce wines that satisfy constraints on aroma concentration, volumes, alcohol levels and price. This paper describes the modeling of wine assemblage as a non linear constrained Min-Max problem (minimizing the gap to the desired concentrations for every aromatic criterion) efficiently handled by the `Ibex` interval branch and bound.

1 Introduction

L'assemblage est l'art de mélanger subtilement des vins de différentes parcelles et/ou de différents cépages, chacun apportant ses propres spécificités. Le choix des assemblages élaborés est généralement effectué par des œnologues. Si la qualité de leur travail est inégalable, une limite forte dans le nombre de dégustations quotidiennes tient à la saturation des goûts. C'est pourquoi la société Nyseos, qui nous a soumis ce problème, propose des outils d'analyse chimique permettant de soulager l'œnologue d'une partie du travail de dégustation. Ces outils analysent les arômes d'un vin en mesurant un ensemble de composés chimiques qui influencent le goût du vin [6]. Il devient ainsi possible de concevoir un outil d'aide à la décision pour le problème suivant : étant donné un ensemble de vins cibles à produire, quelles quantités doit-on prélever dans chaque cuvée pour produire des vins qui respectent des contraintes de concentration en arômes, de volumes, de degré alcoolique et de prix.

Moore et Griffin ont montré que la concentration en arômes d'un mélange de vins obéit à des équations linéaires [10]. Cependant, d'autres impératifs d'assemblage peuvent imposer des contraintes non linéaires. C'est notamment le cas pour la prédiction de la couleur d'un vin cible en fonction de la couleur des vins de base. Même si la modélisation de cette contrainte n'est pas encore aboutie, nous savons déjà qu'elle ne sera pas linéaire. Ce résultat est d'ailleurs confirmé par d'autres études [7]. Une autre difficulté provient du fait qu'il n'est pas possible de transvaser une trop faible quantité de vin entre deux cuves à cause du volume inévitablement perdu dans les tuyaux et du coût des manipulations. Comme nous le montrons dans l'article, cela conduit à définir une contrainte disjonctive qui peut être modélisée par des variables booléennes ou des contraintes non linéaires.

La principale étude sur l'aide à l'assemblage de vin a été publiée dans [7]. Les auteurs utilisent un réseau de neurones pour déterminer les quantités devant être extraites de chaque cuve de base afin de concevoir un vin respectant des critères aromatiques prédéfinis. Dans cette étude, les concentrations en arômes n'ont pas été mesurées chimiquement mais évaluées gusta-

tivement par un panel de personnes (étudiants). Le réseau de neurones est utilisé pour réaliser une optimisation multicritères afin d'ajuster chaque arôme. Il est ainsi difficile de comparer ce travail avec notre démarche dans la mesure où nous réalisons une optimisation mono-critère. Par ailleurs, [7] ne fournit aucune indication de performance (temps CPU).

Nous proposons dans cet article une modélisation mathématique du problème d'assemblage de vin. Le problème se modélise comme un programme non linéaire mixte (discret et continu) que nous transformons en CSP numérique non linéaire traité par des méthodes à intervalles. Pour minimiser, dans chaque vin cible, l'écart entre les concentrations aromatiques souhaitées et celles obtenues, nous modélisons ce problème d'optimisation à l'aide d'un Min-Max que nous parvenons à débarrasser de ses opérateurs valeur absolue et max. Le modèle a été particulièrement étudié pour être traité efficacement par des méthodes à intervalles. Nous montrons des premiers résultats très satisfaisants obtenus sur deux assemblages réels.

2 Le problème de l'assemblage de vins

Les éléments permettant de décrire un problème d'assemblage sont illustrés à la figure 1.

On dispose d'un ensemble de cuvées de base, appelées *bases*, numérotées de 1 à $\mathcal{B}$. On connaît le volume vol_b de chaque base $b \in 1..\mathcal{B}$. Pour des raisons matérielles, on ne peut pas toujours vider entièrement un fond de cuve et l'on doit laisser un fond minimum noté s_b^- (on a : $0 \leq s_b^- \leq \mathrm{vol}_b$).

Toutes les bases sont analysées afin de mesurer leur teneur en composés aromatiques. Ces composés sont numérotés de 1 à $\mathcal{A}$ et on notera $c_{b,a}$ la concentration en arôme a dans la base b.

Un outil d'aide à l'assemblage doit permettre d'étudier globalement l'assemblage de plusieurs vins en puisant dans le même ensemble de bases. On dispose donc d'un ensemble de vins cibles numérotés de 1 à $\mathcal{W}$. Pour chaque vin w, on souhaite produire un volume de vin noté $\hat{\mathrm{vol}}_w$. Ce volume peut cependant fluctuer jusqu'à une borne maximale (resp. minimale) notée vol_w^+ (resp. vol_w^-). Ces bornes permettent de ne pas produire un volume trop important pour être vendu ou au contraire de garantir un volume qui a été commandé. Au final, le volume V_w du vin w devra être le plus proche possible de $\hat{\mathrm{vol}}_w$ tout en respectant :

$$\forall w \in 1..\mathcal{W}, \quad \mathrm{vol}_w^- \leq V_w \leq \mathrm{vol}_w^+ \tag{1}$$

Chaque vin cible w étant obtenu en assemblant du vin issu de plusieurs bases, on notera $V_{w,b}$ le volume de vin w provenant de la base b. Le volume total du vin w est donc la somme de ces volumes partiels :

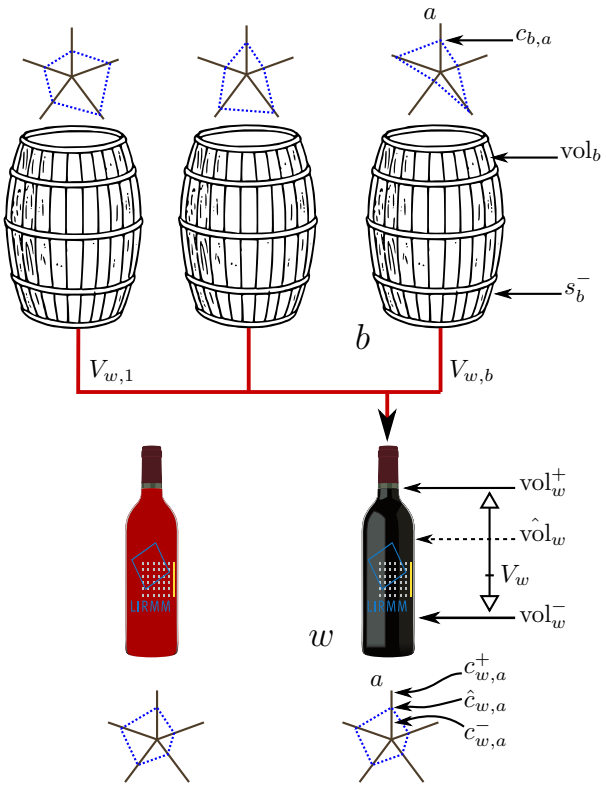


FIGURE 1 – L'assemblage de vins

$$\forall w \in 1..\mathcal{W}, \quad V_w = \sum_{b=1}^{\mathcal{B}} V_{w,b} \tag{2}$$

On doit veiller à ce que la somme des volumes soutirés dans une même base b respecte les contraintes de volume autorisé pour le surplus restant en fond de cuve. On pose la contrainte :

$$\forall b \in 1..\mathcal{B}, \quad s_b^- \leq \mathrm{vol}_b - \sum_{w=1}^{\mathcal{W}} V_{w,b} \tag{3}$$

D'un point de vue pratique, il est impossible de soutirer un trop faible volume d'une cuve, notamment à cause du résidu perdu dans les tuyaux. Si δ_V désigne ce volume minimal, on pose la contrainte *disjonctive* suivante :

$$\forall w \in 1..\mathcal{W}, \forall b \in 1..\mathcal{B}, \ (V_{w,b} = 0) \ \vee \ (\delta_V \leq V_{w,b}) \tag{4}$$

En plus du volume souhaité, chaque vin cible est décrit par un profil aromatique recherché à l'intérieur de plages bien définies. Pour chaque vin cible w, on notera $\hat{c}_{w,a}$ la concentration souhaitée en arôme a. La

concentration minimale (resp. maximale) tolérée sera notée $c^-_{w,a}$ (resp. $c^+_{w,a}$).

On peut donc exprimer le fait que la concentration $C_{w,a}$ en arôme a du vin w doit être comprise entre ces deux bornes : pour tout $w \in 1..\mathcal{W}$ et pour tout $a \in 1..\mathcal{A}$, on a :

$$c^-_{w,a} \;\leq\; \frac{1}{V_w} \sum_{b=1}^{\mathcal{B}} (V_{w,b} \,.\, c_{b,a}) \;\leq\; c^+_{w,a} \qquad (5)$$

De manière analogue, on peut imposer des contraintes sur la concentration en alcool ou le prix au litre des vins cibles. Ces caractéristiques peuvent être considérées comme deux arômes supplémentaires.

3 Un modèle MINLP

Nous pouvons modéliser le problème de l'assemblage de vin comme un programme non linéaire mixte (MINLP). Nous verrons à la section 4 comment transformer ce MINLP en un CSP numérique (NCSP) traité par des méthodes à intervalles. De ce fait, nous allons directement traduire les contraintes de bornes par des domaines bornés, c'est-à-dire des intervalles.

3.1 Variables

Pour tenir compte des contraintes sur le volume minimum pouvant être soutiré (cf. (4)), nous créons pour chaque vin $w \in 1..\mathcal{W}$ et chaque base $b \in 1..\mathcal{B}$:

- une variable $P_{w,b}$ de domaine 0/1 et
- une variable $V'_{w,b}$ de domaine $D(V'_{w,b}) = [\delta_V, \min(\mathrm{vol}^+_w, \mathrm{vol}_b)]$ représentant le volume transvasé de la base b vers le vin cible w.
  (nous avons : $V_{w,b} \equiv P_{w,b} \,.\, V'_{w,b}$. L'introduction de la variable 0/1 évite la définition explicite de la contrainte disjonctive (4).)

Pour chaque vin $w \in 1..\mathcal{W}$, nous définissons aussi une variable V_w de domaine $[\mathrm{vol}^-_w, \mathrm{vol}^+_w]$ représentant son volume (cf. (1)).

3.2 Contraintes

Le système de contraintes de notre MINLP est ainsi défini :

- la contrainte de channeling (2) devient :

$$\forall w,\; V_w - \sum_{b=1}^{\mathcal{B}} (P_{w,b} \,.\, V'_{w,b}) = 0 \qquad (2.\mathrm{i})$$

- la contrainte (3) sur les surplus est similaire :

$$\forall b \in 1..\mathcal{B}, s^-_b \leq \mathrm{vol}_b - \sum_{w=1}^{\mathcal{W}} (P_{w,b} \,.\, V'_{w,b}) \qquad (3.\mathrm{i})$$

- pour améliorer les performances, nous avons introduit une contrainte redondante par rapport à (3.i). Cette contrainte garantit simplement que le volume total des vins cibles est inférieur au volume total des cuves de base :

$$\sum_{w=1}^{\mathcal{W}} V_w \leq \sum_{b=1}^{\mathcal{B}} vol_b \qquad (6)$$

La règle (5) concernant chaque concentration en arômes est décomposée en deux contraintes où chaque partie a été multipliée par le volume positif V_w. On a donc, $\forall w \in 1..\mathcal{W}$ et $\forall a \in 1..\mathcal{A}$:

pour la borne inférieure,

$$0 \leq \sum_{b=1}^{\mathcal{B}} V_{w,b} \,.\, (c_{b,a} - c^-_{w,a})$$

d'où :

$$0 \leq \sum_{b=1}^{\mathcal{B}} P_{w,b} \,.\, V'_{w,b} \,.\, (c_{b,a} - c^-_{w,a}) \qquad (5.\mathrm{i}\text{ -})$$

et pour la borne supérieure :

$$0 \leq \sum_{b=1}^{\mathcal{B}} V_{w,b} \,.\, (c^+_{w,a} - c_{b,a})$$

d'où :

$$0 \leq \sum_{b=1}^{\mathcal{B}} P_{w,b} \,.\, V'_{w,b} \,.\, (c^+_{w,a} - c_{b,a}) \qquad (5.\mathrm{i}\text{ +})$$

3.3 Un Min-Max pour optimiser la qualité des vins

Dans l'assemblage de vins, le principal critère de qualité concerne les profils aromatiques. Cependant, il est tout à fait possible d'étendre les définitions qui vont suivre au degré alcoolique ou au prix des vins cibles.

Nous avons vu que chaque vin cible w est caractérisé par un ensemble de concentrations souhaitées $\hat{c}_{w,a}$ pour chaque arôme a mesuré. Pour optimiser la qualité d'un vin w, nous pouvons minimiser la somme des différences entre chaque concentration $\hat{c}_{w,a}$ souhaitée et la concentration $C_{w,a}$ obtenue (cf. (5)). De plus, nous voulons minimiser l'erreur maximale sur l'ensemble des vins cibles :

$$\max_{w \in 1..\mathcal{W}} \Omega_w (\lambda_{vol_w} \,.\, e_{vol_w} + \sum_{a \in 1..\mathcal{A}} (\lambda_{w,a} \,.\, e_{w,a})) \qquad (7)$$

où :

- Ω_w est un paramètre indiquant l'importance du vin w (on suppose $\Omega_w \in [0,1]$). Il est ainsi possible d'accorder plus d'importance à certains vins cibles, notamment ceux de grande qualité.

– $e_{w,a}$ représente l'écart entre la concentration souhaitée $\hat{c}_{w,a}$ et celle obtenue $C_{w,a}$.
– e_{vol_w} est l'écart entre le volume $\hat{\text{vol}}_w$ désiré pour le vin w et le volume obtenu V_w.
– $\lambda_{w,a} \in [0,1]$ définit le poids de l'arôme a pour le vin w et $\lambda_{vol_w} \in [0,1]$ le poids accordé à son volume. Pour un vin w donné, on suppose que $\lambda_{vol_w} + \sum_{a \in 1..\mathcal{A}} \lambda_{w,a} = 1$.

Tous les paramètres sont mesurés avec un certain taux d'erreur. ε_a représente la marge d'erreur possible sur une mesure de l'arôme a. Nous cherchons donc à minimiser l'écart entre $\hat{c}_{w,a}$ et $C_{w,a}$ dans la limite de la marge d'erreur ε_a. En d'autres termes, si l'écart entre la concentration souhaitée et celle obtenue est inférieur au taux d'erreur, il sera considéré comme nul dans la fonction objectif. La variable $e_{w,a}$ décrit donc l'erreur de concentration normalisée pour l'arôme a dans le vin w :

$$\boldsymbol{e_{w,a}} = \max\left(\frac{|C_{w,a} - \hat{c}_{w,a}|}{\hat{c}_{w,a}} - \varepsilon_a, 0\right) \tag{8}$$

Nous pouvons décrire par une expression similaire l'écart e_{vol_w} entre le volume V_w obtenu pour le vin cible w et le volume souhaité $\hat{\text{vol}}_w$:

$$\boldsymbol{e_{vol_w}} = \max\left(\frac{\hat{\text{vol}}_w - V_w}{\hat{\text{vol}}_w} - \varepsilon_{vol}, 0\right) \tag{9}$$

Par rapport à la formule précédente, la suppression de la valeur absolue signifie simplement que l'erreur n'est pas prise en compte si le volume V_w obtenu est compris entre le volume souhaité $\hat{\text{vol}}_w$ et le volume maximum autorisé vol_w^+. Cette règle est illustrée par la figure 2.

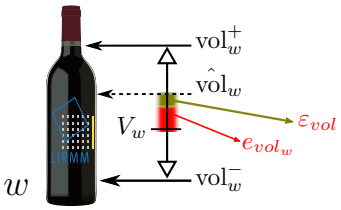


FIGURE 2 – Visualisation de l'écart e_{vol_w} sur le volume

De façon classique pour un problème de Min-Max, nous définissons une variable $E \in [0, +\infty]$ à minimiser, en respectant les contraintes suivantes :

$$\forall w \in 1..\mathcal{W}, \quad \Omega_w(e_{vol_w}.\lambda_{vol_w} + \sum_{a \in 1..\mathcal{A}} (e_{w,a} \,.\, \lambda_{w,a})) \leq E \tag{10}$$

Suppression des opérateurs max et valeur absolue

Afin d'améliorer les performances, nous avons cherché à supprimer les opérateurs max et valeur absolue du modèle précédent. L'opérateur max peut être défini par :

$$e = max(x, y) \;\equiv\; e \geq x \wedge e \geq y \wedge (e = x \vee e = y).$$

De plus, si la quantité e doit être minimisée pour une raison quelconque, la dernière conjonction peut être supprimée, ce qui simplifie la définition de l'opérateur max. Nous pouvons appliquer cette simplification à l'équation (8). Puisque chaque $\lambda_{w,a}$ est positif, minimiser E revient à minimiser chaque variable $e_{w,a}$. On a donc, $\forall w \in 1..\mathcal{W}, \forall a \in 1..\mathcal{A}$:

$$((e_{w,a} + \varepsilon_a)\,\hat{c}_{w,a} \geq |C_{w,a} - \hat{c}_{w,a}|) \;\wedge\; (e_{w,a} \geq 0) \tag{11}$$

La même simplification peut s'appliquer à (9), ce qui donne, $\forall w \in 1..\mathcal{W}$:

$$((e_{vol_w} + \varepsilon_{vol})\,\hat{\text{vol}}_w \geq (\hat{\text{vol}}_w - V_w)) \wedge\; (e_{vol_w} \geq 0) \tag{12}$$

Nous pouvons aussi supprimer l'opérateur *valeur absolue* qui s'exprime sous la forme d'un *max* :

$$\begin{aligned} e = |x| \quad &\equiv \quad e = max(x, -x) \\ &\equiv \quad e \geq x \;\wedge\; e \geq -x \;\wedge\; (e = x \vee e = -x) \end{aligned}$$

Comme précédemment, si la quantité e doit être minimale pour une raison quelconque, nous pouvons remplacer l'opérateur *valeur absolue* par deux inégalités. Cette simplification peut s'appliquer à l'équation (11). Rappelons en effet que chaque variable $e_{w,a}$ doit être minimisée et notons que $\hat{c}_{w,a}$ est toujours positif. Il s'ensuit, $\forall w \in 1..\mathcal{W}, \forall a \in 1..\mathcal{A}$, :

$$(e_{w,a} + \varepsilon_a)\,\hat{c}_{w,a} \geq \frac{1}{V_w} . \sum_{b=1}^{\mathcal{B}} (P_{w,b}.V'_{w,b}.c_{b,a}) - \hat{c}_{w,a}$$

$$(e_{w,a} + \varepsilon_a)\,\hat{c}_{w,a} \geq -\frac{1}{V_w} . \sum_{b=1}^{\mathcal{B}} (P_{w,b}.V'_{w,b}.c_{b,a}) + \hat{c}_{w,a}$$

En multipliant chaque côté de ces inégalités par le volume V_w qui est positif, on obtient finalement trois catégories de contraintes : $\forall w \in 1..\mathcal{W}, \forall a \in 1..\mathcal{A}$,

$$e_{w,a} \geq 0 \tag{13}$$

$$V_w.(e_{w,a} + \varepsilon_a + 1)\,.\,\hat{c}_{w,a} - \sum_{b=1}^{\mathcal{B}} (P_{w,b}.V'_{w,b}.c_{b,a}) \geq 0 \tag{14}$$

$$V_w.(e_{w,a} + \varepsilon_a - 1)\,.\,\hat{c}_{w,a} + \sum_{b=1}^{\mathcal{B}} (P_{w,b}.V'_{w,b}.c_{b,a}) \geq 0 \tag{15}$$

En conséquence, nous avons réussi à supprimer de notre modèle initial tous les opérateurs max et valeur absolue. Bien que le résolveur `Ibex` que nous allons utiliser sache gérer ces opérateurs, les performances seront améliorées et ce modèle simplifié peut être mis en œuvre dans d'autres résolveurs.

3.4 Synthèse : le modèle MINLP final

En plus des variables $P_{w,b}$, $V'_{w,b}$ et V_w définies à la section 3.1, nous avons introduit de nouvelles variables pour le Min-Max : une variable $E \in [0, +\infty]$, $\mathcal{W}.\mathcal{A}$ variables $e_{w,a} \in [0, 1]$ (qui englobent la contrainte unaire (13)) et $\mathcal{W}$ variables $e_{vol_w} \in [0, 1]$ qui englobent la contrainte unaire (12).

En plus des contraintes (2.i), (3.i), (6), (5.i-), (5.i+) définies à la section 3.2, nous ajoutons de nouvelles contraintes pour le Min-Max : (10), (12), (14), (15). La fonction objectif consiste simplement à minimiser la valeur de la variable E.

4 Résolution du MINLP par un B&B à intervalles

Le MINLP détaillé précédemment peut être résolu par n'importe quel résolveur de MINLP [12, 2]. Cependant, aucun d'entre eux n'est *rigoureux*. Cela signifie qu'il peuvent ne pas trouver la meilleure solution à cause d'erreurs d'arrondi de l'arithmétique en virgule flottante à certaines étapes de la résolution. À notre connaissance, un seul *branch and bound* (B & B) rigoureux à intervalles, nommé `IBBA` [11], est doté d'un mécanisme simple de manipulation des variables intégrales. Ainsi, comme :

- notre modèle d'assemblage de vins ne contient qu'un seul type de variables 0/1,
- le résolveur à intervalles `Ibex` [5, 4] fournit un B & B très efficace nommé `IbexOpt` [13], et
- les auteurs ont une bonne connaissance de `Ibex`,

nous avons décidé de coder le MINLP sous la forme d'un NCSP, c'est-à-dire un système standard de contraintes continues non linéaires sur des nombres réels. Pour cela, les variables 0/1 sont codées par des variables réelles $P'_{w,b}$ de domaine $[0, 1]$. Afin de garantir que ces variables ne pourront prendre que la valeur 0 ou 1, nous ajoutons simplement la contrainte quadratique suivante :

$$\forall w \in 1..\mathcal{W} \text{ et } \forall b \in 1..\mathcal{B},\ 4(P_{w,b} - \frac{1}{2})^2 = 1 \qquad (16)$$

Cela signifie que les contraintes disjonctives initiales, qui sont à l'origine des contraintes mixtes dans le MINLP, sont gérées par des contraintes continues quadratiques dans le NCSP.

4.1 Optimisation globale sous contraintes par un B&B sur intervalles

Un problème d'optimisation globale sous contraintes est défini de la manière suivante.

Definition 1 (Optimisation)
Considérons un vecteur de variables $x = (x_1, ..., x_i, ...x_n)$ *de domaine* $[x] = [x_1] \times \cdots \times [x_n]$, *une fonction continue* $f : \mathbb{R}^n \to \mathbb{R}$, *des fonctions* $g : \mathbb{R}^n \to \mathbb{R}^m$ *et* $h : \mathbb{R}^n \to \mathbb{R}^p$. *(Nous notons* $g = (g_1, ..., g_m)$ *et* $h = (h_1, ..., h_j, ..., h_p)$.*)*

Soit le système $S = (f, g, h, [x])$, *le problème d'optimisation globale sous contraintes consiste à trouver*

$$\min_{x \in [x]} f(x) \quad \textit{soumis à} \quad g(x) \leq 0 \wedge h(x) = 0.$$

f est appelée fonction objectif; g et h sont des contraintes d'inégalité et d'égalité respectivement.

Notre optimiseur global (sous contraintes) `IbexOpt` [13] calcule un vecteur de nombres à virgule flottante x ϵ-minimisant [1] :

$$f(x) \ s.t. \ g(x) \leq 0 \wedge (-\epsilon_{eq} \leq h(x) \leq +\epsilon_{eq}).$$

Notons que les égalités $h_j(x) = 0$ sont relâchées par des équations "épaisses" $h_j(x) \in [-\epsilon_{eq},\ +\epsilon_{eq}]$, soit deux inégalités : $-\epsilon_{eq} \leq h_j(x) \leq +\epsilon_{eq}$. `IbexOpt` garantit l'optimum global du système relâché, quoique ϵ_{eq} puisse être souvent choisi arbitrairement petit. (La plupart des optimiseurs globaux comme `Baron` [12] ou `Couenne` [2] ne peuvent fournir aucune garantie.)

Dans notre problème de mélange de vins, nous prenons une constante ϵ^1_{eq} égale à `1e-4` dans les contraintes d'égalité (16). Un autre ϵ^2_{eq} est fixé à `1e-1` dans les contraintes d'égalité (2.i). Cette incertitude correspond à un décilitre, soit moins que 0.1% des volumes des vins cibles (au moins 500 litres).

Cela signifie que les volumes sont calculés avec une approximation significativement meilleure que les erreurs inéluctables faites pendant le mélange effectif, c'est-à-dire les erreurs induites par les mesures et par la perte de matière résiduelle pendant le transvasement d'un vin d'une cuve de base à la cible [2].

4.2 Opérateurs algorithmiques d'Ibex

`IbexOpt` est implanté en `Ibex` (Interval Based EXplorer) et enrichit cette bibliothèque en `C++` dédiée à la "résolution" par intervalles [5].

`IbexOpt` suit un schéma de type *Branch & Contract & Bound* sur les intervalles [13] . Le processus débute avec une boîte initiale $[x]$ subdivisée récursivement par l'opération de branchement. L'arbre est parcouru en meilleur d'abord, en sélectionnant à chaque itération

1. ϵ-minimiser $f(x)$ signifie minimiser $f(x)$ avec une précision ϵ, c'est-à-dire trouver x tel que pour tout y, nous avons $f(y) \geq f(x) - \epsilon$.

2. Notons qu'une prochaine version de `Ibex` permettra de rigoureusement ϵ-optimiser une fonction objectif sous des contraintes d'égalités vraies (non relaxées).

la boîte de coût inférieur le plus petit. Les opérations suivantes s'enchaînent sur chaque boîte (nœud) traité :
Brancher : une variable x_i est choisie et son intervalle $[x_i]$ est découpé en deux sous-intervalles. Les deux sous-boîtes produites subissent les deux opérations suivantes.
Contracter : un processus de filtrage contracte la boîte étudiée (c'est-à-dire améliore les bornes de ses intervalles) sans perte de solutions.
Borner : l'amélioration de la borne inférieure est similiaire à une contraction (en considérant une variable correspondant au coût de l'objectif) : la borne inférieure garantit qu'aucune solution ne se trouve plus bas. L'amélioration de la borne supérieure revient à trouver un point réalisable de coût le plus bas possible, de manière à couper des branches de l'arbre de recherche de coût supérieur.

Le processus débute avec une boîte initiale $[x]$ et s'arrête quand la différence entre la borne supérieure et la borne inférieure atteint une précision donnée (ϵ-optimisation) ou encore quand tous les nœuds explorés atteignent une taille inférieure à une précision donnée en entrée.

A chaque nœud du B&B, `IbexOpt` appelle des opérateurs efficaces pour réduire l'espace de recherche et améliorer les bornes inférieure et supérieure de la fonction objectif :

- L'algorithme de propagation de contraintes de référence (pour le continu) `HC4` [3, 9] est d'abord utilisé pour contracter la boîte traitée.
- Ensuite, l'opérateur algorithmique `X-Newton` utilise une forme de Taylor sur intervalles spécifique pour convexifier l'espace de recherche. Il contracte la boîte et améliore la borne inférieure [1].
- Deux algorithmes cherchent enfin à améliorer la borne supérieure en extrayant d'abord de manière heuristique une région *intérieure* (ou entièrement réalisable) ne contenant que des points - solutions - satisfaisant les contraintes. D'où l'intérêt de relaxer légèrement les équations. En simplifiant quelque peu, l'algorithme `InHC4` est un algorithme dual de `HC4` ; `InnerPolytope` est un algorithme dual de `X-Newton`.

La stratégie d'optimisation par défaut de `Ibex` utilise comme heuristique de branchement (bissection) la variante `SmearSumRel` de l'heuristique de Kearfott basée sur la fonction *Smear* [8]. Les heuristiques de branchement `SmearSumRel` et `SmearMaxRel` sont décrites dans [13].

5 Expérimentations

Nous avons modélisé et résolu plusieurs instances d'assemblage de vins. Nous verrons à la section 5.6 que notre démarche a également été validée par une séance de dégustations réelle.

Pour réaliser l'optimisation, nous avons imposé une précision $\epsilon =$`1e-4` sur le coût (objectif). Cette précision est très inférieure au taux d'erreur ϵ_a des outils chimiques de mesure des concentrations en arômes. La même précision est imposée sur la taille des solutions (boîte) : au dessous de cette taille, une boîte n'est pas étudiée (découpée) par le B & B à intervalles.

5.1 Les instances testées

Nous avons modélisé trois instances du problème d'assemblage des vins. La première (`WineBlending0`) est une petite instance artificielle qui nous a permis de régler le modèle MINLP/NCSP vu plus haut jusqu'à l'obtention d'un temps de réponse suffisamment rapide. L'instance `WineBlending0` contient 21 variables et est résolue en 0.18 secondes avec seulement 6 branchements par la stratégie par défaut, indépendamment de la précision demandée (`1e-4` ou `1e-8`).

Instance réelle 1

La seconde instance (`WineBlending1`) est une instance réelle fournie par la société Nyseos. Il s'agit de produire $\mathcal{W} = 2$ vins cibles à partir de $\mathcal{B} = 7$ vins de base, en prenant en compte $\mathcal{A} = 11$ arômes.

Le problème Min-Max (section 3.4), contient 55 variables et 116 contraintes :

- 2 contraintes de channeling sur les volumes,
- 7 contraintes de surplus pour les bases,
- 44 contraintes de concentration en arômes,
- 49 contraintes liées au Min-Max,
- 14 contraintes quadratiques modélisant les contraintes disjonctives sur les volumes réalistes.

Cette instance forme un système relativement important pour un optimiseur global sous contraintes déterministe (exact).

Instance réelle 2

La troisième instance (`WineBlending2`) décrit l'assemblage de $\mathcal{W} = 3$ vins cibles à partir de $\mathcal{B} = 6$ bases, en tenant compte de $\mathcal{A} = 7$ arômes.

Le problème Min-Max contient 64 variables et 118 contraintes :

- 3 contraintes de channeling sur les volumes,
- 6 contraintes de surplus pour les bases,
- 42 contraintes de concentration en arômes,
- 49 contraintes liées au Min-Max,
- 18 contraintes quadratiques modélisant les contraintes disjonctives sur les volumes réalistes..

5.2 Résultats obtenus avec l'optimiseur par défaut

Tous les résultats présentés dans cet article ont été obtenus sur un ordinateur portable MacBook de 2010 (Intel Core 2 Duo à 2.4 GHz).
Nous avons tout d'abord utilisé l'optimisateur par défaut d'Ibex avec une erreur de `1e-4` et un temps-limite de 5 minutes.

L'optimiseur atteint le temps-limite pour `WineBlending1` et `WineBlending2` tout en retournant des solutions relativement bonnes :

- pour `WineBlending1`, après 5 mn de calcul et 7894 nœuds, une erreur de 0.002 est obtenue sur l'objectif (c.-à-d., l'erreur maximum E).
- pour `WineBlending2`, en 5 mn et 8520 nœuds, une erreur de 0.0054 est obtenue.

5.3 Analyse algorithmique

Les résultats précédents ont été obtenus en utilisant la stratégie par défaut disponible dans `Ibex` et qui est brièvement décrite à la section 4.2.

Nous avons ensuite déterminé quelles options par défaut pouvaient avoir une influence sur la performance. Cette analyse a été très révélatrice.

Concernant l'aspect contraction (filtrage), l'opérateur de linéarisation sur intervalles `X-Newton` est étonnamment contre-productif. À l'inverse, toutes les contractions sont réalisées par l'opérateur `HC4` de programmation par contraintes. Dans la mesure où le modèle d'assemblage de vins est majoritairement linéaire, ce résultat n'est pas intuitif. Nous n'avons pas encore suffisamment d'éléments pour en comprendre la cause.

A l'opposé, nous avons fait une observation plus intuitive sur les régions internes extraites dans la phase d'amélioration de la borne supérieure. L'opérateur `InHC4`, issu des principes de la programmation par contraintes, ne semble pas utile. À l'inverse, l'algorithme `InnerPolytope` (dual de `X-Newton`) s'avère crucial pour extraire un polytope intérieur de l'espace réalisable. Sans cette fonctionnalité, nous ne pourrions pas obtenir de réponse de l'optimiseur.

La stratégie de branchement (bissection) est le second critère ayant une influence majeure sur la performance. Nous avons observé que les heuristiques élémentaires de bissection fournies par `Ibex` sont inefficaces ou peu performantes. L'heuristique `largestFirst`, qui sélectionne une variable de plus grande largeur, empêche le B&B de trouver un point réalisable pendant la recherche arborescente. L'heuristique `roundRobin` offre également de mauvaises performances, sauf si nous modifions l'ordre statique des variables en nous inspirant de la première analyse présentée à la section 5.4.

Seules les stratégies de la famille Smear fonctionnent bien : les approches historiques `SmearMax` et `SmearSum` [8] ; la variante `SmearSumRel` appelée par l'optimiseur par défaut [13], et enfin la variante `SmearMaxRel` qui joue un rôle fondamental dans notre problème d'assemblage de vins.

Partant de cette analyse, nous avons élaboré une nouvelle stratégie. L'opérateur `X-Newton` a tout d'abord été désactivé, ce qui a permis d'améliorer les performances d'un facteur 4 sur les deux instances réelles. Nous avons ensuite comparé les quatre variantes de l'heuristique Smear. Le tableau 1 présente les résultats obtenus.

	WineBlending	1	2
SmearSum	temps cpu (s)	> 300	57
	précision	0.00014	1e-10
	#nœuds	21880	4146
SmearSumRel	temps cpu (s)	> 300	> 300
	précision	0.0018	0.00017
	#nœuds	25864	24270
SmearMax	temps cpu (s)	> 300	111
	précision	0.0013	1e-10
	#nœuds	25864	8851
SmearMaxRel	temps cpu (s)	0.35	27.4
	précision	1e-10	1e-10
	#nœuds	23	2048

TABLE 1 – Comparaison entre les différentes variantes de l'heuristique de branchement Smear. Chaque case contient 3 informations sur les résultats obtenus pour une variante donnée sur une des deux instances réelles. La première valeur indique le temps CPU (avec un temps-limite de 5 mn) ; la seconde valeur correspond à l'erreur finale (`1e-10` signifie que le denier appel à l'algorithme du simplexe réalisé par `InnerPolytope` trouve une solution sans erreur (arrondie à `1e-10`)) ; la troisième valeur indique le nombre de nœuds de branchement.

Nous pouvons remarquer que `SmearSum`, et surtout `SmearMaxRel`, donnent de bons résultats. C'est pourquoi nous avons choisi `SmearMaxRel` pour notre problème d'assemblage de vins.

5.4 Vers une stratégie de branchement dédiée

Nous avons également étudié expérimentalement quelles variables étaient sélectionnées dans différents processus d'optimisation. Nous avons ainsi déterminé expérimentalement que les variables $P_{w,b}$ et $V'_{w,b}$ doivent être souvent sélectionnées alors que les variables liées à l'optimisation du Min-Max doivent l'être moins souvent. Nous avons donc élaboré la stratégie

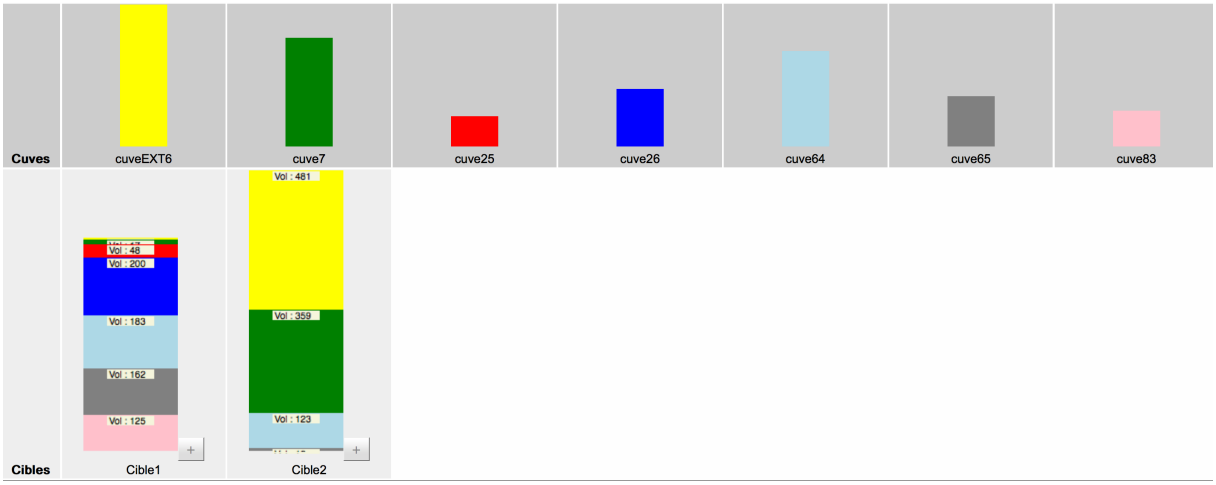


FIGURE 3 – Résultats obtenus pour `WineBlending1` : les volumes des vins cibles (en bas) sont le résultat de l'assemblage des vins de base (en haut).

dédiée suivante :

- seules les variables mentionnées ci-dessus peuvent être sélectionnées pour la bissection ;
- l'heuristique `SmearMaxRel` est appliquée au sous-ensemble de variables sélectionnées.

Cette heuristique dédiée offre des performances à peine meilleures que `SmearMaxRel` (23 secondes contre 27 secondes sur `WineBlending2`). Nous avons bien sûr besoin de plus d'instances pour mieux apprendre à partir des données.

5.5 Les vins résultats

La figure 3 présente la solution calculée pour `WineBlending1`. Les concentrations en arômes ($C_{w,a}$) dans les vins cibles sont détaillées par $\mathcal{W} = 2$ graphes radar à la figure 4. Le fait que les lignes bleues ($\hat{c}_{w,a}$) soient comprises entre les lignes vertes ($\hat{c}_{w,a} - \varepsilon_a$ et $\hat{c}_{w,a} + \varepsilon_a$) montre visuellement que la solution optimale est obtenue sous le seuil de tolérance ε_a.

Les concentrations en arômes pour les vins cibles de l'instance `WineBlending2` sont illustrées par $\mathcal{W} = 3$ graphes radar à la figure 5.

5.6 Séance de dégustation

Une première validation qualitative de notre travail a été réalisée en collaboration avec un œnologue. La société Nyseos lui a demandé d'élaborer un vin cible en assemblant différents vins de base. Elle a ensuite analysé le vin élaboré afin d'identifier les concentrations souhaitées en arômes ainsi que les concentrations dans les vins de base. Nyseos a alors soumis ce problème d'assemblage à notre outil qui a proposé des volumes à assembler différents de ceux choisis par l'œnologue. Une dégustation en aveugle a enfin été réalisée sur les deux vins produits.

Bien que les deux assemblages soient significativement différents, l'œnologue n'a pas été en mesure de différencier les deux vins.

Cette unique expérience n'est bien entendu pas suffisante pour valider la démarche mais c'est un résultat encourageant pour la pertinence de notre outil.

5.7 Un outil de configuration pour l'assemblage de vins

Les résultats expérimentaux sont encore préliminaires et n'ont été obtenus que sur deux cas réels. Ils suggèrent cependant que la stratégie actuelle, éventuellement dotée d'une heuristique de branchement dédiée, est efficace pour traiter la plupart des cas utiles en pratique. Par conséquent, pour mieux répondre aux souhaits des clients, nous pouvons imaginer une utilisation interactive de notre algorithme d'optimisation, à l'intérieur d'un outil de configuration. L'utilisateur pourrait interagir avec le système via des graphes radar correspondant aux différents vins cible, comme ceux des figures 4 et 5.

Une façon de modifier un assemblage est d'augmenter (ou diminuer) le poids relatif Ω_w d'un vin cible w. Un simple curseur sous chaque graphe radar permettrait par exemple de modifier cette valeur, ce qui entraînerait le recalcul d'une solution par l'optimiseur.

En suivant la même idée, l'utilisateur pourrait modifier le poids $\lambda_{w,a}$ d'un arôme a pour un vin w (par exemple, via un menu contextuel apparaissant lorsque le curseur de la souris se trouve sur l'axe correspondant

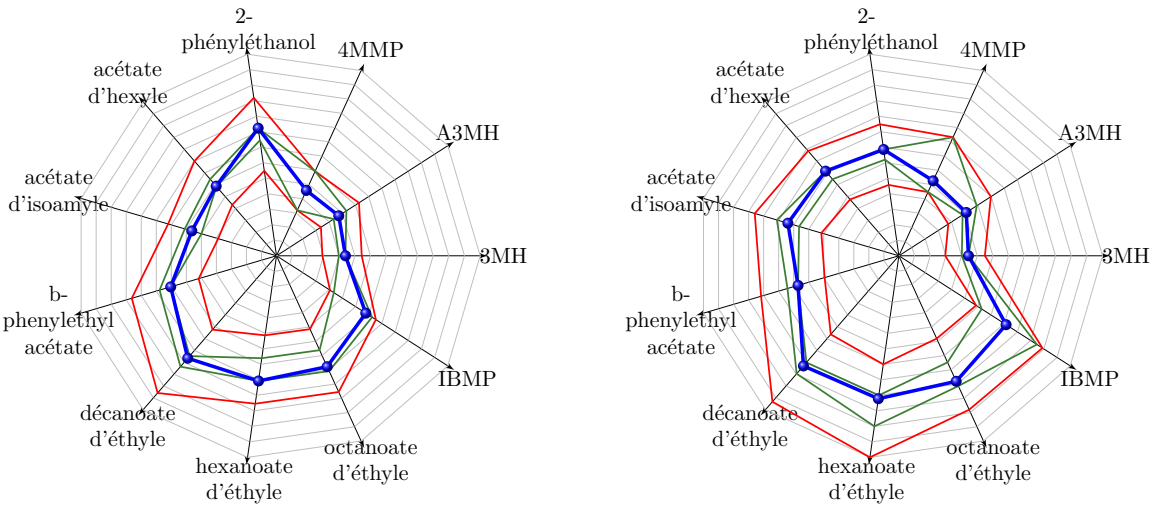


FIGURE 4 – Solution obtenue par notre modèle Min-Max sur les deux vins cibles de l'instance `WineBlending1`. Chaque axe du graphe radar correspond à un composé aromatique. La concentration obtenue $C_{w,a}$ (en bleu) doit rester comprise entre les limites imposées par les bornes $c^-_{w,a}$ et $c^+_{w,a}$ (en rouge) tout en restant le plus proche possible de la concentration souhaitée $\hat{c}_{w,a}$. Cette dernière est représentée par les deux courbes en vert $\hat{c}_{w,a} - \varepsilon_a$ et $\hat{c}_{w,a} + \varepsilon_a$ qui traduisent l'erreur de mesure tolérée pour l'arôme a correspondant.

du graphe radar), et l'outil déclencherait une nouvelle optimisation. Enfin, une dernière possibilité plus intrusive serait de permettre le renforcement des concentrations maximales $c^+_{w,a}$ (ou minimales $c^-_{w,a}$) admissibles pour un arôme a dans un vin w. L'utilisateur pourrait simplement sélectionner une borne et notre système exécuterait deux optimisations afin de déterminer deux informations (en supposant que $c^+_{w,a}$ a été sélectionnée) :

- la plus petite valeur de $c^+_{w,a}$ possible (en respectant les contraintes),
- la plus petite valeur possible de $c^+_{w,a}$ qui n'augmente pas l'erreur globale E.

6 Conclusion

Nous avons présenté dans cet article la première approche en programmation par contraintes pour résoudre le problème de l'assemblage de vins. Ce problème peut être modélisé par un MINLP ou un CSP numérique supportant des contraintes disjonctives qui s'avèrent cruciales en pratique. Ces contraintes sont notamment liées à des impératifs physiques sur le transfert d'un volume de vin entre deux cuves. Nous nous sommes attachés à définir un modèle Min-Max sans opérateurs max ou valeur absolue. Nous avons conçu une stratégie d'optimisation de l'assemblage de vins qui permet d'obtenir, en quelques secondes, une solution (sans erreurs sur les volumes ou les concentrations) pour deux instances réelles fournies par la société Nyseos. Une séance de dégustations réalisée avec un expert œnologue a permis de valider l'intérêt de cette approche. Ces résultats encourageants permettent d'envisager l'utilisation de notre modèle dans un outil interactif de configuration dédié à l'assemblage de vins.

Références

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

[2] P. Belotti. Couenne, a user's manual, 2013. `www.coin-or.org/Couenne/couenne-user-manual.pdf`.

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

[4] G. Chabert. Interval-Based EXplorer, 2013. `www.ibex-lib.org`.

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

[6] L. Dagan. *Potentiel aromatique des raisins de Vitis vinifera L. cv. Petit Manseng et Gros Manseng. Contribution à l'arôme des vins de pays*

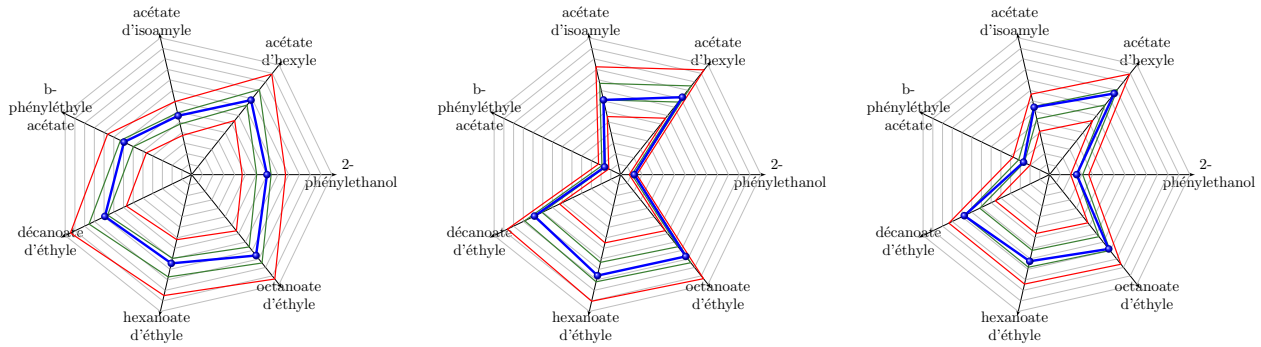


FIGURE 5 – Graphes radar obtenus pour l'instance `WineBlending2`.

Côtes de Gascogne. PhD thesis, École nationale supérieure agronomique (Montpellier), 2006.

[7] J. Ferrier and D. Block. Neural-network-assisted optimization of wine blending based on sensory analysis. *American Journal of Enology and Viticulture*, 52(4) :386–395, 2001.

[8] R.B. Kearfott and M. Novoa III. INTBIS, a portable interval Newton/Bisection package. *ACM Trans. on Mathematical Software*, 16(2) :152–157, 1990.

[9] 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.

[10] D. B. Moore and T. G. Griffin. Computer blending technology. *American Journal of Enology and Viticulture*, 29(1) :50–53, 1978.

[11] J. Ninin, F. Messine, and P. Hansen. A Reliable Affine Relaxation Method for Global Optimization. Technical Report RT-APO-10-05, IRIT, 2010.

[12] M. Tawarmalani and N. V. Sahinidis. A Polyhedral Branch-and-Cut Approach to Global Optimization. *Mathematical Programming*, 103(2) :225–249, 2005.

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

Éliminations des solutions symétriques pour l'isomorphisme de sous-graphe

Philippe Vismara

LIRMM, UMR5506 Université Montpellier II - CNRS, Montpellier, France
MISTEA, UMR0729 Montpellier SupAgro - INRA, Montpellier, France
vismara@lirmm.fr

Résumé

La programmation par contrainte fournit aujourd'hui des modèles efficaces pour résoudre les problèmes d'isomorphisme de sous-graphe. Beaucoup de problèmes réels nécessitent l'énumération de toutes les solutions qui ne sont pas équivalentes à une symétrie près d'un des graphes traités. Dans cet article nous nous intéressons aux conditions sous lesquelles il est possible d'éliminer ces symétries en utilisant des contraintes lexicographiques. Dans le cas où le nombre de symétries devient trop important, nous proposons une méthode basée sur des contraintes ensemblistes.

Abstract

Constraint programming provides effective models to solve subgraph isomorphism problems. Many real-world applications require the enumeration of all solutions that are not equivalent to a symmetry of the processed graphs. In this article we focus on the conditions under which it is possible to eliminate these symmetries using lexicographic constraints. When the number of symmetries becomes too large, we propose a method based on set constraints.

1 Introduction

La prise en compte de la symétrie permet généralement d'améliorer le temps de résolution d'un problème de satisfaction de contraintes. Dans le cas de problèmes comme l'isomorphisme de sous-graphe, il est souvent nécessaire d'énumérer toutes les solutions qui ne sont pas équivalentes à une symétrie d'un des deux graphes. Ces symétries correspondent aux symétries de variable ou de valeur du CSP modélisant le problème. Il est donc possible d'utiliser les travaux sur la symétrie en contraintes pour réaliser l'énumération recherchée. La problématique abordée dans cet article n'est pas l'efficacité des méthodes mais la garantie de ne pas perdre de solutions tout en éliminant les symétries. Nous étudierons plus particulièrement les méthodes faisant appel à des contraintes lexicographiques qui sont souvent privilégiées pour leur facilité de mise en œuvre. Nous étudierons leurs limites lorsque le nombre de symétrie devient trop grand et étudierons les alternatives dans de tel cas.

1.1 Notations et définitions

Un graphe simple non orienté sera noté $G = (N, E)$, où $N = \{0, ...(n_G - 1)\}$ désigne l'ensemble des n_G sommets de G et E l'ensemble des m_G arêtes.

Une arête entre i et j sera notée ij (avec $ij \equiv ji$).

Étant donnés deux graphes $G = (N_G, E_G)$ et $H = (N_H, E_H)$, une fonction injective $\mu : N_G \longrightarrow N_H$ est un isomorphisme de sous-graphe partiel de G dans H ssi $\forall i, j \in N_G,\ ij \in E_G \Rightarrow \mu(i)\mu(j) \in E_H$.

On dit que μ est un isomorphisme de sous-graphe *induit* de G dans H ssi $\forall i, j \in N_G,\ ij \in E_G \Leftrightarrow \mu(i)\mu(j) \in E_H$.

2 Isomorphisme de sous-graphe

L'isomorphisme de sous-graphe partiel est facilement modélisable par un CSP :

- une variable par sommet de G :
$$X = \{x_i\}_{i \in N_G} \text{ avec } D(x_i) = N_H$$
- une contrainte par arête de G (voisinage) :
$$\forall ij \in E_G,\ x_i x_j \in E_H$$
- une contrainte globale de différence : *Alldiff(X)*

Dans cet article nous nous limiterons à ce modèle qui est suffisant pour aborder les problèmes de symétrie. Il a cependant connu de nombreuses améliorations [13, 7]

Le problème de l'isomorphisme de sous-graphe induit a été moins traité dans la littérature. On peut

le résoudre par un CSP construit en remplaçant dans le modèle précédent l'ensemble des m_G contraintes de voisinage par n_G^2 contraintes de la forme :

$$\forall i, j \in N_G, ij \in E_G \Leftrightarrow x_i x_j \in E_H$$

Une autre solution consiste à conserver les m_G contraintes de voisinage pour G et à ajouter m_H contraintes sur de nouvelles variables inspirées de [4]. Nous ajoutons au CSP un nouvel ensemble de variables $\{z_j\}_{j \in N_H}$ correspondant aux sommets de H de telle sorte que $x_i = j$ entraîne $z_j = i$. Le problème n'étant pas surjectif, il faut rajouter des variables $x_{n_G+1}, ..., x_{n_G+n_H}$ de domaine $D(x_{n_G+j}) = \{j\}$ et des contraintes de channeling de la forme $(x_i = j) \Rightarrow (z_j \leq i)$ et $(z_j = i) \Rightarrow (x_i = j)$ pour que z_j soit égal au plus petit i tel que $x_i = j$.

Pour rechercher des isomorphismes de sous-graphe induit on complète le CSP en ajoutant les contraintes :

$$\forall jk \in E_H,\ z_j \leq n_G \wedge z_k \leq n_G \Rightarrow z_j z_k \in E_G$$

Éliminer les symétries de G (variables) On peut facilement vérifier que l'ensemble $\mathcal{S}(G)$ des symétries de G (automorphismes) correspond aux symétries de variables du CSP.

Pour énumérer toutes les solutions qui ne sont pas équivalentes par rapport aux symétries de G, on peut poser $|\mathcal{S}(G)|$ contraintes lexicographiques [1]. Étant donné un ordre $x_{i_1}, \ldots x_{i_n}$ sur les variables, on pose une contrainte $x_{i_1}, \ldots x_{i_n} \leq_{lex} x_{\sigma(i_1)}, \ldots x_{\sigma(i_n)}$ pour chaque symétrie σ. Nous noterons VarLex(G) cet ensemble d'au plus $n_G!$ contraintes.

L'isomorphisme de sous-graphe étant un problème injectif, les auteurs de [12] ont utilisé la méthode de Puget [5]. Elle repose sur l'idée que dans le cas injectif, chaque contrainte de VarLex(G) se réduit à une inégalité $x_i < x_{\sigma(i)}$ où i est le plus petit indice non stable par σ. Par transitivité on obtient un nombre linéaire de contraintes que nous noterons VarLinear(G) . Ces contraintes sont de la forme $x_{r(i)} < x_i$ où $r(i)$ est le plus grand indice, inférieur à i, dont i est l'image par une symétrie qui stabilise l'ensemble $[1..r(i)[$.

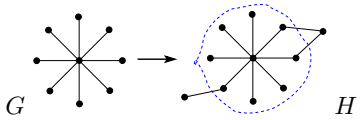
Pour mettre en œuvre cette démarche, il faut déterminer les symétries de G, ce qui est possible en utilisant une librairie comme NAUTY (http ://pallini.di.uniroma1.it/). Il faut également construire une chaîne de stabiliseurs [1] en appliquant l'algorithme de Schreier-Sims. Il existe plusieurs versions de cet algorithme dont la variante de Jerrum [3] qui a une complexité en $\mathcal{O}(n^5)$ et $\mathcal{O}(n^2)$ en espace.

Une vision ensembliste des symétries de variables La méthode précédente nécessite le recours à des algorithmes complexes sur le groupe de symétries. Si l'on ne dispose pas d'implémentation de ces algorithmes, nous proposons de supprimer les symétries de variables en utilisant le theoreme suivant :

Theoreme 1 *Deux isomorphismes de sous-graphe induit μ_1 et μ_2 sont équivalents à une symétrie de G près si et seulement si $\mu_1(N_G) = \mu_2(N_G)$*

Preuve : L'implication directe est évidente. Pour la réciproque, il suffit de montrer que μ_1 et μ_2 sont des isomorphismes de G dans $\mu_1(N_G)$, par définition de l'isomorphisme de sous-graphe induit. Donc $\mu_1^{-1} \circ \mu_2$ est un automorphisme de G □

Par exemple, pour les graphes ci-dessous, dès qu'un isomorphisme de sous-graphe a été trouvé, on peut écarter les 8 !=40320 solutions symétriques qui couvrent le même ensemble de sommets de H.



Pour éliminer les solutions symétriques par rapport à G, on complète le CSP en ajoutant :

- une variable ensembliste u, avec $D(u) = 2^{N_H}$, représentant l'image de N_G
- des contraintes de channeling :
$$\forall i \in N_G,\ x_i \in u \quad \text{et} \quad |u| = n_G$$
- pour chaque nouvelle solution couvrant un ensemble s de valeurs, on ajoute une contrainte supplémentaire : $u \neq s$

Nous noterons VarSet(G) cet ensemble de contraintes. D'après le theoreme 1, $VarSet(G)$ permet d'éliminer les solutions symétriques par rapport à G.

Cette méthode est adaptée au problème de l'isomorphisme de sous-graphe induit mais peut-être généralisée au cas partiel. Tout problème d'isomorphisme de sous-graphe partiel de G dans H est en effet équivalent à un problème de sous-graphe induit dans les graphes des arêtes [2] $L(G)$ et $L(H)$ correspondants. On peut également utiliser une autre transformation basée sur les graphes subdivisés [3] correspondants [8, 9].

Éliminer les symétries de H (valeurs) De manière analogue, les symétries du graphe H correspondent aux symétries de valeurs du CSP.

Pour éliminer les solutions équivalentes par rapport à une symétrie θ de H, on peut poser une contrainte de la forme $x_{i_1}, \ldots x_{i_n} \leq_{lex} \theta(x_{i_1}), \ldots \theta(x_{i_n})$ [10]. Nous noterons ValLex(H) l'ensemble de ces contraintes pour tous les θ de $\mathcal{S}(H)$.

1. à partie de la suite $i_1, ..., i_k, ..., i_n$, on construit la suite de sous-groupes $\mathcal{S}(G)^{[i_k]} = \{\sigma \in \mathcal{S}(G) \mid \forall z < k,\ \sigma(i_z) = i_z\}$.

2. le graphe des aretes (line graph) $L(G)$ est le graphe dont les sommets sont les arêtes de G et deux nœuds sont reliés dans $L(G)$ si les arêtes correspondantes ont une extrémité en commun

3. le graphe subdivisé (subdivision graph) s'obtient en remplaçant chaque arête par un sommet relié aux deux extrémités

L'intérêt de cette méthode est qu'elle est assez facile à mettre en œuvre et compatible avec l'élimination des symétries de G par des contraintes VarLex(G) ou VarLinear(G) [10]. Par contre, elle peut entraîner la définition d'au plus $n_H!$ contraintes.

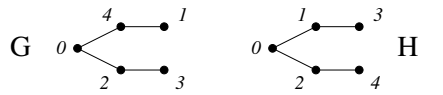
Nous pouvons remarquer ici qu'il est facile d'éliminer certaines contraintes de ValLex(H) en réduisant le domaine de la variable X_{i_1}. Pour chaque symétrie θ, la contrainte lexicographique ne sera vérifiée que si $x_{i_1} \leq \theta(x_{i_1})$. On peut donc supprimer de $D(x_{i_1})$ toute valeur v telle que $\theta(v) < v$. Plus globalement, notons $\mathsf{SmallestOrbits}(\mathcal{S}(H))$ l'ensemble des valeurs minimum de chaque orbite [4] de $\mathcal{S}(H)$.

Theoreme 2 *Poser $x_{i_1} \in \mathsf{SmallestOrbits}(\mathcal{S}(H))$ est compatible avec* ValLex*(H) et permet de supprimer toutes les contraintes de* ValLex*(H) basées sur une symétrie qui ne stabilise aucun élément de $\mathsf{SmallestOrbits}(\mathcal{S}(H))$*

Preuve : Pour tout $v \in D(x_{i_1}) \setminus \mathsf{SmallestOrbits}(\mathcal{S}(H))$, il existe $\theta \in \mathcal{S}(H)$ telle que $\theta(v) < v$. Toute affectation de v à x_{i_1} violera la contrainte de ValLex(H) basée sur θ. Considérons ensuite une symétrie $\theta \in \mathcal{S}(H)$ qui ne stabilise aucun élément de $\mathsf{SmallestOrbits}(\mathcal{S}(H))$. La contrainte Lex associée à θ est nécessairement vérifiée puisque $x_{i_1} \in \mathsf{SmallestOrbits}(\mathcal{S}(H)) \Rightarrow x_{i_1} < \theta(x_{i_1})$ □

Une seconde méthode, proposée dans [4], permet de supprimer les symétries de H. Elle consiste à ajouter au CSP les variables $\{z_j\}_{j \in N_H}$ précédentes qui correspondent aux sommets de H et les contraintes de channeling associées. Il suffit de poser une contrainte VarLinear(H) sur les z_j pour éliminer les symétries de H.

Malheureusement, poser des contraintes VarLinear(H) sur les z_j n'est pas toujours compatible avec les contraintes VarLex(G) ou VarLinear(G) et peut entraîner la perte de solutions [11]. Nous illustrons ce problème dans l'exemple de la figure 1. VarLinear(G) se résume à la contrainte $x_1 < x_3$ alors que VarLinear(H) donne la contrainte $z_1 < z_2$. Or il n'existe aucune solution qui respecte ces deux contraintes.



$\mathcal{S}(G) = \left\{ \begin{array}{ll} \text{id} & 01234 \\ \sigma & 03412 \end{array} \right.$ $r(3) = 1$

$\mathcal{S}(H) = \left\{ \begin{array}{ll} \text{id} & 01234 \\ \theta & 02143 \end{array} \right.$ $r(2) = 1$

FIGURE 1 – Deux graphes et leurs groupes de symétries

Une autre combinaison a été proposée dans [12]. Elle consiste à poser des contraintes VarLinear(G) sur les variables x_i et à utiliser la méthode GE-tree [6] pour éliminer les symétries de valeurs. A chaque nœud de profondeur k de l'arbre de recherche, lorsque la variable x_{i_k} a été choisie, GE-tree cherche dans $D(x_{i_k})$ les valeurs équivalentes à une symétrie de valeur près. À ce stade de la recherche, on ne doit considérer que les symétries de H qui stabilisent toutes les valeurs $v_{i_1}, ..., v_{i_{k-1}}$ déjà affectées. Notons $\mathcal{S}(H)^{[k-1]}$ ce sous groupe de $\mathcal{S}(H)$.

On ne crée un sous-nœud $x_{i_k} = v_j$ que pour tout $v_j \in D(x_{i_k}) \cap \mathsf{SmallestOrbits}(\mathcal{S}(H)^{[k-1]})$

Si l'ordre d'affectation des valeurs n'est pas fixé a priori, il faut déterminer le sous-groupe $\mathcal{S}(H)^{[k-1]}$ à chaque nœud. On peut le faire en $O(n^5)$ via la variante de Jerrum de l'algorithme de Schreier-Sims.

Mais l'utilisation combinée de contraintes VarLinear(G) et GE-tree(H) ne doit pas se faire sans précautions.

Theoreme 3 *Combiner* VarLinear(G) *et* GE-tree(H) *sans contraindre l'ordre d'instanciation des variables peut faire perdre des solutions*

Preuve : considérons les graphes de la figure 1. L'ensemble VarLinear(G) se réduit à la contrainte $x_1 < x_3$. Supposons que la première affectation soit $x_0 = 0$. La propagation des contraintes de connexité donne : $D(x_2) = D(x_4) = \{1, 2\}$ et $D(x_1) = D(x_3) = \{3, 4\}$. Le sous-groupe de $\mathcal{S}(H)$ qui stabilise 0 est égal à $\mathcal{S}(H)$. Supposons que l'on choisisse ensuite la variable x_2. Les valeurs de $D(x_2) = \{1, 2\}$ sont symétriques par rapport à θ. Si on choisit la valeur 1 comme représentant de son orbite, on aboutit à une solution qui ne respecte pas $x_1 < x_3$ □

Dans [12], il est suggéré d'utiliser GE-tree(H) en affectant les valeurs dans l'ordre croissant, ce qui permet de construire toute la chaîne de stabiliseurs $\mathcal{S}(H) \geq \mathcal{S}(H)^{[1]} \geq \ldots \geq \mathcal{S}(H)^{[n_H-1]} \geq \{id.\}$ par un seul appel à l'algorithme de Schreier-Sims. Malheureusement, cet ordre d'instanciation entre dans le cas du theoreme 3 et n'est donc pas toujours compatible avec les contraintes VarLinear(G).

Comme indiqué dans [5], pour combiner VarLinear(G) et GE-tree(H), il faut impérativement utiliser l'ordre d'instanciation statique $x_{i_1}, \ldots x_{i_n}$ correspondant à l'ordre $i_1, ..., i_n$ qui a servi pour construire les contraintes VarLinear(G). Il faut ensuite appliquer l'algorithme GE-Tree en calculant à chaque noeud de profondeur k le sous-groupe $\mathcal{S}(H)^{[k-1]}$. On doit enfin n'affecter x_k qu'à des valeurs de $\mathsf{SmallestOrbits}(\mathcal{S}(H)^{[k-1]})$.

3 Discussion

Le choix d'une méthode pour énumérer toutes les solutions non symétriques d'un problème d'isomor-

4. ces orbites correspondent aux classes d'équivalence de la relation $x \equiv y \Leftrightarrow (\exists \theta \in \mathcal{S}(H), \theta(x) = y)$

phisme de sous-graphes va dépendre du nombre de symétries de G et H.

Si seul le graphe G (resp. H) possède des symétries, il suffit de poser des contraintes VarLinear sur les variables x_i (resp. z_j) correspondant aux sommets de G (resp. H).

Lorsque un des deux graphes possède peu de symétries (jusqu'à 10^5 pour une rapide implémentation avec le solveur Choco), on peut utiliser ValLex sur ce graphe et VarLinear sur l'autre graphe.

Par contre, si les deux graphes possèdent beaucoup de symétries, la méthode la plus simple à mette en œuvre consiste à combiner les contraintes ValLex(H) avec VarSet(G).

L'intérêt de ces méthodes est qu'elles reposent sur des contraintes assez fréquentes dans les solveurs et ne nécessitent de recourir à l'algorithmique des groupes de symétrie que pour construire le modèle du CSP.

À l'inverse, des méthodes comme GE-tree font appel à l'algorithme de Scheierer-Sims pour déterminer les symétries de variables pertinentes à chaque nœud de l'arbre de recherche. Nous avons vu qu'il est possible de combiner VarLinear(G) avec GE-tree(H) à condition d'utiliser un ordre statique sur les variables. On pourrait également étudier une approche comme GAP-SBDS [2] qui détermine des sous-groupes de symétries pendant la recherche. Mais d'après [6], cette méthode peut être moins efficace que GE-tree.

Au delà de la facilité de mise en œuvre, il serait utile de comparer ces méthodes en termes d'efficacité. Cependant, le problème de l'efficacité est différent de celui de l'élimination des solutions symétriques. On pourrait ainsi associer la méthode VarSet à des méthodes de suppression de la symétrie plus efficaces mais incomplètes. L'autre difficulté réside dans le choix d'un jeu de données pertinent. Le problème abordé dans cet article est celui des cas extrêmes où le nombre de symétries devient très important. De tels graphes existent dans les problèmes réels comme en chimie organique où l'on peut rencontrer des graphes moléculaires avec 10^9 symétries. À l'inverse, une base de graphes comme la VFLib Graph Database, souvent utilisée pour l'évaluation d'algorithmes, comporte 20400 paires de graphes mais le graphe le moins symétrique contient au pire 4600 symétries.

4 Conclusion

Dans cet article nous nous sommes intéressés au problème de l'isomorphisme de sous-graphe de G dans H du point de vue de l'énumération des solutions non symétriques par rapport à l'un des deux graphes. Nous avons illustré l'incompatibilité des contraintes lexicographiques lorsqu'elles sont utilisées pour les deux graphes. Nous avons montré que la méthode GE-tree n'est pas compatible avec les contraintes Lex de variables si on utilise un ordre statique sur les valeurs. Nous avons enfin proposé une méthode simple de suppression des symétries de G par des contraintes ensemblistes.

Références

[1] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR*, pages 148–159, 1996.

[2] Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton. Generic sbdd using computational group theory. In *Proc. CP'03*, pages 333–347. Springer, 2003.

[3] M. Jerrum. A compact representation for permutation groups. *J. Algorithms*, 7(1) :60–78, 1986.

[4] J.-F. Puget. Breaking all value symmetries in surjection problems. In *Proc. CP 2005*, pages 490–504, 2005.

[5] J.-F. Puget. Breaking symmetries in all different problems. In *IJCAI*, pages 272–277, 2005.

[6] Colva M. Roney-Dougal, Ian P. Gent, Tom Kelsey, and Steve Linton. Tractable symmetry breaking using restricted search trees. In *ECAI*, pages 211–215, 2004.

[7] C. Solnon. Alldifferent-based filtering for subgraph isomorphism. *Artificial Intelligence*, 174(12-13) :850 – 864, 2010.

[8] Philippe Vismara. Programmation par contraintes pour les problèmes de plus grand sous-graphe commun. In *Actes JFPC 2011*, pages 327–335, Lyon, France, 2011.

[9] Philippe Vismara and Benoît Valery. Finding maximum common connected subgraphs using clique detection or constraint satisfaction algorithms. In *Proc. MCO 2008*, volume 14 of *Communications in Computer and Information Science*, pages 358–368. Springer, 2008.

[10] Toby Walsh. General symmetry breaking constraints. In *CP*, pages 650–664, 2006.

[11] Toby Walsh. Breaking variable and value symmetry. In *Proc. SymCon'11*, page 9, 2011.

[12] S. Zampelli, Y. Deville, M. R. Saïdi, and B. Benhamou. Symmetry breaking in subgraph isomorphism. In *Proc. SymCon'07*, 2007.

[13] S. Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints*, 15 :327–353, 2010.

Maintien asynchrone de la consistance d'arc dans la recherche distribuée synchrone

Mohamed Wahbi[1] **Redouane Ezzahir**[2] **Christian Bessiere**[3] **El Houssine Bouyakhf**[4]

[1] TASC INRIA, LINA CNRS, École des Mines de Nantes, France
[2] ENSA Agadir, Université Ibn Zohr, Maroc
[3] LIRMM, Université Montpellier 2, France
[4] LIMIARF, Université Mohammed V–Agdal, Maroc

mohamed.wahbi@emn.fr red.ezzahir@gmail.com bessiere@lirmm.fr bouyakhf@fsr.ac.ma

Abstract

Nous avons proposé récemment Nogood-Based Asynchronous Forward Checking (AFC-ng), un algorithme efficace et robuste pour résoudre des Problèmes de Satisfaction de Contraintes Distribués (DisCSPs). AFC-ng exécute de manière asynchrone une phase de vérification en avant pendant la recherche synchrone. Dans ce papier, nous proposons deux nouveaux algorithmes basés sur le même mécanisme qu'AFC-ng. Cependant, au lieu d'utiliser le *forward checking* comme propriété de filtrage, nous proposons de maintenir d'une manière asynchrone la consistance d'arc (MACA). Le premier algorithme que nous proposons, MACA-del, maintient la consistance d'arc grâce à un type de messages supplémentaire, des messages de suppression. Le deuxième algorithme, MACA-not, réalise la consistance d'arc sans aucun nouveau type de message. Nous fournissons une analyse théorique et une évaluation expérimentale de l'approche proposée. Nos expérimentations montrent la bonne performance des algorithmes MACA et plus particulièrement ceux de MACA-not.

1 Introduction

Les problèmes de satisfaction de contraintes (CSP) permettent de formaliser de nombreux problèmes réels de forte combinatoire tels que l'allocation de ressources, le car sequencing, etc. Un problème de satisfaction de contraintes consiste en une recherche de solutions pour un réseau de contraintes, c'est à dire, trouver un ensemble d'affectations de valeurs aux variables qui satisfont les contraintes du problème. Ces contraintes spécifient des combinaisons de valeurs admissibles.

Beaucoup d'algorithmes de recherche basés sur le backtrack ont été développés pour résoudre les problèmes de satisfaction de contraintes. Certains parmi ces algorithmes de recherche essayent de construire une solution d'un CSP en alternant l'affectation de variables avec la propagation de contraintes. Forward Checking (FC) [9] et le maintien de la consistance d'arc (Maintaining Arc Consistency (MAC)) [17] sont des exemples de tels algorithmes. Dans les années 80, FC était considéré comme l'algorithme de recherche le plus efficace. Dans le milieu des années 90, plusieurs travaux ont montré de manière empirique que MAC est plus efficace que FC sur les problèmes durs et de grande taille [3, 8].

Les réseaux de capteurs [11, 1] ou les problèmes distribués d'allocation de ressources [15, 16] comme la planification de réunion distribuée [22, 13] sont des applications réelles de nature distribuée. La connaissance est distribuée parmi plusieurs entités. Ces applications peuvent être naturellement formalisées et résolues par un processus de CSP une fois que la connaissance et la description du problème sont livrées à un solveur centralisé. Cependant, dans de telles applications, assembler toute la connaissance dans un solveur centralisé n'est pas indiqué. En général, cette restriction est principalement due à des raisons de confidentialité et/ou de sécurité. Le coût ou l'incapacité de traduire toutes les informations en un format unique peuvent en être une autre raison. Ainsi, un modèle distribué permettant un processus de résolution décentralisé est plus indiqué. Les *problèmes de satisfaction de contraintes*

distribués (DisCSP) ont justement de telles propriétés.

Un DisCSP est composé d'un groupe d'agents autonomes, où chaque agent a le contrôle de quelques éléments d'information sur le problème. Chaque agent possède son réseau de contraintes local. Les variables dans les différents agents sont connectées par des contraintes. Les agents doivent attribuer des valeurs à leurs variables pour que toutes les contraintes soient satisfaites. Ainsi, les agents essayent de générer des affectations localement consistantes (affectation de valeurs à leurs variables), qui sont aussi consistantes avec les contraintes entre agents [24, 23].

Durant les deux dernières décennies, de nombreux algorithmes distribués de résolution de DisCSPs ont été proposés. Synchronous Backtrack (SBT) est l'algorithme de recherche de DisCSP le plus simple. SBT effectue des affectations séquentiellement et de manière synchrone. Dans SBT, seul l'agent disposant de l'affectation partielle courante (CPA pour Current Partial Assignment) effectue une affectation ou un backtrack [25]. Meisels et Zivan [14] ont étendu SBT à l'algorithme Asynchronous Forward Checking (AFC), un algorithme dans lequel l'algorithme FC [9] est exécuté d'une manière asynchrone [14]. Dans AFC, afin d'exécuter FC d'une manière asynchrone, à chaque fois qu'un agent réussit l'extension de l'affectation partielle courante, il envoie la CPA à son successeur et des copies de cette CPA aux autres agents non assignés.

Dans un travail récent [21], nous avons proposé l'algorithme Nogood-Based Asynchronous Forward Checking (AFC-ng), qui est une amélioration d'AFC. Contrairement à AFC, AFC-ng utilise les nogoods comme justification des valeurs supprimées et permet plusieurs backtrack simultanés provenant de différents agents et allant vers différentes destinations. On a montré que AFC-ng est plus performant que AFC.

Bien que de nombreuses études aient incorporé FC avec succès dans les CSPs distribués [5, 14, 7, 21], l'algorithme MAC n'a pas encore été bien examiné. Les seules tentatives pour inclure le maintien de consistance d'arc dans des algorithmes distribués ont été faites sur l'algorithme Asynchronous Backtracking (ABT). Silaghi *et al.* [19] ont introduit Distributed Maintaining Asynchronously Consistency for ABT (*DMAC-ABT*), le premier algorithme capable de maintenir la consistance d'arc dans les CSP distribué [19]. DMAC-ABT considère le maintien de la consistance comme une inférence hiérarchique basée sur les nogoods. Brito et Meseguer [6] ont proposé ABT-uac et ABT-dac, deux algorithmes qui connectent ABT avec la consistance d'arc [6]. ABT-uac propage inconditionnellement des valeurs supprimées pour faire respecter un certain nombre de consistances d'arc complètes. ABT-dac propage conditionnellement et inconditionnellement des valeurs supprimées en utilisant la consistance d'arc directionnelle. ABT-uac montre une amélioration mineure en termes d'envoi de messages. Quant à ABT-dac, il dégrade les performances dans bien des cas.

Dans ce travail, nous proposons deux nouveaux algorithmes de recherche synchrone basés sur le même mécanisme que AFC-ng. Cependant, au lieu de maintenir "forward checking" d'une manière asynchrone sur des agents non encore instanciés, nous proposons de maintenir la consistance d'arc d'une manière asynchrone sur ces agents futurs. Nous appelons ce nouveau schéma MACA, pour maintien de consistance d'arc de manière asynchrone (Maintaining Arc Consistency Asynhronously). Comme dans AFC-ng, seul l'agent disposant de l'affectation partielle courante (CPA) peut accomplir une affectation. Cependant, contrairement à AFC-ng, MACA essaie de maintenir la consistance d'arc au lieu de se contenter uniquement d'exécuter FC. Le premier algorithme que nous proposons, MACA-del, fait respecter la consistance d'arc grâce à un type supplémentaire de messages, des messages de suppression (***del***). Ainsi, à chaque fois que des valeurs sont supprimées pendant l'étape de propagation de contraintes, les agents de MACA-del notifient les autres agents qui peuvent être affectés par ces suppressions en leurs envoyant un message ***del***. Les messages ***del*** contiennent toutes les valeurs supprimées ainsi que le nogood justifiant leur suppression. Le deuxième algorithme, MACA-not, réalise la consistance d'arc sans aucun nouveau type de message. Nous réalisons ceci en stockant toutes les suppressions exécutées par un agent sur les domaines de ses agents voisins et en envoyant ces informations à ces voisins dans le message CPA.

Ce papier est organisé comme suit. La Section 2 donne le contexte nécessaire. La Sections 3 décrit les algorithmes MACA-del et MACA-not. L'analyse théorique et les preuves de correction sont développées dans la Section 4. La Section 5 présente une évaluation expérimentale de nos algorithmes comparés à d'autres algorithmes. Finalement, nous concluons le papier dans la Section 6.

2 Préliminaires

2.1 Définitions et notations basiques

Le Problème de Satisfaction de Contraintes Distribué (DisCSP pour distributed constraint satisfaction problem) a été formalisé dans [24] comme étant un quadruplet $(\mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C})$, où $\mathcal{A}$ est un ensemble de a agents $\{A_1, \ldots, A_a\}$, $\mathcal{X}$ est un ensemble de n variables

$\{x_1, \ldots, x_n\}$, tel que chaque variable x_i est contrôlée par un seul agent dans $\mathcal{A}$. $\mathcal{D}^0 = \{D^0(x_1), \ldots, D^0(x_n)\}$ est un ensemble de domaines, où $D^0(x_i)$ est l'ensemble initial de valeurs qui peuvent être affectées à la variable x_i. Pendant la recherche, les valeurs peuvent être supprimées du domaine. À chaque nœud donné, l'ensemble des valeurs possibles pour la variable x_i est désigné par $D(x_i)$. Ce dernier est appelé le domaine courant de x_i. $\mathcal{C}$ est un ensemble de contraintes qui spécifient les combinaisons de valeurs autorisées pour les variables qu'elles impliquent. Dans ce papier, nous supposons un réseau de contraintes distribué binaire où toutes les contraintes sont binaires (elles impliquent deux variables). Une contrainte $c_{ij} \in \mathcal{C}$ entre deux variables x_i et x_j est un sous-ensemble du produit cartésien de leurs domaines, c.-à-d. $c_{ij} \subseteq D^0(x_i) \times D^0(x_j)$.

Pour des raisons de simplicité, nous considérons une version restreinte de DisCSP où chaque agent contrôle exactement une seule variable. Nous utilisons les termes agent et variable d'une manière interchangeable et nous identifions l'index d'un agent par l'indice de sa variable. En outre, tous les agents mémorisent un ordre unique $\prec$ sur les agents. Les agents qui apparaissent avant A_i dans cet ordre sont les agents les plus prioritaires (prédécesseurs) et inversement, les agents moins prioritaires (successeurs) sont ceux qui apparaissent après A_i. Par souci de clarté, nous supposons que l'ordre total sur les agents $\prec$ est l'ordre lexicographique $[A_1, A_2, \ldots, A_n]$.

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

Définition 1 *Une **affectation** pour un agent $A_i \in \mathcal{A}$ est un uplet (x_i, v_i, t_i), où v_i est une valeur du domaine de x_i et t_i la valeur du compteur de A_i. Quand on compare deux affectations, la plus récente est celle associée avec le plus grand compteur t_i.*

Définition 2 ***L'affectation partielle courante*** *CPA (pour Current Partial Assignment) est un ensemble ordonné d'affectations $\{[(x_1, v_1, t_1), \ldots, (x_i, v_i, t_i)] \mid x_1 \prec \cdots \prec x_i\}$. Deux CPA sont* compatibles *si à chaque variable commune à ces deux sous ensembles est affectée la même valeur.*

Définition 3 ***L'horodatage**, associé à un CPA, est une liste ordonnée des compteurs $[t_1, t_2, \ldots, t_i]$ où t_j est le compteur de la variable x_j. Quand on compare deux CPA, la **plus forte** est celle associée à l'horodatage lexicographiquement supérieur. Autrement dit, celle qui a la plus grande valeur du premier compteur où les deux horodatages diffèrent, le cas échéant, sinon la plus longue.*

Définition 4 *La vue **AgentView** d'un agent $A_i \in \mathcal{A}$ contient les affectations les plus récentes reçues des agents plus prioritaires. Elle a une forme similaire à celle de la CPA. Elle est initialisée par l'ensemble des affectations vides $\{(x_j, empty, 0) \mid x_j \prec x_i\}$.*

En se basant sur les contraintes du problème, les agents peuvent inférer des ensembles d'affectations inconsistants appelés ***nogoods***.

Définition 5 *Un **nogood** écartant la valeur v_k du domaine initial de la variable x_k est une clause sous forme $x_i = v_i \wedge \ldots \wedge x_j = v_j \rightarrow x_k \neq v_k$, signifiant que l'affectation $x_k = v_k$ est inconsistante avec les affectation $x_i = v_i, \ldots, x_j = v_j$. La partie gauche ($lhs$) et la partie droite ($rhs$) sont définies à partir de la position de $\rightarrow$.*

Le domaine courant $D(x_i)$ de la variable x_i contient toutes les valeurs du domaine initial qui ne sont pas écartées par un nogood. Lorsque toutes les valeurs d'un agent x_k sont supprimées par des nogoods (c.-à-d. $D(x_i) = \emptyset$), cet agent procède à la résolution de ces derniers. Cette résolution produit un nouveau nogood ng . Soit x_j la variable ayant la plus basse priorité dans les parties gauches de tous les nogoods enregistrés par x_k et $x_j = v_j$. $lhs(ng)$ est la conjonction de toutes les parties gauches de tous ces nogoods excepté $x_j = v_j$ et $rhs(ng)$ est $x_j \neq v_j$.

2.2 Le maintien de la consistance d'arc

Il est bien connu que la propagation de contraintes joue un rôle central pour la résolution des problèmes de satisfaction de contraintes [2]. La technique de propagation de contraintes la plus ancienne et la plus utilisée est la consistance d'arc.

Définition 6 *Une contrainte c_{ij} est **arc consistante** si et seulement si $\forall\ v_i \in D(x_i), \exists\ v_j \in D(x_j)$ tel que (v_i, v_j) est autorisé par c_{ij} et $\forall\ v_j \in D(x_j), \exists\ v_i \in D(x_i)$ tel que (v_i, v_j) est autorisé par c_{ij}. Un réseau de contraintes est arc consistant ssi toutes ses contraintes sont arc consistantes.*

L'algorithme MAC (Maintaining Arc Consistency) [17] alterne des étapes d'exploration avec d'autres de propagation. Autrement dit, à chaque étape de la recherche, une affectation d'une variable est suivie par un processus de filtrage qui correspond à l'application de la consistance d'arc.

Pour la mise en œuvre de MAC dans un CSP distribué, chaque agent A_i est censé connaître toutes les contraintes dans lesquelles il est impliqué et les agents avec qui il partage ces contraintes (ses voisins). Ces derniers et les contraintes qui les relient à A_i forment le réseau de contraintes local de A_i, désigné par $CSP(i)$.

Définition 7 *Un* ***réseau de contraintes local*** $CSP(i)$ *d'un agent* $A_i \in \mathcal{A}$ *est constitué de toutes les contraintes impliquant* x_i *et toutes les autres variables de ces contraintes.*

Pour permettre aux agents de maintenir la consistance d'arc dans les CSP distribués, notre approche consiste à appliquer la consistance d'arc dans le réseau de contraintes local de chaque agent. Chaque agent A_i enregistre localement des copies de toutes les variables de $CSP(i)$. Nous supposons également que chaque agent connaît le voisinage qu'il a en commun avec ses voisins, sans connaître les contraintes qui les relient. Autrement dit, pour chacun de ses voisins A_k, un agent A_i connaît la liste des agents A_j tel qu'il existe une contrainte entre x_i et x_j et une autre liant x_k à x_j.

3 Le maintien de la consistance d'arc de manière asynchrone

Dans l'algorithme nogood-based Asynchronous Forward Checking (AFC-ng), la phase de forward-checking (FC) vise à anticiper le backtrack. Néanmoins, nous ne tirons pas profit des suppressions produites par FC si le domaine de la variable n'est pas entièrement vidé. Ces suppressions peuvent être utilisées en appliquant la consistance d'arc. Ceci est justifié par le fait que la propagation de la suppression d'une valeur, pour un agent A_i, peut générer un domaine vide pour une variable dans son réseau de contraintes local. Ainsi, nous pouvons détecter promptement l'échec et par conséquence anticiper aussitôt que possible l'opération de backtrack.

Dans les algorithmes de recherche synchrones pour résoudre les DisCSPs, les agents affectent séquentiellement leurs variables. Ainsi, les agents assignent des valeurs à leurs variables seulement lorsqu'ils détiennent l'affectation partielle courante, CPA. Nous proposons un algorithme dans lequel les agents affectent leur variables un par un suivant l'ordre total sur les agents. Par conséquent, quand un agent réussit à étendre la CPA par l'affectation de sa variable, il envoie la CPA à son successeur pour qu'il l'étende à son tour. Des copies de cette CPA sont également envoyées aux autres agents qui ne sont pas encore affectés dans la CPA pour maintenir la consistance d'arc de manière asynchrone (MACA pour *maintain arc consistency asynchronously*). Par conséquent, quand un agent reçoit une copie de la CPA, il maintient la consistance d'arc dans son réseau de contraintes local. Pour appliquer la consistance d'arc sur toutes les variables du problème, les agents communiquent aux autres agents des informations sur les suppressions de valeurs produites localement. Nous proposons deux méthodes pour réaliser ceci. La première, à savoir MACA-del, utilise un nouveau type de messages (des messages ***del***) pour propager ces informations. La deuxième méthode, nommée MACA-not, inclut les informations concernant les suppressions générées localement dans les messages ***cpa***.

3.1 Maintenir l'AC à l'aide des messages del

Dans MACA-del, tout agent A_i maintient la consistance d'arc dans son réseau de contraintes local, $CSP(i)$, chaque fois qu'un domaine d'une variable dans $CSP(i)$ est modifié. Les changements peuvent se produire sur le domaine de A_i soit sur un autre domaine dans $CSP(i)$. Dans l'algorithme MACA-del, l'agent A_i partage avec les autres agents seulement les suppressions produites dans son domaine $D(x_i)$. Elles sont propagées à travers la communication aux autres agents des nogoods qui les justifient. Ces suppressions et ces nogoods sont donc envoyés aux voisins par l'intermédiaire de messages ***del***.

L'agent A_i enregistre tous les nogoods associés à ses valeurs. Ces nogoods sont enregistrés dans $NogoodStore[x_i]$. En plus des nogoods stockés pour ses propres valeurs, A_i a besoin de mémoriser les nogoods des valeurs retirées des domaines des autres variables x_j appartenant à $CSP(i)$. Les nogoods justifiant les suppressions des valeurs de $D(x_j)$ sont stockés dans $NogoodStore[x_j]$. Par conséquent, le NogoodStore d'un agent A_i est un vecteur de plusieurs NogoodStores, un pour chaque variable dans $CSP(i)$.

Le pseudo-code de MACA-del exécuté par chaque agent A_i est illustré en Fig. 1. L'agent A_i démarre la recherche en appelant la procédure `MACA-del()`. Dans cette dernière procédure, A_i appelle `Propagate()` pour maintenir la consistance d'arc (ligne 1) dans son réseau de contraintes local, c.-à-d. $CSP(i)$. Ensuite, si A_i est l'agent d'initialisation IA (le premier agent dans l'ordre), il initialise la recherche en appelant la procédure `Assign()` (ligne 2). Puis, une boucle considère la réception et le traitement des différents types de messages.

Lors de l'appel à la procédure `Assign()`, A_i tente de trouver une affectation consistante avec l'AgentView. Si A_i ne parvient pas à trouver une telle affectation, il appelle la procédure `Backtrack()` (ligne 14). Si A_i a réussi à trouver une affectation, il incrémente son compteur t_i et génère une nouvelle CPA de son AgentView augmentée par sa propre affectation (lignes 11 et 12). Ensuite, A_i appelle la procédure `SendCPA(CPA)` (ligne 13). Si le CPA comprend les affectations de tous les agents (A_i est le dernier agent dans l'ordre ligne 15), A_i rapporte la CPA comme solution du problème et marque le drapeau *end* vrai pour arrêter la boucle principale (ligne 16). Sinon, A_i envoie en avant

```
procedure MACA-del()
01.  end ← false;   Propagate() ;
02.  if ( A_i = IA ) then Assign() ;
03.  while ( ¬end ) do
04.    msg ← getMsg();
05.    switch ( msg.type ) do
06.      cpa  : ProcessCPA(msg);
07.      ngd  : ProcessNogood(msg);
08.      del  : ProcessDel(msg);
09.      stp  : end ← true;

procedure Assign()
10.  if ( D(x_i) ≠ ∅ ) then
11.    v_i ← ChooseValue() ; t_i ← t_i+1 ;
12.    CPA ← {AgentView ∪ (x_i, v_i, t_i)} ;
13.    SendCPA(CPA) ;
14.  else Backtrack() ;

procedure SendCPA(CPA)
15.  if ( size(CPA) = n ) then
16.    broadcastMsg: stp⟨CPA⟩ ;   end ← true ;
17.  else
18.    foreach ( x_k ≻ x_i ) do  sendMsg: cpa⟨CPA⟩ to A_k ;

procedure ProcessCPA(msg)
19.  if ( msg.CPA is stronger than the AgentView ) then
20.    AgentView ← msg.CPA ;
21.    Remove all nogoods incompatible with AgentView ;
22.    if ( ¬Propagate() ) then Backtrack() ;
23.    else if ( msg.sender = A_i−1 ) then Assign() ;

function Propagate()
24.  if ( ¬AC(CSP(i)) ) then return false ;
25.  else if ( D(x_i) was changed ) then
26.    foreach ( x_j ∈ CSP(i) ) do
27.      nogoods ← get nogoods from NogoodStore[x_i]
         that are relevant to x_j ;
28.      sendMsg: del⟨nogoods⟩ to A_j ;
29.  return true ;

procedure ProcessDel(msg)
30.  Let A_k be the agent that sent msg;
31.  foreach ( ng ∈ msg.nogoods ) do
32.    if ( Compatible(ng, AgentView) ) then
33.      add(ng, NogoodStore[x_k]) ;
34.  if ( D(x_k) = ∅ ∧ x_i ∈ NogoodStore[x_k] ) then
35.    newNogood ← solve(NogoodStore[x_k]) ;
36.    add(newNogood, NogoodStore[x_i]) ;
37.    Assign() ;
38.  else if ( D(x_k) = ∅ ∨ ¬Propagate() ) then
39.    Backtrack();

procedure Backtrack()
40.  Let x_k be the variable within the empty domain;
41.  ng ← solve(NogoodStore[x_k]) ;        /* D(x_k) = ∅ */
42.  if ( ng = empty ) then
43.    broadcastMsg: stp⟨∅⟩ ;
44.    end ← true ;
45.  else        /* Let x_j be the variable on the rhs(ng) */
46.    sendMsg: ngd⟨ng⟩ to A_j ;
47.    from ( l = j to i-1 ) do
48.      AgentView[l].value ← empty ;
49.    Remove all nogoods incompatible with AgentView ;

procedure ProcessNogood(msg)
50.  if ( Compatible(lhs(msg.nogood), AgentView) ) then
51.    add(msg.nogood, NogoodStore[x_i]) ;
52.    if ( rhs(msg.nogood).value = v_i ) then Assign() ;
53.    else if ( ¬Propagate() ) then Backtrack() ;
```

FIGURE 1 – L'algorithme MACA-del exécuté par un agent A_i.

la CPA à tous les agents non encore instanciés dans la CPA (ligne 18). Ainsi, le prochain agent dans l'ordre (successeur) tentera d'élargir cette CPA en affectant sa variable dessus pendant que les autres agents maintiennent la consistance d'arc de manière asynchrone.

Quand un agent A_i reçoit un message de type ***cpa***, la procédure `ProcessCPA()` est appelée (ligne 6). Le message reçu sera traité seulement quand il détient une CPA plus forte que l'AgentView de l'agent A_i. Si c'est le cas, A_i met à jour son AgentView (ligne 20). Puis, il met à jour le NogoodStore de chaque variable dans $CSP(i)$ pour être compatible avec sa nouvelle AgentView (ligne 21). Ensuite, A_i appelle la fonction `Propagate()` pour maintenir la consistance d'arc sur $CSP(i)$ (ligne 22). Si l'AC vide un domaine dans $CSP(i)$, A_i appelle la procédure `Backtrack()` (ligne 22). Sinon, A_i vérifie s'il doit affecter sa variable (ligne 23). A_i tente d'affecter sa variable en appelant la procédure `Assign()` seulement s'il a reçu le message ***cpa*** de son prédécesseur.

Lorsque la fonction `Propagate()`, appelant A_i maintient l'AC sur son réseau de contraintes local en considérant les affectations dans son AgentView (ligne 24). Dans notre implémentation, nous avons utilisé *AC-2001* [4] pour maintenir la consistance d'arc, mais tout algorithme générique qui applique l'`AC` peut être utilisé dans MACA. MACA-del a seulement besoin que l'algorithme maintenant l'`AC` mémorise un nogood pour chaque valeur supprimée. Quand deux nogoods sont possibles pour une même valeur, nous choisissons le meilleur en utilisant l'heuristique HPLV (*Highest Possible Lowest Variable*) [10]. Si le maintien de la consistance d'arc sur $CSP(i)$ a échoué, c.-à-d. un domaine a été vidé, la fonction retourne false (ligne 24). Sinon, si le domaine de x_i a été changé (c.-à-d. il existe des suppressions à propager), A_i informe ses voisins en leurs envoyant des messages ***del*** qui contiennent les nogoods justifiant ces suppressions (lignes 27-28). Enfin, la fonction retourne true (ligne 29). Lors de l'envoi d'un message ***del*** à un voisin A_j, les nogoods qui seront communiqués à A_j sont ceux dont toutes les variables dans leurs parties gauches ont une priorité plus élevée que A_j. En outre, tous les nogoods ayant la même partie gauche sont factorisés dans un seul nogood dont la partie droite est l'ensemble de toutes les valeurs supprimées par cette partie gauche.

Quand l'agent A_i reçoit un message de type ***del***, il ajoute au NogoodStore $NogoodStore[x_k]$ de l'expéditeur A_k tous les nogoods compatibles avec l'AgentView de A_i (lignes 31-33). Puis, A_i vérifie si le domaine de x_k est vidé (c.-à-d. les valeurs restantes dans $D(x_k)$ ont été supprimées par les nogoods qui viennent d'être reçus de l'agent A_k) et x_i appartient au Nogood-

Store de x_k (c.-à-d. x_i est déjà affectée et son affectation est incluse dans au moins un nogood qui justifie la suppression d'une valeur de $D(x_k)$) (ligne 34). Si c'est le cas, A_i supprime sa valeur actuelle en mémorisant comme justification de cette suppression le nogood $newNogood$, généré à partir de la résolution du NogoodStore de x_k (ligne 35-36). Ensuite, il appelle la procédure `Assign()` pour tenter d'assigner une autre valeur (ligne 37). Sinon, si $D(x_k)$ est vidé (x_i n'est pas affectée) ou si un échec se produit lors du maintien de la consistance d'arc, A_i doit faire un retour-arrière (`Backtrack()`, ligne 39).

Chaque fois qu'un échec se produit sur le domaine d'une variable x_k dans $CSP(i)$ (comprenant x_i), la procédure `Backtrack()` est appelée. L'agent A_i procède à la résolution de ces nogoods qui ont produit l'échec, et génère un nouveau nogood ng (ligne 41). ng est la conjonction des parties gauches de tous ces nogoods mémorisés par A_i dans $NogoodStore[x_k]$. Si ng est vide, le problème n'a pas de solution. A_i termine son exécution après l'envoi d'un message de type ***stp*** à tous les agents dans le système (lignes 42-44). Sinon, A_i fait un retour-arrière par l'envoi d'un message ***ngd*** à l'agent A_j qui détient la variable dans la partie droite de ng (ligne 46). Ensuite, A_i met à jour son AgentView afin de ne garder que les affectations des agents qui sont placés avant A_j dans l'ordre (lignes 47-48). A_i met également à jour le NogoodStore de toutes les variables dans $CSP(i)$ en enlevant les nogoods incompatibles avec sa nouvelle AgentView (ligne 49).

À la réception d'un message ***ngd***, A_i vérifie la validité du nogood reçu (ligne 50). S'il est compatible avec son AgentView, A_i l'ajoute à son NogoodStore (c.-à-d. $NogoodStore[x_i]$, ligne 51). Puis, A_i vérifie si la valeur dans la partie droite du nogood reçu correspond à sa valeur actuelle. Si c'est le cas, A_i appelle la procédure `Assign()` pour essayer de trouver une autre affectation à sa variable qui soit consistante avec les choix des agents les plus prioritaires (ligne 52). Sinon, A_i appelle la fonction `Propagate()` pour maintenir l'AC. Si un domaine vide apparaît dans son réseau de contraintes local, A_i appelle la procédure `Backtrack()` (ligne 53).

3.2 Maintenir l'AC sans type supplémentaire de message

Dans la suite, nous montrons comment maintenir la consistance d'arc sans utiliser un type supplémentaire de messages. Dans MACA-del, la maintenance de la consistance globale est réalisée en communiquant aux agents voisins (agents dans $CSP(i)$) toutes les valeurs élaguées de $D^0(x_i)$. Cela peut générer de nombreux messages de type ***del*** dans le réseau et donc créer un goulot d'étranglement au niveau de la communication.

```
procedure MACA-not()
01.  end ← false;  Propagate() ;
02.  if ( A_i = IA ) then Assign() ;
03.  while ( ¬end ) do
04.    msg ← getMsg();
05.    switch ( msg.type ) do
06.      cpa  : ProcessCPA(msg);
07.      ngd  : ProcessNogood(msg);
08.      stp  : end ← true;

procedure SendCPA(CPA)
09.  if ( size(CPA) = n ) then
10.    broadcastMsg: stp⟨CPA⟩ ;
11.    end ← true ;
12.  else
13.    foreach ( x_k ≻ x_i ) do
14.      nogoods ← ∅;
15.      foreach ( x_j ∈ {CSP(i) ∩ CSP(k)} | x_j ≻ x_i ) do
16.        nogoods ← nogoods ∪ getNogoods(x_j) ;
17.      sendMsg: cpa⟨CPA, nogoods⟩ to A_k ;

procedure ProcessCPA(msg)
18.  if ( msg.CPA is stronger than the AgentView ) then
19.    AgentView ← CPA ;
20.    Remove all nogoods incompatible with AgentView ;
21.    foreach ( ng ∈ msg.nogoods ) do
22.      let x_k be the variable in the rhs(ng);
23.      add(ng, NogoodStore[x_k]) ;
24.    if ( ¬Propagate() ) then Backtrack() ;
25.    else if ( msg.sender = A_{i-1} ) then Assign();

function Propagate()
26.  return AC(CSP(i)) ;
```

FIGURE 2 – Nouvelles lignes/procédures de MACA-not par rapport à MACA-del.

En outre, les agents doivent effectuer plus d'efforts pour traiter ces nombreux messages ***del***.

Dans MACA-not, la communication des suppressions produites dans $CSP(i)$ est retardée jusqu'à ce que l'agent A_i veuille envoyer un message ***cpa***. Lors de l'envoi d'un message ***cpa*** à un agent moins prioritaire A_k, A_i attache les nogoods justifiant la suppression des valeurs dans $CSP(i)$ au message ***cpa***. Mais il n'attache pas tous ces nogoods car certaines variables ne sont pas pertinentes pour l'agent A_k (non connectées à x_k par une contrainte). MACA-not partage avec A_k tous les nogoods justifiant les suppressions concernant les variables non encore instanciées qui partagent une contrainte avec à la fois A_i et A_k (c.-à-d. les variables dans $\{CSP(i) \cap CSP(k)\} \setminus vars(CPA)$). Ainsi, quand A_k reçoit la CPA dans un message ***cpa***, il reçoit avec toutes les suppressions effectuées dans $CSP(i)$ qui peuvent l'amener à plus de propagation par la consistance d'arc.

Nous présentons dans Fig. 2 le pseudo-code de l'algorithme MACA-not. Seules les procédures qui sont nouvelles ou différentes de celles de MACA-del (Fig. 1) sont présentées. La fonction `Propagate()` n'envoie

plus des messages ***del***, elle ne fait que le maintien de la consistance d'arc dans $CSP(i)$ et renvoie true ssi aucun domaine n'a été vidé.

Dans la procédure `SendCPA(CPA)`, lors de l'envoi d'un message ***cpa*** à un agent A_k, A_i attache au CPA les nogoods qui justifient les suppressions des domaines des variables dans $CSP(i)$ qui sont contraintes avec A_k (lignes 13-17, Fig. 2).

Chaque fois que l'agent A_i reçoit un message ***cpa***, la procédure `ProcessCPA()` est appelée (ligne 6). Le message reçu sera traité seulement quand il contient une CPA plus forte que l'AgentView de A_i. Dans ce cas, A_i met à jour son AgentView (ligne 19). Puis, il met à jour les NogoodStores pour être compatibles avec la CPA qui vient d'être reçue (ligne 20). Ensuite, tous les nogoods contenus dans le message reçu sont ajoutés au NogoodStore (lignes 21-23). Les nogoods sont évidemment ajoutés au NogoodStore des variables dans leurs parties droites (c.-à-d. ng est ajouté au $NogoodStore[x_k]$ si x_k est la variable dans $rhs(ng)$). Puis, A_i appelle la fonction `Propagate()` pour maintenir la consistance d'arc dans $CSP(i)$ (ligne 24). Si le domaine d'une variable dans $CSP(i)$ se vide, A_i appelle la procédure `Backtrack()` (ligne 24). Sinon, A_i vérifie s'il doit affecter sa variable (ligne 25). A_i essaie d'assigner sa variable en appelant la procédure `Assign()` seulement si l'expéditeur du message ***cpa*** est son prédécesseur A_{i-1}.

4 L'analyse théorique

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

Lemme 1 *MACA termine.*

Schéma de preuve. La preuve est proche de celle donnée dans [21]. On peut facilement obtenir par induction sur l'ordre des agents qu'il y aura un nombre limité de nouvelles CPA générées (au plus d^n, où n est le nombre de variables et d est la taille maximale des domaines), et que les agents ne peuvent jamais tomber dans une boucle infinie pour une CPA donnée.

Lemme 2 *MACA ne peut déduire une incohérence si une solution existe.*

Preuve. Chaque fois qu'une forte CPA ou un message ***ngd*** sont reçus, les agents dans MACA mettent à jour leurs NogoodStore. Dans MACA-del, les nogoods contenus dans les messages ***del*** sont acceptés seulement s'ils sont compatibles avec l'AgentView (lignes 31-33, Fig. 1). Dans MACA-not, les nogoods contenus dans les messages ***cpa*** sont compatibles avec la CPA reçue et ils sont acceptés seulement quand la CPA est plus forte que l'AgentView (ligne 18, Fig. 2). Ainsi, pour chaque CPA qui peut potentiellement conduire à une solution, les agents enregistrent seulement des nogoods valides. Puisque tous les nogoods sont générés par des inférences logiques à partir des contraintes existantes, un nogood vide ne peut pas être déduit si une solution existe.

Théoreme 1 *MACA est correct.*

Preuve. Pour montrer que MACA est correct, on suit le même schéma que [21]. Le fait que les agents envoient en avant seulement des solutions partielles consistantes dans des messages ***cpa*** à un seul endroit `Assign()` (ligne 13, Fig. 1), implique que les agents reçoivent seulement des affectations consistantes. La solution est rapportée uniquement par le dernier agent dans la procédure `SendCPA(CPA)` (ligne 16, Fig. 1 et ligne 10, Fig. 2). À ce stade, tous les agents ont affecté leurs variables et leurs affectations sont consistantes. Ainsi, MACA est correct. La complétude provient du fait que MACA termine et ne rapporte pas d'incohérence si une solution existe (Lemmes 1 and 2).

Théoreme 2 *MACA nécessite un espace polynomial par agent.*

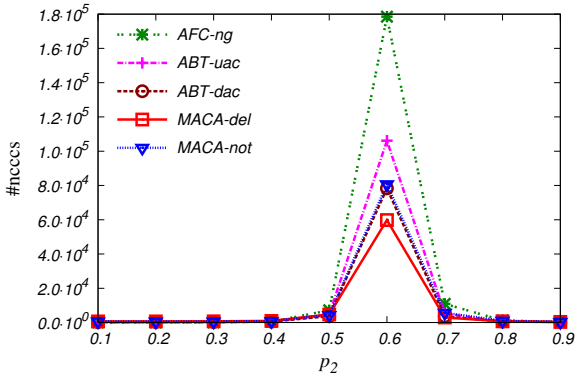
Preuve. Dans chaque agent, MACA mémorise un nogood de taille maximale n par valeur supprimée dans son réseau de contraintes local. Le réseau de contraintes local contient au plus n variables. Ainsi, la complexité spatiale de MACA est en $O(n^2d)$ dans chaque agent où d est la taille maximale des domaines initiaux.

Théoreme 3 *Les messages dans MACA sont polynomialement bornés.*

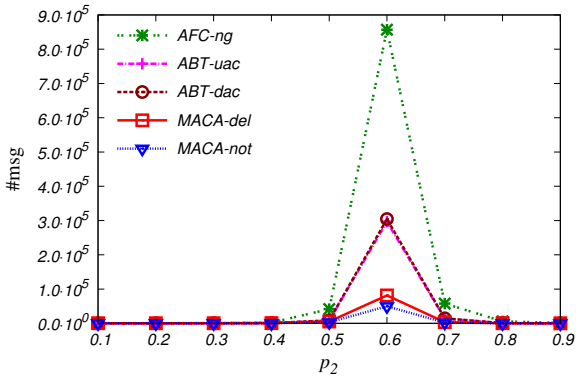
Preuve. Les plus grands messages dans MACA-del sont les messages ***del***. Dans le pire des cas, un message ***del*** contient un nogood pour chaque valeur. Donc, la taille des messages ***del*** est en $O(nd)$. Dans MACA-not, les plus grands messages sont des messages de type ***cpa***. Le pire des cas est un message ***cpa*** qui contient une CPA et un nogood pour chaque valeur dans le réseau de contraintes local. Ainsi, la taille d'un message ***cpa*** est en $O(n + n^2d) = O(n^2d)$.

5 Résultats expérimentaux

Dans cette section, nous comparons expérimentalement les algorithmes MACA avec ABT-uac, ABT-dac [6] et AFC-ng [7]. Ces algorithmes ont été évalués sur des DisCSPs binaires uniformes aléatoires. Toutes

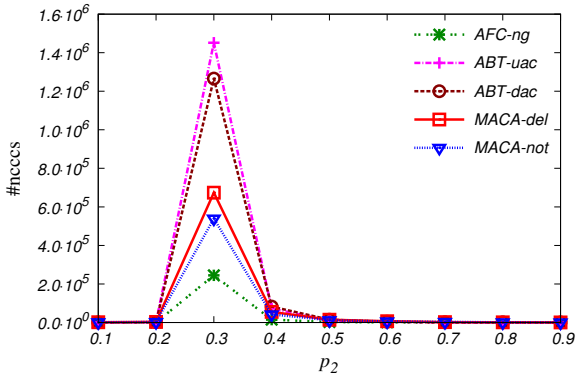


(a) *Le #ncccs effectués*

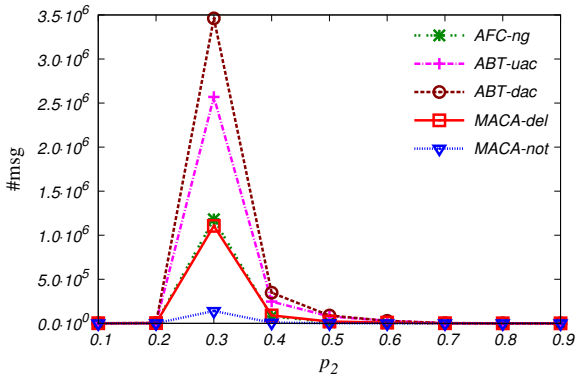


(b) *Le #msg échangés*

FIGURE 3 – Le *#ncccs* effectués et le *#msg* échangés pour résoudre les problèmes binaires uniformes aléatoires avec une faible densité où $p_1 = 0.25$.



(a) *Le #ncccs effectués*



(b) *Le #msg échangés*

FIGURE 4 – Le *#ncccs* effectués et le *#msg* échangés pour résoudre les problèmes binaires uniformes aléatoires avec une densité élevée où $p_1 = 0.70$.

nos expériences ont été réalisées sur la plateforme DisChoco 2.0 platform [1] [20] où les agents sont simulés par des threads java qui communiquent par échange de messages. Tous les algorithmes ont été testés en utilisant le même ordre d'agents (ordre lexicographique) et la même heuristique de sélection de nogood (*HPLV*) [10]. Pour ABT-dac et ABT-uac nous avons implémenté une version améliorée du mécanisme de détection de terminaison de Silaghi [18].

Nous évaluons la performance des algorithmes par le coût de communication [12] et l'effort de calcul. Le coût de communication est mesuré par le nombre total des messages échangés entre les agents durant l'exécution de l'algorithme (*#msg*). Ce nombre total inclut les messages échangés pour détecter la terminaison (messages système). L'effort de calcul est mesuré par le nombre de tests non-concurrents de contraintes (*#nccc*) (non-concurrent constraint checks) [26]. *#ncccs* est la métrique utilisée dans la résolution de contraintes distribuée pour simuler le temps de calcul.

Les algorithmes sont testés sur des DisCSPs uniformes binaires $\langle n, d, p_1, p_2\rangle$, avec n agents/variables, d valeurs par variable, une connectivité définie comme étant la proportion p_1 des contraintes binaires existantes, et une dureté de contraintes définie comme étant la proportion p_2 des paires de valeurs interdites. Nous présentons les résultats pour deux classes de problèmes : des problèmes avec une faible densité $\langle 20,\ 10,\ 0.25, p_2\rangle$ et des problèmes avec une densité élevée $\langle 20,\ 10,\ 0.7, p_2\rangle$. Pour chaque paire fixée (p_1, p_2), nous avons généré 100 instances. Nous reportons les moyennes de 100 exécutions.

La Fig. 3 présente les résultats sur les problèmes avec une faible densité ($p_1 = 0.25$). En terme d'effort de calcul (*#nccc*) (Fig. 3(a)), les algorithmes de maintien de la consistance d'arc sont largement meilleurs que AFC-ng qui maintient seulement le forward checking. MACA-del domine tous les autres al-

1. http ://dischoco.sourceforge.net/

gorithmes. MACA-not a une performance similaire à celle de ABT-dac mais il est plus performant que ABT-uac. En ce qui concerne le coût de communication ($\#msg$) (Fig. 3(b)), l'amélioration des algorithmes qui maintiennent l'AC (par rapport à AFC-ng) est plus nette par rapport à celle d'effort de calcul. ABT-uac et ABT-dac nécessitent presque le même nombre de messages échangés. En comparant les algorithmes qui maintiennent la consistance d'arc entre eux, la figure montre que ceux qui ont un comportement synchrone (les algorithmes MACA) améliorent ceux qui ont un comportement asynchrone (ABT-dac et ABT-uac) d'un facteur 6. Dans les problèmes avec une faible densité, le maintien de la consistance d'arc dans une recherche synchrone semble donner des performances meilleures par rapport à une recherche asynchrone. Dans le pic de complexité, MACA-not échange moins de messages que MACA-del.

Nous présentons les résultats sur les problèmes avec une densité élevée ($p_1 = 0.7$) dans la Fig. 4. En ce qui concerne les $\#msg$ (Fig. 4(a)), la première remarque est que les algorithmes asynchrone sont moins performants que ceux qui affectent leurs variables d'une manière séquentielle. Contrairement aux problèmes avec une faible densité, AFC-ng domine tous les autres algorithmes. Ceci est cohérent avec les résultats dans les CSP centralisés où FC a un meilleur comportement sur les problèmes denses que sur ceux avec une faible densité [3, 8]. De nouveau, ABT-dac est plus performant que ABT-uac. Au contraire des problèmes avec une faible densité, MACA-not est maintenant plus performant que MACA-del. En terme de charge de communication (Fig. 4(b)) dans les problèmes denses, les algorithmes asynchrones (ABT-uac et ABT-dac) nécessitent un grand nombre de messages. MACA-del n'améliore pas AFC-ng en raison du grand nombre de messages ***del*** échangés par MACA-del. Dans ces problèmes, MACA-not est l'algorithme qui nécessite le moins de messages. MACA-not améliore les deux autres algorithmes synchrones (AFC-ng et MACA-del) d'un facteur 11 et les deux autres asynchrones (ABT-uac et ABT-dac) d'un facteur 40.

À partir de ces expérimentations, nous pouvons conclure que dans les algorithmes synchrones, le maintien de la consistance d'arc est mieux que le maintien du forward checking en termes de temps de calcul sur les réseaux à faible densité, et est toujours mieux en termes de charge de communication. Nous pouvons également conclure que le maintien de la consistance d'arc dans les algorithmes synchrones produit plus de bénéfices que le maintien de la consistance d'arc dans les algorithmes asynchrones comme ABT.

6 Conclusion

Nous avons proposé deux nouveaux algorithmes de recherche synchrone pour résoudre les DisCSP. Il s'agit des premières tentatives de maintien de la consistance d'arc dans une recherche synchrone pour les DisCSP. Le premier algorithme, MACA-del, maintient la consistance d'arc grâce à un type de messages supplémentaire, des messages de suppression. Le deuxième algorithme, MACA-not, réalise la consistance d'arc sans aucun nouveau type de message. Malgré le caractère synchrone de la recherche dans ces deux algorithmes, ils exécutent la phase de consistance d'arc d'une manière asynchrone. Nos expérimentations montrent que le maintien de la consistance d'arc dans une recherche synchrone produit des bénéfices bien supérieures à ceux offerts par le maintien de la consistance d'arc dans les algorithmes asynchrones comme ABT. La charge de communication de MACA-del peut être nettement inférieure à celle de AFC-ng, le meilleur algorithme synchrone à ce jour. Grâce à son utilisation plus parcimonieuse des messages, MACA-not montre encore plus de gain.

Références

[1] Ramón Béjar, Carmel Domshlak, Cèsar Fernández, Carla Gomes, Bhaskar Krishnamachari, Bart Selman et Magda Valls : Sensor networks and distributed CSP : communication, computation and complexity. *Artificial Intelligence*, 161:117–147, 2005.

[2] Christian Bessiere : Constraint Propagation. *In Handbook of Constraint Programming (Foundations of Artificial Intelligence)*, chapitre 3. Elsevier, New York, NY, USA, 2006.

[3] Christian Bessiere et Jean-Charles Régin : MAC and combined heuristics : Two reasons to forsake FC (and CBJ ?) on hard problems. *In Proceedings of CP'96*, pages 61–75, 1996.

[4] Christian Bessiere et Jean-Charles Régin : Refining the basic constraint propagation algorithm. *In Proceedings of IJCAI'01*, pages 309–315, 2001.

[5] Ismel Brito et Pedro Meseguer : Distributed Forward Checking. *In Proceeding of CP'03*, pages 801–806, Ireland, 2003.

[6] Ismel Brito et Pedro Meseguer : Connecting ABT with Arc Consistency. *In CP*, pages 387–401, 2008.

[7] Redouane Ezzahir, Christian Bessiere, Mohamed Wahbi, Imade Benelallam et El-Houssine

Bouyakhf : Asynchronous inter-level Forward-Checking for DisCSPs. *In Proceedings of CP'09*, pages 304–318, 2009.

[8] Stuart A. Grant et Barbara M. Smith : The phase transition behaviour of maintaining arc consistency. *In Proceedings of the 12th European conference on Artificial intelligence*, ECAI'96, pages 175–179, 1996.

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

[10] Katsutoshi Hirayama et Makoto Yokoo : The Effect of Nogood Learning in Distributed Constraint Satisfaction. *In Proceedings of ICDCS'00*, pages 169–177, 2000.

[11] Hyuckchul Jung, Milind Tambe et Shriniwas Kulkarni : Argumentation as Distributed Constraint Satisfaction : Applications and Results. *In Proceedings of AGENTS'01*, pages 324–331, 2001.

[12] Nancy A. Lynch : *Distributed Algorithms*. Morgan Kaufmann Series, 1997.

[13] Rajiv T. Maheswaran, Milind Tambe, Emma Bowring, Jonathan P. Pearce et Pradeep Varakantham : Taking DCOP to the real world : Efficient complete solutions for distributed multi-event scheduling. *In Proceedings of AAMAS'04*, 2004.

[14] Amnon Meisels et Roie Zivan : Asynchronous Forward-Checking for DisCSPs. *Constraints*, 12 (1):131–150, 2007.

[15] Adrian Petcu et Boi Faltings : A value ordering heuristic for distributed resource allocation. *In Proceeding of CSCLP'04*, pages 86–97, Feb 2004.

[16] Patrick Prosser, Chris Conway et Claude Muller : A constraint maintenance system for the distributed resource allocation problem. *Intelligent Systems Engineering*, 1(1):76–83, oct 1992.

[17] Daniel Sabin et Eugene Freuder : Contradicting conventional wisdom in constraint satisfaction. *In Proceedings of CP'94*, volume 874, pages 10–20, 1994.

[18] Marius-Calin Silaghi : Generalized dynamic ordering for asynchronous backtracking on DisCSPs. *In DCR workshop, AAMAS-06*, 2006.

[19] Marius-Calin Silaghi, Djamila Sam-Haroud et Boi Faltings : Consistency maintenance for ABT. *In Proceedings of CP'01*, pages 271–285, 2001.

[20] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere et El-Houssine Bouyakhf : DisChoco 2 : A Platform for Distributed Constraint Reasoning. *In Proceedings of DCR'11*, pages 112–121, 2011. URL http://www.lirmm.fr/coconut/dischoco/.

[21] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere et El Houssine Bouyakhf : Nogood-Based Asynchronous Forward-Checking Algorithms. *Constraints*, 18(3):404–433, 2013.

[22] Richard J. Wallace et Eugene C. Freuder : Constraint-based multi-agent meeting scheduling : effects of agent heterogeneity on performance and privacy loss. *In Proceeding of the workshop on DCR'02*, pages 176–182, 2002.

[23] Makoto Yokoo : Algorithms for distributed constraint satisfaction problems : A review. *Autonomous Agents and Multi-Agent Systems*, 3(2): 185–207, 2000.

[24] Makoto Yokoo, Edmund H. Durfee, Toru Ishida et Kazuhiro Kuwabara : The distributed constraint satisfaction problem : Formalization and algorithms. *IEEE Transactions on Knowledge and Data Engineering*, 10:673–685, 1998.

[25] Roie Zivan et Amnon Meisels : Synchronous vs asynchronous search on DisCSPs. *In Proceedings of EUMA'03*, 2003.

[26] Roie Zivan et Amnon Meisels : Message delay and DisCSP search algorithms. *Annals of Mathematics and Artificial Intelligence*, 46(4):415–439, 2006.

Avec le soutien de

afpc · I3 IAF · Inria INVENTEURS DU MONDE NUMÉRIQUE · Aix*Marseille université · FRIIAM Fédération de Recherche en Informatique et Interactions d'Aix-Marseille FRIIAM – FR 3513 · CONSEIL GENERAL BOUCHES-DU-RHÔNE cg13.fr · communauté du PAYS D'AIX www.agglo-paysdaix.fr · COSYTEC · PROLOGIA Expertise - Conseil - Solutions