Actes des Sixièmes Journées Francophones de Programmation par Contraintes

JFPC'2010

9-11 Juin 2010, Caen, France

Organisation

Groupe de Recherche en Informatique, Image, Automatique et Instrumentation de Caen (GREYC)

Présidents des journées

Samir LOUDNI et Bruno ZANUTTINI, GREYC, Université de Caen

Président du comité de programme

Christophe LECOUTRE, CRIL, Université d'artois

Comité de programme

Christian BESSIERE, LIRMM, Montpellier
Lucas BORDEAUX, Microsoft Research (U.K.)
Gilles CHABERT, EMN/LINA, Nantes
Philippe CHATALIC, LRI, Orsay
Simon DE GIVRY, INRA, Toulouse
Yves DEVILLE, UCL, Louvain (Belgique)
Khalil DJELLOUL, LIFO, Orléans
François FAGES, INRIA, Rocquencourt
Pierre FLENER, IT, Uppsala University (Suède)
Arnaud GOTLIEB, IRISA, Rennes
Emmanuel HEBRARD, 4C, Cork (Irlande)
Narendra JUSSIEN, EMN/LINA, Nantes
Matthias KLEINER, LSIS, Marseille
Arnaud LALLOUET, GREYC, Caen
Michel LEMAÎTRE, Onera, Toulouse
Chu-Min LI, LARIA, Amiens
Samir LOUDNI, GREYC, Caen
Laurent MICHEL, University of Connecticut (USA)
Wadi NAANAA, Université de Monastir (Tunisie)
Samba NDOJH NDIAYE, LIRIS, Lyon
Bertrand NEVEU, INRIA, Sophia
Gilles PESANT, Polytechnique, Montréal (Canada)
Thierry PETIT, EMN/LINA, Nantes
Cédric PIETTE, CRIL, Lens
Claude-Guy QUIMPER, Université de Waterloo (Canada)
Olivier ROUSSEL, CRIL, Lens
Frédéric SAUBION, LERIA, Angers
Pierre SCHAUS, Dynadec
Christine SOLNON, LIRIS, Lyon
Sébastien TABARY, CRIL, Lens
Cyril TERRIOUX, LSIS, Marseille
Gilles TROMBETTONI, INRIA, Sophia
Charlotte TRUCHET, LINA, Université de Nantes
Elise VAREILLES, Mines, Albi-Carmaux
Julien VION, LAMIH, Valenciennes

Relecteurs additionnels

Gilles AUDEMARD, Bernard BOTELLA, Thi-Bich-Hanh DAO, Julien DUPUIS, Steven GAY, Frédéric GARDI, Olfat IBRAHIM, Luc JAULIN, Jean-Marie LAGNIEZ, Matthieu LOPEZ, Thierry MARTINEZ, Julien MENANA, Jean-Philippe MÉTIVIER, Sébastien MOUTHUY, Justin PEARSON, Marie PELLEAU, Nicolas PRCOVIC, Abdelilah SAKTI, Sylvain SCHMITZ, Alessandro ZANARINI, Quan ZHE

Comité d'organisation

Patrice BOIZUMAULT
Xavier DUPONT
Arnaud LALLOUET
Jean-philippe MÉTIVIER
Jérémie VAUTARD
Mehdi KHIARI
Mathieu FONTAINE

Table des matières

Index des auteurs

Préface

La programmation par contraintes (PPC) est une discipline à la croisée des domaines de l'intelligence artificielle et de la recherche opérationnelle. La PPC permet la modélisation et la résolution de problèmes s'exprimant sous formes de contraintes portant sur une ou plusieurs variables (inconnues). Certains problèmes nécessitent également la prise en compte d'un (ou plusieurs) critère(s) à optimiser. L'un des objectifs majeurs de la PPC est d'offrir à l'utilisateur un éventail d'outils de modélisation qui soit à la fois large et simple d'utilisation. Ces outils correspondent à des langages et environnements de modélisation ou encore à des "modèles-types" de contraintes appelées contraintes globales. Un autre objectif primordial pour la programmation par contraintes est de rendre les solveurs (systèmes de résolution) robustes. Cela signifie que les solveurs doivent être aptes à compenser une mauvaise modélisation de la part de l'utilisateur par un jeu de méthodes permettant, en cours de résolution, d'identifier et exploiter automatiquement la structure des problèmes à résoudre, typiquement sur la base de mécanismes d'apprentissage (acquisition de contraintes, enregistrement de nogoods, éléments de nature statistiques sur les conflits, ...).

De nombreux progrès ont été réalisés ces dernières années, tant du point de vue de la modélisation que du point de vue de la résolution. De plus, un certain nombre de plate-formes, souvent complémentaires, ont été développées dans la communauté et rendues accessibles (le plus souvent librement) à l'utilisateur. Ces plate-formes proposent des approches complètes et/ou incomplètes, des codages en variables entières et/ou booléennes et/ou réelles, l'utilisation de contraintes globales et/ou contraintes tables, ... Ce constat est une grande satisfaction et une réelle motivation pour les chercheurs en programmation par contraintes, sachant que la marge de progression est encore importante. Le succès de la programmation par contraintes est maintenant largement établi dans le monde industriel, et les applications sont nombreuses dans des domaines aussi variés que la bio-informatique, la planification, la finance, la logistique, le sport, ou encore les télécommunications.

Les Journées Francophones de Programmation par Contraintes (JFPC) sont une manifestation permettant des échanges conviviaux entre les membres de la communauté francophone, qu'ils soient chercheurs, enseignants-chercheurs, ingénieurs, enseignants ou doctorants. Ces journées permettent une diffusion en langue française de travaux qui sont soit originaux, soit publiés en langue anglaise dans les actes de certaines conférences internationales reconnues (CP, AAAI, IJCAI, ECAI, CPAIOR, ...). Les JFPC sont issues de la fusion des conférences JFPLC (Journées Francophones de la Programmation Logique avec Contraintes) nées en 1992 et JNPC (Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets) nées en 1994. Cette sixième édition des JFPC fait suite à celles qui se sont tenues à Orléans (2009), Nantes (2008), Rocquencourt (2007), Nîmes (2006) et Lens (2005).

Quarante articles, provenant de France, mais aussi de Belgique, d'Espagne, d'Algérie et du Canada ont été soumis à cette sixième édition des JFPC. Parmi ces articles, le comité de programme en a sélectionné vingt-sept ; sept autres articles décrivant des travaux moins aboutis, mais comportant des perspectives de recherche prometteuses, ont été sélectionnés pour une présentation sous forme de poster. Les thèmes abordés par les articles sélectionnés concernent notamment la logique propositionnelle (SAT et extensions de SAT), les techniques de filtrage et de résolution, la modélisation, la génération de tests de logiciels, les contraintes qualitatives, les contraintes sur les domaines continus et différentes applications du monde industriel.

Le programme des JFPC'2010 comporte également deux conférences invitées. La première, de Michel RUEHER, présente les apports et potentiels de la programmation par contraintes en optimisation globale sous contraintes. La seconde, de Helmut SIMONIS, porte sur l'amélioration de la modélisation en contraintes à partir de la visualisation. Une session spéciale dédiée aux logiciels libres est également inscrite dans le programme : elle concerne SAT4J (`http://www.sat4j.org`) et son utilisation dans Eclipse, MTSS (`http://www.parallel-sat.net`) un solveur pour machines multi-coeurs, et Qecode (`http://www.univ-orleans.fr/lifo/software/qecode`) un solveur de problèmes de contraintes quantifiées.

Je tiens à remercier tous les auteurs qui ont soumis un article à la conférence, participant ainsi à la vitalité de la recherche francophone dans le domaine de la PPC, aux membres du comité de programme qui ont joué le jeu d'un processus de relecture soigné et constructif, à Christine SOLNON présidente de l'AFPC (Association Française de Programmation par Contraintes) pour ses nombreux conseils, à l'équipe organisatrice du GREYC et en particulier à ses deux présidents

Samir LOUDNI et Bruno ZANUTTINI toujours disponibles, efficaces et de bonne humeur. Je tiens également à remercier Olivier ROUSSEL pour son aide à la réalisation des actes de la conférence.

Enfin, nous remercions nos partenaires académiques et industriels pour leur soutien financier : l'AFPC, le GREYC, l'université de Caen Basse-Normandie, la Région Basse Normandie, l'INRIA, Bouygues et IBM. Le succès de ces journées est aussi à porter à leur crédit.

Christophe LECOUTRE
Président du comité de programme des JFPC'2010

Apports et Potentiels de la Programmation par Contraintes en Optimisation Globale sous Contraintes*

Michel Rueher

Université de Nice-Sophia Antipolis/CNRS I3S
930, Route des Colles - BP 145
06903 Sophia Antipolis Cedex France

michel.rueher@gmail.com

Résumé

Dans cet exposé, nous présentons quelques apports et potentiels de la programmation par contraintes pour l'optimisation globale sous contraintes dans le domaine continu. Nous nous intéressons donc uniquement à des problèmes d'optimisation globale dont l'objectif est de minimiser une fonction objectif en respectant un ensemble d'inégalités et d'égalités. Formellement, ces problèmes sont définis de la manière suivante :

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & g_i(x) = 0,\ i \in \{1,..,k\} \\ & h_j(x) \leq 0,\ j \in \{1,..,m\} \end{array} \quad (1)$$

avec $x \in \mathbf{x}$, $f : I\!R^n \rightarrow I\!R$
$g_i : I\!R^n \rightarrow I\!R$ et $h_j : I\!R^n \rightarrow I\!R$
Les fonctions f, g_i et h_j peuvent être non-linéaires mais elles doivent être continues et différentiables sur un vecteur $\mathbf{x}$ d'intervalles de $I\!R$.

La principale difficulté de ces problèmes vient du fait qu'il existe de très nombreux minimum locaux mais que peu d'entre eux sont des minimum globaux.

Nous montrons d'abord que la PPC (Programmation par Contraintes) offre un cadre élégant et rigoureux pour la résolution de ces problèmes ; par cadre rigoureux nous entendons un cadre qui permet de garantir la correction des résultats. Cette correction, qui exige des calculs avec des arrondis conservatifs, repose sur l'utilisation de l'arithmétique sur les intervalles.

Nous montrons ensuite que l'utilisation des techniques de filtrage et de réfutation permet de mettre en oeuvre de manière élégante et rigoureuse des heuristiques comme l'OBR, qui sont non-rigoureuses, mais qui améliorent significativement les performances.

Nous montrons aussi que l'utilisation de contraintes globales rigoureuses offre une alternative efficace aux méthodes locales pour la réduction de la borne supérieure et pour la recherche de points de départ bien adaptés à la méthode de Newton.

Enfin nous discutons de quelques problèmes ouverts, en particulier des difficultés de la PPC à résoudre efficacement des problèmes de grande taille.

Références

[1] Alexandre Goldsztejn, Yahia Lebbah, Claude Michel, Michel Rueher. Capabilities of Constraint Programming in Safe Global Optimization. Reliable Computing, vol. 15 (to appear).

[2] Claude Michel, Michel Rueher. Handling software upgradeability problems with MILP solvers. Proc. of. LoCoCo 2010, Workshop of SAT at FLoC 2010 on Logics for Component Configuration (to appear in EPCTC).

*Ce travail a été partiellement supporté par 7ème programme européen (FP7/2007-2013), projet MANCOOSI, grant agreement n. 214898.

Improving Constraint Modelling Using Visualization

Helmut Simonis

Cork Constraint Computation Centre
Department of Computer Science
University College Cork
Ireland *
h.simonis@4c.ucc.ie

Abstract

This talk gives an overview of CP-Viz, a constraint solver independent visualization toolkit for understanding search trees, variables and global constraints of modern constraint applications. We show how visualization can play a central role in developing, analyzing and tuning constraint models. Together with the declarative problem modelling framework of CP, and the specialized propagation methods inside global constraints it is a key element for rapid application development with constraints.

This is demonstrated on three practical examples. In the first example we compare different search strategies for rectangle packing problems, which require millions of search steps. We show that, even when a detailed search tree visualization is no longer possible, visualization can still extract key information explaining the differences between the search strategies.

In the second example we consider a sports scheduling example (suggested by R. Finkel) and show how missing redundant constraints are found with the help of the visualization tools.

The last example given is a constraint optimization problem, minizing the makespan in a scheduling problem with cumulative resources. The problem was introduced by R. Nieuwenhuis, based on a real-life, industrial application. The visualization explains a problem with the chosen search strategy caused by missing propagation in the implementation of the cumulative constraint of ECLiPSe. The idea used was generalized in CP-Viz to allow systematic checking of propagation invariants at each step of the the search process.

CP-Viz is a post-mortem visualization tool which collects light-weight execution traces in a generic XML format based on the global constraint catalog [1]. This trace log is then processed by an extensible Java program which produces vector based SVG output for interactive use, but which can also easily be integrated in reports and presentation slides. CP-Viz was initially developed for an ELearning course [2] for ECLiPSe [3], but has since been adapted to SICStus Prolog, Choco and the proposed JSR-331 (Java Constraint API, `http://jcp.org/en/jsr/detail?id=331`) reference implementation. CP-Viz is being developed under an open-source Modzilla licence, more information can be found at `http://4c.ucc.ie/~hsimonis/cpviz.pdf`.

*This work was supported by Science Foundation Ireland (Grant Number 05/IN/I886). The support of Cisco Systems and of the Silicon Valley Community Foundation is gratefully acknowledged.

Références

[1] N. Beldiceanu, M. Carlsson, and J.X. Rampon. Global constraint catalog. Technical Report T2005 :08, SICS, May 2005.

[2] Helmut Simonis. An ECLiPSe ELearning course. `http://4c.ucc.ie/~hsimonis/ELearning/index.htm`, 2009.

[3] Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe : A platform for constraint logic programming. *ICL Systems Journal*, 12 :159–200, May 1997.

HD_DBT : Hypertree Décomposition pour la résolution des problèmes de satisfaction de contraintes basée sur un Dual Backtracking

Kamal Amroun[1] **Zineb Habbas** [2]

[1] Département d'Informatique, Université de Béjaïa, Algérie ,
[2] Laboratoire L.I.T.A, Université de Metz, France
k_amroun25@yahoo.fr zineb@univ-metz.fr

Résumé

L'hypertree decomposition généralisée est l'approche la plus générale connue dans la littérature pour identifier des sous classes traitables de problèmes de satisfaction de contraintes (CSPs) n-aires représentés à l'aide d'hypergraphes. Seulement, quoi que sa complexité théorique est bornée par la largeur de la décomposition, la méthode proposée pour la résolution du CSP donné sous forme d'hypertree n'est pas efficace en pratique. Dans ce papier, nous proposons la méthode HD_DBT (pour résolution des CSP par un algorithme de type BT sur les tuples guidé par un ordre statique induit par une Hypertree Décomposition) comme une nouvelle approche de résolution des CSPs préalablement décomposés en hypertree généralisée. Nous avons implémenté et expérimenté cette approche. Les résultats de comparaison par rapport à la méthode de résolution des CSPs par hypertree decomposition, connue dans la littérature sont encourageants.

1 Introduction

Les problèmes de satisfaction de contraintes (CSPs) sont au coeur de problèmes fondamentaux de l'intelligence artificielle car le formalisme CSP est très puissant et permet de représenter de nombreux problèmes du monde réel. Un CSP est défini comme un ensemble de contraintes impliquant un ensemble de variables. L'objectif est de trouver une instanciation possible pour chaque variable qui satisfait toutes les contraintes. Le formalisme CSP a été introduit par Montanari en 1974 [15]. Depuis, de nombreuses méthodes ont été proposées pour résoudre les CSPs. La méthode naïve de résolution de CSP est le backtracking dont la complexité théorique est exponentielle en $O(md^n)$ où n est le nombre de variables du problème, m est le nombre de contraintes et d est la taille maximale des domaines des variables. Pour réduire cette complexité, beaucoup de travaux ont été menés en exploitant certaines propriétés des instances et plusieurs approches ont été proposées. Parmi ces méthodes, il y a les méthodes de décomposition structurelles. Les méthodes de décomposition structurelles ont toutes pour but de transformer un CSP en un CSP équivalent (ayant les mêmes solutions) en formant des clusters de variables ou de contraintes dont l'interaction a une structure d'arbre. Chaque méthode de décomposition définit son propre concept de largeur : elle peut être interprétée comme une mesure de cyclicité du graphe ou de l'hypergraphe de contraintes. Parmi ces méthodes, on retient principalement la méthode coupe cycle [4], la méthode hypercutset [8], la méthode BICOMP [7], La méthode de tree-clustering [5]. Dans ce papier nous nous intéressons à la méthode la plus générale connue sous le nom d'hypertree-decomposition généralisée [8]. (On ne considère pas la méthode : fractionnal hypertree decomposition due à Grohe et al [12] dans cette comparaison). L'hypertree decomposition généralisée permet d'obtenir des décompositions de meilleures largeurs. Une méthode de décomposition en hypertree généralisée a pour but de transformer l'hypergraphe associé à un CSP en une structure hyper-arborescente dont les noeuds sont des clusters d'hyper-arêtes et des clusters de variables soumis à certaines contraintes dont la résolution se fait en un temps polynomial en la largeur de cette hyper-arborescence (Hypertreewidth). Cependant, la méthode proposée pour la résolution du CSP préalablement décomposé en hypertree décomposition n'est

pas efficace voir inopérante en pratique. Le problème de cette approche vient du fait que la méthode de résolution résout d'abord les sous problèmes au noeuds avant de résoudre le problème tout entier. Pour cela il faut rechercher au niveau des noeuds toutes les solutions et les sauvegarder puis résoudre le problème tout entier par des opérations de semi-jointure. Cette méthode entraîne inévitablement un problème d'explosion mémoire et se trouve limitée à la résolution de problèmes de petite taille. Pour pallier à ce problème, dans ce papier, nous allons présenter une nouvelle méthode de résolution des CSPs dite HD_DBT (pour résolution des CSP par un algorithme de type BT sur les tuples guidé par un ordre statique induit par une Hypertree Décomposition) qui parcourt l'hypertree decomposition de la racine aux feuilles en recherchant au niveau de chaque noeud un seul tuple solution qui soit compatible avec les tuples déjà calculés dans les noeuds précédents et aussi compatible avec l'ensemble des contraintes de ses noeuds fils. Si aucun tuple n'est possible, l'algorithme effectue un backtrack chronologique sur le noeud précédent dans l'hypertree pour calculer un autre tuple et ainsi de suite jusqu'à ce qu'une solution soit trouvée ou alors remonter jusqu'à la racine, auquel cas le CSP n'a pas de solution si tous les tuples possibles sont explorés. Nous allons présenter dans ce papier cette approche et allons montrer par nos premières expérimentations son intérêt.

Ce papier est organisé comme suit : dans la section 2 nous rappelons la définition formelle d'un CSP, les différentes notions de décompositions structurelles et plus particulièrement la méthode qui généralise toutes les autres méthodes : l'hypertree decomposition généralisée (on ne tient pas compte de la méthode fractional hypertree decomposition de Grohe et al [12]). Dans la section 3, nous rappelons l'approche proposée par Gottlob et al [10] pour la résolution du CSP retourné par la décomposition en hypertree généralisée. Nous présenterons les détails de notre méthode de résolution HD_DBT dans la section 4. Les résultats expérimentaux sont présentés dans la section 5 et enfin la section 6 sera consacrée à la conclusion et aux perspectives.

2 Préliminaires

La notion de problèmes de satisfaction de contraintes (CSPs pour Constraint Satisfaction Problems) a été introduite dans [15] selon la définition 1.

Définition 1 . *Un Problème de Satisfaction de Contraintes est un quadruplet* $P = (X, D, C, R)$*, où* $X = \{x_1, x_2, ..., x_n\}$ *est un ensemble de n variables,* $D = \{d_1, d_2, ..., d_n\}$ *est un ensemble de n domaines finis. Chaque domaine* d_i *est associé à une variable* x_i*. C est un ensemble de m contraintes. Chaque contrainte* C_i *est définie par un ensemble de r variables* $X_i = \{x_{i1}, x_{i2}, ..., x_{ir}\}$*.* $R = \{R_1, R_2, ..., R_m\}$ *est un ensemble de m relations. Chaque relation* R_i *définit l'ensemble des tuples autorisés par la contrainte* C_i*.*

Définition 2 *: Hypergraphe de contraintes*

Un hypergraphe de contraintes est un couple $H =< V, E >$ *où V est l'ensemble des sommets et E est l'ensemble des hyperarêtes. Chaque sommet correspond à une variable du CSP et chaque hyperarête correspond à une contrainte. On note souvent V par var(H) et E par hyperedges(H) ou hyperarêtes(H). Aussi,dans ce papier, si h est une hyperarête,* $var(h)$ *indique l'ensemble des variables de h ; si S est un ensemble d'hyperaretes, alors* $var(S)$ *correspond à l'ensemble des variables apparaissant dans les hyperarêtes de S.*

Un *hypertree* pour un hypergraphe H est un triplet $< T, \chi, \lambda >$ où $< T = (N, E) >$ est un arbre enraciné et χ et λ sont deux fonctions d'étiquetage associant à chaque sommet (noeud) p de N un ensemble de variables χ(p) et un ensemble de contraintes λ(p). On note aussi *sommets(T)* l'ensemble des sommets de T et on note la racine de T par *root(T)*. T_p indique l'ensemble des variables du sous arbre qui a pour racine le noeud p.

Proposition 1. [3] : La classe des CSP dont le graphe de contraintes est acyclique est traitable, et ceci indépendamment des relations des contraintes.

Définition 3 *(* décomposition en hypertree généralisée *[10])*

Une décomposition hypertree généralisée d'un hypergraphe $\mathcal{H} =< V, E >$ *est un hypertree* $HD =< T, \chi, \lambda >$ *qui satisfait les conditions suivantes :*

1. *Pour chaque hyperarete* $h \in E$*, il existe* $p \in sommet(T)$ *telle que* $var(h) \subseteq \chi(p)$*. On dit que p couvre h.*
2. *Pour chaque variable* $v \in V$*, l'ensemble* $\{p \in sommets(T) | v \in \chi(p)\}$ *induit un sous arbre connexe de T.*
3. *Pour chaque sommet* $p \in sommets(T)$*,* $\chi(p) \subseteq var(\lambda(p))$

La figure 1 montre l'exemple d'un hypergraphe et son hypertree décomposition généralisée.

L'hypertree decomposition d'un hypergraphe $\mathcal{H} =< V, E >$, est une hypertree decomposition généralisée $HD =< T, \chi, \lambda >$ qui satisfait la condition

supplémentaire suivante :
Pour chaque sommet $p \in sommets(T)$, $var(\lambda(p)) \bigcap \chi(T_p) \subseteq \chi(p)$.

La largeur d'une hypertree décomposition généralisée $< T, \chi, \lambda >$ est $max_{p \in vertices(T)} |\lambda(p)|$. La largeur d'une décomposition en hypertree (généralisée) $(g)hw(\mathcal{H})$ d'un hypergraphe $\mathcal{H}$ est la largeur minimum de toutes ses décompositions en hypertree (généralisée).

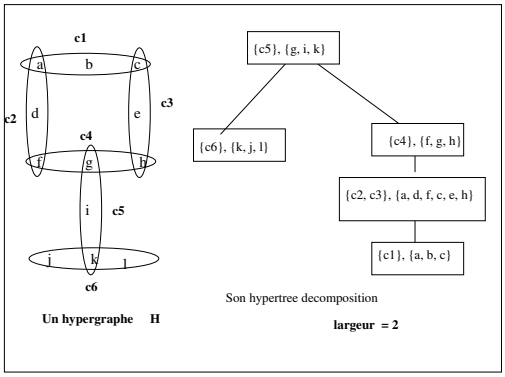


FIG. 1 – Un hypergraphe H et son hypertree decomposition généralisée

Définition 4 *Une hyperarete h est fortement couverte dans un hypertree s'il existe un noeud p tel que $var(h) \subseteq \chi(p)$ et $h \in \lambda(p)$.*

Définition 5 *Une hypertree décomposition $< T, \chi, \lambda >$ d'un hypergraphe $\mathcal{H} =< V, E >$ est **complète** si chaque hyperarête h de $\mathcal{H} =< V, E >$ est fortement couverte dans $HD =< T, \chi, \lambda >$.*

Calculer une hypertree decomposition généralisée optimale est un problème NP difficile [11]. Pour cette dernière, il existe deux approches de calcul : les méthodes exactes et les heuristiques. Parmi les méthodes exactes, nous citons opt-k-decomp [9] qui permet de calculer une décomposition de largeur optimale bornée par une constante k fixée si une telle décomposition existe. Cependant, ces algorithmes dits exacts ne sont efficaces que pour les problèmes de petites tailles. Pour cela, on a vu ces dernières années le développement des heuristiques pour le calcul de cette décomposition. Dans [6], les auteurs ont proposé les heuristiques BE et DBE et dans [16], Musliu et Schafhauser ont exploré les algorithmes génétiques pour calculer une hypertree decomposition généralisée. Dans [2] nous avons proposé des heuristiques comparables à BE.

3 Résolution des CSPs par la méthode hypertree généralisée

Cette approche nécessite d'abord le calcul d'une décomposition par un algorithme exact ou une heuristique. (En pratique, les algorithmes exacts ne sont pas efficaces). Une fois que la décomposition est obtenue, on la complète de telle sorte que toutes les contraintes figurent dans au moins un noeud de l'hypertree.

Ensuite l'approche proposée par Gottlob et al [10] pour résoudre le CSP obtenu est donnée par l'algorithme 1.

Input: Une hypertree decomposition $\mathcal{HD} =< T, \chi, \lambda >$ associée à un CSP donné.
Output: Une solution $\mathcal{A}$
begin
  $\sigma = \{n_1, n_2, \ldots, n_m\}$ un ordre sur les noeuds de l'hypertree decomposition où n_1 est la racine et chaque noeud précède ses fils dans l'ordre. ;
  foreach *p nœud de l'hypertree* **do**
    $R_p = join(\lambda(p)[\chi(p)]$;
  end
  for *i = m to 2* **do**
    begin
      Soit v_j le père de v_i dans l'ordre ;
      $R_j = semijoin(R_j, R_i)$;
    end
  end
  for *i = 2 to m* **do**
    Construire une solution $\mathcal{A}$ en sélectionnant un tuple dans R_i compatible avec tous ceux qui le précèdent.
  end
  return $\mathcal{A}$;
end

Algorithm 1: Méthode de résolution proposée par Gottlob

En pratique, cette méthode est très coûteuse aussi bien en temps qu'en espace mémoire. En effet, cette approche est basée sur deux principales opérations qui sont des jointures au niveau des noeuds de l'hypertree et des semi-jointures entre les différents noeuds de l'hypertree. Malgré les différentes heuristiques introduites précédemment dans le rapport de recherche [1], cette approche souffre toujours du problème de l'espace mémoire.

Pour remédier à ce problème d'explosion mémoire et exploiter les atouts des décompositions arborescentes, P. Jégou et C. Terrioux ont proposé dans [14] une

technique intéressante nommée BTD (Résolution d'un CSP par une méthode de type BT guidée par un ordre statique induit par une tree decomposition). BTD hérite à la fois des avantages des méthodes énumératives pour ce qui concerne l'occupation mémoire et des propriétés structurelles du CSP. Dans ce travail, nous allons proposer une autre approche de résolution qui sera la combinaison entre l'approche de base de Gottlob et l'approche de résolution de type retour arrière pour le choix d'un tuple pour un noeud donné. Cette approche est appelée HD_DBT (Hypertree Decomposition versus Dual BackTracking).

4 La méthode HD_DBT

4.1 Description générale

Nous introduisons dans cette section notre approche nommée HD_DBT. HD_DBT consiste à résoudre un CSP donné en passant par la construction de l'hypertree décomposition généralisée associée à son hypergraphe de contraintes. La procédure HD_DBT décrite par l'algorithme2 considère en entrée une hypertree décomposition généralisée et complètée conformément à la définition 5 de l'hypertree décomposition complète et retourne une solution si elle existe.

L'hypertree décomposition est complétée pour s'assurer que pour chaque contrainte c du CSP, il existe un noeud $n =< \lambda_n, \chi_n >$ de l'hypertree $HD =< T, \chi, \lambda >$ tel que les variables de la contrainte c sont contenues dans χ_n et $c \in \lambda_n$. Si ce noeud n'existe pas, on cherche un noeud n' de l'hypertree tel que χ couvre les variables de c puis on crée un noeud n'' fils de n' dont l'ensemble χ est l'ensemble des variables de c et dont le terme λ contient uniquement la contrainte c. Le noeud n' existe forcément parce que c'est l'une des conditions de l'hypertree decomposition généralisée.

La complexité théorique de cet algorithme est en $O(|r|^{w \times Nb\_noeuds})$ où $|r|$ est la taille de la plus grande relation, w est l'hypertree-width et Nb_Noeuds est le nombre de noeuds de l'hypertree decomposition.

Pour résoudre le problème représenté sous forme d'hypertree décomposition nous proposons un algorithme de type backtrack. A la différence des algorithmes énumératifs classiques celui-ci instancie un ensemble de variables en une seule étape plutôt que d'instancier variable par variable. Le problème crucial dans la méthode de base de Gottlob est le coût des jointures effectuées aux différents noeuds de l'hypertree décomposition, aussi bien en espace qu'en temps d'exécution. C'est pour celà que nous proposons une approche qui ne calcule pas toutes les solutions au niveau d'un

```
Input: Une hypertree decomposition
       HD =< T, χ, λ > associée à un CSP
       donné.
Output: Une solution A
begin
    CHOIX_RACINE ( HD , racine ) ;
    σ ⟵ ORDRE_INDUIT ( HD , racine ) ;
    nc ⟵ racine ;
    while nc ≠ ∅ do
        begin
            consistant ⟵ FALSE ;
            while ¬ consistant do
                begin
                    sol_nc ⟵ resolution (
                    λ(nc),χ (nc) ) ;
                    if Compatible(sol_nc,
                    sol(pere(nc)) then
                        begin
                            A ⟵ A ∪ {x_i ←
                            v_i ∀x_i ∈ χ(i)} ;
                            consistant ⟵ TRUE ;
                        end
                    end
                end
            end
            if ¬ Consistant then
                begin
                    A ⟵ A − {x_i ← v_i ∀x_i ∈ χ(i)}
                    ;
                    nc ⟵pere(nc) ;
                end
            end
            else
                nc ⟵ succ(nc) ;
            end
        end
    end
    return A ;
end
```

Algorithm 2: Procédure générale HD_DBT

noeud mais ne calcule qu'une seule solution. Si cette solution est consistante avec celle des noeuds déjà résolus on continue sinon on effectue un retour arrière chronologique.

La procédure principale *HD_DBT* considère l'hypertree décomposition obtenue précédemment en entrée et se compose des différentes étapes décrites par l'algorithme 2.

4.2 Description détaillée

Les différentes étapes de *HD_DBT* sont les suivantes :

Etape 1 : La première étape réalisée par la procédure générique **CHOIX_RACINE** est le choix du premier noeud de l'hypertree decomposition à résoudre qu'on appelle racine. Plusieurs heuristiques pour déterminer ce noeud racine sont possibles. Elles peuvent être structurelles ou sémantiques. Nous présenterons certaines heuristiques plus loin.

Etape 2 : La deuxième étape est réalisée par la procédure **ORDRE_INDUIT** qui construit un ordre σ sur les noeuds de l'hypertree décomposition. Cet ordre est obtenu à partir de l'hypertree et de la racine choisie par la procédure **ORDRE_INDUIT** et il découle directement de la propriété de connectivité de l'hypertree decompoisition.

Pour illustrer cette notion d'ordre induit, nous présentons dans la figure 2 deux ordres différents pour l'hypertree decomposition complète associée à l'hypergraphe donnée par la figure 1.

Exemple 1

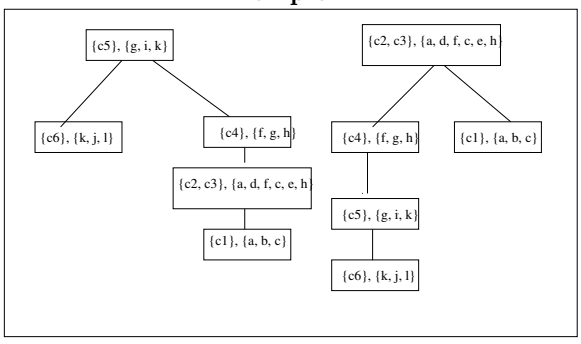


FIG. 2 – Deux ordres différents pour une hypertree décomposition

Etape 3 : Cette étape est la recherche de la solution du CSP par un algorithme énumératif dirigé par l'ordre σ. Pour chaque noeud $nc = (\lambda(nc), \chi(nc))$ de l'ordre σ, *HD_DBT* recherche une solution par la procédure *resolution* . Cette procédure est réalisée par les opérations de jointure et de projection pour trouver une solution (un tuple de jointure). L'algorithme 2 non présenté dans ce papier, pour des raisons de simplicité, est l'algorithme naïf de type BT. Mais cette approche peut se généraliser à tous les autres algorithmes énumératifs tels que FC, Mac, ... Notons toutefois que la connectivité dans un hypertree est liée aux variables et non pas aux contraintes.

4.3 Heuristiques pour le calcul du noeud racine :

Pour améliorer notre approche, nous avons introduit les heuristiques suivantes :

Un noeud dans une hypertree décomposition est composée d'un couple $< \chi, \lambda >$. Pour choisir le noeud racine, il est possible de définir des heuristiques en tenant compte de la cardinalité de l'ensemble λ, de celle de l'ensemble χ ou éventuellement des deux à la fois. On distingue deux catégories d'heuristiques : les heursitiques structurelles et les heuristiques sémantiques.

- **Les heuristiques structurelles** : Elle dépendent uniquement des propriétés de la structures de l'hypertree decomposition (taille des clusters, nombre de fils, ...).
  Nous nous proposons à ce titre d'étudier les heuristiques suivantes :

  PGC (Plus Grand Cluster) : La racine correspond au noeud dont la cardinalité de λ est la plus grande

  PPC (Plus Petit Cluster) : La racine correspond au noeud dont la cardinalité de λ est la plus petite

  PNF (Plus Grand nombre de Fils) : La racine correspond au noeud qui possède le plus grand nombre de fils.

  PGS (Plus Grand Séparateur) : La racine correspond au noeud qui partage avec ses fils le plus grand séparateur par rapport aux variables .

- **Les heuristiques sémantiques** : Celles ci dépendent de la nature des données (taille des relations, densité des noeuds, dureté des contraintes, ...).

Nous nous proposons aussi d'étudier les heuristiques suivantes.

NMC (Noeud le moins contraint) : La racine correspond au noeud qui contient le plus grand nombre de solutions attendues. Le nombre de solution attendu est estimé avant l'exécution.

NPC (Noeud le plus contraint) : La racine correspond au noeud qui contient le plus petit nombre de solutions attendues.

NPD (Noeud le plus Dur) : La racine correspond au noeud dont la propriété de dureté est la plus forte. La dureté d'un noeud est égale à $\frac{Nbsol}{NbMax}$ où Nbsol est le nombre de solutions attendu du noeud et NbMax le nombre de tuples maximal (taille du produit cartésien de toutes les relations contenues dans le cluster.

5 Expérimentation et analyse des résultats

Dans cette section nous allons étudier le comportement de notre approche du point de vue expérimental. Les expérimentations ont été menées sur un PC portable HP Compact 6720s, 1,7 GHZ et 2 GO de RAM, sous LINUX Fedora. Les benchmarks sont téléchargés du site http ://www.cril.univ-artois.fr/lecoutre/research/benchmarks.

Pour le calcul des décompositions en hypertree, nous avons exploité l'heuristique BE [6]. Cette heuristique est connue pour donner des décompositions de largeurs proches des largeurs optimales et ce en des temps très courts.

5.1 Comparaison de notre approche à l'approche de Gottlob

Pour montrer l'intérêt de notre approche nous avons commencé par observer son comportement par rapport au comportement de l'algorithme de base proposé par Gottlob et al.

Dans le tableau 1, nous présentons les résultats de cette comparaison en terme de temps d'exécution. Les résultats obtenus par notre approche en temps d'exécution sont meilleurs pour 11 benchmarks sur 12.

Instance	Taille		Temps(s)	
	$\|V\|$	$\|E\|$	HD_DBT	Gottlob
$renault$	101	134	2	3
$series-6-ext$	11	30	0,04	2,18
$series-7-ext$	12	41	0,1	/
$domino-100-100_ext$	100	100	0,12	2,59
$domino-100-200_ext$	100	100	0,30	18,37
$domino-100-300_ext$	100	100	0,4211	60
$hanoi-5-ext$	30	29	0,55	0,88
$hanoi-6-ext$	62	61	120	14
$hanoi-7-ext$	126	124	58	59
$Langford$	8	32	0,20	2,52
$geom-30a-4-ext$	30	81	0,1	0,1
$pigeons-7-ext$	7	21	2	26

TAB. 1 – Comparaison des temps de résolution des deux approches HD_DBT et la méthode de Gottlob.

5.2 Comportement des différentes heuristiques de choix de racine

Dans cette section nous observons l'impact du choix de la racine sur la résolution ainsi que le gain apporté par le filtrage des relations.

Les tableaux 2 et 3 récapitulent les résultats expérimentaux obtenus pour 10 benchmarks en utilisant une implémentation naïve de l'algorithme HD_DBT, sans filtrage et les tableaux 4 et 5 présentent les résultats pour une implémentation avec filtrage des relations . Premièremons nous constatons que sans filtrage cette approche est totalement inefficace. Trois problèmes sur 10 n'ont pu être résolus aprés 20 secondes.

5.2.1 Algorithme naïf et étude des différents ordres

Dans le tableaux 2, nous reportons les résultats obtenus avec les heuristiques portant sur les données (ou les heuristiques sur la sémantique). Il s'agit des heuristiques NPC (Noeud le Plus Contraint), NMC (Noeud) le Moins Contraint) et NPD (Noeud le Plus Dur). Ici les résultats ne sont pas significatifs.

Instance	Taille		Ordre		
	$\|V\|$	$\|E\|$	NPC	NMC	NPD
$renault$	101	134	3,28	3,1 4	3,18
$series-6-ext$	11	30	2,24	1,88	1,59
$series-7-ext$	12	41	2,02	1,5	2,04
$domino-100-100_ext$	100	100	0,21	0,49	0,53
$domino-100-200_ext$	100	100	5,97	7,02	5,65
$domino-100-300_ext$	100	100	12,48	13,01	12,67
$langford-2-4-ext$	8	32	2,83	1,6	0,93
$geom-30a-4-ext$	30	81	> 20	> 20	> 20
$pigeons-7-ext$	7	21	> 20	> 20	> 20
$haystacks-06_ext$	36	96	> 20	> 20	> 20

TAB. 2 – Temps d'exécution pour les différents ordre sans filtrage.

Dans le tableau 3, nous reportons les résultats sur les heuristiques structurelles même si l'heuristique PGC (Plus Grand Nombre de Contraintes) semble être la meilleure.

Instance	Ordre				
	BE	PGC	PPC	PNF	PGS
$renault$	3,40	3,40	3,09	3,06	3,12
$series-6-ext$	1,54	1,55	1,88	1,50	1,55
$series-7-ext$	2,02	2,08	1,73	2,04	1,99
$domino-100-100_ext$	0,20	0,25	0,58	0,5	0,30
$domino-100-200_ext$	5,90	5,85	6,51	5,5	6,5
$domino-100-300_ext$	12,77	12,68	12,57	12,39	12,80
$langford-2-4-ext$	0,54	0,68	1,5	0,50	0,99
$geom-30a-4-ext$	> 20	3,09	> 20	1,7	> 20
$pigeons-7-ext$	> 20	> 20	> 20	> 20	> 20
$haystacks-06_ext$	> 20	> 20	> 20	> 20	> 20

TAB. 3 – Temps d'exécution pour les différents ordre sans filtrage.

5.2.2 Algorithme avec filtrage et études des différents ordres

Pour améliorer l'algorithme de base, nous avons introduit le filtrage des relations de tous les noeuds fils de chaque noeud instancié. Les tableaux 4 et 5 montrent le comportement des différentes heuristiques du choix de la racine et l'apport du filtrage. Le filtrage améliore clairement les résultats. Nous observons en effet qu'ici toutes les instances sont résolues.

Le tableau 4 montre que parmi les heuristiques portant sur la sémantique, l'heuristique NPD (Noeud le Plus Dur) est la meilleure. Ceci n'est pas surprenant car elle tient compte de la dureté des contraintes mais aussi de la taille des relations, c'est à dire de l'espace possible des solutions.

Instance	Taille		Ordre		
	$\|V\|$	$\|E\|$	NPC	NMC	NPD
$renault$	101	134	3,03	3,02	3,04
$series-6-ext$	11	30	0,10	0,38	0,43
$series-7-ext$	12	41	0,55	3,48	0,13
$domino-100-100_ext$	100	100	0,12	0,70	0,49
$domino-100-200_ext$	100	100	0,23	2,32	0,76
$domino-100-300_ext$	100	100	0,36	5,14	0,75
$Langford$	8	32	0,14	0,36	0,09
$geom-30a-4-ext$	30	81	7	1,23	0,03
$pigeons-7-ext$	7	21	11	4,23	4,2
$haystacks-06_ext$	36	96	6,96	3,32	3,31

TAB. 4 – Temps d'exécution pour les différents ordre en exploitant le filtrage.

Le tableau 5 montre que parmi les heuristiques statiques, l'heuristique PGC (Plus Grand Cluster) est la meilleure. Ceci signifie qu'il serait intéressant de traiter à la racine le noeud qui contient le maximun de contraintes.

Remarque 1 *Nous avons considéré les heuristiques structurelles et sémantiques séparément. Mais il faudra sans doute les combiner. Les décompositions structurelles se contentent de décomposer les problèmes en exploitant uniquement les propriétés structurelles de leurs représentations en graphes ou hypergraphes. Mais la résolution du problème fait intervenir les relations associées aux contraintes. Cette donnée ajoute une dimension non négligeable à la complexité du problème du point de vue expérimental.*

Instance	Ordre				
	BE	PGC	PPC	PNF	PGS
$renault$	3,38	3,02	4,41	3,03	3,31
$series-6-ext$	0,09	0,07	0,43	0,07	0,12
$series-7-ext$	0,08	3,48	3,18	3,5	0,96
$domino-100-100_ext$	0,125	0,14	0,49	0,13	0,22
$domino-100-200_ext$	0,24	0,23	0,27	0,23	0,28
$domino-100-300_ext$	0,35	0,36	0,69	0,34	0,51
$Langford$	0,03	0,31	0,91	8,73	1,02
$geom-30a-4-ext$	0,03	0,03	5	0,23	1,02
$pigeons-7-ext$	4,34	4,19	3,5	12	4,23
$haystacks-06_ext$	3,33	3,35	3,33	3,22	3,21

TAB. 5 – Temps d'exécution pour les différents ordre en exploitant le filtrage.

Il faut des heuristiques de choix du noeud racine qui tiennent compte du nombre de tuples, du nombre de tuples solutions, de la taille des séparateurs, ...etc.

Pour situer notre approche par rapport à la méthode BTD, nous avons testé les problèmes de la famille renault modofiés traités dans le papier de Philippe Jégou et al [13]. Nous n'avons malheureusement pas pu obtenir de solutions pour cette famille de problèmes pour l'instant. Notre approche est à améliorer probablement par l'exploration des goods et des nogoods, comme nous le soulignons dans la conclusion.

6 Conclusion

Pour remedier à l'inconvénient de la méthode de résolution des CSP par l'hypertree decomposition, nous avons proposé dans ce papier une autre approche basée sur un backtracking sur les tuples. Sachant que le choix du noeud à instancier en priorité est primordial pour le comportement de la méthode, nous avons exploré plusieurs stratégies pour le choix de la racine à instancier en priorité. Les résultats expérimentaux montrent effectivement que le choix judicieux consiste à choisir le noeud le plus dur, c'est à dire le noeud qui minimise le rapport entre le nombre de solutions attendues au niveau du noeud et le nombre de solutions dans le pire des cas. Pour estimer l'intérêt pratique de notre approche, nous avons comparé notre méthode à celle de Gottlob et nos résultats sont encourageants.

Nous avons aussi cherché à comparer notre approche à BTD. Sur le plan théorique ces deux approches sont bien sûr comparables dans le sens où toutes les deux recherchent la solution d'un CSP par un algorithme énumératif guidé par un ordre induit par une décomposition structurelle. BTD explore les décompositions arborescentes alors que notre approche explore les décompositions hyper-arborescentes. BTD a été étendue par ailleurs aux décompositions hyper-arborescentes. la spécifité de notre approche repose sur le fait que les clusters correspondent à un ensemble de contraintes et non pas un ensemble de variables. Du point de vue

expérimental, pour comparer notre approche à BTD, nous avons testé la famille des Benchmarks Renault modifés. Les temps d'exécution ne sont pas aussi bons que ceux obtenus par BTD. Notre approche est à améliorer. Nous allons explorer les notions de goods et de nogoods pour accelérer la recherche de solution au niveau d'un noeud donné. Une de nos perspectives serait aussi d'étudier des heuristiques pour construire des hypertree décompositions basées sur les propriétés structurelles et sémantiques des clusters.

Par ailleurs, la complexité théorique de HD_DBT dépendant du nombre de noeuds de l'hypertree décomposition, il faudra trouver des heuristiques minimisant le nombre de noeuds de l'hypertree décomposition et ceci se fera sans doute au détriment de la largeur de l'hypertree decomposition. Ce qui est en contradiction avec les résultats théoriques sur l'hypertree décomposition.

Références

[1] K. Amroun and Z. Habbas. Résolution des problèmes de satisfaction de contraintes par les techniques de décomposition structurelles. Rapport de recherche, Laboratoire de Recheche en Informatique Théorique et Appliquée de Metz, 2009.

[2] M. Aït-Amokhtar, K Amroun, and Z. Habbas. hypertree decomposition for solving constraint satisfaction problems. In *Proceedings of International conference on Agents and Artificial Intelligence,ICAART 2009*, pages 85–92, Portugal, 2009.

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

[4] R. Dechter and J. Pearl. The cycle-cutset method for improving search performance in AI applications. In *Proceedings of the third IEEE on Artificial Intelligence Applications*, pages 224–230, Orlando, 1987.

[5] R. Dechter and J. Pearl. Tree-clustering schemes for constraint-processing. In *Proceedings of the sixth National Conference on Artificial Intelligence (AAAI-88)*, pages 150–154, Saint Paul, MN, 1988.

[6] T. Dermaku, A.and Ganzow, G. Gottlob, B. McMahan, N. Musliu, and M. Samer. Heuristic methods for hypertree decompositions. Technical report, DBAI-R, 2005.

[7] E. C. Freuder. A sufficient condition for backtrack-free search. *Journal of the Association for Computing Machinery*, 29 :24–32, 1982.

[8] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural csp decomposition methods. In *Proceedings of IJCAI'99*, pages 394–399, 1999.

[9] G. Gottlob, N. Leone, and F. Scarcello. On tractable queries and constraints. In *Proceedings of DEXA'99*, 1999.

[10] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions : A survey. In *Proceedings of MFCS '01*, pages 37–57, 2001.

[11] Georg Gottlob, Zoltan Miklos, and Thomas Schwentick. Generalized hypertree decomposition : Np - hardness and tractable variants. In *Proceedings of the 26 th ACM SIGMOD SICACT SIGART Symposium on principles of databases systems*, 2007.

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

[13] P. Jégou, S. Ndiaye, and C. Terrioux. C. stratégies hybrides pour des décompositions optimales et efficaces. In *Actes JFPC 2009*, 2009.

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

[15] U. Montanari. Networks of constraints : Fundamental properties and applications to pictures processing. *Information Sciences*, 7 :95–132, 1974.

[16] Musliu.N and W. .Schafhauser. Genetic algorithms for generalized hypertree decompositions. *European Journal of Industrial Engineering*, 1(3) :317–340, 2005.

Une nouvelle extension de fonctions aux intervalles basée sur le regroupement d'occurrences

Ignacio Araya[1], **Bertrand Neveu**[2], **Gilles Trombettoni**[1]
[1] COPRIN, INRIA Sophia Antipolis, Université Nice Sophia
[2] Imagine, LIGM, Université Paris-Est, France
rilianx@gmail.com {neveu,trombe}@sophia.inria.fr

Résumé

Quand une fonction f est monotone par rapport à une variable sur un domaine donné, il est bien connu que l'*extension aux intervalles par monotonie* de f calcule une image plus étroite que l'*extension naturelle*.

Cet article présente une nouvelle extension aux intervalles d'une fonction f appelée *regroupement d'occurrences* et notée $[f]_{og}$. Quand f n'est pas monotone par rapport à une variable x sur un domaine donné $[B]$, nous essayons de transformer f en une nouvelle fonction f^{og} qui est monotone sur deux variables x_a et x_b, qui regroupent des occurrences de x de telle sorte que f^{og} soit croissante par rapport à x_a et décroissante par rapport à x_b. $[f]_{og}$ est l'extension aux intervalles par monotonie de f^{og} et produit une image plus étroite que l'extension naturelle.

Pour trouver un bon regroupement d'occurrences, nous proposons un programme linéaire et un algorithme qui minimisent une surestimation du diamètre de l'image de $[f]_{og}$ basée sur une forme de Taylor de f. Finalement, des expérimentations montrent les avantages de cette nouvelle extension lors de la résolution de systèmes d'équations.

Abstract

When a function f is monotonic w.r.t. a variable in a given domain, it is well-known that the monotonicity-based interval extension of f computes a sharper image than the natural interval extension does.

This paper presents a so-called "occurrence grouping" interval extension $[f]_{og}$ of a function f. When f is *not* monotonic w.r.t. a variable x in the given domain $[B]$, we try to transform f into a new function f^{og} that is monotonic in two subsets x_a and x_b of the occurrences of x. f^{og} is increasing w.r.t. x_a and decreasing w.r.t. x_b. $[f]_{og}$ is the interval extension by monotonicity of f^{og} and produces a sharper interval image than the natural extension does.

For finding a good occurrence grouping, we propose a linear program and an algorithm that minimize a Taylor-based overestimation of the image diameter of $[f]_{og}$. Finally, experiments show the benefits of this new interval extension for solving systems of equations.

1 Introduction

Le calcul de l'image optimale d'une fonction sur des intervalles est un problème difficile qui est au cœur de l'arithmétique d'intervalles. Il permet d'évaluer une formule mathématique en tenant compte de manière fiable des erreurs d'arrondis dus à l'arithmétique en virgule flottante. Des encadrements étroits permettent aussi aux méthodes par intervalles de converger rapidement vers les solutions d'un système de contraintes sur les réels. A chaque nœud de l'arbre de recherche, un *test d'existence* vérifie que, pour chaque équation $f(X) = 0$, l'extension aux intervalles de f retourne un intervalle incluant 0 (sinon la branche est coupée). Les algorithmes de propagation de contraintes peuvent aussi être améliorés s'ils utilisent de meilleures extensions aux intervalles. Par exemple, l'algorithme `Box` utilise un test d'existence à l'intérieur de son processus itératif de bissection/rognage [3].

Cet article propose une nouvelle extension aux intervalles et nous rappelons d'abord quelques notions de base de l'arithmétique d'intervalles [10, 11, 8] pour introduire les extensions aux intervalles utiles à nos travaux.

Un intervalle $[x] = [a, b]$ est l'ensemble des nombres réels compris entre a et b. $\underline{x} = a$ étant le minimum de $[x]$ et $\overline{x} = b$ le maximum. Le *diamètre* ou taille d'un intervalle est : $diam([x]) = \overline{x} - \underline{x}$, et la valeur absolue d'un intervalle est : $|[x]| = max(|\overline{x}|, |\underline{x}|)$. Un produit cartésien d'intervalles, appelé *boîte*, est noté $[B]$ ou par un vecteur d'intervalles $\{[x_1], [x_2], ..., [x_n]\}$.

Une *fonction sur intervalles* $[f]$ est une fonction de $\mathbb{IR}^n$ dans $\mathbb{IR}$, $\mathbb{IR}$ étant l'ensemble de tous les inter-

valles sur $\mathbb{R}$. $[f]$ est une *extension aux intervalles* d'une fonction $f(x_1, ..., x_n)$ de $\mathbb{R}^n$ dans $\mathbb{R}$ si l'image d'une boîte $[B] = \{[x_1], [x_2], ..., [x_n]\}$ par $[f]$ est un intervalle *conservatif*, c.-à-d. contient l'ensemble $\mathcal{I}f([B]) = \{y \in \mathbb{R}, \exists\{x_1, x_2, ..., x_n\} \in [B], y = f(x_1, x_2, ..., x_n)\}$. (Le calcul de l'image est appelé *évaluation* de f.)

L'image *optimale* $[f]_{opt}([B])$ est le plus petit intervalle contenant $\mathcal{I}f([B])$. Il existe de nombreuses extensions aux intervalles pour une fonction, la difficulté étant de définir une extension qui calcule l'image optimale, ou une approximation étroite de cette dernière.

La première idée est d'utiliser l'arithmétique d'intervalles, qui étend aux intervalles les opérateurs arithmétiques $+$, $-$, $\times$, $/$ et les fonctions élémentaires (*power*, *exp*, *log*, *sin*, *cos*, ...). Par exemple, $[a, b] + [c, d] = [a + c, b + d]$. L'*extension naturelle* $[f]_N$ d'une fonction f évalue avec l'arithmétique d'intervalles tous les opérateurs arithmétiques et toutes les fonctions élémentaires de f.

Si f est continue dans une boîte $[B]$, l'*évaluation naturelle* de f (c.-à-d. le calcul de $[f]_N([B])$) produit l'image optimale si chaque variable n'apparaît qu'une fois dans f. Quand une variable apparaît plusieurs fois, l'évaluation par l'arithmétique d'intervalles produit généralement une surestimation de $[f]_{opt}([B])$, car la corrélation entre les occurrences d'une même variable est perdue : deux occurrences d'une variable sont traitées comme des variables indépendantes. Par exemple $[x] - [x]$, avec $[x] \in [0, 1]$ donne le résultat $[-1, 1]$, au lieu de $[0, 0]$, comme le ferait $[x] - [y]$, avec $[x] \in [0, 1]$ et $[y] \in [0, 1]$.

Ce principal inconvénient de l'arithmétique d'intervalles cause une réelle difficulté pour implanter des résolveurs sur intervalles efficaces, puisque l'évaluation naturelle est un outil de base pour ces résolveurs.

Plusieurs extensions aux intervalles plus sophistiquées ont été proposées pour surmonter cette difficulté. L'*extension de Taylor* $[f]_T$ de f, définie dans [10], utilise la forme de Taylor de f :

$$[f]_T([B]) = f(B_m) + \sum_{i=1}^{n} \left(\left[\frac{\partial f}{\partial x_i} \right] ([B]) \times ([x_i] - x_i^m) \right)$$

où n est le nombre de variables de f, x_i^m est la valeur du milieu de l'intervalle $[x_i]$, B_m est le point au milieu de $[B]$ (i.e., $B_m = (x_1^m, ..., x_n^m)$) et $\left[\frac{\partial f}{\partial x_i} \right]$ est une extension aux intervalles de $\frac{\partial f}{\partial x_i}$. L'extension de Taylor calcule généralement des évaluations étroites quand les diamètres des dérivées partielles sont proches de 0. Dans d'autres cas, elle peut être pire que l'extension naturelle.

Une variante bien connue de l'extension de Taylor, appelée *extension de Hansen* $[f]_H$, calcule une image plus étroite mais à un coût plus élevé [6]. Sans entrer dans les détails, l'extension de Hansen, contrairement à celle de Taylor, calcule les dérivées partielles en utilisant la boîte $[B]$ dans laquelle certains intervalles ont été remplacés par leur point milieu. Cela implique que pour tout $[B] \in \mathbb{IR}^n$: $[f]_H([B]) \subseteq [f]_T([B])$. Cependant, quand on utilise la *différenciation automatique* pour calculer les dérivées partielles de f, l'extension de Taylor utilise la même boîte $[B]$ tandis que l'extension de Hansen doit utiliser une boîte différente par variable. Ainsi, le calcul des dérivées partielles est n fois plus cher avec l'extension de Hansen qu'avec celle de Taylor. L'extension naturelle et l'extension de Hansen (comme l'extension de Taylor) sont incomparables.

Une autre extension aux intervalles utilise la monotonie d'une fonction sur un domaine donné. En fait, quand une fonction est monotone par rapport à toutes ses variables, le problème des occurrences multiples disparaît et l'évaluation (utilisant l'extension par monotonie) devient optimale.

Définition 1 (f_{min}, f_{max}, extension par monotonie) *Soit f une fonction définie sur les variables V de domaines $[V]$. Soit $X \subseteq V$ un sous-ensemble de variables monotones.*

Considérons les valeurs x_i^+ et x_i^- telles que : si $x_i \in X$ est une variable croissante (resp. décroissante), alors $x_i^- = \underline{x_i}$ et $x_i^+ = \overline{x_i}$ (resp. $x_i^- = \overline{x_i}$ et $x_i^+ = \underline{x_i}$).

Soit $W = V \setminus X$ l'ensemble des variables non détectées monotones. Alors, f_{min} et f_{max} sont les fonctions définies par :

$$\begin{aligned} f_{min}(W) &= f(x_1^-, ..., x_n^-, W) \\ f_{max}(W) &= f(x_1^+, ..., x_n^+, W) \end{aligned}$$

Finalement, l'extension par monotonie $[f]_M$ de f dans la boîte $[V]$ produit l'intervalle suivant :

$$[f]_M([V]) = \left[\underline{[f_{min}]_N([W])}, \overline{[f_{max}]_N([W])} \right]$$

Les bornes de l'évaluation par monotonie peuvent être calculées en utilisant n'importe quelle extension aux intervalles (et non nécessairement l'extension naturelle).

Quand l'évaluation par monotonie utilise, récursivement, la même évaluation par monotonie pour calculer les bornes de l'image, nous désignons cette évaluation par *extension récursive par monotonie*.

Considérons par exemple la fonction $f(x_1, x_2) = -6x_1 + x_1x_2^2 + 3x_2$. Les dérivées partielles par rapport à x_1 et x_2 sont $\frac{\partial f}{\partial x_1}(x_2) = -6 + x_2^2$ et $\frac{\partial f}{\partial x_2}(x_1, x_2) = 2x_1x_2 + 3$. Si les intervalles des variables sont $[x_1] = [-2, -1]$ et $[x_2] = [0, 1]$, alors f est décroissante en x_1 et non monotone en x_2. Ainsi, l'évaluation récursive par monotonie devra calculer :

$$[f]_{MR}([B]) = [\underline{[f]_{MR}(-1, [0, 1])}, \overline{[f]_{MR}(-2, [0, 1])}]$$

Dans la boîte $\{-1\} \times [0,1]$ impliquée dans l'évaluation de la borne gauche, il se trouve que f est maintenant croissante en x_2 ($[\frac{\partial f}{\partial x_2}]_N(-1,[0,1]) = [1,3]$). Nous pouvons donc remplacer l'intervalle $[x_2]$ par sa borne gauche. En revanche, dans la boîte $\{-2\} \times [0,1]$, f n'est toujours pas monotone en x_2. Ainsi, l'évaluation récursive par monotonie calcule finalement :

$$[f]_{MR}([B]) = [\underline{[f](-1,0)}, \overline{[f](-2,[0,1])}] = [6,15]$$

où $[f]$ peut être n'importe quelle extension aux intervalles, par exemple l'extension naturelle.

De manière générale, l'évaluation récursive par monotonie calcule un intervalle image plus étroit (ou égal) que ne le fait l'évaluation par monotonie (c.-à-d., $[f]_{MR}([B]) \subseteq [f]_M([B])$) à un coût entre 2 et $2n$ fois plus grand. Dans l'exemple, on vérifie bien que l'évaluation par monotonie produit une évaluation ($[f]_M([B]) = [5,15]$) légèrement plus mauvaise que la variante récursive.

Cet article explique comment utiliser la monotonie quand une fonction n'est pas monotone par rapport à une variable x, mais est monotone par rapport à un sous-ensemble des occurrences de x.

Nous présentons dans la partie suivante l'idée de regrouper les occurrences en trois groupes, croissantes, décroissantes et non monotones. Des programmes linéaires permettant d'obtenir des regroupements d'occurrences *intéressants* sont décrits dans les parties 3 et 4. Dans la partie 5, nous proposons un algorithme qui résout le programme linéaire présenté à la partie 4. Finalement, dans la partie 6, quelques expérimentations montrent les avantages de ce regroupement d'occurrences pour résoudre des systèmes d'équations, en particulier quand on utilise un algorithme de filtrage comme `Mohc` [2, 1] qui exploite la monotonie.

2 Évaluation par monotonie avec regroupement d'occurrences

Dans cette partie, nous étudions le cas d'une fonction qui n'est pas monotone par rapport à une variable à occurrences multiples. Nous pouvons, sans perte de généralité, limiter l'étude à une fonction d'une variable : la généralisation à une fonction de plusieurs variables est immédiate, les évaluations par monotonie étant indépendantes.

Exemple 1 *Considérons* $f_1(x) = -x^3 + 2x^2 + 6x$. *Nous voulons calculer l'image de l'intervalle* $[-1.2, 1]$ *par cette fonction. La dérivée est* $f_1'(x) = -3x^2 + 4x + 6$ *: elle contient un terme positif* (6)*, un terme négatif* $(-3x^2)$ *et un terme contenant zéro* $(4x)$.

$[f_1]_{opt}([B])$ *vaut* $[-3.05786, 7]$*, mais on ne peut pas l'obtenir directement par une simple évaluation de fonction sur intervalles. (On doit résoudre* $f_1'(x) = 0$*, ce qui est dans le cas général un problème en soi.)*

Dans l'intervalle $[-1.2, 1]$*, la fonction* f_1 *n'est pas monotone. L'évaluation naturelle donne* $[-8.2, 10.608]$*, celle de Horner (utilisant une forme factorisée ; voir [7]) produit* $[-11.04, 9.2]$.

Quand une fonction n'est pas monotone par rapport à une variable x, elle peut parfois être monotone par rapport à certaines occurrences. Une première idée naïve pour utiliser la monotonie en ces occurrences est la suivante. On remplace la fonction f par une fonction f^{nog}, regroupant toutes les occurrences croissantes dans une variable x_a, toutes les occurrences décroissantes dans une variable x_b, et les non monotones dans x_c. Le domaine des nouvelles variables auxiliaires est le même : $[x_a] = [x_b] = [x_c] = [x]$.

Pour f_1, ce regroupement aboutit à $f_1^{nog}(x_a, x_b, x_c) = -x_b^3 + 2x_c^2 + 6x_a$. En suivant la définition 1 :

- $\underline{[f_1^{nog}]_M([-1.2,1])} = \underline{[f_1^{nog}]_N(-1.2, 1, [-1.2,1])} = \underline{-1^3 + 2[-1.2,1]^2 - 7.2} = -8.2$
- $\overline{[f_1^{nog}]_M([-1.2,1])} = 10.608$

Finalement, l'évaluation par monotonie produit : $[f_1^{nog}]_M([-1.2,1]) = [-8.2, 10.608]$.

Il apparaît que l'évaluation par monotonie de la nouvelle fonction f^{nog} produit toujours le même résultat que l'évaluation naturelle. En effet, quand un nœud dans l'arbre d'évaluation correspond à une fonction croissante par rapport à un nœud fils, l'évaluation naturelle choisit automatiquement la borne droite du domaine du nœud fils pour calculer la borne droite du domaine du nœud.

L'idée principale de l'article est donc de changer ce regroupement pour réduire le problème de dépendance et obtenir des évaluations plus étroites. Nous pouvons en effet regrouper des occurrences (croissantes, décroissantes ou non monotones) dans une variable croissante x_a tant que la fonction reste croissante par rapport à cette variable x_a. Par exemple, si on place une occurrence non monotone dans un groupe monotone, l'évaluation peut être meilleure (ou rester la même). De même, si on peut transférer toutes les occurrences décroissantes dans la partie croissante, le problème de dépendance n'apparaît plus qu'entre les occurrences des groupes croissant et non monotone.

Pour f_1, si on regroupe le terme à dérivée positive avec le terme à dérivée contenant 0, on obtient la nouvelle fonction : $f_1^{og}(x_a, x_b) = -x_b^3 + 2x_a^2 + 6x_a$. Comme la dérivée par rapport au regroupement des deux occurrences (la variable x_a) est positive : $4[x_a] + 6 = [1.2, 10]$, f_1^{og} est croissante par rapport à x_a. On peut alors utiliser l'évaluation par monotonie et obtenir l'intervalle $[-5.32, 9.728]$. On peut de la même manière

obtenir $f_1^{og}(x_a, x_c) = -x_a^3 + 2x_c^2 + 6x_a$, l'évaluation par monotonie donnant alors $[-5.472, 7.88]$. On remarque alors qu'on trouve des images plus étroites que l'évaluation naturelle de f_1.

Dans la partie 3, nous présentons un programme linéaire qui réalise automatiquement un *regroupement d'occurrences*.

Extension aux intervalles par regroupement d'occurrences

Considérons la fonction $f(x)$ avec des occurrences multiples de x. On obtient une nouvelle fonction $f^{og}(x_a, x_b, x_c)$ en remplaçant dans f chaque occurrence de x par l'une des trois variables x_a, x_b, x_c, de telle sorte que f^{og} soit croissante par rapport à x_a dans $[x]$ et décroissante par rapport à x_b. On définit alors l'*extension aux intervalles par regroupement d'occurrences* de f par :

$$[f]_{og}([B]) := [f^{og}]_M([B])$$

Contrairement aux extensions naturelle et par monotonie, l'extension aux intervalles par regroupement d'occurrences n'est pas unique pour une fonction f puisqu'elle dépend du regroupement d'occurrences (og) qui transforme f en f^{og}.

3 Un programme linéaire en 0,1 pour réaliser le regroupement d'occurrences

Dans cette partie, nous proposons une méthode pour automatiser le regroupement d'occurrences. En nous basant sur l'extension de Taylor, nous calculons d'abord une surestimation du *diamètre* de l'image calculée par $[f]_{og}$. Ensuite, nous proposons un programme linéaire réalisant un regroupement qui minimise cette surestimation.

3.1 Surestimation basée sur Taylor

D'une part, comme f^{og} peut ne pas être monotone par rapport à x_c, l'évaluation par monotonie considère les occurrences de x_c comme des variables différentes, comme le ferait l'évaluation naturelle. D'autre part, comme f^{og} est monotone par rapport à x_a et x_b, l'évaluation par monotonie de ces variables est optimale. Les deux propositions suivantes sont bien connues.

Proposition 1 *Soit $f(x)$ une fonction continue dans une boîte $[B]$ avec l'ensemble des occurrences de x : $\{x_1, x_2, ..., x_k\}$. $f^\circ(x_1, .., x_k)$ est la fonction obtenue en considérant toutes les occurrences de x comme des variables différentes. Alors, $[f]_N([B])$ calcule $[f^\circ]_{opt}([B])$.*

Proposition 2 *Soit $f(x_1, x_2, ..., x_n)$ une fonction monotone en toutes ses variables sur la boîte $[B] = \{[x_1], [x_2], ..., [x_n]\}$. Alors, l'évaluation par monotonie est optimale sur $[B]$, c.-à-d. calcule $[f]_{opt}([B])$.*

En utilisant ces deux propositions, on observe que $[f^{og}]_M([x_a], [x_b], [x_c])$ est équivalent à $[f^\circ]_{opt}([x_a], [x_b], [x_{c_1}], ..., [x_{c_{ck}}])$, en considérant chaque occurrence de x_c dans f^{og} comme une variable indépendante x_{c_j} dans f°, ck étant le nombre d'occurrences de x_c dans f^{og}. En utilisant l'évaluation de Taylor, une surestimation du diamètre $diam([f]_{opt}([B]))$ est donnée par le côté droit de (1) dans la proposition 3.

Proposition 3 *Soit $f(x_1, ..., x_n)$ une fonction sur les domaines $[B] = \{[x_1], ..., [x_n]\}$. Alors,*

$$diam([f]_{opt}([B])) \leq \sum_{i=1}^{n} \big(diam([x_i]) \times |[g_i]([B])| \big) \quad (1)$$

où $[g_i]$ est une extension aux intervalles de $g_i = \frac{\partial f}{\partial x_i}$.

En utilisant la proposition 3, on peut calculer un majorant du **diamètre** de $[f]_{og}([B]) = [f^{og}]_M([B]) = [f^\circ]_{opt}([B])$:

$$diam([f]_{og}([B])) \leq \quad diam([x]) \quad \Big(|[g_a]([B])| + |[g_b]([B])| + \sum_{i=1}^{ck} |[g_{c_i}]([B])| \Big)$$

$[g_a]$, $[g_b]$ et $[g_{c_i}]$ sont les extensions aux intervalles de $g_a = \frac{\partial f^{og}}{\partial x_a}$, $g_b = \frac{\partial f^{og}}{\partial x_b}$ et $g_{c_i} = \frac{\partial f^{og}}{\partial x_{c_i}}$. $diam([x])$ est factorisé car $[x] = [x_a] = [x_b] = [x_{c_1}] = ... = [x_{c_{ck}}]$.

Pour garantir les conditions de monotonie requises par f^{og} : $\frac{\partial f^{og}}{\partial x_a} \geq 0$, $\frac{\partial f^{og}}{\partial x_b} \leq 0$, nous avons les conditions suffisantes $\underline{[g_a]([B])} \geq 0$ et $\overline{[g_b]([B])} \leq 0$, qui impliquent $|[g_a]([B])| = \overline{[g_a]([B])}$ et $|[g_b]([B])| = -\underline{[g_b]([B])}$. Finalement :

$$diam([f]_{og}([B])) \quad \leq \quad diam([x]) \Big(\overline{[g_a]([B])} - \underline{[g_b]([B])} + \sum_{i=1}^{ck} |[g_{c_i}]([B])| \Big) \quad (2)$$

3.2 Un programme linéaire

Nous voulons transformer f en une nouvelle fonction f^{og} qui minimise le côté droit de la relation (2). Le problème peut être facilement transformé en le programme linéaire en nombres entiers suivant :

Trouver les valeurs r_{a_i}, r_{b_i} et r_{c_i} pour chaque occurrence x_i qui minimisent

$$G = \overline{[g_a]([B])} - \underline{[g_b]([B])} + \sum_{i=1}^{k} \left(|[g_i]([B])| r_{c_i} \right) \quad (3)$$

sous les contraintes :

$$\underline{[g_a]([B])} \geq 0 \quad (4)$$

$$\overline{[g_b]([B])} \leq 0 \quad (5)$$

$$r_{a_i} + r_{b_i} + r_{c_i} = 1 \quad \text{for } i = 1, ..., k \quad (6)$$

$$r_{a_i}, r_{b_i}, r_{c_i} \in \{0, 1\} \quad \text{for } i = 1, ..., k,$$

où une valeur r_{a_i}, r_{b_i} ou r_{c_i} égale à 1 indique que l'occurrence x_i de f sera remplacée, respectivement, par x_a, x_b ou x_c dans f^{og}. k est le nombre d'occurrences de x, $[g_a]([B]) = \sum_{i=1}^{k} [g_i]([B]) r_{a_i}$, $[g_b]([B]) = \sum_{i=1}^{k} [g_i]([B]) r_{b_i}$, et $[g_i]([B]), ..., [g_k]([B])$ sont les dérivées par rapport à chaque occurrence.

On peut remarquer que $[g_a]([B])$ et $[g_b]([B])$) sont calculés en utilisant uniquement les dérivées de f par rapport à chaque occurrence de x ($[g_i]([B])$).

Programme linéaire correspondant à l'exemple 1

Nous avons $f_1(x) = -x^3 + 2x^2 + 6x$ et $f_1'(x) = -3x^2 + 4x + 6$ avec $x \in [-1.2, 1]$. Le gradient vaut : $[g_1]([-1.2, 1]) = [-4.32, 0]$, $[g_2]([-1.2, 1]) = [-4.8, 4]$ et $[g_3]([-1.2, 1]) = [6, 6]$. Alors, le programme linéaire est :

Trouver les valeurs r_{a_i}, r_{b_i} et r_{c_i} qui minimisent

$$\begin{aligned} G &= \sum_{i=1}^{3} \overline{[g_i]([B])} r_{a_i} - \sum_{i=1}^{3} \underline{[g_i]([B])} r_{b_i} + \\ &\quad \sum_{i=1}^{3} \left(|[g_i]([B])| r_{c_i} \right) \\ &= (4 r_{a_2} + 6 r_{a_3}) + (4.32 r_{b_1} + 4.8 r_{b_2} - 6 r_{b_3}) \\ &\quad + (4.32 r_{c_1} + 4.8 r_{c_2} + 6 r_{c_3}) \end{aligned}$$

sous les contraintes :

$$\sum_{i=1}^{3} \underline{[g_i]([B])} r_{a_i} = -4.32 r_{a_1} - 4.8 r_{a_2} + 6 r_{a_3} \geq 0$$

$$\sum_{i=1}^{3} \overline{[g_i]([B])} r_{b_i} = 4 r_{b_2} + 6 r_{b_3} \leq 0$$

$$r_{a_i} + r_{b_i} + r_{c_i} = 1 \quad \text{for } i = 1, ..., 3$$

$$r_{a_i}, r_{b_i}, r_{c_i} \in \{0, 1\} \quad \text{for } i = 1, ..., 3$$

On obtient le minimum 10.8, et la solution $r_{a_1} = 1, r_{b_1} = 0, r_{c_1} = 0, r_{a_2} = 0, r_{b_2} = 0, r_{c_2} = 1, r_{a_3} = 1, r_{b_3} = 0, r_{c_3} = 0$, qui est la dernière solution présentée à la section 2. On peut remarquer que la valeur de la surestimation de $diam([f]_{og}([B]))$ est égale à 23.76 ($10.8 \times diam([-1.2, 1])$) alors que $diam([f]_{og}([B])) = 13.352$. Bien que la surestimation soit assez grossière, l'heuristique fonctionne bien sur cet exemple. En effet, $diam([f]_N([B])) = 18.808$ et $diam([f]_{opt}([B])) = 10.06$.

4 Un programme linéaire continu

Le programme linéaire précédent est un programme linéaire en 0,1 qui est connu comme étant NP-difficile en général. On peut le rendre continu et polynomial en permettant à r_{a_i}, r_{b_i} et r_{c_i} de prendre des valeurs réelles. En d'autres termes, on permet à chaque occurrence de x dans f d'être remplacée par une combinaison linéaire convexe des variables auxiliaires x_a, x_b et x_c, f^{og} étant croissante par rapport à x_a et décroissante par rapport à x_b. Chaque occurrence x_i est remplacée dans f^{og} par $r_{a_i} x_a + r_{b_i} x_b + r_{c_i} x_c$, avec $r_{a_i} + r_{b_i} + r_{c_i} = 1$, $\frac{\partial f^{og}}{\partial x_a} \geq 0$ et $\frac{\partial f^{og}}{\partial x_b} \leq 0$. On peut remarquer alors que f et f^{og} ont la même évaluation naturelle.

Dans l'exemple 1, on peut remplacer f_1 par f^{og_1} ou f^{og_2} en respectant les contraintes de monotonie sur x_a et x_b. Considérons l'intervalle $[x] = [-1.2, 1]$:

1. $f_1^{og_1}(x_a, x_b) = -(\frac{5}{18} x_a + \frac{13}{18} x_b)^3 + 2x_a^2 + 6x_a$
   $[f_1^{og_1}]_M([x]) = [-4.38, 8.205]$
2. $f_1^{og_2}(x_a, x_b, x_c) = -x_a^3 + 2(0.35 x_a + 0.65 x_c)^2 + 6x_a$
   $[f_1^{og_2}]_M([x]) = [-5.472, 7]$

Exemple 2 *Considérons la fonction $f_2(x) = x^3 - x$ et l'intervalle $[x] = [0.5, 2]$. f_2 n'est pas monotone et l'image optimale $[f_2]_{opt}([x])$ vaut $[-0.385, 6]$. L'évaluation naturelle donne $[-1.975, 7.5]$, l'évaluation de Horner $[-1.5, 6]$. On peut remplacer f_2 par l'une des fonctions suivantes :*

1. $f_2^{og_1}(x_a, x_b) = x_a^3 - (\frac{1}{4} x_a + \frac{3}{4} x_b)$
   $[f_2^{og_1}]_M([x]) = [-0.75, 6.375]$
2. $f_2^{og_2}(x_a, x_b) = (\frac{11}{12} x_a + \frac{1}{12} x_b)^3 - x_b$
   $[f_2^{og_2}]_M([x]) = [-1.756, 6.09]$

Le nouveau programme linéaire qui prend en compte la combinaison linéaire convexe pour réaliser le regroupement d'occurrences devient :
Trouver les valeurs r_{a_i}, r_{b_i} et r_{c_i} pour chaque occurrence x_i qui minimisent (3) sous les contraintes (4), (5), (6) et

$$r_{a_i}, r_{b_i}, r_{c_i} \in [0, 1] \quad \text{for } i = 1, ..., k. \quad (7)$$

Programme linéaire correspondant à l'exemple 1

Dans cet exemple, nous obtenons comme minimum 10.58 et la nouvelle fonction $f_1^{og}(x_a, x_b, x_c) = -x_a^3 + 2(0.35x_a + 0.65x_c)^2 + 6x_a$: $[f_1^{og}]_M([x]) = [-5.472, 7]$. Le minimum 10.58 est inférieur à 10.8 (obtenu par le programme linéaire en 0,1). L'évaluation par regroupement d'occurrences de f_1 donne $[-5.472, 7]$, ce qui est plus étroit que l'image $[-5.472, 7.88]$ obtenue par le programme linéaire en 0,1 présenté dans la partie 3.

Programme linéaire correspondant à l'exemple 2

Dans cet exemple, nous obtenons comme minimum 11.25 et la nouvelle fonction $f_2^{og}(x_a, x_b) = (\frac{44}{45}x_a + \frac{1}{45}x_b)^3 - (\frac{11}{15}x_a + \frac{4}{15}x_b)$. L'image $[-0.75, 6.01]$ obtenue par regroupement d'occurrences est plus étroite que celles obtenues par les évaluations naturelle et de Horner. Dans ce cas, le programme linéaire en 0,1 de la partie 3 donne le regroupement naïf.

On notera que le programme linéaire continu non seulement rend le problème d'optimisation polynomial, mais peut aussi améliorer le minimum (les conditions d'intégrité étant relâchées).

5 Un algorithme efficace de regroupement d'occurrences

L'algorithme 1 trouve les valeurs r_{a_i}, r_{b_i}, r_{c_i} qui minimisent G sous les contraintes de monotonie. Il génère aussi la nouvelle fonction f^{og} qui remplace chaque occurrence x_i de f par $[r_{a_i}]x_a + [r_{b_i}]x_b + [r_{c_i}]x_c$. On notera que les valeurs sont représentées par de petits intervalles (quelques u.l.p.), pour prendre en compte les erreurs d'arrondis des calculs en virgule flottante.

L'algorithme 1 utilise un vecteur $[g_*]$ de taille k contenant les intervalles des dérivées partielles de f par rapport à chaque *occurrence* x_i de x. Chaque composante de $[g_*]$ est notée $[g_i]$ et correspond à l'intervalle $\left[\frac{\partial f}{\partial x_i}\right]_N([B])$.

Un symbole indicé par une astérisque (*) représente un vecteur ($[g_*]$, $[r_{a_*}]$).

Nous illustrons l'algorithme avec les deux fonctions de nos exemples : $f_1(x) = -x^3 + 2x^2 + 6x$ et $f_2(x) = x^3 - x$ pour les domaines de x : $[-1.2, 1]$ et $[0.5, 2]$ respectivement. Pour ces exemples, les intervalles des dérivées de f_2 par rapport aux occurrences de x sont $[g_1] = [0.75, 12]$ et $[g_2] = [-1, -1]$; ceux des dérivées de f_1 sont $[g_1] = [-4.32, 0]$, $[g_2] = [-4.8, 4]$ et $[g_3] = [6, 6]$.

A la ligne 1, nous calculons la dérivée partielle $[G_0]$ de f par rapport à x en sommant les dérivées partielles de f par rapport à chaque occurrence de x. A la ligne 2, $[G_m]$ donne la valeur de la dérivée partielle de f par rapport aux occurrences monotones de x. Sur les exemples, pour f_1 : $[G_0] = [g_1]+[g_2]+[g_3] = [-3.12, 10]$ et $[G_m] = [g_1] + [g_3] = [1.68, 6]$, et pour f_2 : $[G_0] = [G_m] = [g_1] + [g_2] = [-0.25, 11]$.

Algorithm 1 `OccurrenceGrouping`(**in :** f,$[g_*]$ **out :** f^{og})

1: $[G_0] \leftarrow \sum_{i=1}^{k} [g_i]$
2: $[G_m] \leftarrow \sum_{0 \notin [g_i]} [g_i]$
3: **if** $0 \notin [G_0]$ **then**
4: `OG_case1`$([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
5: **else if** $0 \in [G_m]$ **then**
6: `OG_case2`$([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
7: **else**
8: /* $0 \notin [G_m]$ and $0 \in [G_0]$ */
9: **if** $\underline{G_m} \geq 0$ **then**
10: `OG_case3`$^+([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
11: **else**
12: `OG_case3`$^-([g_*], [r_{a_*}], [r_{b_*}], [r_{c_*}])$
13: **end if**
14: **end if**
15: $f^{og} \leftarrow$ `Generate_New_Function`$(f, [r_{a_*}], [r_{b_*}], [r_{c_*}])$

Selon les valeurs de $[G_0]$ et $[G_m]$, on peut distinguer 3 cas. Le premier cas est bien connu ($0 \notin [G_0]$ à la ligne 3) et apparaît quand x est une variable monotone. La procédure `OG_case1` ne réalise aucun regroupement d'occurrences : *toutes les occurrences* de x sont remplacées par x_a (si $[G_0] \geq 0$) ou par x_b (si $[G_0] \leq 0$). L'évaluation par monotonie de f^{og} est équivalente à l'évaluation par monotonie de f.

Dans le second cas, quand $0 \in [G_m]$ (ligne 5), la procédure `OG_case2` (Algorithme 2) réalise un regroupement des occurrences de x. Les occurrences croissantes sont remplacées par $(1 - \alpha_1)x_a + \alpha_1 x_b$, les occurrences décroissantes par $\alpha_2 x_a + (1 - \alpha_2)x_b$ et les non monotones par x_c (lignes 7 à 13 de l'algorithme 2). f_2 tombe dans ce cas de figure : $\alpha_1 = \frac{1}{45}$ et $\alpha_2 = \frac{11}{15}$ sont calculées aux lignes 3 et 4 de l'algorithme 2 en utilisant $[G^+] = [g_1] = [0.75, 12]$ et $[G^-] = [g_2] = [-1, -1]$. La nouvelle fonction devient : $f_2^{og}(x_a, x_b) = (\frac{44}{45}x_a + \frac{1}{45}x_b)^3 - (\frac{11}{15}x_a + \frac{4}{15}x_b)$.

Le troisième cas apparaît quand $0 \notin [G_m]$ et $0 \in [G_0]$. Sans perte de généralité, supposons $\underline{G_m} \geq 0$. La procédure `OG_case3`$^+$ (Algorithme 3) regroupe d'abord toutes les occurrences monotones dans le groupe croissant (lignes 2–5). Les occurrences non monotones sont alors remplacées par x_a dans un ordre déterminé par le tableau $index$[1] (ligne 7) tant que la contrainte $\sum_{i=1}^{k} r_{a_i}\underline{g_i} \geq 0$ est satisfaite (lignes 9–13). La première occurrence non monotone $x_{i'}$ qui rendrait la contrainte non satisfaite est remplacée par

[1] x_{i_1} est traitée avant x_{i_2} si $|\overline{g_{i_1}}/\underline{g_{i_1}}| \leq |\overline{g_{i_2}}/\underline{g_{i_2}}|$. $index[j]$ rend l'indice de la $j^{ème}$ occurrence dans cet ordre.

Algorithm 2 `OG_case2`(**in:** $[g_*]$ **out:** $[r_{a_*}], [r_{b_*}], [r_{c_*}]$)

1: $[G^+] \leftarrow \sum_{[g_i]\geq 0} [g_i]$
2: $[G^-] \leftarrow \sum_{[g_i]\leq 0} [g_i]$
3: $[\alpha_1] \leftarrow \frac{\underline{G^+}\overline{G^-} + \overline{G^-}\underline{G^-}}{\underline{G^+}\overline{G^-} - \underline{G^-}\overline{G^+}}$
4: $[\alpha_2] \leftarrow \frac{\underline{G^+}\overline{G^+} + \overline{G^-}\underline{G^+}}{\underline{G^+}\overline{G^-} - \underline{G^-}\overline{G^+}}$
5:
6: **for all** $[g_i] \in [g_*]$ **do**
7: **if** $\underline{g_i} \geq 0$ **then**
8: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (1 - [\alpha_1], [\alpha_1], 0)$
9: **else if** $\overline{g_i} \leq 0$ **then**
10: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow ([\alpha_2], 1 - [\alpha_2], 0)$
11: **else**
12: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (0, 0, 1)$
13: **end if**
14: **end for**

$\alpha x_a + (1-\alpha)x_c$, avec α tel que la contrainte est satisfaite et égale à 0, $(\sum_{i=1, i\neq i'}^{k} r_{a_i}\underline{g_i}) + \alpha\underline{g_{i'}} = 0$ (lignes 15–17). Le reste des occurrences non monotones sont remplacées par x_c (lignes 20–22). f_1 tombe dans ce cas de figure. Les première et troisième occurrences de x sont monotones et sont remplacées par x_a. La deuxième occurrence de x, qui est non monotone, est remplacée par $\alpha x_a + (1-\alpha)x_c$, où $\alpha = 0.35$ est obtenu en forçant la contrainte (4) à valoir 0 : $\underline{g_1} + \underline{g_3} + \alpha\underline{g_2} = 0$. On obtient ainsi la nouvelle fonction : $f_1^{og} = -x_a^3 + 2(0.35x_a + 0.65x_c)^2 + 6x_a$.

Finalement, la procédure `Generate_New_Function` (ligne 15 de l'algorithme 1) crée de manière symbolique la nouvelle fonction f^{og}.

Observations

L'algorithme 1 respecte les quatre contraintes (4)–(7). Nous avons démontré que le minimum de la fonction objectif (3) est atteint pour `OG_case2` et sommes en train de finaliser la preuve pour `OG_case3`.

On peut utiliser un algorithme du simplexe standard à la place de l'algorithme 1, à condition que l'implantation prenne en compte les erreurs d'arrondis dues aux calculs en virgule flottante. Nous présentons à la partie 6.3 une comparaison des performances respectives de l'algorithme 1 et du simplexe.

Complexité en temps

La complexité en temps de `Occurrence_Grouping` pour une variable avec k occurrences est en $O(k\,log_2(k))$. Elle est dominée par la complexité de

Algorithm 3 `OG_case3`$^+$(**in:**$[g_*]$ **out:**$[r_{a_*}], [r_{b_*}], [r_{c_*}]$)

1: $[g_a] \leftarrow [0, 0]$
2: **for all** $[g_i] \in [g_*]$, $0 \notin [g_i]$ **do**
3: $[g_a] \leftarrow [g_a] + [g_i]$ /* Toutes les dérivées positives et négatives sont absorbées dans $[g_a]$ */
4: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (1, 0, 0)$
5: **end for**
6:
7: $index \leftarrow$ `ascending_sort` ($\{[g_i] \in [g_*], 0 \in [g_i]\}$, `criterion` $\rightarrow |\overline{g_i}/\underline{g_i}|$)
8: $j \leftarrow 1$; $i \leftarrow index[1]$
9: **while** $\underline{g_a} + \underline{g_i} \geq 0$ **do**
10: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (1, 0, 0)$
11: $[g_a] \leftarrow [g_a] + [g_i]$
12: $j \leftarrow j + 1$; $i \leftarrow index[j]$
13: **end while**
14:
15: $[\alpha] \leftarrow -\frac{\underline{g_a}}{\underline{g_i}}$
16: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow ([\alpha], 0, 1 - [\alpha])$
17: /* $[g_a] \leftarrow [g_a] + [\alpha][g_i]$ */
18: $j \leftarrow j + 1$; $i \leftarrow index[j]$
19:
20: **while** $j \leq$ `length`$(index)$ **do**
21: $([r_{a_i}], [r_{b_i}], [r_{c_i}]) \leftarrow (0, 0, 1)$
22: $j \leftarrow j + 1$; $i \leftarrow index[j]$
23: **end while**

`ascending_sort` dans la procédure `OG_case3`. Comme le montrent les expérimentations de la partie suivante, le temps utilisé en pratique par `Occurrence_Grouping` est négligeable quand la procédure est utilisée dans la résolution de systèmes d'équations.

6 Expérimentations

Le regroupement d'occurrences a été implanté dans le résolveur par intervalles `Ibex` [5, 4] en `C++`. Le but principal de ces expérimentations est de montrer les améliorations en temps de calcul apportées par le regroupement d'occurrences dans la résolution de systèmes d'équations. Seize problèmes tests proviennent du site [9]. Ils correspondent à des systèmes carrés avec un nombre fini de solutions ayant au moins deux contraintes avec des occurrences multiples et demandant plus d'une seconde pour être résolus. Deux instances (`<name>-bis`) ont été simplifiées pour réduire leur temps de résolution : les domaines initiaux des variables ont été arbitrairement réduits.

6.1 Regroupement d'occurrences pour améliorer le test d'existence par monotonie

Tout d'abord, le regroupement d'occurrences a été implanté pour être utilisé dans un test d'existence par

Problème	3BCID	¬OG	OG	**Problème**	3BCID	¬OG	OG
Brent	18.9	19.5	19.1	Butcher-bis	351	360	340
10 1008	3941	3941	3941	8 3	228305	228303	228245
Caprasse	2.51	2.56	2.56	Fourbar	13576	6742	**1091**
4 18	1305	1301	1301	4 3	8685907	4278767	963113
Hayes	39.5	41.1	40.7	Geneig	593	511	**374**
8 1	17701	17701	17701	6 10	205087	191715	158927
I5	55.0	56.3	56.7	Pramanik	100	66.6	**37.2**
10 30	10645	10645	10645	3 2	124661	98971	69271
Katsura	74.1	74.5	75.0	Trigexp2	82.5	87.0	86.7
12 7	4317	4317	4317	11 0	14287	14287	14287
Kin1	1.72	1.77	1.77	Trigo1	152	155	156
6 16	85	85	85	10 9	2691	2691	2691
Eco9	12.7	13.5	13.2	Virasoro-bis	21.1	21.5	19.8
9 16	6203	6203	6203	8 224	2781	2781	2623
Redeco8	5.61	5.71	5.66	Yamamura1	9.67	10.04	9.86
8 8	2295	2295	2295	8 7	2883	2883	2883

TAB. 1 – Résultats expérimentaux obtenus en utilisant le test d'existence par monotonie. Les colonnes 1 et 5 indiquent le nom du problème, avec son nombre de variables (à gauche) et de solutions (à droite). Les colonnes 2 et 6 donnent les temps de calcul en secondes (en haut) et le nombre de nœuds (en bas) obtenus sur un `Intel 6600 2.4 GHz` avec une stratégie basée sur `3BCID`. Les colonnes 3 et 7 donnent les résultats obtenus par la stratégie utilisant un test d'existence par monotonie standard et `3BCID`. Les colonnes 4 et 8 montrent les résultats de notre stratégie utilisant un test d'existence basé sur le regroupement d'occurrences et `3BCID`.

monotonie (`OG` dans le tableau 1). Un regroupement d'occurrences transformant f en f^{og} est appliqué après une bissection et avant une contraction. Ensuite, le test d'existence par monotonie est appliqué à f^{og} : si l'évaluation par monotonie de f^{og} ne contient pas 0, alors la branche est coupée dans l'arbre de recherche.

La stratégie concurrente (¬`OG`) applique directement le test d'existence par monotonie de f sans regroupement d'occurrences.

Les contracteurs utilisés dans les deux cas sont les mêmes : `3BCID` [12] et Newton sur intervalles.

On peut observer à partir de ces premiers résultats que `OG` est nettement meilleur que ¬`OG` sur trois problèmes seulement (`Fourbar`, `Geneig` et `Pramanik`). Sur les autres, l'évaluation par regroupement d'occurrences apparaît inutile. En effet, dans la plupart des problèmes, ce test d'existence ne coupe pas de branche dans l'arbre de recherche. On peut remarquer cependant que `OG` ne demande pas de temps supplémentaire par rapport à ¬`OG`. Cela souligne que le temps de calcul du regroupement d'occurrences est négligeable.

6.2 Regroupement d'occurrences dans un contracteur par monotonie

`Mohc` [2, 1] est un nouveau contracteur basé sur de la propagation de contraintes (comme `HC4` ou `Box`) qui utilise la monotonie d'une fonction pour améliorer la contraction des variables de la contrainte. Appelée à l'intérieur d'un algorithme de propagation, la procédure `Mohc-revise(`f`)` améliore le filtrage obtenu par `HC4-revise(`f`)` en effectuant principalement deux appels additionnels `HC4-revise(`$f_{min} \leq 0$`)` et `HC4-revise(`$f_{max} \geq 0$`)` (f_{min} et f_{max} sont introduits à la définition 1). Il appelle aussi une version monotone de la procédure `BoxNarrow` utilisée par `Box` [3].

Le tableau 2 montre les résultats de `Mohc` sans l'algorithme de regroupement d'occurrences `OG` (¬`OG`), et avec (`OG`), la fonction f étant transformée en f^{og} avant d'appeler `Mohc-revise(`f^{og}`)`.

On observe que pour 7 des 16 benchmarks, le regroupement d'occurrences est capable d'améliorer les résultats de `Mohc` sur `Butcher-bis`, `Fourbar`, `Virasoro-bis` et `Yamamura1` pour lesquels les gains en temps de calcul ($\frac{\neg \texttt{OG}}{\texttt{OG}}$) sont respectivement 30, 11, 5.6 et 5.4.

6.3 Comparaison avec le simplexe

Nous avons comparé les performances de deux implantations du regroupement d'occurrences, l'une utilisant notre algorithme ad hoc (`Occurrence_Grouping`) et l'autre la méthode du simplexe. L'algorithme du simplexe utilisé a été pris sur la page `pagesperso-orange.fr/jean-pierre.moreau/Cplus/tsimplex_cpp.txt`. Il n'est pas fiable, c.-à-d. ne prend pas en compte les erreurs d'arrondis de l'arithmétique en virgule flottante.

Deux résultats importants ont été obtenus. Tout d'abord, nous avons vérifié expérimentalement que notre algorithme est correct, c.-à-d. obtient le mini-

Problème	Mohc			**Problème**	Mohc		
	¬OG	OG	#OG calls		¬OG	OG	#OG calls
Brent	20	20.3		Butcher-bis	220.64	**7.33**	
	3811	3805	30867		99033	2667	111045
Caprasse	2.57	2.71		Fourbar	4277.95	**385.62**	
	1251	867	60073		1069963	57377	8265730
Hayes	17.62	17.45		Geneig	328.34	**111.43**	
	4599	4415	5316		76465	13705	2982275
I5	57.25	58.12		Pramanik	67.98	**21.23**	
	10399	9757	835130		51877	12651	395083
Katsura	100	103		Trigexp2	90.57	88.24	
	3711	3625	39659		14299	14301	338489
Kin1	1.82	1.79		Trigo1	137.27	**57.09**	
	85	83	316		1513	443	75237
Eco9	13.31	13.96		Virasoro-bis	18.95	**3.34**	
	6161	6025	70499		2029	187	241656
Redeco8	5.98	6.12		Yamamura1	11.59	**2.15**	
	2285	2209	56312		2663	343	43589

TAB. 2 – Résultats expérimentaux avec Mohc. Les colonnes 1 et 3 indiquent le nom de chaque problème, les colonnes 2 et 6 montrent les résultats obtenus par la stratégie 3BCID(Mohc) sans OG. Les colonnes 3 et 7 présentent les résultats de notre stratégie utilisant 3BCID(OG+Mohc). Les colonnes 4 et 8 indiquent le nombre d'appels au regroupement d'occurrences.

mum de la fonction objectif G. Ensuite, comme nous l'espérions, la performance de l'algorithme du simplexe général est moins bonne que celle de notre algorithme. Il prend entre 2.32 (Brent) et 10 (Virasoro) fois plus de temps.

6.4 Comparaison des diamètres lors de l'évaluation

Le tableau 3 présente une comparaison entre l'évaluation par regroupement d'occurrences ($[f]_{og}$) et un ensemble d'évaluations sur intervalles incluant Taylor, Hansen [6] et des extensions par monotonie. Les différentes colonnes sont liées aux différentes extensions $[f]_{ext}$ et donnent la moyenne des rapports $\rho_{ext} = \frac{Diam([f]_{og}([B]))}{Diam([f]_{ext}([B]))}$ calculés dans chaque procédure (Mohc-)Revise d'une fonction f à chaque nœud de l'arbre de recherche d'une stratégie de résolution utilisant l'algorithme Mohc.

La liste des extensions aux intervalles correspondant aux colonnes du tableau sont : l'extension naturelle $[f]_N$, l'extension de Taylor $[f]_T$, l'extension de Hansen $[f]_H$, l'évaluation par monotonie $[f]_M$, l'évaluation récursive par monotonie $[f]_{MR}$, l'évaluation récursive par monotonie qui calcule les bornes en utilisant l'extension de Hansen $[f]_{MR+H}$ et une évaluation récursive par monotonie qui calcule les bornes en utilisant l'extension par regroupement d'occurrences $[f]_{MR+og}$.

Il est bien connu que les extensions de Taylor et de Hansen ne sont pas comparables avec l'extension naturelle. C'est pourquoi, pour obtenir des comparaisons plus raisonnables, nous avons redéfini $[f]_T([B]) = [f]'_T([B]) \cap [f]_N([B])$ et $[f]_H([B]) = [f]'_H([B]) \cap [f]_N([B])$, où $[f]'_T$ et $[f]'_H$ sont les véritables extensions de Taylor et Hansen.

Le tableau montre que $[f]_{og}$ calcule, en général, des évaluations plus étroites que tous ses concurrents. (Seul $[f]_{MR+og}$ obtient des évaluations plus étroites, mais il utilise aussi le regroupement d'occurrences.) Les améliorations par rapport aux deux évaluations par monotonie $[f]_M$ et $[f]_{MR}$ confirment les avantages de notre approche. Par exemple, sur Fourbar, $[f]_{og}$ produit un diamètre d'intervalle qui vaut 42.7% de celui produit par $[f]_{MR}$.

$[f]_{MR+H}$ obtient les meilleures évaluations sur trois problèmes (Caprasse, Fourbar et Virasoro). Cependant, $[f]_{MR+H}$ est plus coûteux que $[f]_{og}$. $[f]_{MR+H}$ requiert le calcul de $2n$ dérivées partielles, traversant ainsi $4n$ fois l'arbre de l'expression si on utilise une méthode de dérivation automatique [6].

$[f]_{MR+og}$ produit une évaluation nécessairement meilleure que (ou égale à) $[f]_{og}$. Cependant, les expérimentations sur nos problèmes tests soulignent que le gain en diamètre n'est que de 1.6% en moyenne (entre 0% et 6.2%), si bien que nous pensons que cela ne compense pas son coût supplémentaire.

7 Conclusion

Nous avons proposé une nouvelle méthode pour améliorer l'évaluation par monotonie d'une fonction f. Cette méthode, basée sur un regroupement d'occurrences, crée pour chaque variable trois variables auxiliaires sur lesquelles f est respectivement croissante,

NCSP	$[f]_N$	$[f]_T$	$[f]_H$	$[f]_M$	$[f]_{MR}$	$[f]_{MR+H}$	$[f]_{MR+og}$
`Brent`	0.857	0.985	0.987	0.997	0.998	0.999	1.000
`Butcher-bis`	0.480	0.742	0.863	0.666	0.786	0.963	1.028
`Caprasse`	0.602	0.883	0.960	0.856	0.953	1.043	1.051
`Direct kin.`	0.437	0.806	0.885	0.875	0.921	0.979	1.017
`Eco9`	0.724	0.785	0.888	0.961	0.980	0.976	1.006
`Fourbar`	0.268	0.718	0.919	0.380	0.427	1.040	1.038
`Geneig`	0.450	0.750	0.847	0.823	0.914	0.971	1.032
`Hayes`	0.432	0.966	0.974	0.993	0.994	0.998	1.001
`I5`	0.775	0.859	0.869	0.925	0.932	0.897	1.005
`Katsura`	0.620	0.853	0.900	0.993	0.999	0.999	1.000
`Kin1`	0.765	0.872	0.880	0.983	0.983	0.995	1.001
`Pramanik`	0.375	0.728	0.837	0.666	0.689	0.929	1.017
`Redeco8`	0.665	0.742	0.881	0.952	0.972	0.997	1.011
`Trigexp2`	0.904	0.904	0.904	0.942	0.945	0.921	1.002
`Trigo1`	0.483	0.766	0.766	0.814	0.814	0.895	1.000
`Virasoro`	0.479	0.738	0.859	0.781	0.795	1.025	1.062
`Yamamura1`	0.272	0.870	0.870	0.758	0.758	0.910	1.000
MOYENNE	0.564	0.822	0.888	0.845	0.874	0.973	1.016

TAB. 3 – Différentes évaluations comparées à $[f]_{og}$

décroissante et non monotone. Elle transforme alors f en une fonction f^{og} qui remplace les occurrences d'une variable par une combinaison linéaire convexe de ces variables auxiliaires. Il en résulte l'*évaluation par regroupement d'occurrences* de f, c.-à-d. l'évaluation par monotonie de f^{og}, qui est meilleure que l'évaluation par monotonie de f.

L'extension de monotonie par regroupement d'occurrences a montré de bonnes performances quand elle est utilisée comme test d'existence et quand elle est incluse dans le contracteur `Mohc` qui exploite la monotonie des fonctions.

Références

[1] Ignacio Araya, Gilles Trombettoni, and Bertrand Neveu. Exploitation de la monotonie des fonctions dans la propagation de contraintes sur intervalles. In *Actes JFPC*, 2010.

[2] Ignacio Araya, Gilles Trombettoni, and Bertrand Neveu. Exploiting Monotonicity in Interval Constraint Propagation. In *Proc. AAAI (to appear)*, 2010.

[3] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-François Puget. Revising Hull and Box Consistency. In *Proc. ICLP*, pages 230–244, 1999.

[4] Gilles Chabert. `www.ibex-lib.org`, 2009.

[5] Gilles Chabert and Luc Jaulin. Contractor Programming. *Artificial Intelligence*, 173 :1079–1100, 2009.

[6] Eldon Hansen. *Global Optimization using Interval Analysis*. Marcel Dekker inc., 1992.

[7] William G. Horner. A new Method of Solving Numerical Equations of all Orders, by Continuous Approximation. *Philos. Trans. Roy. Soc. London*, 109 :308–335, 1819.

[8] Luc Jaulin, Michel Kieffer, Olivier Didrit, and Eric Walter. *Applied Interval Analysis*. Springer, 2001.

[9] Jean-Pierre Merlet. `www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html`.

[10] Ramon Moore. *Interval Analysis*. Prentice Hall, 1966.

[11] Arnold Neumaier. *Interval Methods for Systems of Equations*. Cambridge University Press, 1990.

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

Exploitation de la monotonie des fonctions dans la propagation de contraintes sur intervalles

Ignacio Araya[1] **Gilles Trombettoni**[1] **Bertrand Neveu**[2]
[1] COPRIN, INRIA Sophia Antipolis, Université Nice Sophia
[2] Imagine, LIGM, Université Paris-Est, France
rilianx@gmail.com {neveu,trombe}@sophia.inria.fr

Résumé

Nous proposons un nouvel algorithme de propagation de contraintes sur intervalles, appelé *consistance d'enveloppe monotone* (Mohc), qui exploite la monotonie des fonctions. La propagation est standard, mais la procédure de *révision* Mohc-Revise, utilisée pour contracter le domaine d'une variable par rapport à une contrainte individuelle, utilise des versions monotones des procédures classiques HC4-Revise et BoxNarrow. Mohc-Revise semble être la première procédure de révision adaptative en programmation par contraintes (sur intervalles). Quand une fonction est monotone en toutes ses variables, nous montrons que Mohc-Revise calcule la boîte englobante optimale de la contrainte correspondante (*Hull-consistance*). Des résultats expérimentaux très prometteurs suggèrent que Mohc a le potentiel de devenir une alternative aux algorithmes classiques HC4 et Box.

Abstract

We propose a new *interval* constraint propagation algorithm, called *MOnotonic Hull Consistency* (Mohc), that exploits monotonicity of functions. The propagation is standard, but the Mohc-Revise procedure, used to filter/contract the variable domains w.r.t. an individual constraint, uses monotonic versions of the classical HC4-Revise and BoxNarrow procedures.

Mohc-Revise appears to be the first *adaptive* revise procedure ever proposed in constraint programming. Also, when a function is monotonic w.r.t. every variable, Mohc-Revise is proven to compute the optimal/sharpest box enclosing all the solutions of the corresponding constraint (hull consistency). Very promising experimental results suggest that Mohc has the potential to become an alternative to the state-of-the-art HC4 and Box algorithms.

1 Introduction

Les résolveurs de contraintes sur intervalles traitent les systèmes d'équations et d'inégalités sur les réels. Leur fiabilité et leur performance croissantes leur permettent de résoudre des systèmes qui apparaissent dans divers domaines comme la robotique [12], les systèmes dynamiques de commande robuste ou la localisation de robots autonomes [9].

Deux principaux types d'algorithmes de contraction permettent de réduire les domaines des variables. Les algorithmes de type *Newton sur intervalles* sont des généralisations aux intervalles des méthodes standard d'analyse numérique [13]. Les algorithmes de contraction/filtrage provenant de la programmation par contraintes sont aussi au cœur des résolveurs sur intervalles. Les algorithmes de propagation de contraintes HC4 et Box [3, 15] sont très souvent utilisés dans les stratégies de résolution. Ils réalisent une boucle de propagation et réduisent les domaines des variables (c-à-d améliorent leurs bornes) avec des procédures de *révision* spécifiques (appelées HC4-Revise et BoxNarrow) qui traitent les contraintes individuellement.

HC4-Revise calcule la boîte optimale englobant toutes les solutions d'une contrainte c quand la fonction correspondante est continue et que chaque variable apparaît une seule fois dans c. Si *une* variable apparaît plusieurs fois dans c, HC4-Revise n'est généralement *pas* optimal. Dans ce cas, BoxNarrow calcule une enveloppe plus étroite. Le nouvel algorithme présenté dans cet article, appelé Mohc-Revise, essaie de traiter le cas général où *plusieurs* variables ont des occurrences multiples dans c.

Quand une fonction est monotone par rapport à une variable x dans une boîte, il est bien connu que l'ex-

tension aux intervalles de f basée sur la monotonie ne produit pas de surestimation due aux occurrences multiples de x. `Mohc-Revise` exploite cette propriété pour améliorer la contraction. La monotonie n'est généralement vraie que pour quelques paires (f, x) au début de la recherche, mais peut être détectée pour plus de paires quand on traite des boîtes plus petites en descendant dans l'arbre de recherche.

Après quelques rappels sur les CSP numériques, nous présentons l'algorithme `Mohc-Revise` et établissons quelques conditions qui en augmentent l'efficacité. Nous montrons que, si une fonction est monotone en toutes ses variables, `Mohc-Revise` calcule alors la boîte optimale englobant toutes les solutions de la contrainte (propriété de *Hull-consistance*). Des expérimentations soulignent les performances de `Mohc`.

2 Intervalles et CSP numériques

Les intervalles permettent des calculs fiables en gérant les arrondis des calculs sur les nombres en virgule flottante.

Définition 1 (Définitions de base, notations)
Un **intervalle** $[v] = [a, b]$ *est l'ensemble* $\{x \in \mathbb{R},\ a \leq x \leq b\}$. $\mathbb{IR}$ *est l'ensemble de tous les intervalles.*
$\underline{v} = a$ *(resp.* $\overline{v} = b$*) est le nombre à virgule flottante qui est la* **borne gauche** *(resp. la* **borne droite***) de* $[v]$.
`Mid`$([v])$ *est le* **milieu** *de* $[v]$.
`Diam`$([v]) := \overline{v} - \underline{v}$ *est le* **diamètre***, ou taille, de* $[v]$.
Une **boîte** $[V] = [v_1], ..., [v_n]$ *représente le produit cartésien* $[v_1] \times ... \times [v_n]$.

L'*arithmétique d'intervalles* a été définie pour étendre à $\mathbb{IR}$ les fonctions élémentaires sur $\mathbb{R}$ [13]. Par exemple, la somme sur intervalles est définie par $[v_1]+[v_2] = [\underline{v_1}+\underline{v_2}, \overline{v_1}+\overline{v_2}]$. Quand une fonction f est la composition de fonctions élémentaires, une *extension* de f aux intervalles doit être définie pour assurer un calcul d'image conservatif.

Définition 2 (Extension d'une fonction à $\mathbb{IR}$)
Soit une fonction $f : \mathbb{R}^n \to \mathbb{R}$.
$[f] : \mathbb{IR}^n \to \mathbb{IR}$ *est une* **extension** *de* f *aux intervalles si :*

$$\begin{array}{ll} \forall [V] \in \mathbb{IR}^n & [f]([V]) \supseteq \{f(V),\ V \in [V]\} \\ \forall V \in \mathbb{R}^n & f(V) = [f](V) \end{array}$$

L' **extension naturelle** $[f]_N$ d'une fonction réelle f correspond à l'utilisation directe de l'arithmétique d'intervalles. L'*extension par monotonie* est particulièrement utile quand une fonction f est *monotone* par rapport à une variable v dans une boîte donnée $[V]$, ce qui est le cas si l'évaluation de la dérivée partielle de f par rapport à v est positive (ou négative) en tout point de $[V]$. Par abus de langage, nous dirons parfois que v est monotone.

Définition 3 (f_{min}, f_{max}, extension par monotonie)
Soit f *une fonction définie sur les variables* V *de domaines* $[V]$*. Soit* $X \subseteq V$ *un sous-ensemble de variables monotones.*

Considérons les valeurs x_i^+ *et* x_i^- *telles que : si* $x_i \in X$ *est une variable croissante (resp. décroissante), alors* $x_i^- = \underline{x_i}$ *et* $x_i^+ = \overline{x_i}$ *(resp.* $x_i^- = \overline{x_i}$ *et* $x_i^+ = \underline{x_i}$*).*

Soit $W = V \setminus X$ *l'ensemble des variables non détectées monotones. Alors,* f_{min} *et* f_{max} *sont les fonctions définies par :*

$$\begin{array}{rcl} f_{min}(W) & = & f(x_1^-, ..., x_n^-, W) \\ f_{max}(W) & = & f(x_1^+, ..., x_n^+, W) \end{array}$$

Finalement, l'extension par monotonie $[f]_M$ *de* f *dans la boîte* $[V]$ *produit l'intervalle image suivant :*

$$[f]_M([V]) = \left[\underline{[f_{min}]_N([W])}, \overline{[f_{max}]_N([W])}\right]$$

La monotonie des fonctions est généralement utilisée comme un **test d'existence** calculant si 0 appartient à l'intervalle image d'une fonction. Elle a aussi été utilisée dans les CSP numériques quantifiés pour contracter facilement une variable quantifiée universellement qui est monotone [8].

Considérons par exemple :
$f(x_1, x_2, w) = -x_1^2 + x_1x_2 + x_2w - 3w$ dans la boîte $[V] = [6, 8] \times [2, 4] \times [7, 15]$.

$[f]_N([x_1], [x_2], [w]) = -[6, 8]^2 + [6, 8] \times [2, 4] + [2, 4] \times [7, 15] - 3 \times [7, 15] = [-83, 35]$.

$\frac{\partial f}{\partial x_1}(x_1, x_2) = -2x_1 + x_2$, et $[\frac{\partial f}{\partial x_1}]_N([6, 8], [2, 4]) = [-14, -8]$. Comme $[-14, -8] < 0$, nous en déduisons que f est décroissante par rapport à x_1. Avec le même raisonnement, nous déduisons que x_2 est croissante. Finalement, $0 \in [\frac{\partial f}{\partial w}]_N([x_1], [x_2], [w]) = [-1, 1]$, ainsi w n'est pas détectée monotone. Suivant la définition 3, l'évaluation par monotonie donne :
$[f]_M([V]) = \left[\underline{[f](\overline{x_1}, \underline{x_2}, [w])}, \overline{[f](\underline{x_1}, \overline{x_2}, [w])}\right]$
$= \left[\underline{[f](8, 2, [7, 15])}, \overline{[f](6, 4, [7, 15])}\right] = [-79, 27]$

2.1 Le problème de dépendance (occurrences multiples)

Le *problème de dépendance* est le point d'achoppement de l'arithmétique d'intervalles. Il est dû au fait que les *occurrences multiples* d'une même variable dans une expression sont traitées comme des variables différentes par l'arithmétique d'intervalles. Dans notre

exemple, il explique pourquoi l'intervalle image calculé par $[f]_M$ est différent et plus étroit que celui produit par $[f]_N$. De plus, si on utilisait une forme factorisée comme $-x_1^2 + x_1x_2 + (x_2 - 3)w$, on obtiendrait une image encore meilleure. Le problème de dépendance rend en fait NP-difficile le problème de trouver l'intervalle image optimal d'un polynôme [10]. (L'extension correspondante est notée $[f]_{opt}$.) Le fait que l'extension par monotonie remplace les intervalles par leurs bornes explique la proposition suivante.

Proposition 1 *Soit f une fonction continue sur $[V]$. Alors,*

$$[f]_{opt}([V]) \subseteq [f]_M([V]) \subseteq [f]_N([V])$$

De plus, si f est monotone dans la boîte $[V]$ par rapport à toutes ses variables apparaissant plusieurs fois *dans f, alors l'extension par monotonie calcule l'image optimale :*

$$[f]_M([V]) = [f]_{opt}([V])$$

2.2 CSP numériques

L'algorithme `Mohc` présenté dans cet article contribue à la résolution de systèmes de contraintes non linéaires ou CSP numériques.

Définition 4 (NCSP)
Un **CSP numérique** $P = (V, C, [V])$ *comprend un ensemble de contraintes C, un ensemble V de n variables ayant pour domaines $[V] \in \mathbb{IR}^n$.*
Une **solution** $S \in [V]$ *de P satisfait toutes les contraintes de C.*

Pour trouver toutes les solutions d'un NCSP avec des techniques par intervalles, le processus de résolution commence avec une boîte initiale représentant l'espace de recherche et construit un arbre de recherche, suivant un schéma *Brancher & Contracter*.

- *Brancher :* la boîte courante est **bissectée** sur une dimension (variable) et produit deux sous-boîtes.
- *Contracter :* les algorithmes de filtrage (aussi appelés de **contraction**) réduisent les bornes de la boîte sans perdre de solution.

Le processus se termine avec des **boîtes atomiques** de taille au plus ω sur chaque dimension.

Les algorithmes de contraction comprennent les algorithmes de type *Newton sur intervalles* issus de la communauté numérique d'*analyse par intervalles* [13] ainsi que des algorithmes venant de la programmation par contraintes. L'algorithme de contraction présenté dans cet article prend en compte la monotonie des fonctions, en adaptant les procédures classiques `HC4-Revise` et `BoxNarrow` de programmation par contraintes sur intervalles. L'algorithme `HC4` effectue une boucle de propagation de type AC3. Sa procédure de révision, appelée `HC4-Revise`, traverse deux fois l'arbre représentant l'expression mathématique de la contrainte pour contracter les intervalles des variables de la contrainte. Un exemple est donné à la figure 1.

`Box` est un autre algorithme de propagation. Pour chaque paire (f, x), où f est une fonction du NCSP considéré et x est une variable de f, `BoxNarrow` remplace d'abord les a autres variables de f par leurs intervalles $[y_1], ..., [y_a]$. Ensuite, la procédure réduit les bornes de $[x]$ de telle sorte que la nouvelle borne gauche (resp. droite) soit la solution la plus à gauche (resp. à droite) de l'équation $f(x, [y_1], ..., [y_a]) = 0$. Les procédures existantes utilisent un principe de *rognage* qui élimine de $[x]$ les sous-intervalles $[x_i]$ aux bornes de $[x]$ qui ne satisfont pas la contrainte.

Contracter de manière optimale une boîte par rapport à une contrainte individuelle revient à atteindre la propriété que l'on appelle **hull-consistance**. Comme pour le calcul de l'intervalle image optimal, la *hull*-consistance n'est pas atteignable en temps polynomial, à cause du problème de la dépendance. `HC4-Revise` calcule la hull-consistance des contraintes sans variable avec occurrences multiples, à condition que la fonction et ses fonctions de projection soient continues. La *Box-consistance* obtenue par l'algorithme `BoxNarrow` est plus forte [7] et produit la *hull*-consistance quand la contrainte ne contient qu'*une* variable avec occurrences multiples. En effet, le processus de rognage réalisé par `BoxNarrow` sur une variable x limite fortement l'effet de surestimation sur x. Par contre, il n'est *pas optimal* dans le cas où d'autres variables y_i possèdent aussi des occurrences multiples.

Ces algorithmes sont parfois utilisés dans nos expérimentations comme des sous-contracteurs de `3BCID` [14], une variante de `3B` [11]. `3B` utilise un principe de réfutation par rognage. Un sous-intervalle $[x_i]$ à une borne d'un intervalle $[x]$ est supprimé si l'appel au sous-contracteur (par exemple `HC4`) sur le sous-problème correspondant (où $[x]$ est remplacé par $[x_i]$) aboutit à un échec (sous-problème sans solution). On procède ainsi de chaque côté jusqu'à ce qu'on obtienne une tranche ne pouvant pas être supprimée ou que toutes les tranches l'aient été.

3 L'algorithme Mohc

L'algorithme de consistance d'enveloppe monotone (*MOnotonic Hull-Consistency*, `Mohc`) est un nouvel algorithme de propagation de contraintes qui exploite la monotonie des fonctions pour mieux contracter une boîte. La boucle de propagation est exactement le

même algorithme de type `AC3` mis en œuvre par `HC4` et `Box`. Sa nouveauté réside dans la procédure `Mohc-Revise` traitant une contrainte $f(V) = 0$ individuellement[1] et décrite à l'algorithme 1.

Algorithm 1 `Mohc-Revise` (**in-out** $[V]$; **in** f, V, ρ_{mohc}, τ_{mohc}, ϵ)

```
HC4-Revise (f(V) = 0, [V])
if MultipleOccurrences(V) and ρ_mohc[f] < τ_mohc then
    (X, Y, W, f_max, f_min, [G]) ← PreProcessing(f, V, [V])
    MinMaxRevise ([V], f_max, f_min, Y, W)
    MonotonicBoxNarrow ([V], f_max, f_min, X, [G], ε)
end if
```

`Mohc-Revise` commence par appeler la procédure bien connue et peu coûteuse `HC4-Revise`. Les procédures de contraction par monotonie (`MinMaxRevise` et `MonotonicBoxNarrow`) ne sont appelées que si V contient au moins une variable apparaissant plusieurs fois (fonction `MultipleOccurrences`). L'autre condition rend `Mohc-Revise` adaptatif. Cette condition dépend d'un paramètre utilisateur τ_{mohc} détaillé dans la partie suivante. Le second paramètre ϵ de `Mohc-Revise` est un ratio de précision utilisé par `MonotonicBoxNarrow`.

La procédure `PreProcessing` calcule le gradient de f. Le gradient est stocké dans le vecteur $[G]$ et utilisé pour répartir les variables de V en trois sous-ensembles X, Y et W :

- les variables de X sont monotones et ont des occurrences multiples dans f,
- les variables de Y n'ont qu'une seule occurrence dans f (elles peuvent être monotones),
- les variables w de W apparaissent plusieurs fois dans f et ne sont pas détectées monotones, c.-à-d. $0 \in [\frac{\partial f}{\partial w}]_N([V])$.

La procédure `PreProcessing` détermine aussi les deux fonctions f_{min} et f_{max}, introduites dans la définition 3, qui approximent f en utilisant sa monotonie.

Les deux procédures suivantes sont au cœur de `Mohc-Revise` et sont détaillées plus loin. Utilisant les monotonies de f_{min} et f_{max}, `MinMaxRevise` contracte $[Y]$ et $[W]$ tandis que `MonotonicBoxNarrow` contracte $[X]$.

`HC4-Revise`, `MinMaxRevise` et `MonotonicBoxNarrow` calculent quelquefois une boîte vide $[V]$, prouvant alors l'absence de solution. Une exception terminant la procédure est alors levée.

A la fin, si `Mohc-Revise` a contracté un intervalle de $[W]$ (de plus qu'un ratio donné par le paramètre utilisateur τ_{propag}), alors la contrainte est mise dans la queue de propagation pour être traitée de nouveau par un appel suivant à `Mohc-Revise`. Sinon, nous savons qu'un point fixe en terme de filtrage a été atteint (voir lemmes 2 et 4).

[1] La procédure peut facilement être étendue pour traiter une inégalité.

3.1 La procédure MinMaxRevise

Nous savons que :

$(\exists X \in [X])(\exists Y \in [Y])(\exists W \in [W]) : f(X \cup Y \cup W) = 0 \implies f_{min}(Y \cup W) \leq 0 \;\; et \;\; 0 \leq f_{max}(Y \cup W)$

La contraction apportée par `MinMaxRevise` est simplement obtenue en appelant `HC4-Revise` sur les contraintes $f_{min}(Y \cup W) \leq 0$ et $0 \leq f_{max}(Y \cup W)$ pour réduire les intervalles des variables de Y et W (voir l'algorithme 2).

Algorithm 2 `MinMaxRevise` (**in-out** $[V]$; **in** f_{max}, f_{min}, Y, W)

```
HC4-Revise(f_min(Y∪W) ≤ 0, [V]) /* MinRevise */
HC4-Revise(f_max(Y∪W) ≥ 0, [V]) /* MaxRevise */
```

La figure 1 illustre comment `MinMaxRevise` contracte la boîte $[x] \times [y] = [4, 10] \times [-80, 14]$ par rapport à la contrainte : $f(x, y) = x^2 - 3\,x + y = 0$.

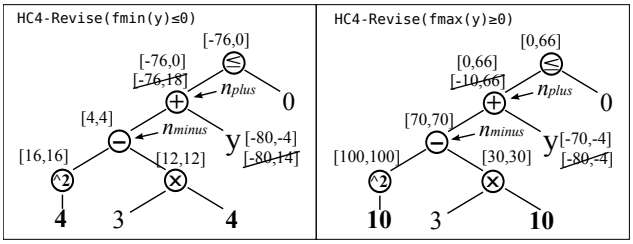


FIG. 1 – `MinRevise` (à gauche) et `MaxRevise` (à droite) appliquées à $x^2 - 3\,x + y = 0$.

La figure 1-gauche montre la première étape de `MinMaxRevise`. L'arbre représente l'inégalité $f(4, y) = f_{min}(y) \leq 0$. `HC4-Revise` procède en deux phases. La phase d'évaluation évalue chaque nœud de bas en haut avec l'arithmétique d'intervalles et attache le résultat au nœud. La seconde phase, à cause de l'inégalité, commence par intersecter l'intervalle du haut $[-76, 18]$ avec $[-\infty, 0]$ et, si le résultat est non vide, redescend dans l'arbre en appliquant *les fonctions de projection ("inverses")*. Par exemple, comme $n_{plus} = n_{minus} + y$, la fonction inverse de cette somme est la différence $[y] \leftarrow [y] \cap ([n_{plus}] - [n_{minus}]) = [-80, 14] \cap ([-76, 0] - [4, 4]) = [-80, -4]$. Suivant le même principe, `MaxRevise` applique `HC4-Revise` à $f(10, y) = f_{max}(y) \geq 0$ et réduit $[y]$ à $[-70, -4]$ (cf. la figure 1-droite).

Notons qu'un appel direct à `HC4-Revise` sur la contrainte $x^2 - 3\,x + y = 0$ (sans utiliser la monotonie de f) n'aurait apporté aucune contraction à $[x]$ ni à $[y]$.

3.2 La procédure MonotonicBoxNarrow

Cette procédure effectue une boucle sur chaque variable monotone x_i de X pour contracter $[x_i]$.

À chaque itération, elle travaille avec deux fonctions sur intervalles, dans lesquelles toutes les variables de X, sauf x_i, ont été remplacées par une borne de l'intervalle correspondant :

$$[f^{x_i}_{min}](x_i) = [f]_N(x_1^-, ..., x_{i-i}^-, x_i, x_{i+1}^-, ..., x_n^-, [Y], [W])$$
$$[f^{x_i}_{max}](x_i) = [f]_N(x_1^+, ..., x_{i-i}^+, x_i, x_{i+1}^+, ..., x_n^+, [Y], [W])$$

Comme Y et W ont été remplacées par leurs domaines, $[f^{x_i}_{max}]$ et $[f^{x_i}_{min}]$ sont des fonctions intervalles de la seule variable x_i (cf. la figure 2).

`MonotonicBoxNarrow` appelle deux sous-procédures :

- Si x_i est croissante, alors sont appelées :
  - `LeftNarrowFmax` sur $[f^{x_i}_{max}]$ pour améliorer $\underline{x_i}$,
  - `RightNarrowFmin` sur $[f^{x_i}_{min}]$ pour améliorer $\overline{x_i}$.
- Si x_i est décroissante, alors sont appelés :
  - `LeftNarrowFmin` sur $[f^{x_i}_{min}]$ pour améliorer $\underline{x_i}$,
  - `RightNarrowFmax` $[f^{x_i}_{max}]$ pour améliorer $\overline{x_i}$.

Nous détaillons dans l'algorithme 3 comment la borne gauche de $[x]$ est améliorée par la procédure `LeftNarrowFmax` qui utilise $[f^x_{max}]$.

Algorithm 3 `LeftNarrowFmax` (**in-out** $[x]$; **in** $[f^x_{max}]$, $[g]$, ϵ)

if $\overline{[f^x_{max}]_N(\underline{x})} < 0$ /* test d'existence */ **then**
  $size \leftarrow \epsilon \times$ `Diam`$([x])$
  $[l] \leftarrow [x]$
  while `Diam`$([l]) > size$ **do**
    $x_m \leftarrow$ `Mid`$([l])$; $z_m \leftarrow \overline{[f^x_{max}](x_m)}$
    /* $z_m \leftarrow \underline{[f^x_{min}](x_m)}$ dans `{Left|Right}NarrowFmin` */
    $[l] \leftarrow [l] \cap x_m - \frac{z_m}{[g]}$ /* itération de Newton */
  end while
  $[x] \leftarrow [\underline{l}, \overline{x}]$
end if

Le processus est illustré par la fonction représentée graphiquement à la figure 2. Le but est de contracter $[l]$ (initialisé à $[x]$) pour obtenir un encadrement étroit du point L. L'utilisateur spécifie le paramètre de précision ϵ (comme un rapport de diamètres d'intervalles) donnant la qualité de l'approximation. `LeftNarrowFmax` conserve seulement $\underline{l}$ à la fin, comme le montrent la dernière ligne de l'algorithme 3 et l'étape 4 à la figure 2.

Un test d'existence préliminaire vérifie que $\overline{[f^x_{max}]_N(\underline{x})} < 0$, c.-à-d. que le point A sur la figure 2 est en dessous de zéro. Sinon, $[f^x_{max}]_N \geq 0$ est satisfait en $\underline{x}$ et $[x]$ ne peut être réduit, ce qui aboutit à la terminaison de la procédure. Nous effectuons un processus dichotomique jusqu'à ce que l'on obtienne

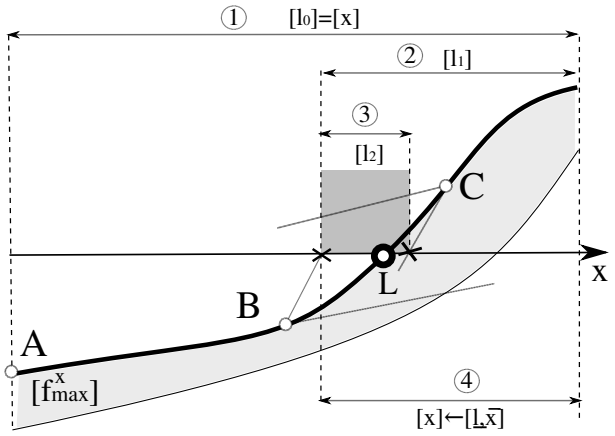


FIG. 2 – Itérations de Newton sur intervalles pour réduire $\underline{x}$.

`Diam`$([l]) \leq size$. Des itérations de l'algorithme classique *Newton sur intervalles* univarié sont lancées à partir du point milieu x_m de $[l]$, comme sur la figure 2 :

1. à partir du point B (milieu de $[l_0]$, c.-à-d., $[l]$ à l'étape 0) et
2. à partir du point C (milieu de $[l_1]$).

Graphiquement, une itération de *Newton sur intervalles* univarié intersecte $[l]$ avec la projection sur l'axe x d'un cône (deux demi-droites à partir de B, puis de C). Les pentes de ces droites sont égales aux bornes de la dérivée partielle $[g] = [\frac{\partial f^x_{max}}{\partial x}]_N([x])$. Notons que le cône forme un angle d'au plus 90 degrés car la fonction est monotone et $[g]$ est positif. Ceci explique pourquoi `Diam`$([\mathtt{l}])$ est divisé au moins par 2 à chaque itération.

Lemme 1 *Soit ϵ la précision exprimée comme un rapport de diamètres d'intervalles. Alors,* `LeftNarrowFmax` *et les procédures symétriques terminent et s'exécutent en un temps en $O(log(\frac{1}{\epsilon}))$.*

Observons que les itérations de Newton appelées à l'intérieur de `LeftNarrowFmax` et `RightNarrowFmax` travaillent sur $z_m = \overline{[f^x_{max}](x_m)}$, qui est une courbe (en gras sur la figure), et non la fonction sur intervalles $[f^x_{max}](x_m)$.

3.3 Points notables de Mohc-Revise

Comment rendre Mohc-Revise adaptatif

Le paramètre utilisateur $\tau_{mohc} \in [0, 1]$ permet d'appeler plus ou moins souvent, au cours de la recherche arborescente, les procédures utilisant la monotonie (voir l'algorithme 1). Pour chaque contrainte, les procédures exploitant la monotonie de f ne sont appelées que si $\rho_{mohc}[f] < \tau_{mohc}$. Cette condition demande que l'image d'une fonction par l'évaluation par monotonie

soit plus étroite que celle donnée par l'évaluation naturelle d'un facteur $\rho_{mohc}[f]$ au moins égal au paramètre τ_{mohc} :

$$\rho_{mohc}[f] = \frac{Diam([f]_M([V]))}{Diam([f]_N([V]))}$$

Comme nos expérimentations le confirment, ce ratio est pertinent pour les phases d'évaluation de `MinRevise` et `MaxRevise`, et aussi pour `MonotonicBoxNarrow` qui effectue beaucoup d'évaluations.

ρ_{mohc} est calculé dans une procédure de prétraitement appelée après chaque bissection. Comme de plus en plus de cas de monotonie apparaissent au fur et à mesure que l'on descend dans l'arbre de recherche (les boîtes devenant plus petites), `Mohc-Revise` est capable d'activer de manière adaptative la machinerie liée à la monotonie. `Mohc-Revise` apparaît ainsi comme la première procédure de révision adaptative en programmation par contraintes (sur intervalles).

Le regroupement d'occurrences pour utiliser plus de monotonie

Un appel à une nouvelle procédure appelée *Regroupement d'occurrences* a en fait été ajouté à `Mohc-Revise` juste après le prétraitement. Si f n'est *pas* monotone par rapport à une variable x, il est toutefois possible que f soit monotone par rapport à un sous-groupe des occurrences de x. Pour trouver de tels sous-groupes d'occurrences croissantes et décroissantes, cette procédure utilise une approximation de f basée sur l'évaluation de Taylor et résout à la volée un programme linéaire. Ceci permet d'améliorer l'évaluation par monotonie de f. Des détails et une évaluation expérimentale se trouvent dans [2].

4 Propriétés

Proposition 2 (Complexité en temps)
Considérons une contrainte c comprenant n variables et e opérateurs unaires et binaires ($n \leq e$). Soit ϵ la précision exprimée comme un ratio de diamètres d'intervalles. Alors,
la complexité en temps de Mohc-Revise est en $O(n\, e\, log(\frac{1}{\epsilon})) = O(e^2\, log(\frac{1}{\epsilon}))$.

La complexité en temps est dominée par `MonotonicBoxNarrow` (voir le lemme 1). Un appel à `HC4-Revise` et un calcul du gradient sont tous deux en $O(e)$ [3].

Proposition 3 *Soit $c : f(X) = 0$ une contrainte telle que f est continue, dérivable et monotone par rapport à chaque variable dans la boîte $[X]$. Alors, avec une précision ϵ,* `MonotonicBoxNarrow` *calcule la hull-consistance de c.*

On peut trouver les preuves dans [1] et [5]. Cependant, la nouvelle proposition 4 ci-après est plus forte car les variables apparaissant une seule fois (Y) sont traitées par `MinMaxRevise` et non par `MonotonicBoxNarrow`.

Proposition 4 *Soit $c : f(X, Y) = 0$ une contrainte, où les variables dans Y apparaissent une seule fois dans f. Si f est continue, dérivable et monotone par rapport à chaque variable dans la boîte $[X \cup Y]$, alors, avec une précision ϵ,* `Mohc-Revise` *calcule la hull-consistance of c.*

On peut trouver la démonstration complète dans [1]. Il est aussi prouvé qu'aucune hypothèse de monotonie n'est requise pour les variables de Y, à condition que `Mohc-Revise` utilise la variante combinatoire `TAC-Revise` [6] de `HC4-Revise`.

Les lemmes 2, 3 et 4 ci-dessous permettent de démontrer les propositions 3 et 4. Ils démontrent aussi la correction de `Mohc-Revise`.

Lemme 2 *Quand* `MonotonicBoxNarrow` *réduit l'intervalle d'une variable $x_i \in X$ en utilisant $[f^{x_i}_{max}]$ (resp. $[f^{x_i}_{min}]$), alors, pour tout $j \neq i$, $[f^{x_j}_{min}]$ (resp. $[f^{x_j}_{max}]$) ne peut pas apporter de contraction additionnelle à l'intervalle $[x_j]$.*

Le lemme 2 est une généralisation de la proposition 1 de [5] aux fonctions sur intervalles ($[f^{x_i}_{max}]$ et $[f^{x_i}_{min}]$).

Lemme 3 *Si $0 \in [z] = [f_{max}]([Y \cup W])$ (resp. $0 \in [z] = [f_{min}]([Y \cup W])$), alors* `MonotonicBoxNarrow` *ne peut pas contracter un intervalle $[x_i]$ ($x_i \in X$) en utilisant $[f^{x_i}_{min}]$ (resp. $[f^{x_i}_{max}]$).*

Lemme 4 *Si* `MonotonicBoxNarrow` *(suivant un appel à* `MinMaxRevise`*) contracte $[x_i]$ (avec $x_i \in X$), alors un second appel à* `MinMaxRevise` *ne peut pas contracter davantage $[Y \cup W]$.*

Les lemmes 2 et 4 expliquent pourquoi il n'est pas nécessaire d'effectuer de boucle dans `MohcRevise` pour atteindre un point fixe en termes de filtrage.

4.1 Démonstrations des lemmes 3 et 4

La figure 3 illustre ces démonstrations dans le cas où f est croissante. Nous distinguons deux cas suivant la borne droite initiale de l'intervalle $[x_i]$.

Dans le lemme 3, nous avons $0 \in [z] = [f_{max}]([Y \cup W])$, c.-à-d. $\underline{z} \leq 0 \leq \overline{z}$. Cette condition est en particulier vraie quand `MaxRevise` apporte une contraction. On peut vérifier que $\overline{x_i}$ ne peut être amélioré par `RightNarrowFmin` : puisque $\overline{x_i}$ est une solution de $[f^{x_i}_{max}](x_i) = 0$ (le segment en gras sur la figure 3),

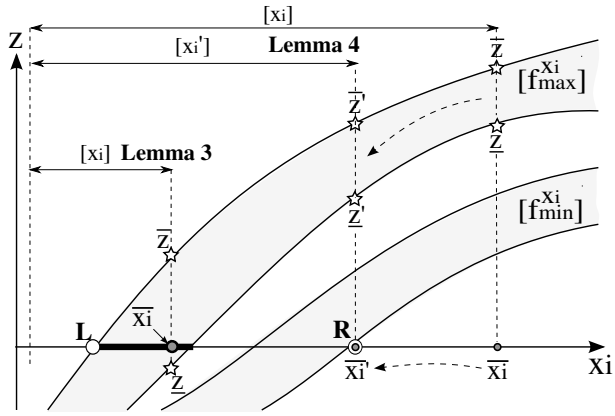


FIG. 3 – Démonstration des lemmes 3 et 4 montrant la dualité de `MinMaxRevise` et `MonotonicBoxNarrow`. $[z]$ est l'image de $[f_{max}]$ obtenue par la phase d'évaluation de `MaxRevise`.

$\overline{x_i}$ satisfait aussi la contrainte $[f^{x_i}_{min}](x_i) \leq 0$ qui est utilisée par `RightNarrowFmin`.

Dans le lemme 4, nous avons $0 < [z] = [f_{max}]([Y \cup W])$ (`MaxRevise` n'apporte pas de contraction). Après la contraction réalisée par `RightNarrowFmin`, la borne droite de l'intervalle devient $\overline{x'_i}$. Une nouvelle évaluation de $[f_{max}]([Y \cup W])$ donne $[z']$ qui est encore au dessus de 0, de telle sorte qu'un deuxième appel à `MaxRevise` n'apporterait pas de contraction additionnelle. □

4.2 Amélioration de MonotonicBoxNarrow

Finalement, les lemmes 2 et 3 apportent des conditions simples pour éviter des appels à `LeftNarrowFmax` (et aux procédures symétriques) à l'intérieur de `MonotonicBoxNarrow`.

Avec l'ajout de ces conditions, comme le montrent les tests détaillés dans [1], 35% du temps CPU de `Mohc-Revise` est passé dans `MinMaxRevise` alors que seulement 9% est passé dans la procédure plus coûteuse `MonotonicBoxNarrow` (entre 1% et 18% selon l'instance).

5 Expérimentations

Nous avons implanté `Mohc` dans la bibliothèque en `C++ Ibex` [4] de résolution par intervalles. Tous les algorithmes concurrents, `HC4`, `Box`, `Octum` [5], `3BCID(HC4)`, `3BCID(Box)` et `3BCID(Octum)` sont aussi disponibles dans `Ibex`, ce qui rend la comparaison équitable.

`Mohc` et ses compétiteurs ont été testés sur la même machine `Intel 6600 2.4 GHz` sur 17 NCSP avec un nombre fini de solutions ponctuelles disponibles sur le site de COPRIN[2]. Nous avons sélectionné tous les NCSP avec des variables à occurrences multiples qui se trouvent dans les deux premières parties (systèmes *polynomiaux* et *non polynomiaux*) du site. Nous avons ajouté `Brent`, `Butcher`, `Direct Kin.` et `Virasaro` qui proviennent de la partie appelée *problèmes difficiles*.

Toutes les stratégies de résolution utilisent comme choix de variable le tour de rôle. Entre deux branchements dans l'arbre de recherche, trois procédures sont appelées en séquence. D'abord, un test d'existence utilisant la monotonie, amélioré par le regroupement d'occurrences, vérifie que l'image de chaque fonction contient zéro[3]. Ensuite, le contracteur est appelé : `Mohc`, `3BCID(Mohc)`, ou l'un des concurrents cités. Enfin, l'algorithme *Newton sur intervalles* est appelé si la boîte courante a un diamètre de 10 ou moins. Tous les paramètres ont été fixés à des valeurs par défaut. Le ratio de précision du rognage dans `3B` et `Box` est 10% ; une contrainte est mise dans la queue de propagation si l'intervalle d'une de ses variables est réduit de plus de plus de $\tau_{propag} = 1\%$ avec tous les contracteurs sauf `3BCID(HC4)` et `3BCID(Mohc)` (10%). Pour `Mohc`, le paramètre τ_{mohc} a été fixé à 70% ou 99%.
ϵ vaut 3% dans `Mohc` et 10% dans `3BCID(Mohc)`.

5.1 Résultats

Le tableau 1 compare le temps CPU utilisé et le nombre de points de choix obtenus par `Mohc` et `3BCID(Mohc)` avec ceux obtenus par leurs concurrents. La dernière colonne donne le gain obtenu par `Mohc`, c.-à-d. $Gain = \frac{tempsCPU\ (meilleur\ compétiteur)}{tempsCPU\ (meilleure\ stratégie\ de\ type\ Mohc)}$

Le tableau montre les très bons résultats obtenus par `Mohc`, en termes de pouvoir de filtrage (petit nombre de points de choix) et de temps CPU. Les résultats obtenus par `3BCID(Box)`, `Octum` et `3BCID(Octum)` ne sont pas indiqués car ces méthodes ne se sont pas avérées compétitives avec `Mohc`. Par exemple, `Octum` est moins rapide que `Mohc` d'un ordre de grandeur. La supériorité de `Mohc` sur `Box` montre qu'il vaut mieux un plus grand effort de filtrage (appel à BoxNarrow) moins souvent, c.-à-d. quand on a détecté qu'une variable était monotone. `Mohc` et `HC4` obtiennent des résultats similaires sur 9 des 17 problèmes testés. Avec $\tau_{mohc} = 70\%$, on peut noter que la perte en performance de `Mohc` (resp. `3BCID(Mohc)`) par rapport à `HC4` (resp. `3BCID(HC4)`) sur ces problèmes est négligeable. Elle est inférieure à 5%, sauf pour `Katsura` (25%).

Sur 6 NCSP, `Mohc` obtient un gain compris entre 2.4 et 8. Sur `Butcher` et `Direct kin.`, on observe même un gain de resp. 163 et 49. Sans le test d'existence par

[2] www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html

[3] Ce test d'existence ne prend qu'une petite part du temps total (en général moins de 10% et moins de 1% pour `3BCID`), tout en améliorant grandement la performance des concurrents.

TAB. 1 – Résultats expérimentaux. La première colonne inclut le nom du système, le nombre d'équations et le nombre de solutions. Les autres colonnes donnent le temps CPU en secondes (en haut) et le nombre de points de choix (en bas) pour tous les compétiteurs. Les meilleurs temps sont en gras.

NCSP	HC4	Box	3BCID(HC4)	Mohc 70%	Mohc 99%	3BCID(Mohc) 70%	3BCID(Mohc) 99%	Gain
Butcher 8 3	>4e+5	>4e+5	282528 1.8e+8	>4e+5	>4e+5	5431 2.2e+6	**1722** 288773	163 623
Directkin. 11 2	>2e+4	>2e+4	17507 1.4e+6	2560 777281	2480 730995	428 8859	**356** 5503	49.1 253
Virasoro 8 224	>2e+4	>2e+4	7173 2.5e+6	1180 805047	1089 715407	1051 71253	**897** 38389	8.00 66.8
Yamam.1 8 7	32.4 29513	12.6 3925	11.7 3017	19.2 24767	27.0 29973	**2.2** 345	2.87 295	5.30 10.2
Geneig 6 10	1966 4.1e+6	3721 1.3e+6	390 161211	463 799439	435 655611	107 13909	**81.1** 6061	4.81 26.6
Hayes 8 1	163 541817	323 214253	41.6 17763	30.9 73317	27.6 49059	17.0 4375	**13.8** 1679	3.02 10.6
Trigo1 10 9	93 5725	332 6241	151 2565	**30** 1759	30.6 1673	57.7 459	73.2 443	3.10 5.79
Fourbar 4 3	863 1.6e+6	2441 1.1e+6	1069 965343	361 437959	**359** 430847	366 58571	373 45561	2.40 21.2
Pramanik 3 2	26.9 103827	91.9 81865	35.9 69259	30.3 87961	25.0 69637	**20.8** 12691	21.3 8429	1.29 8.22
Caprasse 4 18	2.04 7671	11.5 5957	2.73 1309	**1.87** 4577	2.69 3741	2.64 867	4.35 383	1.09 3.42
Kin1 6 16	6.91 1303	26.9 689	1.96 87	5.68 1055	5.65 931	**1.79** 83	3.43 83	1.09 1.05
Redeco8 8 8	3769 1.0e+7	9906 7.9e+6	6.28 2441	3529 6.8e+6	2936 4.6e+6	**6.10** 2211	10.65 1489	1.03 1.64
Trigexp2 11 0	1610 1.6e+6	>2e+4	**86.9** 14299	1507 1417759	1027 935227	**87** 14299	165 7291	1.00 1.96
Eco9 9 16	39.9 115445	94.1 110423	**13.9** 6193	46.8 97961	44.2 84457	**14.0** 6025	26.6 4309	0.99 1.44
I5 10 30	9310 2.4e+7	>2e+4	**55.9** 10621	7107 1.6e+7	7129 1.5e+7	57.5 9773	84.1 8693	0.97 1.22
Brent 10 1008	497 1.8e+6	151 23855	**18.9** 3923	244 752533	232 645337	19.9 3805	41.4 3189	0.95 1.23
Katsura 12 7	182 271493	2286 251727	**77.8** 4251	106 98779	143 94249	104 3573	251 3471	0.75 1.22

monotonie appelé avant les contracteurs, on a obtenu un gain de 37 sur le système `Fourbar`.

En conclusion, la combinaison `3BCID(Mohc)` semble être très prometteuse.

5.2 Travaux connexes

Un algorithme de propagation de contraintes exploitant la monotonie se trouve dans le résolveur par intervalles ALIAS[4]. Sa procédure de révision n'utilise pas un arbre pour représenter une expression f (contrairement à `HC4-Revise`). A la place, une fonction de projection f^o_{proj} est créée pour réduire l'intervalle de chaque occurrence o et est évaluée avec une extension par monotonie $[f^o_{proj}]_M$. C'est plus coûteux que `MinMaxRevise` et n'est pas optimal puisqu'aucune procédure `MonotonicBoxNarrow` n'est utilisée.

L'article [5] décrit un algorithme de propagation de contraintes appelé `Octum`. `Mohc` et `Octum` ont été conçus indépendamment pendant le premier semestre de 2009. Pour le décrire rapidement, `Octum` appelle `MonotonicBoxNarrow` quand *toutes* les variables de la contrainte sont monotones. Comparé à `Octum` :

- `Mohc` ne demande pas qu'une fonction soit monotone par rapport à toutes ses variables simultanément ;
- `Mohc` utilise `MinMaxRevise` pour contracter rapidement les intervalles des variables (de Y) apparaissant une seule fois (voir la proposition 4) ;
- `Mohc` utilise un algorithme de regroupement d'occurrences pour détecter plus de cas de monotonie.

Une première étude expérimentale (non décrite ici) montre que la bien meilleure performance de `Mohc` est due principalement à la condition établie dans le lemme 3 (et testée durant `MinMaxRevise`), qui permet d'éviter des appels à `LeftNarrowFmax` et à ses procédures symétriques.

[4] www-sop.inria.fr/coprin/logiciels/ALIAS/ALIAS.html

6 Conclusion

Cet article a présenté un algorithme de propagation de contraintes utilisant la monotonie des fonctions. En utilisant les ingrédients présents dans les procédures existantes `HC4-Revise` et `BoxNarrow`, `Mohc` a le potentiel pour remplacer avantageusement `HC4` et `Box`, comme le montrent nos premières expérimentations. De plus, `3BCID(Mohc)` semble constituer une combinaison très prometteuse.

La procédure `Mohc-Revise` utilise deux paramètres utilisateurs, notamment τ_{mohc} pour régler la sensibilité à la monotonie. Un travail futur consistera à rendre `Mohc-Revise` auto-adaptatif en permettant de régler automatiquement τ_{mohc} durant la recherche combinatoire.

Références

[1] I. Araya. *Exploiting Common Subexpressions and Monotonicity of Functions for Filtering Algorithms over Intervals.* PhD thesis, Université de Nice–Sophia, 2010.

[2] I. Araya, B. Neveu, and G. Trombettoni. Une nouvelle extension de fonctions aux intervalles basée sur le regroupement d'occurrences. In *Proc. JFPC*, 2010.

[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. `www.ibex-lib.org`, 2010.

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

[6] G. Chabert, G. Trombettoni, and B. Neveu. Box-Set Consistency for Interval-based Constraint Problems. In *Proc. SAC (ACM)*, pages 1439–1443, 2005.

[7] H. Collavizza, F. Delobel, and M. Rueher. Comparing Partial Consistencies. *Reliable Computing*, 5(3) :213–228, 1999.

[8] A. Goldsztejn, C. Michel, and M. Rueher. Efficient Handling of Universally Quantified Inequalities. *Constraints*, 14(1) :117–135, 2009.

[9] M. Kieffer, L. Jaulin, E. Walter, and D. Meizel. Robust Autonomous Robot Localization Using Interval Analysis. *Reliable Computing*, 3(6) :337–361, 2000.

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

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

[12] J-P. Merlet. Interval Analysis and Robotics. In *Symp. of Robotics Research*, 2007.

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

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

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

Modélisation par contraintes et exploitation d'une gamme automobile : nouveaux problèmes, nouvelles requêtes, nouveaux besoins en programmation par contraintes

Jean Marc Astesana[1] **Laurent Cosserat**[1] **Hélène Fargier**[2]

[1] Renault SA, 13 avenue Paul Langevin 92359 Plessis Robinson
[2] IRIT-CNRS, 118 route de Narbonne 31062 Toulouse Cedex 9
{jean-marc.astesana,laurent.cosserat}@renault.com fargier@irit.fr

Abstract

La mise au point et l'exploitation de la gamme commerciale d'un constructeur automobile présentent de multiples problèmes métier qui peuvent se modéliser comme des problèmes CSP : détection et analyse d'incohérence dans la spécification de la gamme, configuration véhicule, configuration de la prévision. Ces problèmes mettent en jeu des données de taille importante, et leur résolution en situation réelle doit être effectuée dans un temps de calcul limité. Cet article présente, formalise et étudie la complexité de plusieurs de ces problèmes. Au-delà de l'étude d'une application réelle, son objectif est de proposer à la communauté CSP non pas de nouvelles approches de résolution, mais de nouvelles requêtes, de nouveaux problèmes, voire de nouveaux besoins.

1 Introduction

Le formalisme des problèmes de satisfaction de contraintes (CSP) offre un cadre de formalisation puissant pour l'expression d'une multitude de problèmes, en particulier de nombreux problèmes académiques ou issus du monde réel (par exemple les problèmes de coloration de graphes, d'ordonnancement, d'affectation de fréquence, . . .). Résoudre un CSP c'est, quasiment toujours, déterminer s'il est cohérent et/ou lui trouver une solution -dans certains cas une solution optimale- et c'est une question difficile : le problème de décision associé est NP-complet. La grande majorité des travaux de la communauté a donc porté sur ce problème, soit en l'abordant de front (proposer les algorithmes les plus efficaces possibles pour déterminer la cohérence d'un CSP), soit en cherchant comment le contourner, par l'étude de sous-classes qui seraient polynomiales ou par la définition d'algorithmes sains mais incomplets.

Or si les CSP, ou les langages de programmation par contraintes, sont largement utilisés, ce n'est pas seulement parce qu'ils sont munis d'algorithmes de résolution efficaces. C'est aussi parce qu'ils permettent d'exprimer simplement les données des applications modélisées. Mais dans ces applications, la question n'est pas toujours de savoir si le CSP est cohérent ni même d'en trouver une solution. Les applications en configuration de produit, par exemple, définissent des CSP cohérents (pourvus de centaines de milliers de solutions) ... et la machine n'est pas utilisée pour extraire une solution, mais pour aider l'utilisateur à construire la sienne.

L'ambition de cet exposé est de proposer à la communauté non pas de nouvelles approches de résolution, mais de nouvelles requêtes, de nouveaux problèmes, voire de nouveaux besoin en modélisation, en exposant quelques problèmes métier de Renault qui ont pour point commun la nécessité d'inférer sur le système de contraintes représentant la gamme, et notamment la configuration de véhicules automobiles.

2 Modélisation à base de contraintes de la gamme Renault

La gamme de véhicules de Renault est très étendue. Le nombre de véhicules différents dans la gamme d'un modèle peut être énorme. Par exemple, l'ensemble des petites fourgonnettes "Trafic" comprend 10^{21} véhicules différents. Ce nombre rend impossible l'énumération de l'ensemble de ces véhicules dans une base

de données. Cet ensemble doit alors être représenté en intention. La gamme peut être vue comme spécifiée par modèle, et dans la suite de cet article on entend par "gamme" l'ensemble de toutes les différentes variantes possibles d'un modèle. Les véhicules sont définis par des variables (*carburant*, *couleur*, etc.) qui prennent des valeurs dans des domaines énumérés ($\{essence, diesel, gpl\}$, $\{blanc, noir, rouge\}$, etc.). Les choix possibles pour ces variables ne sont pas indépendants les uns des autres (typiquement, un niveau d'équipement donné n'est pas disponible sur toutes les motorisations) : des dépendances sont modélisées par un système de contraintes entre ces variables. On a ainsi tous les ingrédients d'un problème CSP, dont chaque solution définit un véhicule précis de la gamme.

Plusieurs travaux ont proposé d'utiliser des extensions des CSP pour formaliser des problèmes de configuration, traitant généralement de difficultés de représentation spécifiques à ces types de problèmes, comme par exemple le fait que l'existence d'une variable puisse dépendre la valeur affectée à une autre. L'étude des problèmes de configuration a ainsi poussé la communauté à étendre le modèle CSP de base, définissant ainsi les CSP dynamiques [8], CSP composites [10], CSP interactifs [7], CSP à hypothèses [1], etc. Dans cet article, nous négligerons la prise en compte de ces difficultés de représentation pour nous tourner plutôt vers les requêtes adressées au modèle, vers la question de l'exploitation des données. *On supposera donc que la gamme est représentée par un CSP "classique".*

Les notations et définitions précises nécessaires à la formalisation de l'ensemble des questions étudiées ici seront introduites en Section 4. Disons pour l'instant simplement qu'un CSP est un triplet $P =< X, D, C >$ où X est l'ensemble de ses variables, D l'ensemble des domaines respectifs de ces variables et C l'ensemble (supposé fini) de ses contraintes ; on désignera par $Sols(P)$ l'ensemble des solutions de P.

Comme la plupart des instances issues de problèmes réels, les CSP représentant les modèles de la gamme Renault possèdent des spécificités particulières, en termes sémantiques (type de variables et contraintes) et syntaxiques (forme du graphe de contraintes).

La gamme de Renault est construite suivant ce qu'on appelle le "modèle version-options". Selon ce modèle, les variables se répartissent en deux catégories :

- un petit nombre de variables servant à définir les versions : appelons-les variables majeures. Soit $Maj \subset X$ leur ensemble : pour fixer les idées, on peut considérer que la restriction de $Sols(P)$ aux variables de Maj définit l'ensemble des versions de la gamme. Une variable encodant précisément la version peut exister, mais ce n'est pas systématiquement le cas : dans le cas contraire, la notion de version n'existe qu'implicitement, comme combinaison de valeurs de variables majeures.
- le reste des variables, la grande majorité, dont les valeurs possibles dépendent de la version : ce sont les variables mineures éléments de Min. Maj et Min forment une partition de X. Une valeur du domaine d'une variable mineure pourra ainsi être impossible sur une version, optionnelle sur une autre et obligatoire sur une troisième.

On peut regrouper comme suit les contraintes de spécification de la gamme :

- Les *contraintes majeures* portent sur des variables majeures qui définissent l'ensemble des versions. Typiquement, une contrainte majeure est directement définie par une Table, c'est-à-dire par l'énumération explicite des combinaisons autorisées de valeurs des variables de la contrainte.
- Les *contraintes options* sont des contraintes de la forme "$(x = v) \Rightarrow (y \in A)$", x étant une variable majeure, v une valeur de son domaine, y une variable mineure, et A un sous-ensemble du domaine de x. Par exemple, on peut utiliser ce type de contrainte pour spécifier la liste des autoradios disponibles sur les versions haut de gamme.
- Les *contraintes packs*. Il arrive qu'un lot de plusieurs options soit regroupé en une seule : un pack (ces options s'appellent alors les composants du pack). Les packs donnent lieu à de nombreuses contraintes qui complexifient le CSP. Par exemple, la contrainte de définition d'un pack exprime que le choix d'un pack *pack* implique la sélection de la totalité de ses n composants opt_i : $(x = pack) \Rightarrow \wedge_{i=1}^{n}(x_i = opt_i)$. De plus, une même option peut être un composant de plusieurs packs, qui s'excluent alors mutuellement.
- Les *contraintes quelconques*. Il reste des contraintes ne faisant pas partie des catégories précédentes. Elles peuvent inclure des variables majeures et mineures.

Notons que certaine contraintes, comme les contraintes options, s'expriment par des combinaisons booléennes de conditions ($variable = valeur$) ou de conditions ($variable \in$ sous-domaine) ; par exemple la contrainte $(carburant = diesel) \wedge (type\_boite = boite\_auto) \Rightarrow (type\_chauffage = clim\_regulee)$. On parle alors d'expression booléenne[1].

Dans les paramètres les plus dimensionnants des

[1] Il est bien entendu possible de modéliser la gamme de manière entièrement booléenne et d'obtenir ainsi des formules de la logique propositionnelle ; plusieurs codages sont possibles - voir par ex [13], mais pour des raisons de lisibilité, c'est souvent l'encodage direct qui est utilisé : pour chaque variable *var* qui admet les valeurs possibles $val_1, ..., val_n$ on définit n variables booléennes $val_1, ..., val_n$. Ces variables sont classiquement liées par des contraintes d'exhaustivité et d'exclusivité.

problèmes CSP définissant une gamme, on peut citer :

- La combinatoire des variables majeures. En configuration commerciale il y a au plus une centaine de versions, mais sur des gammes dites techniques, la projection de $Sols(P)$ sur les variables majeures peut en comporter plusieurs dizaines de milliers.
- le nombre de variables mineures et majeures. On peut supposer que $card(Min) \leq 150$. Le nombre de variables majeures est de l'ordre de 10.

La complexité est aggravée par les packs : au pire on peut en avoir une dizaine, chacun impliquant typiquement de 2 à 5 options composantes.

Mais l'essentiel de la difficulté vient des contraintes quelconques. Contrairement aux contraintes options, qui relient les variables de manière essentiellement hiérarchique, ces contraintes peuvent porter sur n'importe quelles variables, et donc complexifier le graphe de contraintes. Il peut y en avoir jusqu'à une centaine : elles portent généralement sur moins de quatre variables (sur deux variables dans la moitié des cas).

3 Exploitation de la gamme

Nous décrivons maintenant les requêtes métier qui se posent chez Renault lors des tâches d'élaboration, la maintenance et l'exploitation d'une gamme de véhicules - dit autrement, lors de la définition, la modification ou l'exploitation de ce CSP. Leur formalisation en termes de CSP sera abordée en Section 4.

3.1 Élaboration de la documentation véhicule

Les premières tâches interviennent en amont du problème de configuration client, en phase de modélisation de la gamme par un système de contraintes. On parle d'élaboration de la "documentation véhicule". Il s'agit de savoir si la gamme décrite par le système est cohérente, dans un sens étendu : dans certains cas, il ne suffit pas qu'il existe des véhicules compatibles avec les contraintes, il faut aussi que les possibilités spécifiées par les contraintes soient effectives. Supposons par exemple qu'une contrainte signifie "Les véhicules ayant le moteur M3 peuvent avoir comme type de chauffage : climatisation manuelle ou climatisation régulée." Si d'autres contraintes rendent incompatible le moteur M3 et la climatisation manuelle, la documentation véhicule est considérée comme incohérente. Autrement dit, certaines contraintes de la documentation véhicule ont une double sémantique : négative (elles interdisent des combinaisons de valeurs), et positive (les combinaisons autorisées par une contrainte doivent être effectivement possibles) : nous appelons cette propriété cohérence positive. C'est le cas des contraintes majeures et des contraintes options. Les questions qui se posent sont :

Cohérence (contextuelle) de la gamme : Y a-t-il au moins un véhicule répondant à cette définition de gamme ? Étant donnée une liste de quelques variables dont on impose la valeur, y a-t-il dans la gamme au moins un véhicule compatible avec cette affectation ? Plus généralement, étant donnée une expression booléenne c portant sur des paires $(variable = valeur)$, y a-t-il dans la gamme au moins un véhicule vérifiant c ?

Lorsque la réponse à ces questions est non, c'est que la documentation véhicule est erronée dans son état courant. Il faut alors analyser l'incohérence, c'est-à-dire déterminer un jeu minimal de contraintes qui suffit à expliquer l'impossibilité des véhicules prévus :

Conflits : Trouver un ensemble de contraintes incohérent, qui soit minimal pour l'inclusion.

Conflits contextuels : Étant donnée une liste de variables dont la valeur est imposée, trouver un ensemble de contraintes incohérent avec ces affectations qui soit minimal pour l'inclusion.

Cohérence positive Pour chaque contrainte à sémantique positive, et chaque tuple qu'elle autorise, existe il un véhicule de la gamme qui réalise ce tuple. De la même façon, pour chaque variable et chaque valeur de son domaine, existe-t-il un véhicule qui affecte cette valeur à cette variable.

3.2 Restitution de la gamme

Lorsque la gamme a été définie de manière cohérente, on cherche à en extraire des données significatives techniquement en considérant des ensembles de variables $\{x_1, \ldots, x_k\}$ (généralement, $k \leq 6$), et générant automatiquement des descriptions synthétiques des combinaisons des valeurs possibles de ces variables, typiquement sous la forme d'expressions booléennes sur des propositions de type $variable = valeur$.

Validité de description synthétique : Tester si tous les véhicules de la gamme vérifient une description synthétique c donnée.

Simplification de description synthétique : Une description synthétique c donnée est elle minimale (peut-on, sans supprimer une variable de c sans compromettre sa validité ?)

Génération de description synthétique : Étant donné un ensemble de variables, quelles sont les combinaisons de valeurs pour ces variables qui sont autorisées par la gamme (idéalement, on voudrait générer cette projection sous la forme d'une description synthétique minimale) ?

On cherche enfin à dimensionner la diversité de la gamme, en termes de nombres de véhicules différents :

Comptage : Combien de véhicules la gamme

compte-t-elle ? Combien de véhicules ayant un ensemble donné de valeurs imposées la gamme compte-t-elle ? Combien de combinaisons de valeurs différentes sont autorisées pour un sous-ensemble de variables Y : typiquement, on peut vouloir dimensionner la "diversité technique", c'est-à-dire le nombre de véhicules existant dans la gamme, hors considérations de teinte , appellation commerciale et pays de commercialisation.

3.3 Élaboration de la documentation pièces

La documentation pièces de Renault spécifie les pièces montées sur les véhicules de la gamme considérée. Elle est distincte de la documentation véhicule, dont les variables représentent des prestations perçues par les clients et non des pièces. Le lien est assuré par ce qu'on appelle le "cas d'emploi" : le cas d'emploi d'une pièce p_i est une expression booléenne c_i portant sur des atomes $variable = valeur$ de la documentation véhicule - cette contrainte spécifie les véhicules sur laquelle la pièce est montée. Par exemple, une pièce se monte sur les véhicules qui ont un moteur essence et une boîte automatique mais pas de toit ouvrant.

Les pièces sont regroupées en "pièces génériques". Une pièce générique, par exemple l'autoradio, est une fonction réalisée par la pièce. Pour que la documentation pièces soit cohérente, il faut que chaque véhicule ait un autoradio (propriété d'exhaustivité), qu'aucun véhicule n'en ait plusieurs (propriété de non-dualité) et que tout autoradio soit utilisé dans au moins un véhicule. On peut voir une pièce générique comme une variable supplémentaire p dont les valeurs sont les pièces associées $p_1, ..., p_n$. A chaque p_i correspond une expression booléenne c_i définissant son cas d'emploi. Les questions qui se posent alors sont :

Concision de la doc. : Pour chaque pièce associée (chaque cas d'emploi), existe-t-il au moins un véhicule de la gamme qui le valide ?

Exhaustivité de la doc. : Existe-t-il un véhicule ne montant aucune des pièces associées à une pièce générique donnée (ne validant aucun des cas d'emploi de la pièce générique) ?

Non-dualité des Emplois : Existe-t-il un véhicule montant deux des pièces, ou plus, associées à une pièce générique donnée (validant au moins deux des cas d'emplois de la pièce générique) ?

Si la réponse est "oui" à l'un des deux derniers points, on veut caractériser ces véhicules ou ces pièces :

Emplois Inconnus : Donner une condition sur les atomes "$variable = valeur$" qui caractérise l'ensemble des véhicules ne montant aucune des pièces associées à une pièce générique donnée ?

Emplois Multiples : Quelles sont les ensembles de pièces qui peuvent être montées simultanément sur un véhicule ; pour chaque ensemble, donner une condition sur les atomes "$variable = valeur$" qui caractérise l'ensemble des véhicules supportant ces pièces simultanément ?

Il peut sembler relativement simple de construire ces conditions caractéristiques en formant des combinaisons logiques à partir des cas d'emplois et des contraintes du CSP, si tant est qu'elles sont exprimées par des conditions logiques. La difficulté est que chaque condition caractéristique recherchée doit être simplifiée (aucune variable ne doit pouvoir en être supprimée sans compromettre l'ensemble qu'elle définit).

3.4 Configuration en ligne

Les problèmes liés à la configuration en ligne ont été déjà abordés dans plusieurs articles (voir par exemple [1]). Le principe de la configuration en ligne est que l'utilisateur (un client potentiel) définit le produit qu'il désire en choisissant interactivement des valeurs pour les variables ; il peut également revenir en arrière en relâchant un ou plusieurs de ses choix. Après chaque action, les domaines de valeurs possibles fournis à l'utilisateur doivent être modifiés de manière à contenir toutes les valeurs compatibles avec les choix courants, et seulement les valeurs compatibles avec ces choix. L'utilisateur peut forcer le choix d'une valeur détectée comme incompatible : le CSP devient alors incohérent. Quand l'ensemble des choix est incohérent avec les contraintes du CSP, il doit pouvoir revenir sur ses choix précédents et les relâcher.

L'utilisateur doit pouvoir également avoir une idée du prix du véhicule qu'il est en train de construire, et du délai d'obtention. Les informations prix sont décrites par des coûts associés aux valeurs de variables, ou par des fonctions de coût portant sur quelques variables. Le prix d'un véhicule est la somme des prix unitaires. Notons que certains coûts peuvent être négatifs (ce qui correspond à un retrait d'option par rapport au choix "standard"). On peut, en simplifiant, supposer que les informations délai sont portées par des conditions composées sur la gamme et définissant des intervalles de valeurs ("si le moteur choisit est hybride, alors le délai est d'au moins deux mois").

Pour aider l'utilisateur dans sa démarche, le système doit répondre aux requêtes suivantes :

Cohérence globale : Assurer, après chaque modification de domaine (choix de valeur ou relaxation) que les domaines correspondent exactement aux valeurs compatibles avec les choix courants.

Estimation du prix / du délai : Calculer le prix (ou le délai) min (ou max) des véhicules de la gamme compatibles avec les choix courants.

Calcul de restauration : En cas d'incohérence, identifier un ou des sous-ensembles maximaux

cohérents de restrictions utilisateur.

Calcul de conflit : Identifier des sous-ensembles minimaux de choix utilisateur incohérents, ou incohérents avec des choix de valeur de variable.

Simulation des choix : Déterminer quel serait l'état des domaines si l'utilisateur choisissait telle valeur pour telle variable (ce qui revient à la première requête, la cohérence globale)

Complétion de la configuration : Lorsqu'un certain nombre de variables "obligatoires" ont été renseignées, complétion automatique de la configuration courante à la demande de l'utilisateur (éventuellement sur un critère de minimisation de prix ou de délai).

3.5 Probabilisation de la gamme

La spécification de la gamme permet également à l'entreprise de construire des prévisions sur les ventes - typiquement, pour dimensionner sa production et celle de ses fournisseurs. Pour un modèle, ces prévisions sont équivalentes à la donnée d'un volume global et de probabilités (des taux prévisionnels) sur les valeurs de variables, conditionnées éventuellement par un contexte. ("On prévoit le choix de sièges en cuir dans 30% des véhicules vendus", "En Pologne, 25% des véhicules vendus seront diesel"). Ces probabilités individuelles sont en principe les projections d'une distribution de probabilité jointe sur la gamme. Cette démarche - construire une telle distribution à partir de taux - est appelée probabilisation de la gamme.

Cette distribution n'est pas connue, mais elle est spécifiée partiellement par la donnée de certaines de ses projections sur des variables (on parle de distributions marginales) et par la définition de la gamme. Il est possible qu'elle soit sur-spécifiée, c'est-à-dire qu'il n'existe aucune distribution de probabilité sur la gamme dont les marginales correspondent aux taux renseignés, ceci à cause des contraintes définissant la gamme. Dans le cas contraire, elle est généralement sous-spécifiée : plusieurs distributions jointes différentes peuvent produire les mêmes marginales. Dans ce cas la probabilité d'un véhicule n'est pas connue avec précision, mais l'on peut en connaître la plus petite et la plus grande valeurs possibles. De la même façon, plusieurs distributions peuvent être envisagées pour les marginales (pour les taux) non renseignées. Les requêtes en rapport avec la probabilisation de la gamme sont :

Cohérence des taux : Les taux sont-ils cohérents (existe-t-il une distribution de probabilité sur la gamme dont les marginales sont précisément les taux renseignés) ?

Taux conflictuels : Si les taux sont incohérents, en exhiber un sous-ensemble (minimal) incohérent modulo la définition de la gamme.

Configuration des taux : Si les taux sont cohérents, calculer une borne min et une borne max de la probabilité de l'affectation d'une valeur à une variable de véhicule. Utilisé itérativement avec la définition de taux, ce mécanisme permet de configurer progressivement un jeu de taux qui reste cohérent à toutes les étapes de la construction.

Extrapolation de taux : Si les taux sont cohérents, calculer une borne min et une borne max de la probabilité d'une expression booléenne sur des affectations - typiquement, sur des cas d'emploi. Ce calcul permet notamment d'encadrer les volumes prévisionnels de pièces.

Génération d'échantillon : Les taux étant cohérents, générer un échantillon de n véhicules conformément à la distribution de probabilité sous-jacente ; si elle est sous-spécifiée, on peut se baser sur la distribution d'entropie maximale. Cet échantillon sera utilisé comme une commande prévisionnelle par d'autres logiciels (simulation de la production, calcul de prix de revient, etc.).

3.6 Temps de réponse

Pour les applications interactives (typiquement, la configuration en ligne) le temps de réponse doit être (au pire) de l'ordre de la seconde. Les applications hors ligne, elles, peuvent compter leur temps de calcul en heures. Mais si elles ont un grand nombre de requêtes à effectuer, cela signifie que chaque requête devra également prendre un temps réduit (entre la milliseconde et la seconde, suivant les cas). Cependant, il reste possible de consacrer un temps beaucoup plus élevé (par exemple se comptant en minutes) pour les requêtes les plus difficiles, tant que cela ne modifie pas l'ordre de grandeur du temps de calcul total. C'est cette logique qui prévaut pour les contrôles de cohérence de la documentation véhicule et de la documentation pièces.

4 Formalisation des requêtes ; complexité

Nous étudions ici comment les requêtes présentées en Section 3 peuvent être formalisées en termes de CSP, et présentons nos premiers résultats quant à leur complexité théorique.

4.1 Notations et définitions

CSP Un *CSP* est, on l'a vu, un triplet $P =< X, D, C >$. Pour toute variable x de X, $D(x)$ est le domaine (supposé fini) de x. Pour toute contrainte c de C, on note $vars(c)$ l'ensemble des variables dont elle restreint les valeurs.

Une *affectation* d est une fonction qui associe à chaque $x \in X$ une valeur de son domaine ou le sym-

bole $*$, qui indique que la variable n'a pas été affectée. d est complète (resp. partielle) si le symbole $*$ n'est pas (resp. est) présent. Plus largement, on dit que d affecte $Y \subseteq X$ (resp. c) ssi pour tout $x \in Y$ (resp. pour tout $x \in vars(c)$), $d(x) \neq *$. $D(Y) = \{d, \forall x, d(x) = * \Leftrightarrow x \notin Y\}$ est l'ensemble des affectations de $Y \subseteq X$. Enfin, on note $vars(d)$ les variables affectées dans d, et on dit que d' étend d (à $vars(d')$) ssi $var(d) \subseteq vars(d')$ et $\forall x \in vars(d), d(x) = d'(x)$; ceci est noté $d' \models d$.

Une contrainte peut être vue comme une fonction c de l'ensemble des affectations de $vars(c)$ dans $\{\top, \bot\}$: $c(d) = \top$ ssi d satisfait la contrainte. On ne fait pas d'hypothèse sur la manière dont sont représentées les contraintes, mais on suppose que, pour toute affectation d affectant (au moins) toutes les variables de c, on peut tester en temps polynômial (idéalement, linéaire) si d satisfait c (noté $d \models c$) ou la viole (noté $d \not\models c$). Une *solution* d'un CSP est une affectation complète de ses variables qui satisfait toutes ses contraintes. On note $Sols(P)$ l'ensemble des solutions de P. Si $Sols(P) = \emptyset$, P est dit *incohérent*, sinon, il est dit *cohérent*.

$Sols(P)^{\downarrow Y} = \{d \in D(Y)$ t.q. $\exists d' \in Sols(P), d' \models d\}$ dénote la *projection* de $Sols(P)$ sur $Y \subseteq X$.

On dit que v est une valeur *globalement cohérente* pour x ssi $v \in Sols(P)^{\downarrow\{x\}}$. Lorsque toute les valeurs des domaines du CSP sont globalement cohérentes pour la variable correspondante, on dit que les domaines du CSP sont globalement cohérents (ou simplement que le CSP est globalement cohérent).

Conditions Booléennes Un *fluent* est un couple (x, A), avec $x \in X$ (et généralement, $A \subseteq D(x)$, mais pas nécessairement) ; il est élémentaire quand A est un singleton - il représente alors une affectation de x. Une *condition booléenne* est une formule logique (utilisant les opérateurs $\vee, \wedge, \Longrightarrow, \neg, \Leftrightarrow$ dans leur acception classique) dont les atomes sont des fluents. $vars(c)$ dénote l'ensemble des variables de c.

Une affectation d telle que $x \in vars(d)$ satisfait le fluent $f = (x, A)$ si et seulement si $d(x) \in A$ (on note $f(d) = \top$). Soit c une condition booléenne et d une affectation de $vars(c)$. La valeur de vérité de c dans d, notée $c(d)$ est calculée conformément à l'interprétation habituelle des opérateurs logiques. d satisfait c (noté $d \models c$) ssi $c(d) = \top$.

Certaines contraintes de la gamme seront représentées par des conditions booléennes - typiquement les contraintes options (c.f. section 2). Les description synthétiques utiles aux tâches de restitution de la gamme (c.f. Section 3.2) et les cas d'emplois relatifs à la documentation pièces (c.f. Section 3.3)le seront également. Dans la suite, on suppose généralement que les conditions booléennes manipulées sont intrinsèquement cohérentes ($\exists d \in D(vars(c)), c(d) = \top$) et que l'on peut tester en temps polynômial (idéalement, linéaire) leur satisfaction par une affectation de leurs variables.

4.2 Résultats

Les Tables 1 à 4 décrivent comment les requêtes présentées en section 3 peuvent être formalisées, et donnent nos premiers résultats quant à leur complexité. Certaines définissent clairement des problèmes de décision au sens de la théorie de la complexité, par exemple le problème de cohérence de la gamme (il revient à déterminer si $< X, D, C >$ est cohérent). D'autres sont des problèmes de recherche souvent NP-difficiles. Dans les Tables qui suivent, nous essayons d'affiner cette information : la complexité d'un problème de recherche d'objet (ex : d'une solution d'un CSP, d'un ensemble minimal incohérent de contraintes) est en effet directement liée à celle de l'existence d'un tel objet ou à celle du test de conformité de l'objet. Il peut arriver que le test de conformité soit linéaire alors que le test d'existence est difficile (typiquement, pour l'existence / le test de solution d'un CSP) ; il peut également arriver que le test d'existence soit trivial, alors que le test de conformité est difficile (par exemple, pour l'existence / le test d'ensembles (minimaux) incohérents de contraintes). Faute de place, seules les preuves de complexité les moins évidentes seront évoquées[2].

Documentation véhicule (voir Table 1). La requête de *Cohérence* (éventuellement contextuelle) correspond au test de cohérence, classique, d'un CSP, d'où sa NP-complétude. Sans surprise, on appellera *conflit* un ensemble incohérent de contraintes, et l'objectif est de connaître des conflits minimaux pour l'inclusion. On retrouve des notions de sous ensemble minimaux incohérents étudiées depuis la fin des années 80 en IA et plus récement en configuration [3][4][1] [6] (voir en particulier [4][6] pour le cas contextuel). La recherche de conflits minimaux (éventuellement contextuels) est NP-difficile, puisque le simple test d'(in)cohérence de $< X, D, C' >$ appelle la résolution d'un problème de cohérence. D'après [1], la question est BH_2-complète.[3]

La question de la cohérence positive du modèle est évidement une question de cohérence globale des contraintes et des domaines. Elle est donc NP-difficile.

[2] Pour plus de détails voir `ftp://ftp.irit.fr/IRIT/RPDMP/PapersFargier/wsecai10.pdf`

[3] BH_2 est l'ensemble des problèmes qui se ramènent à un problème de NP et un problème de CO-NP. Le problème BH_2-complet canonique est le problème SAT-UNSAT : il s'agit de déterminer si, étant donnée une paire (ϕ, ψ) de 3-CNF, la première est satisfiable et la seconde insatisfiable).

Nom	Donnée	Requête	Complexité
Cohérence de la gamme	$< X, D, C >$	$< X, D, C >$ admet-il une solution ?	NP-complet
Cohérence contextuelle	$< X, D, C >$ c une cond. booléenne cohérente	$< X, D, C \cup \{c\} >$ admet-il une solution ?	NP-complet
Conflits	$< X, D, C >$ $C' \subseteq C$	Déterminer si (i) $< X, D, C' >$ incohérent et (ii) $\forall C" \subsetneq C', < X, D, C" >$ cohérent	BH_2-complet
Conflits contextuels	$< X, D, C >$ Cxt un ensemble de cond. booléennes cohérentes , $C' \subseteq C$	Déterminer si (iii) $< X, D, C' \cup Cxt >$ incohérent et (iv) $\forall C" \subsetneq C', < X, D, C" \cup Cxt >$ cohérent	BH_2-complet
problème de recherche :	*trouver un /tous les C'*	*respectant (i) et (ii) (resp. (iii) et (iv))*	
Cohérence positive	$< X, D, C >$ Cohérent $c \in C$ une contrainte d'arité inférieure à k	Déterminer si c est globalement cohérente $(\forall d \in D(vars(c)), d \models c$ $\Rightarrow \exists d' \in Sols(P), d' \models d)$	NP-complet y compris si c est unaire

TAB. 1 – Résultats : Élaboration de la documentation véhicule

Nom	Donnée	Requête	Complexité
Validité de description	$P =< X, D, C >$ cohérent, c une cond. booléenne cohérente	Déterminer si c est valide (i.e. si $\forall d \in Sols(< X, D, C >), d \models c$)	CO-NP complet
Simplification de description	c une cond. booléenne cohérente	Déterminer si c est minimale $\forall x \in vars(c), \exists d, d' \in D(vars(c))$ t.q. $(\forall y = xd(y) \neq d'(y)) \wedge (d \models c)(\wedge d' \not\models c))$	$O(d^{\lvert vars(c)\rvert})$
Génération de description	$P =< X, D, C >$ cohérent, c une cond. booléenne cohérente	Déterminer si c équivaut à $Sols(P)^{\downarrow vars(c)}$ (si $\forall d \in D(vars(c))$: $c(d) = \top \Leftrightarrow d \in Sols(P)^{\downarrow vars(c)}$)	Π_2^P-complet
problème de recherche :	*trouver un c t.q.*	$vars(c) \subseteq Y$ *et c équivaut à* $Sols(P)^{\downarrow vars(c)}$	
Comptage	$P =< X, D, C >$, cohérent, $Y \subseteq X$	Calculer $\lvert Sols(P)^{\downarrow Y}\rvert$	#P-difficile

TAB. 2 – Résultats : Restitution de la gamme

Nom	Donnée	Requête	Complexité
Concision de la doc.	$P =< X, D, C >$, cohérent C' un ensemble de cond. booléennes cohérentes	Déterminer si C' est concis $(\forall c' \in C', < X, D, C \cup \{c'\} >$ cohérent)	NP-complet
Exhaustivité de la doc.	$P =< X, D, C >$, cohérent C' un ensemble de cond. booléennes cohérentes	Déterminer si C' est exhaustif (si $< X, D, C \cup \bigcup_{c' \in C'} \{\neg c'\} >$ est incohérent)	CO-NP complet
problème de recherche	*trouver une cond. booléenne cohérente c*	*telle que* $d \models c \Leftrightarrow (\forall c' \in C', d \not\models c')$	NP-difficile
Non-dualité des emplois	$P =< X, D, C >$, cohérent C' un ensemble de cond. booléennes cohérentes	Déterminer si C' comporte des cas de dualité (si $\exists c' \neq c" \in C'$ t.q. $< X, D, C \cup \{c', c"\} >$ cohérent)	NP-complet
problème de recherche	*trouver une cond. booléenne cohérente c*	*telle que étant donnés* $c', c'' \in C'$: $\forall d \in Sols(P) : (d \models c \Leftrightarrow d \models c'$ et $d \models c")$	NP-difficile

TAB. 3 – Résultats : élaboration de la documentation pièces

Nom	Donnée	Requête	Complexité
Cohérence globale	$P =< X, D, C >$, cohérent $x \in X \setminus vars(d), v \in D(x)$	Existe t il $d' \in Sols(P)$ t.q. $d'(x) = v$?	NP-complet
Cohérence globale (version incrémentale)	$P =< X, D, C >$, glb. cohérent d tq. $\exists d' \in Sols(P), d' \models d$ $x \in X$, $v \in D(x)$	Existe t il $d' \in Sols(P)$ $d' \models d$ et $d'(x) = v$?	NP-complet
Estimation du prix	$P =< X, D, C >$, cohérent S un ens. fini de fct de coût sur X d une affectation (partielle)	Calculer $\alpha_m = \min_{d' \in Sols(P), d' \models d} \Sigma_{s \in S} s(d')$ $\alpha_m = \max_{d' \in Sols(P), d' \models d} \Sigma_{s \in S} s(d')$	NP-difficile NP-difficile
Estimation du délai	$P =< X, D, C >$, cohérent x_t une variable codant le délai d une affectation (partielle)	Calculer $\alpha_m = \min_{d' \in Sols(P), d' \models d} d'(x_t)$ $\alpha_m = \max_{d' \in Sols(P), d' \models d} d'(x_t)$	NP-difficile NP-difficile
Calcul de restauration	$P =< X, D, C >$ $Choix$ un ens. de cond. booléennes $R \subseteq Choix$	Est il vrai que $< X, D, C \cup R >$ est cohérent et que $\forall R' \subseteq Choix$ t.q. $R \subsetneq R', < X, D, C \cup R' >$ est incohérent ?	BH-2 complet
Calcul de conflit	$P =< X, D, C >$ $Choix$ un ens. de cond. booléennes $R \subseteq Choix$	Est il vrai que $< X, D, C \cup R >$ est incohérent et que $\forall R' \subsetneq R$ $< X, D, C \cup R' >$ est cohérent ?	BH-2 complet
Complétion	d une affectation (partielle) $P =< X, D, C >$ glb. coherent pour d	Existe-t-il $d' \in Sols(P)$ t.q. $d' \models d$?	trivial
Problème de recherche		*Trouver* $d' \in Sols(P)$ *t.q* $d' \models d$	? ? ? ? ?
Problème de recherche *version optimisation*	*S un ens. fini de fct de coût*	*Trouver* $d' \in Sols(P)$ *tel que* $d' \models d$ *et que* $\forall d" \in Sols(P)$ si $d" \models d$ alors $\Sigma_{s \in S} s(d') \leq \Sigma_{s \in S} s(d")$	NP-Difficile
	x_t *une variable codant le délai*	*Trouver* $d' \in Sols(P)$ *t.q.* $d' \models d$ *et* $\forall d" \in Sols(P), d" \models d \Rightarrow d"(x_t) \leq d'(x_t)$	NP-difficile

TAB. 4 – Résultats : Configuration en ligne

Restitution de la gamme (voir Table 2). En restitution de la gamme, on cherche à obtenir des descriptions synthétiques sur des sous-ensembles de variables - techniquement, ce sont des projections de l'ensemble des solutions du CSP sur ces sous-ensembles. Idéalement, on les voudrait les plus simples possibles, c'est-à-dire impliquant le moins de variables possible. Dans le contexte de notre application, ces description doivent être données sous la forme d'une condition booléenne cohérente (la cohérence interne des descriptions ne doit pas être une source de complexité). Techniquement, nous dirons donc qu'une description booléenne c est *valide* ssi $\forall d \in Sols(P), d \models c$, qu'elle est *synthétique* si elle coïncide avec la projection de la gamme sur ses variables $(\forall d \in D(vars(d)), c(d) = \top \Leftrightarrow d \in Sols(P)^{\downarrow vars(c)})$ et qu'elle est minimale si il n'existe pas de $Y \subsetneq vars(C)$ telle que la restriction de c à Y lui soit équivalente. Le problèmes de *validité* est CO-NP complet, les problème d'inférence clausale ou de redondance de contrainte en étant des cas particuliers.

Le test de conformité associé à la *Génération de description* est Π_2^P-complet[4]. Le principe de la démonstration de est dans notre cas : c est la condition $\bigwedge_{x \in Y}(x, D(x))$: cette condition sélectionne un certain nombre de variables (celles de Y) sans restreindre les valeurs. Dans ce cas c équivaut à $Sols(< X, D, C >)^{\downarrow vars(c)}$ si et seulement si, pour toute affectation de Y, il existe une affectation de $X \setminus Y$ qui satisfasse le CSP. On peut donc réduire le problème canonique de Π_2^P ($\forall Y_1, \exists Y_2 : \Sigma$, Σ étant une CNF) au problème de test de l'équivalence entre la projection sur Y_2 d'un CSP représentant la CNF et la condition $c = \bigwedge_{x \in Y_2}(x, \{true, false\})$. Le test d'appartenance à Π_2^P est plus simple. L'implication $d \in Sols(< X, D, C >)^{\downarrow vars(c)} \implies c(d) = \top$ se falsifie avec un oracle (deviner d, vérifier que d satisfait toutes les contraintes puis vérifier que $c(d) = \bot$). Pour falsifier $c(d) = \top \implies d \in Sols(< X, D, C >)^{\downarrow vars(c)}$, il faut deviner une affectation de $vars(c)$, montrer qu'elle satisfait c, puis montrer qu'elle est inconsistante avec le CSP (ce qui est un problème de CO-NP).

Les problèmes de décision liés à la restitution de la gamme sont typiquement des problèmes d'inférence. Cela dit, la projection et la simplification sont basiquement des problèmes de recherche auxquels on peut ré-

[4]Rappelons qu'un problème appartient à $\Pi_2^p = \mathsf{NP}^{\mathsf{CO-NP}}$ si on peut résoudre son co-problème en (1) faisant appel à un NP-oracle et (2) établissant un certificat non pas de manière polynomiale mais en résolvant un problème de CO-NP. Le problème canonique de Π_2^p est celui qui consiste à déterminer si une formule booléenne quantifiée de la forme $\forall Y_1 \ \exists Y_2 \ \phi$ est valide , ϕ étant une 3-CNF et $\{Y_1, Y_2\}$ une partition de ses variables.

pondre par des algorithmes d'élimination de variables - en utilisant un espace potentiellement exponentiel.

Élaboration de la documentation pièces (voir Table 3). L'idée est de représenter chaque cas d'emploi par une condition booléenne cohérente (la cohérence interne d'un cas d'emploi n'est pas une source de complexité), et de considérer l'ensemble C' des cas d'emploi d'une pièce générique : C' est exhaustif ssi $\forall d \in Sols(P), \exists c' \in C'$ t.q. $d \models c'$; on dit que $c', c'' \in C'$ forme une paire d'emplois duale ssi $\exists d \in Sols(P), (d \models c')$ et $(d \models c'')$. C' est non exhaustif ssi $< X, D, C \cup \bigcup_{c' \in C'} \{\neg c'\} >$ est cohérent et il est non dual ssi chacun des problème $< X, D, C \cup \{c', c''\} >$ est incohérent (où $c', c'' \in C'$) . Enfin, la description de la pièce générique est concise ssi chacun des problèmes $< X, D, C \cup \{c'\} >$, $c' \in C'$ est cohérent.
Ces trois questions demandent de tester la cohérence ou l'incohérence d'un CSP, cohérent au départ, auquel on ajoute une ou plusieurs contraintes. Les preuves de complexité associées ne présentent aucune difficulté.

Enfin, on veut caractériser par une condition booléenne c les véhicules de la gamme "en erreur" par rapport à l'exigence de non-dualité ou d'exhaustivité de la pièce générique. c représente les cas de non-exhaustivité ssi $\forall d \in Sols(P), d \models c \Leftrightarrow \forall c' \in C, d$ viole c' ; il représente les cas de non dualité ssi $\forall d \in Sols(P), d \models c \Leftrightarrow \exists c', c'' \in C, d \models c'$ et $d \models c''$. Dans le premier cas, la CO-NP difficulté se prouve par réduction du problème d'inférence clausale ($C = \emptyset$, c est la clause à inférer et les c' sont les négations des clauses de la CNF). La CO-NP difficulté du second problème se prouve par réduction du même problème (c est la clause à inférer et c' et c" représentent une partition des clauses).

Configuration en ligne (voir Table 4) La *cohérence globale* des domaines doit être assurée en amont de la configuration utilisateur, pour ne lui laisser comme choix de configuration que des valeurs d'option présentes dans au moins un véhicule de la gamme. La version incrémentale est la répétition de ce principe en cours de configuration : l'utilisateur fixe une affectation d, cohérente, et l'on veut, pour toute variable x, ne laisser que des valeurs v pouvant effectivement conduire à un véhicule d' tel que $d' \models d$ et $d'(x) = v$. Dans ses deux versions, le problème de la maintenance de la cohérence globale est un problème NP-difficile.

Pour la prise en compte des coûts, on utisera une modélisation proche de celle des problèmes à contraintes souples (c.f. [12, 2]) : une "fonction de coût" est une fonction s sur $D(vars(s))$ prenant ses valeurs dans une échelle, ici $\mathbb{Z}$ et le "coût" d'une affectation est la somme des coûts qui lui sont associés par ces fonctions. Notons que certains coûts pouvant être négatifs, il faut théoriquement relaxer l'exigence de monotonie de l'opérateur d'agrégation (ici, l'addition). Lorsque tous les coûts sont positifs, que d n'instancie aucune variable et que $C = \emptyset$, le problème *d'estimation du prix* est un problème classique de minimisation de coûts dans un CSP souple, réputé NP-difficile.

Pour *l'estimation du délai*, on peut représenter le délai par une variable particulière x_t et coder les connaissances par des conditions logiques du type $c \implies x_t \in I_1 \cup \cdots \cup I_k$, les I_j étant des intervalles (ex : $x_t \in (-\infty, 8] \cup [10, +\infty)$). L'estimation du délai min (et du délai max par ailleurs) est un problème NP-difficile : lorsque d n'instancie aucune variable, et que seules des bornes inférieures sont données quant au délai, on reconnaît le problème d'optimisation dans les CSP à priorité [11] où il faut minimiser la priorité de la plus importante des contraintes violées.

La complétion de configuration est au sens d'un critère de coût ou de délai est évidement un problème NP-difficile. En revanche, on ne sait pas si il existe un algorithme polynomial permettant d'obtenir une solution à partir d'un problème globalement cohérent.Si un tel algorithme existe (ce qui est peu probable), il ne procède pas par réétablissement de la cohérence globale. Notre résultat sur la cohérence globale incrémentale montre en effect que, sous l'hypothèse P $\neq$ NP, il n'existe pas d'algorithme polynomial ré-établissant la cohérence globale suite à l'affectation d'une variable.

Pour les problèmes de *calcul de restauration et de conflits*, voir [1].

4.3 Probabilisation d'un CSP

Les problèmes de gestion de taux ne peuvent pas être modélisés de manière simple sous la forme d'un CSP ou d'un CSP valué, les taux n'étant pas indépendants les uns des autres. Il faudrait pouvoir définir une notion de CSP probabilisé comme étant un quadruplet $P^* =< X, D, C, F >$, où $< X, D, C >$ est un CSP et F est un ensemble de distributions de probabilités sur des sous-ensembles de X. Pour tout $p \in F$, on note $vars(p) \subseteq X$ les variables sur lesquelles porte p.

On dira que P^* est *cohérent* avec F si et seulement si il existe une distribution de probabilité p^* sur les affectations complètes de X telle que :

- $d \notin Sols(< X, D, C >) \implies p^*(d) = 0$
- $\forall p \in F$, d tel que $vars(d) = vars(p)$: $p(d) = \Sigma_{d' \in Sols(<X,D,C>), d' \models d} \; p^*(d')$

On note $Sols(P^*)$ l'ensemble des distributions de probabilité p^* vérifiant les deux conditions ci dessus. Le CSP probabilisé peut être sur-spécifié (aucune distribution n'est cohérente avec les taux de F) ou sous-spécifié, dans ce cas $Sols(P^*)$ contient plusieurs p^* .

La requête principale n'est pas le calcul explicite d'un p^*, qui est une distribution sur un ensemble de taille exponentielle ($Sols(< X, D, C >)$). Dans notre application, on cherche plutôt à connaître la probabilité d'affectations de X, de valeurs v dans les domaines, ou plus généralement, d'événements. Pour toute condition booléenne c, on peut définir :
$P_{inf}(c) = Inf_{p^* \in Sols(P*)} \Sigma_{d' \in Sols(<X,D,C>), d' \models c}\ p^*(d')$
$P_{sup}(c) = Sup_{p^* \in Sols(P*)} \Sigma_{d' \in Sols(<X,D,C>), d' \models c}\ p^*(d')$
Notons que P_{inf} et P_{sup} ne sont pas des mesures de probabilité. Elle encadrent la mesure de probabilité.

La notion de CSP probabilisé est bien différente des notions de CSP stochastique [14] ou probabiliste[5]. Dans ces deux formalismes (1) il existe deux types de variables, les variables de décision et les variables aléatoires (dans un CSP probabilisé, toutes les variables sont aléatoires) (2) les variables aléatoires sont indépendantes (il existe une unique p^* et son calcul est immédiat) et (3) on ne calcule pas la probabilité d'une affectation mais la probabilité qu'une affectation satisfasse le CSP : plusieurs affectations peuvent avoir une probabilité de 1 (dans un cSP probabilisé la somme des probabilités des affectations est égale à 1).

Les CSP probabilisés seraient une généralisation des réseaux bayésiens, relachant l'hypothèse qu'il existe une *unique* distribution jointe dont les projections seraient les distribution $p \in F$ (les Tables du réseau bayésien), et que le réseau est acyclique. Par conséquent le simple calcul d'une affectation de probabilité inférieure maximale serait un problème NP-difficile.

Sans entrer plus dans les détails, disons simplement que l'analyse de la tâche de gestion des taux en termes de programmation par contraintes fait apparaître le besoin d'un nouveau modèle qui serait une généralisation de réseaux bayesiens. L'analyse de la complexité des requêtes associées et la définition d'algorithmes efficaces sortent du cadre de cet article.

5 Conclusion

La formalisation d'un certain nombre d'applications métier utilisant une modélisation à base de contraintes d'une gamme montre que la preuve de la cohérence d'un CSP est loin d'être la seule requête que doivent addresser les algorithmes de programmation par contraintes : on rencontre notamment des problèmes d'inférence, des problèmes d'optimisation, des problèmes de comptage et des problèmes de marginalisation, qui se situent souvent au dessus de NP.

Dans la plupart des cas, le CSP considéré est notoirement cohérent, connu à l'avance, et commun à toute une série de requêtes. Chez Renault, l'équipe "Intelligence Artificielle Appliquée " en charge de ces applications a choisi une approche par compilation du CSP représentant la gamme [9], approche qui permet de répondre en temps satisfaisant (de l'ordre du centième de seconde pour la configuration en ligne, par exemple) à la plupart des requêtes.

La question est maintenant de savoir (1) si les algorithmes proposés par la communauté CSP peuvent donner des résultats comparables, (2) si la communauté peut proposer des algorithmes efficaces pour les problèmes qui ne sont pas typiquement des problèmes de satisfaction (par exemple, l'établissement de la cohérence globale, le calcul de marginalisations), et (3) comment étendre le modèle CSP pour qu'il sache traiter de questions d'inférence complexes - typiquement, effectuer de l'inférence probabiliste.

Références

[1] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoration and explanations in dynamic CSP - application to configuration. *Artificial Intelligence*, 135(1-2) :199–234, 2002.

[2] M. C. Cooper and T. Schiex. Arc consistency for soft constraints. *Artificial Intelligence*, 154(1-2) :199–227, 2004.

[3] Johan de Kleer. Problem solving with the atms. *Artificial Intelligence*, 28(2) :197–224, 1986.

[4] J. L. de Siqueira N. and J-F Puget. Explanation-based generalisation of failures. In *ECAI*, pages 339–344, 1988.

[5] H. Fargier and J. Lang. Uncertainty in constraint satisfaction problems : a probalistic approach. In *ECSQARU*, pages 97–104, 1993.

[6] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Stumptner. Consistency-based diagnosis of configuration knowledge bases. *Artif. Intell.*, 152(2) :213–234, 2004.

[7] E. Gelle and R. Weigel. Interactive configuration using constraint satisfaction techniques. In *PACT-96*, pages 37–44, 1996.

[8] S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI*, pages 25–32, 1990.

[9] B. Pargamin. Vehicle sales configuration : the cluster tree approach. In *ECAI'02 Configuration Workshop*, 2002.

[10] D. Sabin and E. C. Freuder. Configuration as composite constraint satisfaction. In *AI and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[11] T. Schiex. Possibilistic constraint satisfaction problems or "how to handle soft constraints ?". In *UAI*, pages 268–275, 1992.

[12] T. Schiex. Arc consistency for soft constraints. In *CP*, pages 411–424, 2000.

[13] T. Walsh. Sat v csp. In *CP*, pages 441–456, 2000.

[14] T. Walsh. Stochastic constraint programming. In *ECAI*, pages 111–115, 2002.

Une restriction de la résolution étendue pour les démonstrateurs SAT modernes*

Gilles Audemard[1] **George Katsirelos**[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 gkatsi@gmail.com simon@lri.fr

Résumé

La plupart des solveurs SAT modernes se basent, avec succès, sur les mécanismes d'analyse de conflits et d'apprentissage initialement introduits dans les solveurs GRASP et CHAFF. D'un point de vue théorique, ce succès a été partiellement expliqué à l'aide de l'équivalence, en termes de puissance, entre le système de preuves implanté par l'apprentissage (avec redémarrages) et la résolution générale, alors que les précédents démonstrateurs SAT implantent des systèmes de preuve moins puissants. Néanmoins, des bornes inférieures exponentielles subsistent pour la résolution générale, ce qui suggère une voie prometteuse – mais théorique – permettant une amélioration significative des démonstrateurs SAT : l'utilisation de systèmes de preuves plus puissants. Transformer cette voie théorique en améliorations pratiques dans le cas général a cependant toujours échoué.

Dans cet article, nous présentons un solveur SAT qui utilise une restriction de la résolution étendue. Ce solveur améliore les solveurs existant, en pratique, sur des instances issues des dernières compétitions, mais également sur des instances difficiles pour la résolution comme les instances XOR-SAT.

Abstract

Modern complete SAT solvers almost uniformly implement variations of the clause learning framework introduced by Grasp and Chaff. The success of these solvers has been theoretically explained by showing that the clause learning framework is an implementation of a proof system which is as poweful as resolution. However, exponential lower bounds are known for resolution, which suggests that significant advances in SAT solving must come from implementations of more powerful proof systems.

We present a clause learning SAT solver that uses extended resolution. It is based on a restriction of the application of the extension rule. This solver outperforms existing solvers on application instances from recent SAT competitions as well as on instances that are provably hard for resolution, such as XOR-SAT instances.

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

1 Introduction

Depuis une vingtaine d'années des progrès spectaculaires ont été réalisé dans la résolution pratique du problème de satisfaisabilité (SAT). Dans le cadre des problèmes structurés (par opposition aux problèmes aléatoires uniformes), l'apprentissage, initialement introduit dans GRASP [11] représente maintenant la brique de base de tout solveur « moderne ». Le passage à l'échelle des instances industrielles repose quant à lui principalement sur l'utilisation des structures de données paresseuses, associés à l'heuristique proposée dans CHAFF [12]. Peu après, en proposant une réécriture simplifiée et compacte, le solveur MINISAT [7] a également aidé à mieux comprendre les divers composants essentiels aux solveurs SAT modernes d'un point de vue pratique. Si on se place maintenant du côté théorique, d'importants progrès ont aussi été obtenus. Ainsi, l'efficacité de ces solveurs a pu être partiellement expliqué d'un point de vue théorique en montrant que l'apprentissage permettait d'atteindre la puissance des systèmes de preuves par résolution générale [2], ce qui n'est pas le cas pour les algorithmes de type DPLL [6, 1]. Ces derniers sont en effet limités à la puissance d'une forme resteinte de la résolution (*tree like resolution*). Il existe ainsi théoriquement des exemples admettant des preuves de réfutation de taille minimale exponentiellement plus longues pour les algorithmes de type DPLL que pour les algorithmes CDCL (*conflict driven, clause learning*), alors

que l'inverse n'est pas vrai.

D'un point de vue pratique, même si les solveurs CDCL sont très efficaces sur des instances issues de la vérification formelle [14], de nouveaux problèmes provenant de domaines comme la cryptographie et la bio-informatique [16] sont régulièrement proposés comme nouveaux challenges à la communauté. Afin de permettre une amélioration pratique significative, nous proposons dans ce travail d'étendre les solveurs CDCL à un système de preuves plus puissant que la résolution. Un candidat naturel pour cela est la résolution étendue (ER) [19]. La résolution étendue ajoute à la résolution une règle simple : la possibilité d'introduire des lemmes à la formule. Malgré son apparente simplicité, ER est aussi efficace que le système de preuve le plus efficace connu actuellement : L'*Extended Frege Systems* [21]. En effet, il n'a toujours pas été identifié de familles d'instances difficiles pour ER (*i.e.* n'admettant que des réfutations « longues »).

Néanmoins, il existe un obstacle majeur à franchir avant d'implanter un solveur basé sur la résolution étendue : comment choisir les bons lemmes à ajouter à la formule ? D'une part les possibilités d'extension sont gigantesques à chaque étape, et d'autre part trouver les bons lemmes est « difficile ». Par exemple, dans [5], Stephen COOK propose une preuve courte du problème des pigeons en utilisant la résolution étendue, mais les lemmes ajoutés tiennent compte de la sémantique du problème. Ainsi, même si ER peut être utilisé pour construire des preuves courtes de problèmes difficiles pour RES, aucun solveur implantant ER n'a encore été proposé. Choisir judicieusement les bons lemmes est au moins aussi difficile que résoudre directement le problème initial. Il est cependant à noter que certains travaux ont déjà, dans une certaine mesure, utilisé avec succès la résolution étendue, mais de manière indirecte seulement. Ainsi, les solveurs basés sur les BDD peuvent être vus comme l'utilisation de ER [18, 4], tout comme l'utilisation des symétries [17]. Le challenge est ici de construire un solveur implantant la résolution étendue et fonctionnant en pratique mieux que les solveurs CDCL, sur leurs problèmes favoris.

Nous proposons dans ce travail une restriction simple à ER qui réduit de manière significative le nombre potentiel de lemmes à ajouter à la formule. Cette restriction est en relation étroite avec le coeur des algorithmes CDCL : l'observation de régularités dans la succession de clauses apprises durant la construction de la preuve d'insatisfaisabilité. Cette approche est bien différente des approches mentionnées préalablement, dans la mesure où la référence à ER est explicite et peut permettre d'obtenir un solveur implantant un système de preuves aussi puissant que ER non restreinte. L'autre caractère novateur de notre approche vient du fait que nous traitons les nouvelles variables ajoutées comme une ressource similaire aux clauses apprises. Nous apprenons beaucoup de nouvelles variables, mais en oublions également beaucoup afin de limiter la consommation mémoire et de maintenir une bonne vitesse de propagation unitaire.

Le reste de l'article est organisé de la manière suivante : nous commençons par rappeler certaines notions nécessaires à la compréhension de l'article : les solveurs CDCL, les systèmes de preuves et la résolution étendue. Nous décrivons ensuite la restriction de la résolution étendue que nous utilisons et montrons comment elle peut améliorer la résolution. Nous continuons en décrivant comment cette restriction peut être efficacement implantée dans un solveur CDCL. Avant de conclure, nous montrons que le solveur obtenu peut améliorer de manière significative les solveurs actuels sur de nombreuses instances difficiles.

2 Rappels

Le problème SAT consiste à trouver une affectation des variables d'une formule propositionnelle ϕ exprimée sous forme normale conjonctive (CNF) afin que toutes les clauses de ϕ soient satisfaites. Nous considérons que la formule ϕ contient n variables propositionnelles $x_1 \dots x_n$. Pour chaque variable x_i, il existe deux *littéraux*, à savoir x_i et $\overline{x}_i$. Nous notons l un litéral arbitraire et $\bar{l}$ sa négation. Une formule CNF est une conjonction de clauses, chaque clause étant une disjonction de littéraux. Par convention, une CNF pourra être vue comme un ensemble de clauses et chaque clause pourra être vue comme un ensemble de littéraux. Nous supposons que le lecteur est familier avec les notions de sauts arrières (*backtracking*) et de de propagation unitaire.

Dans cet article, nous utilisons les solveurs CDCL [11, 12, 7], qui sont des algorithmes complets étendant la fameuse procédure de Davis et Putnam [6]. Ils incorporent des structures de données paresseuses, des sauts arrières, des nogoods des heuristiques dynamiques et des redémarrages. L'agorithme 1 décrit le schéma général d'un solveur CDCL. À chaque itération de la boucle principale la propagation unitaire est effectuée (ligne 3). Si un conflit apparaît (ligne 4), un nogood (également appelé clause assertive) est construit, en utilisant un schéma d'apprentissage basé sur la résolution [23], puis enregistré dans la base des clauses (ligne 5 ; 8). Un saut arrière est effectué jusqu'à ce que le nogood redevienne unitaire et permette un nouveau processus de propagation unitaire (ligne 9). Dans le cas où aucun conflit n'apparait, une nouvelle variable de décision est choisie de manière heuristique (ligne 13–14). La recherche est terminée lorsque toutes les variables sont affectées (lignes 11–12) ou lorsque la clause vide est générée (lignes 6–7).

Algorithm 1 Solveur Conflict Driven - Clause Learning

```
1: procedure CDCL(φ)
2:     loop
3:         Propagation Unitaire
4:         if une clause C est FAUSSE then
5:             C' = AnalyseConflit(C)
6:             if C' = ∅ then
7:                 retourner UNSAT
8:             Enregistrer C'
9:             Retour arrière jusqu'à ce que C' soit unitaire
10:        else
11:            if toutes les variables sont affectées then
12:                retourner SAT
13:            Choisir un littéral l
14:            Affecter l à un nouveau niveau de décision
```

2.1 Systèmes de preuves

Un système de preuves propositionel est un algorithme en temps polynômial SP tel que, pour toute formule propositionnelle insatisfaisable ϕ, il existe une preuve p tel que SP accepte l'entrée (ϕ, p), c'est à dire que SP montre que p est bien une preuve d'insatisfaisabilité pour ϕ. En d'autres termes, un système de preuve est un moyen efficace de vérifier la validité d'une preuve d'insatisfaisabilité (trouver une telle preuve reste cependant difficile dans le cas général). Les systèmes de preuves ont été étudiés en informatique théorique comme une façon de comparer les problèmes NP et co-NP. Ils permettent également de comparer les systèmes d'inférences les uns par rapport aux autres en permettant par exemple d'établir une hiérarchie de ces derniers reflétant leurs puissances relatives.

Un système de preuves SP admet des preuves courtes (resp. longues) pour une famille d'instances $\mathcal{F}$ si la taille de la plus petite preuve croit polynomialement (resp. exponentiellement) avec la taille des instances de $\mathcal{F}$. On dit alors que $\mathcal{F}$ est facile (resp. difficile) pour SP.

Un système de preuves SP_1 p-simule un système de preuves SP_2 si pour toute formule insatisfaisable ϕ la plus petite preuve de ϕ dans SP_1 est au pire polynomialement plus grande que la plus petite preuve de ϕ dans SP_2. S'ils se p-simulent l'un l'autre, ils sont alors polynomialement équivalents. La notion de p-simulation exprime une hiérarchie entre les systèmes de preuves. En effet, si SP_1 p-simule SP_2 alors SP_1 est au moins aussi puissant que SP_2.

Le système de preuves le plus connu et le plus utilisé est le système de preuves par résolution [15] (RES). Une preuve dans ce système est une suite de clauses $\langle C_1 \dots C_m, C_{m+1}, \dots, C_s = \bot \rangle$ où les m premières clauses sont celles de la formule initiale et les clauses suivantes sont produites par résolution. C'est à dire que la clause C_k est générée à partir des clauses C_i et C_j, où $i < j < k$ et il existe un littéral l tel que $C_i \equiv l \vee \alpha$, $C_j \equiv \bar{l} \vee \beta$, $C_k \equiv \alpha \vee \beta$ et $l \in \alpha \implies \bar{l} \notin \beta$. La clause C_k est alors la résolvante de C_i et C_j, ce qui est noté $C_k = C_i \otimes_l C_j$. La clause C_s est la clause vide.

La trace d'exécution d'un solveur CDCL peut se voir comme un système de preuves [2, 22, 9, 13] qui, dès lors, permet de p-simuler RES. Ce résultat est d'autant plus important que les autres algorithmes de résolution pour SAT, y compris la version originelle de DPLL [6] sont connus pour être plus faibles que la résolution. Par exemple, la procédure DPLL peut-être p-simulée par une restriction de la résolution où chaque clause qui n'appartient pas à la formule de départ ne peut être utilisée qu'une fois durant la preuve. Les bonnes performances pratiques des solveurs SAT modernes peuvent être expliquées, au moins en partie, par ce résultat théorique.

Néanmoins, le système de preuves RES à ses propres limites et plusieurs familles de problèmes ont été identifiées comme difficiles pour RES. Ainsi, le fameux problème des pigeons a été identifié comme difficile pour RES il y a déjà 25 ans [8], suivi par d'autres travaux et un résultat plus général [3]. Ces limitations suggèrent que des améliorations significatives dans la résolution du problème SAT peuvent venir de l'implantation de solveurs qui ne peuvent pas être p-simulés par la résolution, comme nous allons le proposer dans la suite de cet article.

3 Résolution étendue

Le système de preuve que nous visons pour augmenter la puissance des CDCL est bien connu. Il s'agit de la *résolution étendue* ER [19]. C'est une extension de la résolution à laquelle on ajoute la simple règle suivante : à chaque étape de la construction de la preuve, il est possible d'ajouter des lemmes dans la formule, sous la forme d'une nouvelle variable y (qui n'apparaissait pas dans la preuve jusque-là) associée aux clauses qui codent $y \Leftrightarrow l_1 \vee l_2$, où l_1 et l_2 sont deux littéraux apparaissant précédemment dans la preuve. Une preuve dans ER est donc une suite $\langle C_1 \dots C_m, \{C|E\}_{m+1}, C_s = \bot \rangle$ où C_i est une clause obtenue par résolution et E_i est l'ajout d'une nouvelle variable par la règle mentionnée ci-dessus. Sans perte de généralité, il est possible d'introduire l'ensemble des nouvelles variables au début de la preuve, qui peut donc être réécrite ainsi : $\langle C_1 \dots C_m, E_{m+1}, \dots E_{m+k}, C_{m+k+1}, \dots, C_s = \bot \rangle$. Tout en étant d'apparence une règle très simple, l'ajout de nouvelles variables permet à la résolution étendue d'être très efficace : aucun système de preuve n'a été prouvé comme étant plus fort que la résolution étendue. Ainsi, dans un *superbe* article [5], Stephen COOK montre comment utiliser la résolution étendue pour construire une preuve polynômiale des pigeons prenant en compte le coté inductif de ce problème.

Cependant, cette puissance ne va pas sans contre-partie. Il est en effet très difficile de trouver judicieusement et automatiquement les bonnes paires de littéraux à étendre.

C'est certainement ce qui explique l'absence de la résolution étendue dans les solveurs SAT actuels. À titre d'exemple, dans [5], COOK utilise la sémantique du problème pour choisir les lemmes à ajouter. Dans notre approche, cette difficulté est en partie écartée grâce à une restriction des possibilités d'extension.

3.1 Restriction proposée

Soient deux littéraux l_1 et l_2 provenant de deux variables différentes. Supposons que nous ayons appris durant la recherche les clauses $C_i = \bar{l}_1 \vee \alpha$ et $C_j = \bar{l}_2 \vee \alpha$. Supposons enfin que ces clauses soient toutes deux utilisées dans un processus de résolution avec la même séquence de clauses $C_{m_1}, \ldots, C_{m_k}$ afin de dériver $C'_i = \bar{l}_1 \vee \beta$ et $C'_j = \bar{l}_2 \vee \beta$. Dans ce cas, il peut être judicieux d'étendre la formule initiale avec la nouvelle variable $x \Leftrightarrow l_1 \vee l_2$. Ceci va alors permetttre d'effectuer une résolution entre C_i et C_j avec $\overline{x} \vee l_1 \vee l_2$ afin d'obtenir $C_k = \overline{x} \vee \alpha$. La clause C_k peut alors être résolue avec $C_{m_1}, \ldots, C_{m_k}$ pour dériver $\overline{x} \vee \beta$. Ainsi, toute étape commune dans la résolution entre C'_i et C'_j ne sera pas dupliquée, ce qui permettra une *compression* de la preuve.

Cet argument peut être généralisé. Supposons que les clauses C_i et C_j diffèrent de plus de un littéral et qu'elles soient de la forme $C_i = \bar{l}_1 \vee \alpha \vee \beta$ et $C_j = \bar{l}_2 \vee \alpha \vee \gamma$, où $\alpha \vee \beta \vee \gamma$ ne soit pas une tautologie. Si il existe une séquence de clauses $C_{m_1}, \ldots, C_{m_k}$ telle que C_i et C_j entrent en résolution avec $C_{m_1}, \ldots, C_{m_k}$ pour dériver $C'_i = \bar{l}_1 \vee \delta$ et $C'_j = \bar{l}_1 \vee \delta$ où $\beta \subseteq \delta$ et $\gamma \subseteq \delta$, nous pouvons alors appliquer la résolution étendue pour éviter de dupliquer la résolution sur $C_{m_1}, \ldots, C_{m_k}$.

Notons que dans ce schéma, les clauses C_i et C_j n'ont pas besoin d'être des clauses de la formule initiale, mais peuvent être des clauses dérivées. Ce point est primordial pour permettre l'introduction des variables en examinant la preuve au fur et à mesure de sa construction. La définition suivante formalise ce schéma comme un système de preuve par résolution.

Definition 1 (Résolution étendue locale) LER *est le système de preuve propositionel qui inclut la règle de résolution mais aussi la règle de résolution étendue telle que, à une étape k, il est possible d'introduire la nouvelle variable $z \Leftrightarrow l_1 \vee l_2$ si il existe deux clauses $C_i \equiv \bar{l}_1 \vee \alpha$ et $C_j \equiv \bar{l}_2 \vee \beta$ avec $i < k, j < k$, où α et β sont des disjonctions vérifiant $l \in \alpha \implies \bar{l} \notin \beta$.*

3.2 Discussion sur le pouvoir de LER

En pratique, LER diffère de ER dans la mesure où elle impose la présence de clauses d'une forme donnée avant d'introduire de nouvelles variables. En tout généralité, certaines de ces clauses peuvent ne jamais être dérivées, empêchant du même coup l'introduction des variables les plus pertinentes. Cependant, il faut noter un avantage indéniable de LER sur ER. Lors de la construction de la preuve, l'introduction des nouvelles variables est moins imprévisible que celle de ER. Intuitivement, on voit combien chaque variable introduite est directement reliée à la preuve. Cela atténue la nature non déterministe de ce système de preuve.

Néanmoins, la restriction de LER n'est pas trop sévère. En fait, il est facile de vérifier que des problèmes qui sont difficiles pour RES et qui acceptent des preuves polynômiales pour ER peuvent être transformées en des preuves polynômiales pour LER. C'est le cas par exemple pour la preuve ER de Stephen COOK sur le problème des pigeons [5], ce qui permet de montrer par la même occasion que LER est plus puissant que la résolution générale. D'un autre coté, il ne faut pas oublier qu'il peut être difficile de générer les deux clauses nécessaires à l'introduction d'une nouvelle variable. On peut donc s'attendre à ce que la preuve minimale pour LER soit plus grande que la preuve minimale pour ER pour certaines familles de problèmes. Actuellement, nous ne savons pas encore s'il existe des familles d'instances difficiles pour LER, mais faciles pour ER ou si LER p-simule ER.

Dans la mesure où les solveurs CDCL ont été décrit comme des algorithmes basés sur la résolution [10], nous pensons avoir assez d'informations durant la recherche pour implémenter LER et étendre la formules avec des variables qui aideront à produire des preuves plus courtes. Ces idées sont exploitées dans la section ci-dessous au travers d'un nouveau type de solveur utilisant cette restriction de la résolution étendue.

4 Un Solveur CDCL basé sur les extensions locales

Le schéma expliqué dans la section précédente restreint le nombre de paires de littéraux candidats pour la résolution étendue. Malgré cela, le nombre de candidats potentiels est trop grand pour espérer étendre systématiquement la formule avec tous les candidats possibles.

Rappelons ici que notre but initial est de construire un solveur efficace sur les instances applicatives (industrielles). Il est donc primordial de mettre au point une heuristique efficace (coût de calcul réduit, nombre de candidats raisonnable) de sélection des paires candidates. Cette restriction, si importante soit-elle, est prioritaire. On peut s'attendre à ce qu'elle ne nous permette plus de tirer pleinement parti des bénéfices de LER, en ne permettant plus la résolution efficace de certaines instances. Nous n'avons d'autres choix que de contrôler de manière drastique le nombre de variables que nous souhaitons ajouter.

Typiquement, dans un solveur CDCL, la majeure partie du raisonnement a lieu durant l'analyse de conflit. Ceci inclut l'apprentissage (voir algorithme 1), mais aussi la mise à jour des scores heuristiques (variables et clauses)

et plus réçemment l'évaluation des stratégies de redémarrages. Comme c'est là que nous pouvons examiner facilement les clauses dérivées, c'est également un endroit logique pour détecter les paires de littéraux candidates à l'extension.

4.1 Détection rapide (mais incomplète) des paires de littéraux intéressants

La nouvelle restriction que nous imposons à notre solveur pour qu'il puisse rester efficace, en comparaison avec LER est la suivante. Nous concentrons notre effort sur les paires de clauses de la forme $\overline{l}_1 \vee \alpha, \overline{l}_2 \vee \alpha$. Encore une fois, cette nouvelle restriction peut réduire la puissance du système de preuves sous-jaçent à notre solveur. Elle va cependant permettre de conserver un nombre de candidats à l'extension relativement faible, ce qui, on l'espère, va s'avérer payant en pratique.

De plus, cette nouvelle restriction n'est pas sans lien avec les mécanismes d'apprentissages des solveurs CDCL. Ansi, intuitivement, les heuristiques des solveurs (notemment grâce à la sauvegarde de phases) leur permettent de rester dans le même espace de recherche malgré les redémarrages. Il semble donc naturel de penser que les paires de clauses correspondant au schéma $\overline{l}_1 \vee \alpha$, $\overline{l}_2 \vee \alpha$ soient produites de manière rapprochées durant la recherche. On peut donc penser réduire la fenêtre des clauses candidates à examiner pour détecter ces paires de clauses. Dans notre implantation, nous avons poussé ce raisonnement à l'extrême, en ne cherchant que les paires de clauses produites *successivement*. Ceci nous permet, de plus, de supposer que les littéraux $\overline{l}_1$ et $\overline{l}_2$ sont en fait les littéraux FUIP découverts durant l'analyse de conflit. Ainsi, notre processus de sélection des candidats est le suivant : si deux clauses successivement apprises par le solveur sont de la forme $\overline{l}_1 \vee \alpha$, $\overline{l}_2 \vee \alpha$, alors le solveur introduit la variable $z \Leftrightarrow l_1 \vee l_2$ (si la paire de littéraux l_1, l_2 n'a pas déjà étendue précédemment). Cette détection s'intègre hormonieusement au cadre classique des solveurs CDCL. Nous verrons dans la partie expérimentale que ce schéma de détection, même s'il semble très restrictif, se produit assez souvent durant la recherche.

Applications successives de la résolution étendue Si n clauses apprises successives sont de la forme $l_i \vee C$ $(1 \leq i \leq n)$ alors nous ajoutons la variable $z_1 \Leftrightarrow \overline{l}_1 \vee \overline{l}_2$, mais aussi $z_i \Leftrightarrow \overline{l}_i \vee \overline{l}_{i+1}$ pour $2 \leq i < n$. Il était également possible d'ajouter à la place les lemmes $z_i \Leftrightarrow \overline{z}_i \vee \overline{l}_{i+1}$ en lieu et place de $z_i \Leftrightarrow \overline{l}_i \vee \overline{l}_{i+1}$ mais les résultats expérimentaux (non présentés ici) ont montré que la première approche était la plus efficace.

4.2 Injection des variables étendues dans les nouvelles clauses

À chaque fois qu'une nouvelle variable $z \Leftrightarrow l_1 \vee l_2$ est introduite, nous devons nous assurer que toutes les nouvelles clauses de la forme $l_1 \vee l_2 \vee \beta$ seront remplacées par $z \vee \beta$. Ce remplacement systématique est important. En effet, il permet à la nouvelle variable z d'être propagée même si cela n'était pas le cas pour la clause initiale. Par exemple, si la sous-clause β est falsifiée, la clause $l_1 \vee l_2 \vee \beta$ n'est pas unitaire, mais la clause $z \vee \beta$ l'est. Les tests expérimentaux ont montré que sans cela, la variable z n'apparaît pas dans l'analyse de conflit et ne participe donc pas à la preuve. Par soucis d'efficacité, nous ne réduisons pas les clauses apprises avant la création de la variable.

Afin de gérer efficacement cette étape de réduction, nous maintenons une table de hachage de chaque paire de littéraux (l_1, l_2). Celle-ci est sondée à chaque clause assertive générée afin de remplacer les paires par le littéral étendu associé. Notez que cela peut impliquer un choix. Supposons la clause $C \equiv l_1 \vee l_2 \vee l_3 \vee \beta$ et les variables étendues $z_1 \Leftrightarrow l_1 \vee l_2$ et $z_2 \Leftrightarrow l_2 \vee l_3$. Suivant l'ordre utilisé pour le remplacement, il est possible d'obtenir soit la clause $C_1 \equiv z_1 \vee l_3 \vee \beta$ soit la clause $C_2 \equiv l_1 \vee z_2 \vee \beta$. Pour trancher, nous avons choisi de favoriser le remplacement des variables étendues ayant le plus grand score VSIDS. Il faut tout de même noter ici que, même si cette étape est importante pour les performances du solveur, ce n'est qu'une heuristique qui encourage la résolution sur les variables étendues, mais n'affecte pas les propriétés théoriques du solveur.

On notera aussi que, contrairement au cas des clauses de la forme $l_1 \vee l_2 \vee \beta$, il n'est pas nécessaire de compresser les clauses de la forme $\overline{l}_1 \vee C$, $\overline{l}_2 \vee C$ en les remplacant par la seule clause $\overline{z} \vee C$. En effet, si jamais $\overline{l}_1$ ou $\overline{l}_2$ sont propagés en utilisant les deux clauses précédentes, z le sera aussi et sera potentiellement utilisé durant l'analyse de conflits.

4.3 Réduction des effets de bords indésirables

Dès lors qu'une étape de résolution étendue $z \Leftrightarrow \overline{l}_1 \vee \overline{l}_2$ est déclenchée, les trois clauses $\overline{z} \vee l_1 \vee l_2$, $z \vee \overline{l}_1$ et $z \vee \overline{l}_2$ sont ajoutées. Il est importat d'être alors certain qu'aucune de ces nouvelles clauses n'est unitaire à un niveau plus haut que le niveau de retour arrière, sans quoi des cycles pourraient apparaître dans le graphe de propagations unitaires. Fort heureusement, grâce à nos restrictions, ce cas n'arrive jamais. En effet, considérons à nouveau les deux clauses assertives consécutives $\overline{l}_1 \vee \alpha$ et $\overline{l}_2 \vee \alpha$. Comme l_1 et l_2 sont les littéraux assertifs, la clause $\overline{z} \vee l_1 \vee l_2$ est vraie et z va être propagé à vrai grâce à la clause $z \vee \overline{l}_1$. Ainsi, les effets de bords induits par la résolution étendue sont, nous l'espérons, relativement circonscris à l'adjonction des lemmes eux-mêmes. Bien entendu, ceci aura un impact sur la suite du calcul, mais il n'est pas nécessaire de forcer un

redémarrage ni de remettre à jour les décision et les raisons qui en découlent.

4.4 Nettoyage des lemmes jugés inutiles

Afin d'optimiser les performances, les solveurs CDCL réduisent régulièrement la base des clauses apprises durant la recherche. Le même argument peut être appliqué aux variables introduites ainsi qu'aux clauses correspondantes. Nous avons donc opté pour supprimer les variables étendues ayant un petit score VSIDS, indiquant qu'elles ne sont pas très utiles à la preuve. Chaque fois que l'on effectue un nettoyage de la base de clauses, on supprime ainsi la moitié des variables étendues.

5 Évaluation expérimentale

Dans cette traditionnelle section expérimentale, nous proposons de comparer deux solveurs CDCL de référence, MINISAT et GLUCOSE, dans leurs versions initiales, avec leurs versions modifiées (intégrant la résolution étendue), respectivements appelées MINISATER et GLUCOSER. Nous avons tout de même ajouté à la comparaison PRECOSAT qui est, avec GLUCOSE, l'autre gagnant de la dernière compétition SAT. Notez que la version de MINISAT utilisée par la suite diffère quelque peu de la version disponible sur le web : nous avons changé la stratégie de redémarrage en utilisant une série de Luby (demarrant à 32) et nous avons également ajouté la sauvegarde de la phase. Ces modifications améliorent grandement les performances de MINISAT sur les problèmes industriels et sont maintenant adoptées par de nombreux solveurs.

Nous utilisons deux types de familles de benchmarks : (1) les instances de type application (industrielles) issues des compétitions SAT'07 et SAT'09 [16] et (2) des instances connues pour être dures pour les algorithmes basés sur la résolution (problème des pigeons, problèmes Urquhart [20]) ou expérimentalement connues pour être difficiles. Nous avons utilisé un cluster de quad-core Intel XEON X5550 – 2.66 GHz avec 32 go de RAM. Le temps limite est fixé à 5000 secondes et les résultats sont reportés en secondes.

5.1 Résultats obtenus sur les compétitions

La table 1 résume les résultats obtenus sur les instances industrielles des deux dernières compétitions. Il faut noter ici combien ces instances sont difficiles. Même une petite amélioration dans le nombre de problèmes résolus doit être vu comme un progrès non négligeable (notons que PRECOSAT et GLUCOSE, les deux vainqueurs de 2009, ont résolu le même nombre d'instances dans la catégorie SAT +UNSAT). De plus le réel challenge dans notre travail était de montrer que l'on pouvait augmenter la puissance de raisonnement des solveurs CDCL tout en gardant de bonnes performances générales.

	SAT 07	SAT 09
PRECOSAT	167 (91 - 76)	211 (129 - **82**)
GLUCOSE	185 (111 - 74)	211 (132 - 79)
GLUCOSER	**191** (**113** - **78**)	**213** (**133** - 80)
MINISAT	142 (81 - 61)	190 (118 - 72)
MINISATER	146 (85 - 61)	198 (123 - 75)

TAB. 1 – Performance de PRECOSAT et de MINISAT et GLUCOSE avec et sans LER. Les instances proviennent de la catégorie industrielle/applicaiton des compétitions 2007 (234 instances) et 2009 (292 instances). Pour chaque solveur, nous reportons le nombre d'instances résolues avec entre parenthèses, le nombre d'instances SAT et UNSAT.

5.2 Résultats obtenus sur des instances reconnues difficiles

La table 2 met en avant les resultats obtenus sur deux types d'instances problématiques. Les premières sont les instances connues pour être difficiles pour la résolution, comme celles codant le problème des pigeons et les instances Urquhart. Tout d'abord, notons que MINISATER n'améliore pas le solveur de base sur les instances des pigeons. En fait, MINISAT est clairement le meilleur solveur sur ce type d'instances. Nous pouvons néanmoins proposer une explication pour ces mauvais résultats, certes assez surprenant. Nous avons ainsi, par ailleurs, généré les clauses qui codent le problème des pigeons en y ajoutant les variables étendues utilisées par COOK pour faire une preuve courte [5]. Il a été assez surprenant de voir que ces instances ont également été difficiles pour MINISAT. Cela montre que l'utilisation des variables étendues n'est pas suffisante pour obtenir à coup sûr de bons résultats. Nous devons également modifier l'heuristique de branchement pour forcer ces variables à être utilisées durant le processus de résolution. Le second argument que nous pouvons apporter à ces mauvais résultats est que MINISATER et GLUCOSER améliorent de manière significative leur solveur de base lors de la résolution des instances qui encodent le problème des pigeons fonctionnel, admettant plus de contraintes. Malgré tout, même dans ce cas, les solveurs étendus n'ont pas un comportement polynômial par rapport à la taille des problèmes.

À l'opposé, les résultats sont clairement plus positifs sur les instances Urquhart. MINISATER et GLUCOSER améliorent grandement leurs solveurs de base même si, ici encore, leurs résultats ne semblent pas évoluer polynomialement avec la taille des instances.

Enfin, la partie la plus importante de la table 2 est en rapport avec un ensemble d'instances connues expé-

rimentalement comme étant difficiles. Certaines d'entre elles n'ont pas été résolues durant la dernière compétition. Cela montre clairement que GLUCOSER, et dans une moindre mesure, MINISATER, résolvent des instances inaccessibles à leur solveur de base. Avant de conclure, voici quelques points qui peuvent être soulignés. Tout d'abord, les solveurs introduisent de nombreuses nouvelles variables, en gros, une tous les 1000 conflits. De plus, et cela n'est pas rapporté sur les tables, les solveurs branchent fréquemment sur les nouvelles variables. Ce qui implique qu'elles ont un score VSIDS elevé, et donc qu'elles sont utilisées durant le processus de résolution. Cela montre que GLUCOSER et MINISATER construisent une preuve LER, et pas seulement une preuve de résolution. Ainsi, malgré la restriction apportée à LER nous arrivons à introduire de nouvelles variables et LER admet des preuves beaucoup plus courtes sur ces instances que les solveurs CDCL classiques.

6 Conclusions

Dans cet article, nous avons proposé une restriction de la résolution étendue présentant deux avantages non négligeables par rapport à la résolution étendue classique. Tout d'abord, les variables introduites sont en relation directe avec la preuve construite. De plus, le nombre de paires de littéraux candidates à l'extension est fortement réduit. Ces deux avantages simplifient la création d'un solveur CDCL basé sur ce système de preuves. L'implantation résultante sur deux solveurs différents, y compris un solveur état de l'art, en a amélioré leurs performances sur des instances difficiles issues des dernières compétitions.

Ce travail laisse de nombreuses questions ouvertes. D'un point de vue pratique nous souhaitons mettre au point des heuristiques permettant d'introduire et de supprimer beaucoup plus fréquemment des variables, ceci, sans surcoût important. On peut par exemple imaginer tenir compte d'un ensemble de nogoods à comparer plutôt que de se limiter aux deux derniers. D'un point de vue théorique, nous souhaitons de plus déterminer la place exacte de LER dans la hiérarchie des systèmes de preuves. Si l'on arrive à prouver que ce dernier p-simule ER, cela pourrait être une étape importante pour comprendre la puissance de ER et aider à son exploitation dans les solveurs SAT.

Remerciements

Les auteurs tiennent à remercier Nicolas Prcovic pour avoir suggéré l'utilisation des instances de pigeons « préétendues ».

Références

[1] R. J. Bayardo and R. C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pages 203–207, Providence, RI, 1997.

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

[3] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow—resolution made simple. *J. ACM*, 48(2) :149–169, 2001.

[4] Philippe Chatalic and Laurent Simon. Multiresolution for SAT checking. *International Journal on Artificial Intelligence Tools*, 10(4) :451–481, 2001.

[5] Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4) :28–32, 1976.

[6] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5 :394–397, 1962.

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

[8] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39 :297–308, 1985.

[9] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-2008)*, pages 283–290, 2008.

[10] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 2318–2323, 2007.

[11] J. P. Marques Silva and K. A. Sakallah. GRASP—a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5) :506–521, May 1999.

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference*, pages 530–535, 2001.

[13] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers with restarts. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, volume 5732, chapter 51, pages 654–668. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

benchmark	SAT ?	PRECOSAT	GLUCOSE	GLUCOSER	MINISAT	MINISATER
hole-11	UNSAT	90s / 1,314	78s / 552	127s / 848 / 930	7s / 148	20s / 359 / 2195
hole-12	UNSAT	963s / 8,112	1,167s / 4,588	208s / 1,333 / 1,334	40s / 405	363s / 1,825 / 6,927
Urq3_5	UNSAT	142s / 4,570	21s / 791	8s / 235 / 1,478	398s / 2,119	10s / 318 / 2,702
Urq4_5	UNSAT	–	3,106s / 28,181	26s / 527 / 3,103	–	11s / 333 / 3,191
Urq5_5	UNSAT	–	–	545s / 5,021 / 7,298	–	107s / 1,488 / 7,503
aloul-chnl11-13	UNSAT	–	–	310s / 1,794 / 1,233	130s / 808	88s / 859 / 4,665
SAT_dat.k75†	UNSAT	–	–	614s / 5,544 / 23,042	–	–
dated-5-19-u†	UNSAT	–	–	3,127s / 9,063 / 9,202	–	–
9dlx_vliw_at_b_iq8†	UNSAT	–	4,322s / 6,672	3,609s / 4,733 / 14,490	–	–
sortnet-8-ipc5 -h19-sat†	SAT	3,395s / 2,179	–	3,672s / 7,147 / 6,921	–	–
vmpc_34	SAT	–	2,616s / 15,833	1,565s / 10,022 / 31,576	–	–
rbcl_xits_08	UNSAT	1,286s / 3,600	–	3,067s / 8,069 / 8,916	–	–
simon-s02b- k2f-gr-rcs-w8	UNSAT	–	–	3,622s / 7,904 / 13,567	–	3,797s / 5,121 / 30,890

TAB. 2 – Résultats sur une sélection d'instances difficiles. Pour chaque instance, nous reportons, si elle est satisfaisable, le temps nécessaire par chaque solveur pour la résoudre, le nombre de conflits (en milliers) et le nombre de variables étendues (après le "/", si résolution étendue il y a eu) ou "–" si le solveur ne résoud pas l'instance dans le temps imparti. Les instances annotées par †n'ont pas été résolues aux compétitions SAT'07 et SAT'09. Les autres ont été résolues par au plus 5 solveurs.

[14] M. Prasad, A. Biere, and A. Gupta. A survey of recent advances in SAT-based formal verification. *journal on Software Tools for Technology Transfer*, 7(2) :156–173, 2005.

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

[16] `http://www.satcompetition.org`, 2009.

[17] Bas Schaafsma, Marijn Heule, and Hans van Maaren. Dynamic symmetry breaking by simulating zykov contraction. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584, chapter 22, pages 223–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[18] Carsten Sinz and Armin Biere. Extended resolution proofs for conjoining bdds. In Dima Grigoriev, John Harrison, and Edward A. Hirsch, editors, *CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 600–611. Springer, 2006.

[19] G. Tseitin. On the complexity of proofs in propositional logics. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning : Classical Papers in Computational Logic 1967–1970*, volume 2. Springer-Verlag, 1983.

[20] A. Urquhart. Hard examples for resolution. *JACM*, 34(1) :209–219, 1987.

[21] Alasdair Urquhart. The complexity of propositional proofs. pages 332–342, 2001.

[22] Allen Van Gelder. Pool resolution and its relation to regular resolution and DPLL with clause learning. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Lecture Notes in Computer Science, chapter 40, pages 580–594. 2005.

[23] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of IEEE/ACM International Conference on Computer Design (ICCAD)*, pages 279–285, 2001.

Résolution de contraintes sur les nombres à virgule flottante par une approximation sur les nombres réels*

Mohammed Said BELAID, Claude MICHEL, Michel RUEHER

I3S-CNRS, 2000 route des Lucioles, BP 121, 06903 Sophia Antipolis Cedex, France
{MSBelaid, Claude.Michel}@i3s.unice.fr, Michel.Rueher@gmail.com

Résumé

La mise en œuvre effective de méthodes de vérification de programmes comportant des calculs sur les nombres à virgule flottante reste encore problématique. C'est pourquoi nous présentons dans cet article une nouvelle méthode de résolution de contraintes sur les nombres à virgule flottante qui consiste à les approximer sur les réels. Elle est basée sur la construction d'approximations sur les réels, précises et conservatives des solutions des contraintes sur les nombres à virgule flottante. Cette méthode permet de s'appuyer sur l'utilisation d'algorithmes de filtrage sur les réels pour résoudre des problèmes sur les nombres à virgule flottante. Il devient ainsi possible de repousser les limitations actuelles des solveurs de contraintes sur les nombres à virgules flottantes, telles que le problème du passage à l'échelle, pour générer des jeux de tests, ou vérifier des programmes plus conséquents que ceux traités jusqu'à maintenant.

Keywords : contraintes sur les nombres à virgule flottante, approximation des solutions, vérification de programme.

1 Introduction

La vérification et le test de programmes sont des étapes très importantes dans un processus de développement de logiciels, plus particulièrement, lorsqu'il s'agit de logiciels critiques [6]. Par exemple, dans l'aéronautique beaucoup de ces logiciels effectuent des calculs basés sur l'arithmétique des nombres à virgule flottante. Or l'utilisation sans précaution de cette arithmétique peut causer de graves dégâts. Citons, par exemple, le crash de la fusée Ariane V lié à un problème de conversion d'un nombre à virgule flottante vers un entier 16 bits [11]. Un autre exemple qui montre bien que, même une erreur mineure peut avoir de graves conséquences, est l'échec de la mission de l'anti-missile Patriot [17]. Une imprécision dans le calcul du temps (de l'ordre de 10^{-7}) n'a pas permis un calcul assez précis de la position, le cumul de l'erreur étant d'autant plus grand que ce programme était lancé bien avant le départ de l'anti-missile. Ces événements montrent bien que la vérification et le test sont encore plus cruciaux pour les programmes avec des nombres à virgule flottante.

Plusieurs approches de vérification de programmes avec des calculs sur les nombres flottants ont été développées. On peut notamment citer les approches basées sur l'interprétation abstraite [14, 9] qui, si elles permettent de garantir pour certains programmes que les opérations arithmétiques sont correctes, rejettent, malheureusement, de nombreux programmes corrects. D'autres approches de la vérification de programmes avec des nombres à virgule flottante se basent sur la programmation par contraintes. Les contraintes ont été utilisées pour le test de programmes [2], ou encore, la vérification de la conformité des programmes vis-à-vis leurs spécifications. Les outils conçus pour ces approches se limitent souvent au traitement de l'arithmétique des entiers comme, par exemple, Euclide [8] pour le test de programmes et CPBPV [4] pour la vérification de conformité d'un programme vis à vis de sa spécification. Il existe cependant peu d'exemples d'outils de vérification de programmes avec des nombres à virgule flottante basés sur les contraintes, en particulier, à cause de l'absence d'un solveur efficace capable de traiter des contraintes sur les nombres à virgule flottante. En effet, les méthodes disponibles de résolu-

*Ce travail a été partiellement financé par l'ANR, programme SESUR, projet CAVERN (ANR-07-SESUR-003).

tion de contraintes sur les nombres à virgule flottante [13, 12, 3], qui s'appuient sur une adaptation aux flottants d'algorithmes de filtrage sur les réels, peinent à passer à l'échelle. Cela est en partie dû aux difficultés inhérentes à l'arithmétique des nombres à virgule flottante dont la pauvreté des propriétés rend souvent impossible la transposition de résultats établis sur les réels.

Afin d'améliorer le potentiel des solveurs de contraintes sur les nombres à virgule flottante, nous proposons dans cet article de construire une approximation sur les réels, précise et conservative, des solutions des contraintes sur les nombres à virgule flottantes. Une telle approximation pourra bénéficier de l'ensemble des outils développés pour résoudre des problèmes sur les réels et, ainsi, offrir des perspectives jusqu'alors interdites par les limitations de l'arithmétique des nombres à virgule flottante.

L'article est organisé comme suit : la section suivante rappelle quelques notations et notions de base utiles à la compréhension de cet article. La section 3 introduit notre méthode à l'aide d'un exemple illustratif. Les approximations sur les réels sont détaillées dans la section 4 alors que la section 5 décrit leur mise en œuvre. Enfin, la section 6 compare notre méthode aux travaux connexes.

2 Notations et définitions

2.1 Les nombres à virgule flottante

Les nombres à virgule flottante -souvent appelés nombres flottants, ou encore flottants- sont une représentation d'un sous-ensemble fini des nombres réels. On note par $\mathbb{F}$ l'ensemble des nombres flottants et $\mathbb{R}$ l'ensemble des nombres réels.

Les nombres flottants sont utilisés dans les programmes informatiques pour approximer des valeurs de type réel. Ils se composent de trois parties : le signe s (0 ou 1), la mantisse m et l'exposant e [7]. Un flottant est représenté par

$$(-1)^s m \times \beta^e$$

où β est la base de calcul (2 dans le cas général).

La mantisse m est représentée par un nombre fixe de chiffres.

$$d_0.d_1 d_2 \cdots d_{p-1} \text{ avec } (0 \leq d_i < \beta)$$

où p est le nombre de chiffres qui composent la mantisse. Notons qu'il existe une relation directe entre p et la précision du calcul. Afin de garantir une représentation unique d'un flottant, la mantisse est normalisée, i.e. $d_0 \neq 0$.

L'exposant est un entier signé délimité par $e_{min} \leq e \leq e_{max}$.

2.2 Arrondi et erreurs d'arrondi

En utilisant les opérations de l'artihmétique sur les réels, le résultat d'une opération sur les flottants n'est, le plus souvent, pas un nombre flottant. Aussi, afin de n'avoir que des nombres flottants à manipuler, ce résultat doit être arrondi vers le flottant le plus proche. L'opération d'arrondi introduit donc une erreur qui peut être mesurée de différentes manières.

L'erreur absolue

L'erreur absolue err_{abs} est la différence entre la valeur réelle et son approximation sur les flottants.

$$err_{abs} = |valeur_{relle} - valeur_{flottante}|$$

L'erreur relative

L'erreur relative ($\epsilon = err_{rel}$) est le rapport entre l'erreur absolue et la valeur réelle.

$$\epsilon = err_{rel} = \frac{err_{abs}}{|valeur_{relle}|}$$

L'erreur en ulp

L'*Ulp* (*Unit in the Last Place*) est la distance qui sépare deux flottants consécutifs [16] et donc, l'erreur maximale qui peut être faite lors d'un arrondi. Plus formellement, la valeur de l'*ulp* pour un nombre flottant x est

$$ulp(x) = \beta^{e_x} \times \beta^{-p+1}$$

où e_x est l'exposant de x, β sa base de calcul (2 en général) et p désigne le nombre de chiffres de la mantisse.

2.3 La norme IEEE-754

En 1985, l'IEEE normalise l'arithmétique des nombres à virgule flottante [10]. La plupart des processeurs récents utilisent cette norme. En 2008, la norme a été révisée [1] afin de, entre autres, lever certaines ambiguïtés, notamment dans les choix d'implémentation.

La norme IEEE-754 définit les nombres à virgule flottante en base 2. Elle définit deux formats principaux pour représenter les nombres à virgule flottante : le format simple précision sur 32 bits (dont 24 pour la mantisse, et 8 pour l'exposant) et le format double précision sur 64 bits (dont 53 pour la mantisse, et 11 pour l'exposant). Il existe également d'autres formats moins spécifiés comme le double étendu ainsi que le quadruple précision introduit lors de la révision de la norme.

Les nombres normalisés et dénormalisés

La norme IEEE-754 distingue les nombres normalisés des nombres dénormalisés. Les nombres normalisés ont une mantisse où la partie entière est égale à 1, i.e. $x = m_x 2^{e_x}$ est normalisé si $m_x = 1.x_1 \cdots x_{p-1}$ et $e_{max} < e_x < e_{min}$. Les nombres dénormalisés servent à représenter les nombres très proches du zéro. Un nombre $x = m_x 2^{e_x}$ est dénormalisé si la partie entière de la mantisse est nulle, i.e. $m_x = 0.x_1 \cdots x_{p-1}$, et l'exposant est le plus petit exposant possible, i.e. $e_x = e_{min}$.

La norme introduit également des valeurs symbolique telles que les infinis ($-\infty$, $+\infty$) et les NaN (*Not a Number*). Les infinis représentent des dépassements de capacité de calcul alors que les NaN représentent le résultat d'une opération impossible comme une division par zéro, ou encore, la racine carrée d'un nombre négatif.

Les modes d'arrondi

La norme IEEE-754 fournit quatre modes d'arrondi : l'arrondi vers 0, vers $-\infty$, vers $+\infty$ et au plus proche. Avec le mode d'arrondi vers $-\infty$, le réel x est arrondi vers le plus grand flottant inférieur à x, alors qu'avec le mode d'arrondi à $+\infty$, il est arrondi vers le plus petit flottant supérieur à x. Le mode d'arrondi vers 0 se comporte comme un arrondi vers $-\infty$ si $x >= 0$, et comme un arrondi vers $+\infty$ dans les autres cas. Pour le mode d'arrondi au plus proche, le nombre réel est arrondi au flottant le plus proche de x. Lorsque x est à équidistance de deux flottants, celui avec une mantisse paire est choisi. La version révisée de la norme [1] définit une variante du mode d'arrondi au plus près appelée *Round to nearest away* qui, dans le cas où x est à équidistance de deux flottants, choisit celui dont la valeur absolue est la plus grande.

Les opérations

La norme définit cinq opérations de base : les 4 opérations arithmétiques ($+$, $-$, $\times$, $/$), et la racine carrée. La norme impose à ces opérations d'être correctement arrondies, i.e. le résultat de ces opérations sur les flottants doit être égal à l'arrondi du résultat obtenu en effectuant l'opération équivalente sur les réels. Par exemple, pour l'addition, si r est la fonction d'arrondi, $+$ l'addition sur les réels et $\oplus$, l'addition sur les flottants, alors

$$x \oplus y = r(x + y)$$

2.4 Notations

Dans la suite de cet article, $\oplus$, $\ominus$, $\otimes$ et $\oslash$ dénotent respectivement une addition, une soustraction, une multiplication ou une division sur les nombres flottants. De la même manière, $+$, $-$, $\times$ et / dénotent les opérations équivalentes sur les réels. Pour un nombre flottant x, x^+ et x^- dénotent respectivement, le successeur et le prédécesseur de x.

3 La méthode proposée

Le principe de base de notre méthode consiste à construire des approximations sur les réels des contraintes sur les nombres flottants. Cette transformation doit conserver le maximum d'information des contraintes originales, et ne doit éliminer aucune solution du système de contraintes initiales. L'intuition de ce principe est illustrée par l'exemple suivant

$$x \oplus 16.0 == 16.0$$

Interprétée sur $\mathbb{R}$, cette contrainte a une solution unique $x = 0$. Par contre, elle en a plusieurs dans l'ensemble des nombres à virgule flottante. Supposons que l'opération est faite avec un mode d'arrondi vers le plus proche. $x \oplus 16$ est égal à 16 si et seulement si le résultat réel de l'opération $x + 16$ est compris entre $16 - \frac{1}{2}ulp(16)$ et $16 + \frac{1}{2}ulp(16)$, i.e.

$$16 - \frac{1}{2}ulp(16) \leq x + 16 \leq 16 + \frac{1}{2}ulp(16)$$

qui est une contrainte sur les réels. Pour cette contrainte, l'ensemble des flottants $x \in [-\frac{1}{2}ulp(16), \frac{1}{2}ulp(16)]$, sont des solutions de la contrainte initiale sur $\mathbb{F}$. Cependant, dans le cas général, il n'est pas possible d'obtenir une approximation aussi précise des solutions.

Pour illustrer ceci, considérons un type particulier de nombres flottants binaires représentés sur 4 bits, dont deux bits pour la mantisse ($p = 2$) et deux pour l'exposant. Ces flottants ne permettent que de représenter des nombres positifs car, pour des raisons de simplicité, le bit de signe est absent. Les valeurs réelles des nombres représentables dans ce format sont

$$\{0, 0.25, 0.5, 0.75, 1.0, 1.5, 2.0, 3.0, 4.0, 6.0\}$$

Par exemple, 0.75 est représenté par $1.1_{(2)} \times 2^{-1}$ et 4.0 par $1.0_{(2)} \times 2^2$. Le mode d'arrondi considéré étant l'arrondi vers $-\infty$, on a, par exemple, $1 \oplus 0.75 = r(1.75) = 1.5$.

Considérons le système de contraintes sur les

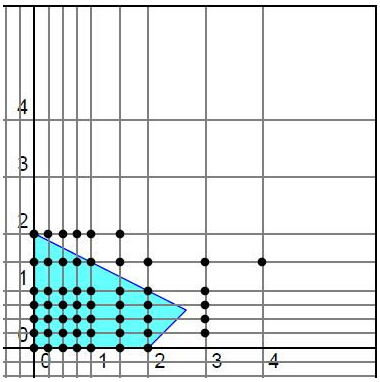


FIG. 1 – Solutions réelles vs solutions flottantes. *L'espace en bleu représente les solutions des contraintes sur les réels. Les points en noir représentent les solutions sur les flottants.*

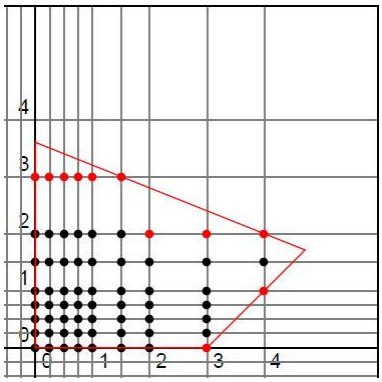


FIG. 2 – Premier niveau d'approximation. *Les points en noirs représentent les solutions des contraintes sur les flottants. Les points en rouge représentent les éléments non-solution des contraintes sur les flottants et qui appartiennent à l'approximation.*

nombres flottants suivant

$$\mathcal{CSP}_{\mathbb{F}} = \begin{cases} x & \geq 0 \\ y & \geq 0 \\ (x \oplus y) \oplus y & \leq 4 \\ x \ominus y & \leq 2 \end{cases}$$

où $x \in \mathbb{F}$ et $y \in \mathbb{F}$. Les solutions flottantes de ces contraintes sont représentées par les points de couleur noire dans la figure 1. À partir de ce système de contraintes, il est possible de construire un système sur les réels en interprétant directement chaque opération comme étant une opération sur les réels

$$\mathcal{CSP}_{\mathbb{R}} = \begin{cases} x & \geq 0 \\ y & \geq 0 \\ x + 2 \times y & \leq 4 \\ x - y & \leq 2 \end{cases}$$

où $x \in \mathbb{R}$ et $y \in \mathbb{R}$. La zone bleu de la figure 1 représente les solutions réelles de $\mathcal{CSP}_{\mathbb{R}}$. Notez que toutes les solutions flottantes de $\mathcal{CSP}_{\mathbb{F}}$ ne sont pas incluses dans l'espace des solutions réelles de $\mathcal{CSP}_{\mathbb{R}}$. Par exemple, le point $(3, 0.75)$ satisfait les contraintes sur les flottants, puisque $(3 \oplus 0.75) \oplus 0.75 = 3 \leq 4$ et $(3 \ominus .75) = 2 \leq 2$, alors qu'il ne satisfait pas les contraintes sur les réels, puisque $(3 + 0.75) + 0.75 = 4.5 > 4$ et $(3 - .75) = 2.25 > 2$. Ceci met en évidence la nécessité d'utiliser une approche spécifique pour résoudre correctement les contraintes sur les flottants.

Bien qu'une transposition directe des opérations sur les flottants en opérations sur les réels ne soit pas conservative des solutions sur les flottants, il est possible d'utiliser de construire des approximations correctes des contraintes sur les flottants, c'est à dire, qui conservent toutes les solutions. La figure 2 présente la première approximation conservative que nous avons défini. Cette approximation englobe non seulement toutes les solutions des contraintes sur les flottants mais contient aussi des éléments non-solutions représentés ici par des points rouges. Il est toutefois possible de définir une meilleure approximation, i.e. qui contient moins de points non-solutions sur les flottants. Une telle approximation est présentée dans la figure 3.

Notez qu'avec cette seconde approximation, le nombre de points non-solutions est réduit par huit. Si le nombre d'éléments éliminés semble faible, il peut être plus significatif pour des systèmes de taille importante.

De telles approximation peuvent directement être traitées à l'aide d'algorithmes de filtrage sur les réels et permettre ainsi une réduction notable des domaines des variables. En combinant ce processus avec un solveur de contraintes sur les flottants, il devient possible d'obtenir plus rapidement des solutions correctes du système de contraintes sur les flottants.

En se basant sur ce principe, nous avons proposé une nouvelle méthode dont les étapes principales sont illustrées dans la figure 4. Les contraintes sur les flottants sont d'abord approximées par des contraintes sur les réels. Les détails de cette transformation sont donnés dans la section 4. Les contraintes obtenues sont ensuite traitées par des algorithmes de filtrage issus d'un solveur sur les réels. S'il n'existe aucune solution pour le système de contraintes sur les réels, alors les contraintes initiales sur les nombres à virgule flottante n'ont pas de solutions. Dans le cas contraire, une recherche combinée avec un processus valide d'énumération sur les flottants. Ce dernier point n'est toutefois pas abordé dans cet article.

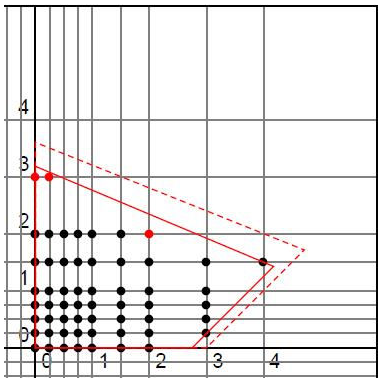


FIG. 3 – Approximation plus précise *Les points noirs représentent les solutions des contraintes sur les flottants. Les points en rouge représentent les éléments non-solution des contraintes sur les flottants et qui appartiennent à l'approximation.*

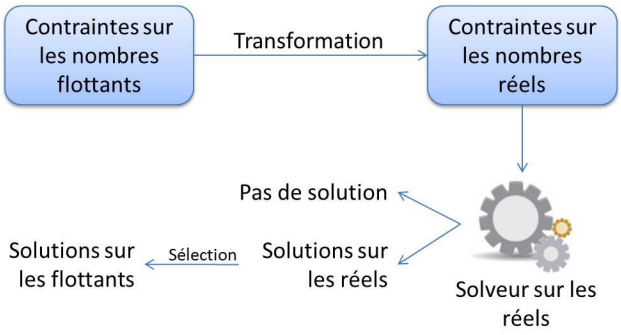


FIG. 4 – Les étapes de la méthode proposée. *La première étape fait une transformation de contraintes entre les réels et les flottants. La deuxième consiste à résoudre les contraintes obtenus. Puis Les solutions flottantes sont choisies*

4 Transformation des contraintes sur les nombres à virgule flottante vers des contraintes sur les réels

Cette étape à pour but de construire un système de contraintes sur les réels qui approxime correctement les contraintes sur les nombres à virgule flottante. Le système construit doit avoir les qualités suivantes :

- Il doit être conservatif en terme de solutions, i.e. aucune solution des contraintes sur les flottants ne doit être perdue.
- La transformation doit être la plus précise possible, i.e. elle doit minimiser le nombre de points qui ne satisfont pas les contraintes sur les nombres flottants et qui sont des solutions pour les contraintes sur les réels.

4.1 Méthode de transformation

La méthode appliquée ici consiste à construire des approximations précises et conservatives des contraintes élémentaires sur les flottants. Cependant, et dans le cas général, les contraintes sur les flottants sont composées d'expressions complexes qu'il s'agit aussi de prendre en compte. Pour cela, notre approche s'appuie sur les propriétés de l'arithmétique des intervalles. Les intervalles vont nous permettre de combiner les approximations obtenues pour les opérations élémentaires afin de traiter des expressions plus complexes. Plus formellement, considérons une contrainte C_i.

$$C_i : f(x_1, \cdots, x_n) \diamond g(x_1, \cdots, x_n)$$

où $\diamond \in \{<, \leq, \geq, >, =\}$, et f et g sont des fonctions sur les nombres flottants avec les opérations de base sur les flottants. Chaque membre de la relation doit être approximé par un intervalle sur les réels

$f(x_1, \cdots, x_n) \in [f_{inf}(x_1, \cdots, x_n), f_{sup}(x_1, \cdots, x_n)]$
$g(x_1, \cdots, x_n) \in [g_{inf}(x_1, \cdots, x_n), g_{sup}(x_1, \cdots, x_n)]$

où f_{inf}, f_{sup}, g_{inf} et g_{sup} sont des fonctions sur les réels. Pour obtenir un système conservatif des solutions, les intervalles doivent contenir toutes les solutions possibles de la fonction sur les flottants. Des approximations initiales sont d'abord définies pour chaque opération de base (cf 4.2) pour être ensuite combinées entre elles grâce à l'arithmétique par intervalle. Le système résultant offre ainsi une approximation générale du problème initial.

Après avoir traité chaque expression constitutive d'une contrainte sur les flottants, les contraintes sur les réels correspondantes peuvent être générées. Pour ce faire, à chaque expression flottante traitée est substituée une variable. Le système résultant prend alors la forme suivante

$$f_{inf}(x_1, \cdots, x_n) \leq x_f \leq f_{sup}(x_1, \cdots, x_n)$$
$$g_{inf}(x_1, \cdots, x_n) \leq x_g \leq g_{sup}(x_1, \cdots, x_n)$$
$$x_f \diamond x_g$$

4.2 Différentes approximations

Nous présentons ici une méthode pour approximer sur les réels les opérations de base sur les flottants. Sans perte de généralité, nous nous limiterons à l'addition $\oplus$ avec un mode d'arrondi vers $-\infty$. Le même raisonnement peut être suivi pour les autres opérations et les autres modes d'arrondi. Nous nous limitons également aux nombres flottants positifs normalisés.

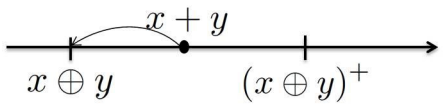


FIG. 5 – La valeur réelle est comprise entre la valeur flottante de l'arrondi et sa valeur suivante.

Approximation par l'erreur en *ulp*

Cette première approximation s'appuie sur le fait que les opérations sont correctement arrondies, i.e. le résultat flottant doit être égal à l'arrondi du résultat réel.

Pour que $(x+y)$ soit arrondi à $(x \oplus y)$ pour un mode d'arrondi vers $-\infty$, il doit être compris entre $(x \oplus y)$ et son successeur, i.e. $(x \oplus y)^+$ (voir figure figure 5). On a donc

$$x \oplus y \leq x + y < (x \oplus y)^+$$

avec $(x \oplus y)^+ = (x \oplus y) + ulp(x \oplus y)$ De cette relation on peut tirer une approximation sur $\mathbb{R}$ pour $(x \oplus y)$

$$(x + y) - ulp(x \oplus y) < x \oplus y \leq x + y$$

Cependant, dans cette relation, la valeur d'$ulp(x \oplus y)$ dépend des valeurs de x et de y. S'il est possible de trouver une borne supérieure de cet Ulp, une approximation plus précise peut cependant être construite. Elle résulte de la propriété suivante

Proposition 1. *Soient x et y des nombres flottants positifs normalisés, et pour un mode d'arrondi fixé à $-\infty$, alors $x \oplus y$ est borné par*

$$\alpha \times (x + y) < x \oplus y \leq x + y$$

avec $\alpha = \dfrac{1}{1 + 2^{-p+1}}$ et où p est le nombre de chiffres de la mantisse.

Démonstration. Pour obtenir cette relation, nous devons d'abord calculer la borne supérieure et la borne inférieure de l'erreur relative ϵ

$$\epsilon = \left| \frac{valeur_{relle} - valeur_{flottante}}{valeur_{relle}} \right|$$

$$\epsilon = \left| \frac{(x + y) - (x \oplus y)}{(x + y)} \right|$$

Sachant que, pour un mode d'arrondi fixé à $-\infty$, la valeur réelle de $(x+y)$ est toujours supérieure au résultat flottant de $(x \oplus y)$, et que le résultat réel est toujours positif puisque x et y sont positifs, ϵ est toujours positif. Notons aussi que le cas $x + y = 0$ correspond à $x \oplus y = 0$, i.e. à une erreur absolue nulle et donc une erreur relative que l'on peut considérer comme nulle. Par la suite, on suppose que $x + y > 0$, x et y étant des nombres flottants positifs normalisés. On a donc

$$\epsilon = \frac{(x + y) - (x \oplus y)}{(x + y)} \geq 0$$

$$= 1 - \frac{x \oplus y}{x + y}$$

Il nous faut maintenant trouver une borne supérieure pour ϵ.

L'addition étant correctement arrondie, on a

$$x \oplus y \leq x + y < x \oplus y + ulp(x \oplus y)$$

donc

$$\frac{1}{x \oplus y + ulp(x \oplus y)} < \frac{1}{x + y}$$

En multipliant chaque membre de la relation par $(x \oplus y)$, on obtient

$$\frac{x \oplus y}{x \oplus y + ulp(x \oplus y)} < \frac{x \oplus y}{x + y}$$

Ce qui, multiplié par -1 et en ajoutant 1 donne

$$1 - \frac{x \oplus y}{x + y} < 1 - \frac{x \oplus y}{x \oplus y + ulp(x \oplus y)}$$

donc

$$0 \leq \epsilon < \frac{ulp(x \oplus y)}{x \oplus y + ulp(x \oplus y)}$$

Posons $z = x \oplus y = m_z 2^{e_z}$ où m_z est la mantisse de z (elle est normalisée puisque les deux opérandes sont normalisées) et e_z l'exposant de z. On a alors

$$0 \leq \epsilon < \frac{ulp(x \oplus y)}{x \oplus y + ulp(x \oplus y)}$$

$$0 \leq \epsilon < \frac{ulp(z)}{z + ulp(z)}$$

La valeur de l'$ulp(z)$ est donnée par $ulp(z) = 2^{-p+1}2^{e_z}$. En remplaçant $ulp(z)$ par cette formule, on obtient

$$0 \leq \epsilon < \frac{ulp(z)}{z + ulp(z)}$$

$$0 \leq \epsilon < \frac{2^{-p+1}2^{e_z}}{m_z 2^{e_z} + 2^{-p+1}2^{e_z}}$$

$$0 \leq \epsilon < \frac{2^{-p+1}}{m_z + 2^{-p+1}}$$

Donc il suffit de trouver une borne supérieure pour $2^{-p+1}/(m_z + 2^{-p+1})$ qui est maximal lorsque $m_z \in [1.0, 2.0[$ est minimal i.e. $m_z = 1.0$.

$$0 \leq \epsilon < \frac{2^{-p+1}}{m_z + 2^{-p+1}} \leq \frac{2^{-p+1}}{1 + 2^{-p+1}}$$

donc

$$0 \leq \epsilon < \frac{2^{-p+1}}{1 + 2^{-p+1}}$$

Il ne reste qu'à trouver une approximation pour $(x \oplus y)$. On a

$$\begin{aligned} 0 \leq \quad & \epsilon & < \frac{2^{-p+1}}{1 + 2^{-p+1}} \\ 0 \leq \quad & \frac{(x+y) - (x \oplus y)}{(x+y)} & < \frac{2^{-p+1}}{1 + 2^{-p+1}} \\ 0 \leq \quad & (x+y) - (x \oplus y) & < (x+y)\frac{2^{-p+1}}{1 + 2^{-p+1}} \end{aligned}$$

On obtient une approximation pour l'opération d'addition

$$(x+y)\frac{1}{1 + 2^{-p+1}} < x \oplus y \leq x + y$$

Donc la valeur de $(x \oplus y)$ peut être approximée par l'intervalle $]\alpha(x+y), x+y]$, avec $\alpha = 1/(1 + 2^{-p+1})$. □

Avec le même raisonnement, un résultat similaire est obtenu pour les autres opérations, toujours avec un mode d'arrondi vers $-\infty$

$$\boxed{\begin{array}{rcl} (x \oplus y) & \in &]\alpha(x+y), x+y] \\ (x \ominus y) & \in &]\alpha(x-y), x-y] \\ (x \otimes y) & \in &]\alpha(x \times y), x \times y] \\ (x \oslash y) & \in &]\alpha(x/y), x/y] \\ \alpha & = & \frac{1}{1+2^{-p+1}} \end{array}}$$

Approximation plus précise

L'approximation précédente peut encore être affinée. En effet, le résultat précédent est basé sur la relation suivante

$$x \oplus y \leq x + y < x \oplus y + ulp(x \oplus y)$$

Or, $(x+y)$ n'atteindra jamais $x \oplus y + ulp(x \oplus y)$. En recherchant le pire cas qui peut réellement être atteint, il est possible de construire une approximation avec une inégalité large.

Comme dans la méthode précédente, nous recherchons une délimitation de l'erreur relative, mais, cette fois, dont les bornes soient atteignable par ϵ. La propriété suivante donne ces bornes

Proposition 2. *Si x et y sont des nombres flottants positifs normalisés, et si le mode d'arrondi est fixé à $-\infty$, alors l'erreur relative de l'opération $x \oplus y$ est bornée par*

$$0 \leq \epsilon \leq \frac{y}{x+y}$$

De plus, l'erreur relative maximal correspond à $(x \oplus y = x)$.

Démonstration. Pour un x donné, nous avons $x \leq x \oplus y$ puisque y est positif (l'addition sur les flottants étant commutative, x et y sont ici interchangeables). Par ailleurs, le résultat réel est toujours supérieur à $x \oplus y$ pour un mode d'arrondi vers $-\infty$. On a donc

$$x \leq x \oplus y \leq x + y$$

À partir de cette relation, on obtient

$$\begin{aligned} x \leq \quad & x \oplus y & \leq x + y \\ \frac{x}{x+y} \leq \quad & \frac{x \oplus y}{x+y} & \leq 1 \\ 0 \leq \quad & 1 - \frac{x \oplus y}{x+y} & \leq \frac{y}{x+y} \end{aligned}$$

Donc l'erreur relative est bornée par

$$0 \leq \epsilon \leq \frac{y}{x+y}$$

Remarquons que $y/(x+y)$ est atteignable par ϵ. En effet, lorsque $(x \oplus y = x)$, i.e. lorsque y est absorbé par x lors de l'opération d'arrondi, l'erreur relative est alors égale à

$$\epsilon = \frac{(x+y) - (x \oplus y)}{(x+y)} = \frac{y}{x+y}$$

□

Cet encadrement plus fin de l'erreur relative nous permet de construire une approximation plus précise de l'addition sur les flottants donnée par la propriété suivante

Proposition 3. *Si x et y sont des nombres flottants positifs normalisés, et si le mode d'arrondi est fixé à $-\infty$, alors $x \oplus y$ est délimité par*

$$\gamma \times (x+y) \leq x \oplus y \leq x + y$$

avec $\gamma = \dfrac{1}{1 + (2^{-p+1} - 2^{-2p+1})}$.

Démonstration. À la fin de la démonstration de la propriété précédente, nous avons établi que l'erreur relative atteint sa borne supérieure pour

$$x \oplus y = x$$

Nous savons aussi que, pour que l'addition soit correctement arrondie, il faut que

$$x \oplus y \leq \quad x + y \quad < (x \oplus y) + ulp(x \oplus y)$$

En combinant ces deux résultats, nous obtenons

$$\begin{array}{rcl} x \leq & x + y & < x + ulp(x) \\ 0 \leq & y & < ulp(x) \end{array}$$

Donc, pour que $x \oplus y = x$, on a $y \in [0, ulp(x)[$.

Revenons à l'erreur relative ϵ

$$0 \leq \epsilon \leq \frac{y}{x+y}$$

Nous devons trouver une limite supérieure pour la fonction $y/(x+y)$. La fonction $y/(x+y)$ est croissante par rapport à y. Donc elle est maximale lorsque y est maximal. Sachant que $y \in [0, ulp(x)[$ et $y \in \mathbb{F}$, la plus grande valeur flottante de y est le prédécesseur de $ulp(x)$, i.e. $(ulp(x))^-$. Posons $x = m_x 2^{e_x}$, l'$ulp(x)$ est donc

$$ulp(x) = 2^{e_x} 2^{-p+1} = 1.0 \times 2^{e_x - p + 1}$$

et son prédécesseur a pour valeur

$$\begin{array}{rcl} (ulp(x))^- & = & (1.0 \times 2^{e_x - p + 1})^- \\ & = & 1.11 \cdots 11 \times 2^{e_x - p} \\ & = & (2 - 2^{-p+1}) \times 2^{e_x - p} \\ & = & 2^{e_x - p + 1} - 2^{e_x - 2p + 1} \\ & = & 2^{e_x}(2^{-p+1} - 2^{-2p+1}) \end{array}$$

Remplaçons dans la propriété 2 y par la valeur obtenue. On a

$$\begin{array}{rcl} 0 \leq \quad \epsilon \quad & \leq & \dfrac{y}{x+y} \\ 0 \leq \quad \epsilon \quad & \leq & \dfrac{2^{e_x}(2^{-p+1} - 2^{-2p+1})}{m_x 2^{e_x} + 2^{e_x}(2^{-p+1} - 2^{-2p+1})} \\ 0 \leq \quad \epsilon \quad & \leq & \dfrac{2^{-p+1} - 2^{-2p+1}}{m_x + (2^{-p+1} - 2^{-2p+1})} \end{array}$$

Il suffit donc de trouver une borne supérieure à $2^{-p+1} - 2^{-2p+1}/(m_x + (2^{-p+1} - 2^{-2p+1}))$. Cette fonction est maximale lorsque m_x est minimal, i.e. $m_x = 1.0$. On a donc

$$0 \leq \quad \epsilon \quad \leq \frac{2^{-p+1} - 2^{-2p+1}}{1 + (2^{-p+1} - 2^{-2p+1})}$$

En reprenant le même raisonnement que celui de la fin de la preuve de la première propriété, nous pouvons maintenant trouver l'approximation qui correspond à cette erreur relative

$$\gamma \times (x+y) \leq x \oplus y \leq x + y$$

$$\gamma = \frac{1}{1 + (2^{-p+1} - 2^{-2p+1})}$$

□

Notons que $1 + 2^{-p+1} - 2^{-2p+1} < 1 + 2^{-p+1}$ et que donc $\gamma > \alpha$. Cette seconde approximation est donc meilleure que la première.

Un résultat identique peut être obtenu pour la soustraction lorsque $x \geq y$. Cependant, ce n'est pas possible pour la multiplication et la division pour lesquelles seule la première approximation est correcte.

Application à l'exemple illustratif

Revenons à l'exemple introduit en section 3. En utilisant les approximations de la première proposition, la contrainte $x \ominus y \leq 2$ est approximée par

$$\begin{array}{rcl} x \ominus y & \in &]\alpha(x-y), x-y] \\ 2 & \in & [2,2] \\ x \ominus y \leq 2 & \Rightarrow &]\alpha(x-y), x-y] \leq [2,2] \\ & \Rightarrow & \alpha(x-y) \leq 2 \\ & \Rightarrow & \frac{1}{1+2^{-1}}(x-y) \leq 2 \\ & \Rightarrow & \frac{2}{3}(x-y) \leq 2 \\ & \Rightarrow & x - y \leq 3 \end{array}$$

La contrainte $(x \oplus y) \oplus y \leq 4$ est approximée par

$$\begin{array}{rcl} x \oplus y & \in &]\alpha(x+y),\ x+y] \\ (x \oplus y) \oplus y & \in &]\alpha(x+y),\ x+y] \oplus [y,y] \\ (x \oplus y) \oplus y & \in &]\alpha^2(x+y) + \alpha y,\ x+2y] \\ (x \oplus y) \oplus y & \in &](\frac{2}{3})^2(x+y) + \frac{2}{3}y,\ x+2y] \\ (x \oplus y) \oplus y & \in &]\frac{4}{9}x + \frac{10}{9}y,\ x+2y] \\ 4 & \in & [4,4] \\ (x \oplus y) \oplus y \leq 4 & \Rightarrow &]\frac{4}{9}x + \frac{10}{9}y,\ x+2y] \leq [4,4] \\ & \Rightarrow & \frac{4}{9}x + \frac{10}{9}y \leq 4 \\ & \Rightarrow & x + 2.5y \leq 9 \end{array}$$

Nous obtenons donc le système de contraintes suivant

$$\left\{ \begin{array}{rcl} x & \geq & 0 \\ y & \geq & 0 \\ x + 2.5y & \leq & 9 \\ x - y & \leq & 3 \end{array} \right.$$

Les solutions de ce système sont représentées dans la figure 2. En utilisant la seconde approximation, nous obtenons (avec $\gamma = \frac{8}{11}$)

$$\left\{ \begin{array}{rcl} x & \geq & 0 \\ y & \geq & 0 \\ x + 2.375y & \leq & 7.5625 \\ x - y & \leq & 2,75 \end{array} \right.$$

Les solutions de ce second système de contraintes sont représentées par la figure figure 3.

Le solveur de contraintes sur les réels utilisé peut imposer des limitations sur les coefficients des contraintes. Le plus souvent, ces solveurs utilisent les flottants de type double pour effectuer leurs calculs. Les coefficients des contraintes construites ici doivent prendre en compte ces limitations. Par exemple, lors du calcul de α pour la contrainte $\alpha(x+y) < x \oplus y \leq x+y$, $1+2^{-p+1}$ est calculé avec un arrondi à $+\infty$ et son inverse avec un arrondi vers $-\infty$ afin garantir que toutes les solutions sont préservées.

5 Traitement des approximations sur les réels

L'ensemble des approximations sur les réels précédement construites forme un système de contraintes sur les réels qu'il s'agit de traiter à l'aide d'algorithmes de filtrage sur les réels. Le choix de cet algorithme dépend du problème traité. Si les contraintes sont linéaires, on peut utiliser un solveur de programmation linéaire sur les réels tel que Cplex[1]. Lorsque le système obtenu n'est pas linéaire, il est possible d'utiliser un solveur non-linéaire tel que Icos ou encore de linéariser le problème.

Dans tout les cas, si l'algorithme de filtrage sur les réels utilisé montre qu'il n'existe aucune solution pour ce système de contraintes, alors le système initial de contraintes sur les flottants n'a, lui non plus, pas de solution.

$$S_{\mathbb{R}} = \emptyset \text{ et } S_{\mathbb{F}} \subseteq S_{\mathbb{R}} \Rightarrow S_{\mathbb{F}} = \emptyset$$

Cependant, dans le cas général, l'application de l'algorithme de filtrage sur les réels va uniquement se traduire par des réductions des domaines des variables. Potentiellement, un tel système admet des solutions sur les réels, solutions qui sont rarement des solutions du système de contraintes sur $\mathbb{F}$. Il est donc nécessaire d'utiliser un processus différent pour énumérer sur les flottants.

6 Travaux connexes

Michel et al. furent les premiers à s'intéresser à la problématique de la résolution de contraintes sur les nombres à virgule flottante [13]. Ces premiers travaux adaptent l'algorithme de *Box*-consistence aux nombres à virgule flottante, un algorithme initialement conçu pour le filtrage de contraintes sur les réels. Il consiste à réduire les intervalles de variables en éliminant les parties qui ne contiennent pas de solutions. Dans une seconde approche proposée dans [12], des projections de contraintes élémentaires sur les flottants ont permis de définir une version de la $2B$-consistence dédiée aux nombres à virgule flottante. Les fonctions de projection ont également été étendues dans [3]. Si ces deux approches permettent de résoudre des systèmes de contraintes sur les flottants, elles peinent à traiter des systèmes de contraintes conséquents. La démarche proposée dans cet article peut être vue comme une approche complémentaire de ces premiers solveurs sur les flottants : grâce à ces algorithmes sur les réels, des réductions supplémentaires des domaines des variables deviennent possibles.

D'autres travaux adoptent une démarche similaire à la notre, i.e. la sur-approximation de contraintes. Par exemple, les travaux de Miné [15, 14] étendent l'interprétation abstraite au cas des nombres à virgule flottante pour prouver l'absence des erreurs d'exécution telle que la division par zéro et le débordement. Cette approche consiste aussi à faire des sur-approximations sur les réels en calculant un sur ensemble des états atteignables. Cette méthode à été mise en œuvre dans l'outil Astrée [5]. Cependant, les sur-approximations introduites sont cependant moins précise que les approximations que nous avons présentées. Il existe d'autres travaux basée sur l'interprétation abstraite qui consistent à estimer l'erreur d'arrondi pour une expression sur les nombres flottants [9]. Ces travaux requièrent, à priori, les valeurs des variables pour en estimer l'erreur. Par ailleurs, cette dernière approche ne correspond pas à nos besoins qui sont le calcul de l'ensemble des états atteignables.

7 Conclusion

Nous avons présenté dans cet article une méthode de résolution de contraintes sur les nombres à virgule flottante. L'approche consiste à construire une approximation précise et conservative sur les réels du système initial de contraintes sur les nombres à virgule flottante. L'approximation construite peut alors être traitée par des algorithmes sur les réels sans qu'aucune solution sur les flottants ne soit perdue. Si les résultats préliminaires restent encourageants, une phase de validation de notre approche reste encore nécessaire. Actuellement, nous nous concentrons sur son implémentation et sa mise en œuvre effective dans le cadre de la vérification et du test de programmes.

[1] http ://www-01.ibm.com/software/integration/ optimization/cplex/

Références

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1 –58, 29 2008.

[2] S. Bardin, B. Botella, F. Dadeau, F. Charreteur, A. Gotlieb, B. Marre, C. Michel, M. Rueher, and N. Williams. Constraint-Based Software Testing. *Journée du GDR-GPL*, 9.

[3] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability*, 16(2), 2006.

[4] H. Collavizza, M. Rueher, and P. Van Hentenryck. CPBPV : A Constraint-Programming Framework for Bounded Program Verification. *Proc. of CP2008*, pages 327–341.

[5] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Mine, D. Monniaux, and X. Rival. The ASTREE analyzer. *Lecture Notes in Computer Science*, 3444 :21–30, 2005.

[6] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7) :1165–1178, 2008.

[7] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1) :5–48, 1991.

[8] A. Gotlieb. Euclide : A Constraint-Based Testing Framework for Critical C Programs. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation-Volume 00*, pages 151–160. IEEE Computer Society Washington, DC, USA, 2009.

[9] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations : a simple abstract interpreter. In *European Symposium on Programming, ESOP'02*, volume 2305. Springer.

[10] IEEE. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard*, 754, 1985.

[11] J.L. Lions et al. Ariane 5 flight 501 failure. *Report by the Inquiry Board, Paris*, 19, 1996.

[12] C. Michel. Exact projection functions for floating point number constraints. In *Proc. of 7th AIMA Symposium, Fort Lauderdale (US)*, 2002.

[13] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. *Lecture Notes in Computer Science*, 2239 :524–538, 2001.

[14] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. *Lecture Notes in Computer Science*, 2986 :3–17, 2004.

[15] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004.

[16] J.M. Muller, N. Brisebarre, F. de Dinechin, C.P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, S. Torrès, S. Pion, et al. *Handbook of Floating-Point Arithmetic*. 2009.

[17] Washington DC 20548 Report Number : GAOfiMTEC-92-26 Or Publisher : US General Accounting Office, GAO. Patriot Missile Defense : Software Problem Led to System Failure at Dhahran, Saudi Arabia. 1992.

La contrainte Increasing Nvalue

Nicolas Beldiceanu, Fabien Hermenier, Xavier Lorca et Thierry Petit

Mines-Nantes, LINA UMR CNRS 6241,
4, rue Alfred Kastler, FR-44307 Nantes, France.
{Nicolas.Beldiceanu,Fabien.Hermenier,Xavier.Lorca,Thierry.Petit}@mines-nantes.fr

Résumé

Cet article introduit la contrainte INCREASING_NVALUE, qui restreint le nombre de valeurs distinctes affectées à une séquence de variables, de sorte que chaque variable de la séquence soit inférieure ou égale à la variable la succédant immédiatement. Cette contrainte est une spécialisation de la contrainte NVALUE, motivée par le besoin de casser des symétries. Il est bien connu que propager la contrainte NVALUE est un problème NP-Difficile. Nous montrons que la spécialisation au cas d'une séquence ordonnée de variables rend le problème polynomial. Nous proposons un algorithme d'arc-consistance ayant une complexité temporelle en $O(\Sigma_{\mathrm{Di}})$, où Σ_{Di} est la somme des tailles des domaines. Cette algorithme est une amélioration significative, en termes de complexité, des algorithmes issus d'une représentation de la contrainte INCREASING_NVALUE à l'aide d'automates ou de la contrainte SLIDE. Nous utilisons notre contrainte dans le cadre d'un problème d'allocation de ressources.

1 Introduction

La contrainte NVALUE [9] exprime une restriction du nombre de valeurs distinctes affectées à un ensemble de variables. Bien que tester si NVALUE admet une solution ou pas soit un problème NP-Difficile [5], un certain nombre d'algorithmes de filtrage ont été développés ces dernières années [3, 2]. Dans le but de casser des symétries, nous présentons dans cet article la contrainte INCREASING_NVALUE, qui représente la conjonction de NVALUE avec une chaîne de contraintes binaires d'inégalité non strictes. Nous proposons un algorithme de filtrage qui réalise la consistance d'arc généralisée (GAC) pour INCREASING_NVALUE avec une complexité temporelle en $O(\Sigma_{\mathrm{Di}})$, où Σ_{Di} est la somme des tailles des domaines. Cet algorithme est plus efficace les algorithmes issus d'une représentation de la contrainte INCREASING_NVALUE à l'aide d'automates finis déterministes [11] ou de la contrainte SLIDE [4], qui réalisent le même filtrage en, respectivement, $O(n(\cup_{Di})^3)$ et $O(nd^4)$, où n est le nombre de variables, $\cup_{Di}$ la taille de l'union des domaines, et d la taille maximale d'un domaine. Cette efficacité provient en partie d'une structure de données spécifique, une matrice creuse avec accès ordonnés aux colonnes.

Nos expérimentations sont réalisées sur un problème réel d'allocation de ressources, relatif à la gestion de grappes dans un réseau de machines. *Entropy* est une machine virtuelle (VM) pour la gestion de grappes [7], qui propose un moteur flexible et autonome pour manipuler l'état et la position de VM sur les différents nœuds composant une grappe. La programmation par contraintes intervient dans l'affectation des VM (tâches) sur un nombre réduit de nœuds (les ressources) dans la grappe. La contrainte NVALUE intervient pour maintenir le nombre de nœuds nécessaires pour couvrir toutes les VM. Cependant, en pratique on constate que les ressources consommées sont le plus souvent équivalentes d'une VM à une autre. Cette propriété induit qu'il existe un nombre limité de classes d'équivalences de VM, et on peut profiter du filtrage polynomial et linéaire de la contrainte INCREASING_NVALUE pour casser les symétries induites par les classes d'équivalence.

La section 2 rappelle quelques définitions et introduit formellement la contrainte INCREASING_NVALUE. La section 3 décrit une condition nécessaire et suffisante pour tester l'existence d'une solution pour INCREASING_NVALUE. La section 4 présente l'algorithme réalisant GAC. La section 5 évalue l'impact de la contrainte sur le problème d'allocation de ressource. Enfin, nous décrivons dans la section 6 les approches génériques pour reformuler INCREASING_NVALUE, qui s'avèrent être moins efficaces que notre algorithme.

2 Définitions

Étant donnée une séquence X de variables, le *domaine* $\mathrm{D}(x)$ d'une *variable* $x \in X$ est l'ensemble fini des valeurs entières pouvant être affectées à cette variable. $\mathcal{D}$ est l'union des domaines des variables X. On note $\min(x)$ la valeur minimale de $D(x)$ et $\max(x)$ sa valeur maximale. La somme des tailles des domaines de $\mathcal{D}$ est $\Sigma_{\mathrm{Di}} = \sum_{x_i \in X} |D(x_i)|$. $A[X]$ désigne une affectation de valeurs aux variables X. Étant donnée $x \in X$, $A[x]$ est la valeur de x dans $A[X]$. $A[X]$ est *valide* ssi $\forall x_i \in X, A[x_i] \in \mathrm{D}(x_i)$. Une *instantiation* $I[X]$ est une affectation complète de valeurs aux variables X. Étant donnée $x \in X$, $I[x]$ est la valeur de x dans $I[X]$. Une contrainte $C(X)$ spécifie les combinaisons de valeurs acceptées pour un ensemble de variables X. Elle définit un sous ensemble $\mathcal{R}_C(\mathcal{D})$ du produit cartésien des domaines $\Pi_{x_i \in X} \mathrm{D}(x_i)$. Toute instantiation solution de $C(X)$ appartient à $\mathcal{R}_C(\mathcal{D})$. On emploie le terme d'instantiation *faisable* ou *réalisable*, ou encore, on dira que si $I[X]$ est une solution de $C(X)$ alors $I[X]$ *satisfait* $C(X)$. Sans perte de généralité, nous considérons dans cet article que X contient au moins deux variables. Étant donnés $X = [x_0, \ldots, x_{n-1}]$ et deux entiers i et j tels que $0 \leq i < j \leq n-1$, $I[x_i, \ldots, x_j]$ est la projection de $I[X]$ sur $[x_i, \ldots, x_j]$.

Définition 1 Increasing_Nvalue(N, X) *est définie par une variable N et une séquence de n variables $X = [x_0, \ldots, x_{n-1}]$. Soit une instantiation de $[N, x_0, \ldots, x_{n-1}]$. Elle satisfait* Increasing_Nvalue(N, X) *ssi :*

1. *N est égale au nombre de valeurs distinctes affectées aux variables X,*
2. *$\forall i \in [0, n-2]$, $x_i \leq x_{i+1}$.*

3 Faisabilité de Increasing Nvalue

Cette section présente une condition de faisabilité nécessaire et suffisante pour Increasing_Nvalue. Nous montrons tout d'abord que le nombre de valeurs distinctes d'une instantiation $I[X]$ telle que $\forall i \in [0, n-2]$, $I[x_i] \leq I[x_{i+1}]$ est égal au nombre de stretchs dans $I[X]$: un *stretch* [10] est une séquence de variables consécutives de longueur maximale qui sont affectées à la même valeur. Pour chaque valeur v de chaque domaine $D(x)$, nous calculons les nombres minimal et maximal de stretchs parmi toutes les instantiations $I[X]$ telles que $I[x] = v$. Ensuite, étant donnée une variable $x \in X$, nous établissons les propriétés liant l'ordre naturel des valeurs dans $D(x)$ et les nombres de stretchs minimal et maximal qui peuvent être obtenus en affectant une valeur à x. À partir de ces propriétés, nous prouvons qu'il existe une instantiation satisfaisant Increasing_Nvalue(N, X) pour chaque valeur de $\mathrm{D}(N)$ comprise entre le minimum $\underline{s}(X)$ et le maximum $\overline{s}(X)$ des nombres de stretchs possibles. Cette propriété nous amène au résultat majeur de la section : Increasing_Nvalue(N, X) est réalisable ssi $\mathrm{D}(N) \cap [\underline{s}(X), \overline{s}(X)] \neq \emptyset$ (Proposition 3 de la Section 3.3).

3.1 Estimation du nombre de stretchs

Une instantiation réalisable $I[X]$ de Increasing_Nvalue(N, X) satisfait la propriété suivante : $I[x_i] \leq I[x_j]$ pour tout $i < j$. Dans la suite de l'article, une instantiation $I[x_0, x_1, \ldots, x_{n-1}]$ est dite *bien ordonnée* ssi pour tout i et tout j tels que $0 \leq i < j \leq n-1$, on a $I[x_i] \leq I[x_j]$. Une valeur $v \in \mathrm{D}(x)$ est bien ordonnée pour x ssi elle peut appartenir à au moins une instantiation bien ordonnée.

Lemme 1 *Soit $I[X]$ une instantiation. Si $I[X]$ satisfait* Increasing_Nvalue(X, N) *alors $I[X]$ est bien ordonnée.*

Preuve : D'après la Définition 1, si $I[X]$ satisfait Increasing_Nvalue alors $\forall i \in [0, n-2]$, $I[x_i] \leq I[x_{i+1}]$. Par transitivité de $\leq$, le lemme est correct.

Définition 2 (stretch) *Soit $I[x_0, \ldots, x_{n-1}]$ une instantiation. Soient i et j tels que $0 \leq i \leq j \leq n-1$. Un* stretch *de $I[X]$ est une séquence de variables consécutives $[x_i, \ldots, x_j]$ telle que dans $I[X]$: (1) $\forall k \in [i, j]$, $\forall \ell \in [i, j]$, $x_k = x_\ell$. (2) soit $i = 0$ soit $x_{i-1} \neq x_i$. (3) soit $j = n-1$ soit $x_j \neq x_{j+1}$.*

Lemme 2 *Soit $I[X]$ une instantiation bien ordonnée. Le nombre de stretchs dans $I[X]$ est égal au nombre de valeurs distinctes dans $I[X]$.*

Preuve : $I[X]$ est bien ordonnée, donc pour tout i et tout j tels que $0 \leq i < j \leq n-1$, on a $I[x_i] \leq I[x_j]$. En conséquence, si x_i et x_j appartiennent à deux stretchs distincts (et $i < j$) alors $I[x_i] < I[x_j]$. La réciproque est trivialement vraie.

Il est possible d'évaluer pour chaque valeur v dans chaque domaine $\mathrm{D}(x_i)$ les minima et maxima exacts du nombre de stretchs d'instantiation suffixes bien ordonnées $I[x_i, \ldots, x_n]$ telles que $I[x_i] = v$, et similairement pour des instantiations préfixes.

Notation 1 *Soit $X = [x_0, x_1, \ldots, x_{n-1}]$ une séquence de variables et soit v une valeur dans $\mathcal{D}$. Le nombre minimal (exact) de stretchs parmi toutes les instantiations bien ordonnées $I[x_i, \ldots, x_{n-1}]$ telles que $I[x_i] = v$ est noté $\underline{s}(x_i, v)$. Par convention, si $v \notin \mathrm{D}(x_i)$*

alors $\underline{s}(x_i, v) = +\infty$. *Similairement, le nombre minimal (exact) de stretchs parmi toutes les instantiations bien ordonnées* $I[x_0, \ldots, x_i]$ *telles que* $I[x_i] = v$ *est noté* $\underline{p}(x_i, v)$. *Par convention, si* $v \notin \mathrm{D}(x_i)$ *alors* $\underline{p}(x_i, v) = +\infty$.

Lemme 3 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables.* $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, $\underline{s}(x_i, v)$ *peut être calculé ainsi :*

1. *Si* $i = n-1$: $\underline{s}(x_i, v) = 1$,
2. *Si* $i < n-1$: $\underline{s}(x_i, v) = \min(\ \underline{s}(x_{i+1}, v),\ \min_{w>v}(\underline{s}(x_{i+1}, w)) + 1\ )$.

Preuve : par induction. Si $|X| = 1$ il y a un stretch, donc quelque soit $v \in \mathrm{D}(x_i)$, on a $\underline{s}(x_i, v) = 1$. Considérons à présent une variable x_i, $i < n-1$, et une valeur $v \in \mathrm{D}(x_i)$. Les instantiations telles que $I[x_{i+1}] < v$ ne peuvent pas être étendues avec la valeur v pour x_i pour former une instantiation bien ordonnée $I[x_i, \ldots, x_{n-1}]$. Soit $I[x_{i+1}, \ldots, x_{n-1}]$ une instantiation telle que $I[x_{i+1}] \geq v$, minimisant le nombre de stretchs dans $[x_{i+1}, \ldots, x_{n-1}]$. Soit $I[x_{i+1}] = v$ et $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$, car le premier stretch de $I[x_{i+1}, \ldots, x_{n-1}]$ est étendu lorsqu'on ajoute à $I[x_{i+1}, \ldots, x_{n-1}]$ la valeur v pour x_i, soit $I[x_{i+1}] \neq v$ et $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, I[x_{i+1}]) + 1$ car v crée un nouveau stretch. Par construction, les instantiations des variables $[x_{i+1}, \ldots, x_{n-1}]$ ne minimisant pas le nombre de stretchs ne peuvent pas aboutir sur une valeur $\underline{s}(x_i, v)$ strictement inférieure à $\min(\underline{s}(x_{i+1}, w), w > v)+1$, et ce même si $I[x_{i+1}] = v$.

Étant donnée une séquence de variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, calculer $\underline{p}(x_i, v)$ est symétrique : si $i = 0$: $\underline{p}(x_i, v) = 1$. Si $i > 0$: $\underline{p}(x_i, v) = \min(\ \underline{p}(x_{i-1}, v),\ \min_{w<v}(\underline{p}(x_{i-1}, w)) + 1\ )$.

De plus, étant donnée une variable x_i, nous évaluons pour chaque valeur v le nombre maximal (exact) de stretchs qui peuvent apparaître, parmi toutes les instantiations suffixes bien ordonnées $I[x_i, \ldots, x_{n-1}]$ avec $I[x_i] = v$, et similairement pour les instantiations préfixes.

Notation 2 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables et* v *une valeur dans* $\mathcal{D}$. *Le nombre maximal (exact) de stretchs parmi toutes les instantiations bien ordonnées* $I[x_i, \ldots, x_{n-1}]$ *avec* $I[x_i] = v$ *est noté* $\overline{s}(x_i, v)$. *Par convention, si* $v \notin \mathrm{D}(x_i)$ *alors* $\overline{s}(x_i, v) = 0$. *Similairement, le nombre maximal (exact) de stretchs parmi toutes les instantiations bien ordonnées* $I[x_1, \ldots, x_i]$ *avec* $I[x_i] = v$ *est noté* $\overline{p}(x_i, v)$. *Par convention, si* $v \notin \mathrm{D}(x_i)$ *alors* $\overline{p}(x_i, v) = 0$.

Lemme 4 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables.* $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, $\overline{s}(x_i, v)$ *peut être calculé ainsi :*

1. *Si* $i = n-1$: $\overline{s}(x_i, v) = 1$,
2. *Si* $i < n-1$: $\overline{s}(x_i, v) = \max(\ \overline{s}(x_{i+1}, v),\ \max_{w>v}(\overline{s}(x_{i+1}, w)) + 1\ )$.

Preuve : Similaire au Lemme 3.

Étant donnée une séquence de variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, le calcul de $\overline{p}(x_i, v)$ est symétrique : si $i = 0$: $\overline{p}(x_i, v) = 1$, si $i > 0$: $\overline{p}(x_i, v) = \max(\ \overline{p}(x_{i-1}, v),\ \max_{w<v}(\overline{s}(x_{i-1}, w)) + 1\ )$.

3.2 Propriétés sur le nombre de stretchs

Nous considérons exclusivement des valeurs bien ordonnées, pouvant appartenir à une instantiation réalisable pour Increasing_Nvalue.

3.2.1 Propriétés sur une unique valeur

Les propriétés suivantes sont directement déduites, par construction, des Lemmes 3 et 4.

Propriété 1 *Toute valeur* $v \in \mathrm{D}(x_i)$ *bien ordonnée pour* x_i *est telle que* $\underline{s}(x_i, v) \leq \overline{s}(x_i, v)$.

Propriété 2 *Soit* $v \in \mathrm{D}(x_i)$ *(*$i < n-1$*) une valeur bien ordonnée pour* x_i. *Si* $v \in \mathrm{D}(x_{i+1})$ *et* v *est bien ordonnée pour* x_{i+1} *alors* $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$.

Propriété 3 *Soit* $v \in \mathrm{D}(x_i)$ *(*$i < n-1$*) une valeur bien ordonnée pour* x_i. *Si* $v \in \mathrm{D}(x_{i+1})$ *et* v *est bien ordonnée pour* x_{i+1} *alors* $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, v)$.

Preuve : D'après le Lemme 4, s'il existe une valeur $w \in D(x_{i+1})$, $w > v$ qui est bien ordonnée pour x_{i+1} et telle que $\overline{s}(x_{i+1}, w) \geq \overline{s}(x_{i+1}, v)$ alors $\overline{s}(x_i, v) > \overline{s}(x_{i+1}, v)$. Dans le cas contraire, $\overline{s}(x_i, v) = \overline{s}(x_{i+1}, v)$.

3.2.2 Ordre sur les valeurs

Les deux propriétés suivantes établissent les liens entre l'ordre naturel des valeurs dans un domaine $D(x_i)$ avec le nombre minimal et maximal de stretchs que l'on peut obtenir dans la sous-séquence $[x_i, x_{i+1}, \ldots, x_{n-1}]$.

Propriété 4 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables et* $i \in [0, n-1]$ *un entier.* $\forall v, w \in \mathrm{D}(x_i)$ *deux valeurs bien ordonnées,* $v \leq w \Rightarrow \underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$.

Preuve : Si $v = w$ alors la propriété est vraie. Si $i = n-1$, d'après le Lemme 3, $\underline{s}(x_{n-1}, v) = \underline{s}(x_{n-1}, w) = 1$. La propriété est vérifiée. Étant donné $i < n-1$, soit v', w' deux valeurs bien ordonnées de $\mathrm{D}(x_{i+1})$ telles que $v' \geq v$ et $w' \geq w$, qui minimisent le nombre de stretchs débutant en x_{i+1} : $\forall \alpha \geq v$, $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_{i+1}, \alpha)$ et $\forall \beta \geq w$, $\underline{s}(x_{i+1}, w') \leq \underline{s}(x_{i+1}, \beta)$. Ces valeurs existent car v et w sont bien ordonnées. Par construction, on a $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_{i+1}, w')$, et d'après le Lemme 3, $\underline{s}(x_{i+1}, w') \leq \underline{s}(x_i, w)$, ce qui entraîne $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_i, w)$. D'après le Lemme 3, $\underline{s}(x_i, v) \leq \underline{s}(x_{i+1}, v') + 1$. Ainsi, $\underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$.

Une propriété symétrique existe concernant le nombre maximal de stretchs.

Propriété 5 *Soit $X = [x_0, x_1, \ldots, x_{n-1}]$ une séquence de variables et $i \in [0, n-1]$ un entier. $\forall v, w \in \mathrm{D}(x_i)$ deux valeurs bien ordonnées, $v \leq w \Rightarrow \overline{s}(x_i, v) \geq \overline{s}(x_i, w)$.*

Preuve : si $v = w$ alors la propriété est vraie. Si $i = n-1$, d'après le Lemme 4, $\overline{s}(x_{n-1}, v) = \overline{s}(x_{n-1}, w) = 1$. La propriété est vérifiée. Étant donné $i < n-1$, soit $w' \in \mathrm{D}(x_{i+1})$ une valeur bien ordonnée telle que $w' \geq w$, qui maximise le nombre de stretchs débutant en x_{i+1} : $(\forall \beta \geq w,\ \overline{s}(x_{i+1}, w') \geq \overline{s}(x_{i+1}, \beta))$. D'après le Lemme 4, $\overline{s}(x_i, w) \leq \overline{s}(x_{i+1}, w') + 1$. Sachant que $v < w$ et donc $v < w'$, $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, w') + 1$. La propriété est vraie.

3.2.3 Ordre sur le nombre maximal de stretchs

L'intuition de la propriété 6 réside dans le fait que, plus une valeur bien ordonnée pour une variable x_i est petite, plus le nombre de stretchs qui peuvent être obtenus sur la séquence $[x_i, x_{i+1}, \ldots, x_{n-1}]$, à partir de cette valeur, est grand.

Propriété 6 *Soit $X = [x_0, x_1, \ldots, x_{n-1}]$ une séquence de variables et soit i un entier dans $[0, n-1]$. $\forall v, w \in \mathrm{D}(x_i)$ deux valeurs bien ordonnées, $\overline{s}(x_i, w) < \overline{s}(x_i, v) \Rightarrow v < w$.*

Preuve : Nous prouvons que si $v \geq w$ alors il existe une contradiction avec $\overline{s}(x_i, w) < \overline{s}(x_i, v)$. Si $i = n-1$, le Lemme 4 assure $\overline{s}(x_{n-1}, w) = \overline{s}(x_{n-1}, v) = 1$, une contradiction. Considérons le cas $i < n-1$. Si $v = w$ alors $\overline{s}(x_i, w) = \overline{s}(x_i, v)$, une contradiction. Sinon $(v > w)$, soit v' une valeur de $\mathrm{D}(x_{i+1})$ telle que $v' \geq v$, qui maximise $\overline{s}(x_{i+1}, \alpha)$, $\alpha \geq v$. Une telle valeur existe car v est bien ordonnée. Par construction $w < v'$. D'après le Lemme 4, $\overline{s}(x_i, w) \geq \overline{s}(x_{i+1}, v') + 1$ (1). Par construction on a aussi $v \leq v'$, qui entraîne $\overline{s}(x_{i+1}, v') + 1 \geq \overline{s}(x_i, v)$ (2). D'après (1) et (2) on a $\overline{s}(x_i, w) \geq \overline{s}(x_i, v)$, une contradiction.

3.2.4 Ordre sur le nombre minimal de stretchs

Il n'existe pas d'implication liant le nombre minimal de stretchs et l'ordre des valeurs dans les domaines. Par exemple, soit $X = [x_0, x_1, x_2]$ avec $D(x_0)$=$D(x_1)$=$\{1, 2, 3\}$ et $D(x_2) = \{1, 2, 4\}$. $\underline{s}(x_0, 1) = 1$ et $\underline{s}(x_0, 3) = 2$, alors $\underline{s}(x_0, 1) < \underline{s}(x_0, 3)$ et $1 < 3$. Supposons à présent que $D(x_2) = \{2, 3, 4\}$. $\underline{s}(x_0, 1) = 2$ et $\underline{s}(x_0, 3) = 1$, alors $\underline{s}(x_0, 3) < \underline{s}(x_0, 1)$ et $3 > 1$.

3.2.5 Résumé

Le tableau ci-dessous récapitule les relations qui existent entre des valeurs bien ordonnées v et w d'un domaine $D(x_i)$ et les estimations du nombre minimal et maximal de stretchs parmi toutes les instantiations débutant par ces valeurs (*i.e.*, $I[x_i, \ldots, x_{n-1}]$ telle que $I[x_i] = v$ ou telle que $I[x_i] = w$).

Pré-condition	Propriété	Prop.
$v \in D(x_i)$ est bien ordonnée	$\underline{s}(x_i, v) \leq \overline{s}(x_i, v)$	Prop. 1
$v \in D(x_i)$ est bien ordonnée, $i < n-1$ et $v \in D(x_{i+1})$	$\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$ $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, v)$	Prop. 2 Prop. 3
$v \in D(x_i)$, $w \in D(x_i)$ sont bien ordonnées et $v \leq w$	$\underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$ $\overline{s}(x_i, v) \geq \overline{s}(x_i, w)$	Prop. 4 Prop. 5
$v \in D(x_i)$, $w \in D(x_i)$ sont bien ordonnées et $\overline{s}(x_i, w) < \overline{s}(x_i, v)$	$v < w$	Prop. 6

3.3 Condition nécessaire et suffisante de faisabilité

Notation 3 *Étant donnée une séquence de variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\underline{s}(X)$ est la valeur minimale de $\underline{s}(x_0, v)$, $v \in \mathrm{D}(x_0)$, et $\overline{s}(X)$ est la valeur maximale de $\overline{s}(x_0, v)$, $v \in \mathrm{D}(x_0)$.*

Proposition 1 *Étant donnée* IncreasingNvalue(N, X), *si $\underline{s}(X) > \max(\mathrm{D}(N))$ alors la contrainte n'a pas de solution. symétriquement, si $\overline{s}(X) < \min(\mathrm{D}(N))$ alors la contrainte n'a pas de solution.*

Preuve : Par construction d'après les Lemmes 3 et 4.

Sans perte de généralité, $\mathrm{D}(N)$ peut être réduit à $[\underline{s}(X), \overline{s}(X)]$. Néanmoins, on peut observer que $\mathrm{D}(N)$ peut avoir des trous ou bien être strictement inclus dans $[\underline{s}(X), \overline{s}(X)]$. Nous montrons que pour toute valeur k dans $[\underline{s}(X), \overline{s}(X)]$ il existe une valeur dans $v \in \mathrm{D}(x_0)$ telle que $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$. Ainsi, toute valeur de $D(N) \cap [\underline{s}(X), \overline{s}(X)]$ est une valeur réalisable.

Proposition 2 *Soit* $X = [x_0, x_1, \ldots, x_{n-1}]$ *une séquence de variables. Pour tout entier* k *dans* $[\underline{s}(X), \overline{s}(X)]$ *il existe* v *dans* $\mathrm{D}(x_0)$ *telle que* $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$.

Preuve : Soit $k \in [\underline{s}(X), \overline{s}(X)]$. Si $\exists v \in \mathrm{D}(x_0)$ telle que $k = \underline{s}(x_0, v)$ ou $k = \overline{s}(x_0, v)$, alors la propriété est vraie par construction. Supposons que $\forall v \in \mathrm{D}(x_0)$, dans ce cas soit $k > \overline{s}(x_0, v)$ soit $k < \underline{s}(x_0, v)$. Soient v', w' deux valeurs telles que v' soit la plus grande valeur de $\mathrm{D}(x_0)$ telle que $\overline{s}(x_0, v') < k$ et w' soit la plus grande valeur telle que $k < \underline{s}(x_0, w')$. Alors, $\overline{s}(x_0, v') < k < \underline{s}(x_0, w')$ (1). D'après la propriété 1, $\underline{s}(x_0, w') \leq \overline{s}(x_0, w')$. D'après la propriété 6, $\overline{s}(x_0, v') < \overline{s}(x_0, w') \Rightarrow w' < v'$. D'après les propriétés 4 et 1, $w' < v' \Rightarrow \underline{s}(x_0, w') \leq \overline{s}(x_0, v') + 1$, qui est en contradiction avec (1).

Algorithm 1: Construction d'une solution pour Increasing_Nvalue(k, X).

```
1 if k ∉ [s(X), s̄(X)] ∩ D(N) then return "no solution" ;
2 v := a value ∈ D(x_0) s.t. k ∈ [s(x_0, v), s̄(x_0, v)] ;
3 for i := 0 to n − 2 do
4 |   I[x_i] := v;
5 |   if ∀v_{i+1} ∈ D(x_{i+1}) s.t. v_{i+1} = v,
  |   k ∉ [s(x_{i+1}, v_{i+1}), s̄(x_{i+1}, v_{i+1})] then
6 |   |   v := v_{i+1} in D(x_{i+1}) s.t. v_{i+1} > v ∧
  |   |   k − 1 ∈ [s(x_{i+1}, v_{i+1}), s̄(x_{i+1}, v_{i+1})];
7 |   |   k := k − 1 ;
8 I[x_{n−1}] := v ; return I[X];
```

Lemme 5 *Étant donnée une contrainte* Increasing_Nvalue(N, X) *et un entier* k*, si* $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$ *alors l'algorithme 1 retourne une solution pour* Increasing_Nvalue(N, X) *avec* $N = k$*. Sinon, il n'existe aucune solution avec* $N = k$.

Preuve : la première ligne de l'algorithme assure que soit $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$ et $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$, soit il n'existe pas de solution (d'après les Propositions 1 et 2). À chaque itération de la boucle **for**, d'après les lemmes 3 et 4 et la proposition 2, soit la condition (ligne 6) est satisfaite et un nouveau stretch débute en $i + 1$ avec une valeur supérieure (qui garantit que $I[\{x_1, \ldots, x_{i+1}\}]$ est bien ordonnée) et k est décrémenté de 1, soit il est possible de garder la valeur courante v pour $I[x_{i+1}]$. C'est pourquoi au début de la boucle (ligne 4), $\exists v \in \mathrm{D}(x_i)$ telle que $k \in [\underline{s}(x_i, v), \overline{s}(x_i, v)]$. Lorsque $i = n - 1$, par construction $k = 1$ et $\forall v_{n-1} \in \mathrm{D}(x_{n-1})$, $\underline{s}(x_{n-1}, v_{n-1}) = \overline{s}(x_{n-1}, v_{n-1}) = 1$; $I[X]$ est bien ordonnée et contient k stretchs. D'après le Lemme 2, l'instantiation $I[\{N\} \cup X]$ with $I[N] = k$ est une solution de Increasing_Nvalue(N, X) avec k valeurs distinctes pour les variables X.

Le Lemme 5 nous amène à une condition nécessaire et suffisante de faisabilité.

Proposition 3 *Étant donnée une contrainte* Increasing_Nvalue(N, X)*, les deux propositions suivantes sont équivalentes :*

1. Increasing_Nvalue(N, X) *admet une solution.*
2. $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$.

Preuve : ($\Rightarrow$) Supposons que la contrainte Increasing_Nvalue(N, X) admette une solution. Soit $I[\{N\} \cup X]$ une telle solution. D'après le Lemme 2 la valeur k affectée à N est égale au nombre de stretchs dans $I[X]$. Par construction (Lemmes 3 et 4) $k \in [\underline{s}(X), \overline{s}(X)]$. Il s'en suit $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$. ($\Leftarrow$) Soit $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$. D'après le Lemme 5 il est possible de construire une solution pour la contrainte Increasing_Nvalue(N, X).

4 GAC pour Increasing Nvalue

Cette section présente un algorithme de filtrage réalisant GAC pour Increasing_Nvalue(N, X) avec une complexité temporelle en $O(\Sigma_{\mathrm{Di}})$, où Σ_{Di} est la somme des tailles des domaines des variables X. Pour une variable $x_i \in X$ et une valeur $v \in D(x_i)$, le principe consiste à déterminer le nombre minimal et le nombre maximal de stretchs parmi toutes les instantiations $I[X]$ telles que $I[x_i] = v$, puis à comparer l'intervalle dérivé de ces deux bornes avec $D(N)$. Dans ce but, et sans perte de généralité, nous déterminons le nombre minimal et le nombre maximal de stretchs relatifs aux instantiations préfixes $I[x_0, \ldots, x_i]$ et aux instantiations suffixes $I[x_i, \ldots, x_{n-1}]$.

Définition 3 (GAC) *Soit une contrainte* $C(X)$*. Un* **support** *sur* $C(X)$ *est une instantiation* $I[X]$ *qui satisfait* $C(X)$*. Un domaine* $\mathrm{D}(x)$ *est* **arc-consistant** *sur* $C(X)$ *ssi* $\forall v \in \mathrm{D}(x)$*,* v *appartient à un support sur* $C(X)$*.* $C(X)$ *a la propriété d'***arc-consistance généralisée** *(GAC) ssi* $\forall x_i \in X$*,* $\mathrm{D}(x_i)$ *est arc-consistant.*

4.1 Condition nécessaire et suffisante de filtrage

D'après le Lemme 5, les valeurs de $D(N)$ qui ne sont pas dans $[\underline{s}(X), \overline{s}(X)]$ peuvent être supprimées de $D(N)$. D'après la Proposition 3, toutes les valeurs restant dans $D(N)$ sont réalisables. Nous fournissons à présent une condition nécessaire et suffisante pour supprimer une valeur d'un domaine $D(x_i)$, $x_i \in X$.

Proposition 4 *Considérons une contrainte* Increasing_Nvalue(N, X)*. Soit* $i \in [0, n-1]$ *un entier et* v *une valeur dans* $D(x_i)$*. Les deux propositions suivantes sont équivalentes :*

1. $v \in D(x_i)$ *est arc-consistant pour* Increasing_Nvalue
2. v *est bien ordonnée relativement à* $D(x_i)$ *et l'intersection* $[\underline{p}(x_i,v)+\underline{s}(x_i,v)-1, \overline{p}(x_i,v)+\overline{s}(x_i,v)-1] \cap \mathrm{D}(N)$ *n'est pas vide.*

Preuve : si v n'est pas bien ordonnée alors d'après le Lemme 1, v n'est pas arc-consistante sur Increasing_Nvalue. Sinon, $\underline{p}(x_i,v)$ est le nombre minimal exact de stretchs parmi les instantiations bien ordonnées $I[x_0,\ldots,x_i]$ telles que $I[x_i]=v$ et $\underline{s}(x_i,v)$ est le nombre minimal exact de stretchs parmi les instantiations bien ordonnées $I[x_i,\ldots,x_{n-1}]$ telles que $I[x_i]=v$. Donc, par construction, $\underline{p}(x_i,v)+\underline{s}(x_i,v)-1$ est le nombre minimal exact de stretchs parmi les instantiations bien ordonnées $I[x_0,\ldots,x_{n-1}]$ telles que $I[x_i]=v$. Soit $\mathcal{D}_v \subseteq \mathcal{D}$ l'ensemble de domaines tels que tous les domaines dans $\mathcal{D}_v$ soient identiques aux domaines de $\mathcal{D}$ sauf $D(x_i)$ qui est réduit au singleton $\{v\}$. Nous appelons X_v l'ensemble des variables associées aux domaines de $\mathcal{D}_v$. D'après la Definition 3, $\underline{p}(x_i,v)+\underline{s}(x_i,v)-1=\underline{s}(X_v)$. Par un raisonnement symétrique, $\overline{p}(x_i,v)+\overline{s}(x_i,v)-1=\overline{s}(X_v)$. D'après la proposition 3, la proposition est vraie.

4.2 Algorithmes

D'après la proposition 4, nous dérivons un algorithme de filtrage réalisant GAC en $O(\Sigma_{\mathrm{Di}})$. Pour une variable x_i $(0 \le i < n)$, nous avons besoin de calculer les informations préfixes et suffixes $\underline{p}(x_i,v)$, $\overline{p}(x_i,v)$, $\underline{s}(x_i,v)$ et $\overline{s}(x_i,v)$, et ce que la valeur v appartienne ou non à $D(x_i)$. Afin d'atteindre la complexité temporelle en $O(\Sigma_{\mathrm{Di}})$, nous tirons parti de deux points :

1. L'algorithme itère toujours sur $\underline{p}(x_i,v)$, $\overline{p}(x_i,v)$, $\underline{s}(x_i,v)$ et $\overline{s}(x_i,v)$ en parcourant $D(x_i)$ en ordre croissant ou décroissant de valeurs.
2. Pour une valeur v n'appartenant pas à $D(x_i)$, 0 (resp. n) est la valeur par défaut pour $\overline{p}(x_i,v)$ et $\overline{s}(x_i,v)$ (resp. $\underline{p}(x_i,v)$ et $\underline{s}(x_i,v)$).

Dans ce cadre, nous utilisons une structure de données pour gérer ces matrices creuses, pour lesquelles les accès en lecture et en écriture sont toujours réalisés en itérant sur les lignes d'une colonne donnée, selon un ordre croissant ou décroissant d'indices sur une colonne . La partie haute du tableau ci-dessous décrit les trois primitives sur ces matrices creuses avec accès ordonné aux colonnes, et fournit leur complexité en temps. La partie basse décrit les primitives utilisées pour accéder ou modifier le domaine d'une variable. Les primitives réduisant le domaine d'une variable x retournent vrai si $D(x) \neq \emptyset$ après l'operation et faux dans le cas contraire.

Algorithm 2: build_suffix($[x_0,\ldots,x_{n-1}], \underline{s}[][], \overline{s}[][]$).

```
1  Allocate mins, maxs;
2  ScanInit({s, s̄}, n − 1, ↓) ; v := max(x_{n−1}) ;
3  repeat
4      Set(s, n − 1, v, 1) ; Set(s̄, n − 1, v, 1) ; w := v ;
       v :=getPrev(x_{n−1}, v);
5  until w = v ;
6  for i := n − 2 downto 0 do
7      ScanInit({s, s̄, mins, maxs}, i + 1, ↓) ; v := max(x_{i+1}) ;
8      repeat
9          if v < max(x_{i+1}) then
10             Set(mins, i + 1, v, min(Get(mins, i + 1, v +
               1),Get(s, i + 1, v)));
11             Set(maxs, i + 1, v, max(Get(maxs, i + 1, v +
               1),Get(s̄, i + 1, v)));
12         else
13             Set(mins, i + 1, v,Get(s, i + 1, v));
14             Set(maxs, i + 1, v,Get(s̄, i + 1, v));
15         w := v ; v :=getPrev(x_{i+1}, v);
16     until w = v ;
17     ScanInit({s, s̄}, i, ↓) ;
       ScanInit({s, s̄, mins, maxs}, i + 1, ↓) ; v := max(x_i) ;
18     repeat
19         if v = max(x_{i+1}) then
20             Set(s, i, v,Get(s, i + 1, v)) ;
               Set(s̄, i, v,Get(s̄, i + 1, v));
21         else
22             if v ≥ min(x_{i+1}) then
23                 Set(s, i, v, min(Get(s, i +
                   1, v),Get(mins, i + 1, v + 1) + 1));
24                 Set(s̄, i, v, max(Get(s̄, i +
                   1, v),Get(maxs, i + 1, v + 1) + 1));
25             else
26                 Set(s, i, v,Get(mins, i + 1, min(x_{i+1})) + 1);
27                 Set(s̄, i, v,Get(maxs, i + 1, min(x_{i+1})) + 1);
28         w := v ; v :=getPrev(x_i, v);
29     until w = v ;
```

Primitives (*accès aux matrices*)	**Description**	**Complex.**
ScanInit(mat, i, dir)	indique que l'on itère sur la i-ème colonne de la matrice mat en ordre croissant d'indices ($dir = \uparrow$) ou décroissant ($dir = \downarrow$)	$O(1)$
Set($mat, i, j, info$)	réalise l'affectation $mat[i, j] := info$	$O(1)$
Get(mat, i, j)	retourne le contenu de l'entrée $mat[i, j]$ ou la valeur par défaut si cette entrée n'appartient pas à la matrice creuse (q appels consécutifs à Get sur la même colonne i et en ordre croissant ou décroissant des indices des lignes coûte $O(q)$).	amortie

Primitives (*accès aux variables*)	**Description**	**Complex.**
adjust_min(x, v) :bool	ajuste le minimum de la variable x à la valeur v	$O(1)$
adjust_max(x, v) :bool	ajuste le maximum de la variable x à la valeur v	$O(1)$
remove_val(x, v) :bool	supprime v de $D(x)$	$O(1)$
instantiate(x, v) :bool	instancie x à la valeur v	$O(1)$
get_prev(x, v) :bool,int	retourne vrai et la plus grande valeur w de $D(x)$ telle que $w < v$, ou retourne faux	$O(1)$
get_next(x, v) :bool,int	retourne vrai et la lus petite valeur w de $D(x)$ telle que $w > v$, ou retourne faux	$O(1)$

L'algorithme 3 correspond à l'algorithme de filtrage principal, qui implémente la Proposition 4. Dans une première phase il réduit les valeurs minimum et maximum des domaines des variables $[x_0, x_1, \ldots, x_{n-1}]$ en fonction des contraintes d'inégalité (*i.e.*, seules les valeurs bien ordonnées sont conservées). Dans une seconde phase, il calcule les informations relatives au nombre de stretchs minimum et maximum sur les matrices de préfixes et de suffixes $\underline{p}$, $\overline{p}$, $\underline{s}$, $\overline{s}$. Enfin, en fonction de ces informations, les bornes de N sont ajustées et l'algorithme filtre les domaines des variables $x_0, x_1, \ldots, x_{n-1}$. En utilisant les Lemmes 3 et 4, l'algorithme 2 construit les matrices de suffixes $\underline{s}$ et $\overline{s}$ utilisées par l'algorithme 3 ($\underline{p}$ et $\overline{p}$ sont construites de façon similaire) :

1. Dans une première phase, les colonnes $n-1$ des matrices $\underline{s}$ et $\overline{s}$ sont initialisées à 1 (premier item des Lemmes 3 et 4).
2. Dans une deuxième phase, les colonnes $n-2$ jusqu'à 0 sont initialisées (deuxième item des lemmes 3 et 4). Afin d'éviter de recalculer les quantités $\min(\underline{s}(x_{i+1}, v), \min_{w>v}(\underline{s}(x_{i+1}, w)) + 1)$ et $\max(\overline{s}(x_{i+1}, v), \max_{w>v}(\overline{s}(x_{i+1}, w)) + 1)$, nous introduisons deux matrices creuses avec accès ordonné aux colonnes $mins[i, j]$ et $maxs[i, j]$. Lorsques les i-èmes colonnes des matrices $\underline{s}$ et $\overline{s}$ sont initialisées, nous calculons d'abord les $i+1$-èmes colonnes des matrices *mins* et *maxs* (premier **repeat** de la boucle **for**). Ensuite, dans le second **repeat** de la boucle **for** nous initialisons les i-èmes colonnes de $\underline{s}$ et $\overline{s}$. On peut observer que l'on balaye les colonnes $i+1$ des matrices *mins* et *maxs* par indice de ligne décroissant.

L'algorithme 2 a une complexité temporelle en $O(\Sigma_{\text{Di}})$ et l'algorithme 3 supprime toutes les valeurs qui ne sont pas arc-consistantes avec Increasing_Nvalue en $O(\Sigma_{\text{Di}})$.[1]

Algorithm 3: Increasing_Nvalue($N, [x_0, \ldots, x_{n-1}]$) : boolean.

```
1  if n = 1 then return instantiate(N, 1);
2  for i = 1 to n − 1 do if ¬adjust_min(x_i, min(x_{i−1})) then
   return false;
3  for i = n − 2 downto 0 do if ¬adjust_max(x_i, max(x_{i+1}))
   then return false;
4  Allocate p, p̄, s, s̄;
5  build_prefix p, p̄ ; build_suffix s, s̄;
6  ScanInit({s, s̄}, 0, ↑);
7  if ¬adjust_min(N, min_{v∈D(x_0)}(Get(s, 0, v))) then return
   false;
8  if ¬adjust_max(N, max_{v∈D(x_0)}(Get(s̄, 0, v))) then return
   false;
9  for i := 0 to n − 1 do
10     ScanInit({p, p̄, s, s̄}, i, ↑) ; v := min(x_i);
11     repeat
12        N_v :=Get(p, i, v)+Get(s, i, v) − 1 ;
          N̄_v :=Get(p̄, i, v)+Get(s̄, i, v) − 1;
13        if [N_v, N̄_v] ∩ D(N) = ∅ and ¬remove_val(x_i, v)
          then return false;
14        w := v ; v := getNext(x_i, v);
15     until w = v ;
16 return true;
```

[1] Le code source de Increasing_Nvalue est disponible à l'adresse suivante : `http ://choco.emn.fr`.

5 Expérimentations

Cette section présente une série d'expérimentations de la contrainte Increasing_Nvalue. Tout d'abord, la section 5.1 présente une reformulation de la contrainte Nvalue utilisant Increasing_Nvalue, motivée par le fait de casser des symétries. Ensuite, la section 5.2 évalue les performances de Increasing_Nvalue sur une application réelle. Toutes ces expériences ont été réalisées à l'aide de la librairie de programmation par contraintes Choco, sur un processeur Intel Core 2 Duo 2.4GHz avec 4GB de RAM, et 128Mo alloués à la machine virtuelle Java.

5.1 Amélioration de la propagation de Nvalue

Assurer GAC pour une contrainte Nvalue est un problème NP-Difficile, et les algorithmes de filtrage de cette contrainte sont connus pour être assez peu efficaces pour des domaines de variables énumérés [3]. Dans notre implémentation, nous utilisons une représentation de Nvalue qui est basée sur les contraintes d'occurrence de Choco. Nous évaluons alors l'effet de la contrainte Increasing_Nvalue utilisée comme contrainte implicite sur des classes d'équivalence : Étant donné un ensemble $\mathcal{E}(X)$ de classes d'équivalence sur l'ensemble X, le filtrage de la contrainte Nvalue(X, N) peut être renforcé ainsi :

$$Nvalue(N, X) \quad (1)$$

$$\forall E \in \mathcal{E}(X), Increasing\_Nvalue(N_E, E) \quad (2)$$

$$\max_{E \in \mathcal{E}(X)}(N_E) \leq N \leq \sum_{E \in \mathcal{E}(X)}(N_E) \quad (3)$$

où N_E est la variable d'occurrence associée à la classe $E \in \mathcal{E}(X)$ (avec $E \subseteq X$).

Les paramètres enregistrés sont le nombre de nœuds dans l'arbre de recherche, le nombre d'échecs et le temps d'une résolution nécessaires à l'obtention d'une solution. Les paramètres variables de nos expériences sont le nombre de valeurs dans les domaines des variables, le pourcentage de trous dans ces domaines, et le nombre de classes d'équivalences. Le comportement observé n'est pas dépendant du nombre de variables : des tailles de 20, 40 et 100 variables ont été testées.

Les tableaux 1 et 2 montrent les résultats de nos expérimentations pour 40 variables, avec des domaines contenant au plus 80 valeurs (des tailles de 20 et 40 sont aussi testées). Concernant le tableau 1, 50 instances ont été générées pour chaque taille de classe d'équivalence. Concernant le tableau 2, 350 instances ont été générées pour chaque densité évaluée. Une limite de temps de résolution a été fixée à 60 secondes. Un paramètre est inclus dans la moyenne si les deux modèles résolvent l'instance. Les deux modèles sont strictement comparables si le pourcentage d'instances résolues est le même. Autrement, les paramètres enregistrés ne peuvent être comparés que pour les instances résolues par les deux modèles.

Le tableau 1 illustre le fait que le nombre de classes d'équivalence ait un impact sur les performances du modèle intégrant la contrainte Increasing_Nvalue. Avec sept classes, le nombre moyen de variables dans chaque classe est suffisant pour que le gain en propagation comble le coût de calcul supplémentaire dû à l'application de l'algorithme de filtrage de Increasing_Nvalue. Ce n'est plus le cas quand on a plus 10 classes d'équivalences.

La table 2 montre que le nombre de trous dans les domaines a un impact sur les performances du modèle incluant la contrainte Increasing_Nvalue. On peut remarquer que lorsque ce nombre de trous augmente, le nombre d'instances résolues décroît. Ce phénomène est dû à la propagation de la contrainte Nvalue, moins efficace quand il y a des trous dans les domaines.

5.2 Intégration dans un problème d'allocation de ressources

Entropy[2] [7] fournit un moteur autonome et flexible pour manipuler l'état et la position de VMs (machines virtuelles) sur les différents nœuds actifs composant une grappe. Ce moteur est basé sur la programmation par contraintes. Il fournit un modèle central dédié à l'affectation de VMs à des nœuds, et permet de personnaliser cette affectation à l'aide de contraintes qui modélisent des besoins exprimés par des utilisateurs et des administrateurs.

Le modèle central définit chaque nœud (les ressources) par son CPU et sa capacité mémoire, et chaque VM (les tâches) par sa demande en CPU et en mémoire pour s'exécuter. La partie traitée via un modèle en programmation par contraintes consiste à calculer une affectation de chaque VM qui (i) satisfait la demande en ressources (CPU et mémoire) des VMs et (ii) utilise un nombre minimum de nœuds. Au final, la libération de nœuds peut faire que davantage de tâches seront acceptées dans la grappe, ou bien peut permettre d'éteindre les nœuds non utilisés pour économiser l'énergie. Dans ce problème, deux parties peuvent être distinguées : (1) l'affectation de nœuds en fonction de la capacité de ressources : il s'agit d'un problème de bin-packing 2D. Il est modélisé par un ensemble de contraintes *sac à dos* associées avec chaque nœud. La propagation est basée sur l'utilisation de la contrainte CostRegular [6], afin de traiter simultanément les deux dimensions de ressource. (2) Ré-

[2] http ://entropy.gforge.inria.fr

Nb de	Nvalue				Nvalue + Increasing_Nvalue			
cl. d'quiv.	nœuds	échecs	temps (ms)	résolus(%)	nœuds	échecs	temps (ms)	résolus(%)
1	2798	22683	6206	76	**28**	**0**	**51**	**100**
3	1005	12743	4008	76	**716**	**7143**	**3905**	**82**
5	1230	14058	**8077**	72	**1194**	**12067**	8653	72
7	850	18127	**6228**	64	**803**	**16384**	6488	**66**
10	387	3924	**2027**	58	387	**3864**	2201	58
15	1236	16033	**6518**	38	**1235**	**16005**	7930	38
20	379	7296	**5879**	58	379	7296	6130	58

Tab. 1: Évaluation de Increasing_Nvalue en fonction du nombre de classes d'équivalence sur les variables.

Trous(%)	Nvalue				Nvalue + Increasing_Nvalue			
	nœuds	échecs	temps (ms)	résolus(%)	nœuds	échecs	temps (ms)	résolus(%)
25	1126.4	13552	5563.3	63.1	**677.4**	**8965.5**	**5051.1**	**67.7**
50	2867.1	16202.6	**4702.1**	50.8	**1956.4**	**12345**	4897.5	**54.9**
75	5103.7	16737.3	**3559.4**	**65.7**	**4698.7**	**15607.8**	4345.5	65.1

Tab. 2: Évaluation de Increasing_Nvalue en fonction du pourcentage de trous dans les domaines.

duction du nombre de nœuds utilisés pour instancier toutes les VMs. Les VMs sont classées en fonction de leur consommation en CPU et en mémoire (il existe donc naturellement des classes d'équivalence entre les VMs). Les contraintes Nvalue et Increasing_Nvalue (Section 5.1) sont utilisées pour modéliser cette partie du problème.

En pratique, les résultats obtenus par un modèle incluant des contraintes Increasing_Nvalue montrent un gain modéré en terme de temps de résolution (3%), alors que le gain en nombre d'échecs est plus significatif (35% en moyenne).

6 Travaux connexes

Un filtrage GAC pour Increasing_Nvalue peut être obtenu via deux autres techniques : l'usage d'automates déterministes avec un nombre polynomial de transitions [11], et la contrainte Slide [4].

Soit C une contrainte d'arité k et soit X une séquence de n variables. La contrainte Slide(C,X) [4] est un cas particulier de la contrainte *cardpath*. Cette contrainte est satisfaite ssi $C(X_i, X_{i+1}, \ldots, X_i + k - 1)$ est vérifiée pour tout $i \in [1, n - k + 1]$. La GAC peut être établie en temps $O(nd^k)$ où d est la taille maximale d'un domaine. Une extension appelée slide$_j$(C,X) est satisfaite $C(X_{ij+1}, X_{ij+2}, \ldots, X_{ij+k})$ est vérifiée pour tout $i \in [0, \frac{n-k}{j}]$. Étant donné $X = \{x_i \mid i \in [1; n]\}$, la contrainte Increasing_Nvalue peut être codée par Slide$_2$(C,$[x_i, c_i]_{i\in[1;n]}$) où (a) $c_1, c_2, \ldots, c_n$ sont des variables prenant leurs valeurs dans $[1, n]$ avec $c_1 = 1$ et $c_n = N$, et (b) $C(x_i, c_i, x_{i+1}, c_{i+1})$ est la contrainte $b \Leftrightarrow x_i \neq x_{i+1} \wedge c_{i+1} = c_i + b \wedge x_i \leq x_{i+1}$. La complexité en temps est alors en $O(nd^4)$ pour établir GAC sur la reformulation de Increasing_Nvalue.

La reformulation basée sur des automates fini déterministes est détaillée dans le catalogue de contraintes globales [1]. En utilisant l'algorithme de Pesant [11], la complexité obtenue est $O(n{\cup_{Di}}^3)$ pour GAC, où $\cup_{Di}$ est la taille de l'union des domaines des variables.

7 Conclusion

Nous avons proposé une technique pour réaliser un filtrage complet (GAC) pour une spécialisation de la contrainte Nvalue, où les variables de décision doivent satisfaire une séquence de contraintes binaires d'inégalité. Alors que prouver si une contrainte Nvalue admet ou non une solution est un problème NP-Difficile, notre algorithme a une complexité linéaire en la somme des tailles des domaines des variables. Nous pensons que la structure de données pour gérer ces matrices creuses avec accès ordonné aux colonnes peut être utile au delà du cadre de Increasing_Nvalue, afin d'améliorer la complexité d'algorithmes de filtrage d'autres contraintes globales. Nos travaux futurs se concentreront également sur une amélioration pratique de l'implémentation de l'algorithme de filtrage de Increasing_Nvalue, en considérant des intervalles de valeurs consécutives dans un domaine donné, plutôt que chaque valeur séparément. De façon plus générale, ce travail aborde le thème de l'intégration générique de méthodes pour casser des symétries dans les systèmes

à contraintes, via des contraintes globales [8, 12].

Références

[1] N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog, working version of January 2010. Technical Report T2005-08, Swedish Institute of Computer Science, 2005. Available at `www.emn.fr/x-info/sdemasse/gccat`.

[2] N. Beldiceanu, M. Carlsson, and S. Thiel. Cost-Filtering Algorithms for the two Sides of the *sum of weights of distinct values* Constraint. Technical report, Swedish Institute of Computer Science, 2002.

[3] C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. Filtering Algorithms for the *nvalue* Constraint. In *CP-AI-OR'05*, volume 3524 of *LNCS*, pages 79–93, 2005.

[4] C. Bessière, E. Hebrard, B. Hnich, Z. Kızıltan, and T. Walsh. SLIDE : A useful special case of the CARDPATH constraint. In *ECAI 2008, Proceedings*, pages 475–479, 2008.

[5] C. Bessière, E. Hebrard, B. Hnich, and T. Walsh. The Complexity of Global Constraints. In *19th National Conference on Artificial Intelligence (AAAI'04)*, pages 112–117. AAAI Press, 2004.

[6] S. Demassey, G. Pesant, and L.-M. Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4) :315–333, 2006.

[7] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy : a consolidation manager for clusters. In *VEE '09 : Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 41–50, 2009.

[8] G. Katsirelos, N. Narodytska, and T. Walsh. Combining Symmetry Breaking and Global Constraints. In *Recent Advances in Constraints, Joint ERCIM/CoLogNet International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2008*, volume 5655 of *LNCS*, pages 84–98, 2009.

[9] F. Pachet and P. Roy. Automatic Generation of Music Programs. In *CP'99*, volume 1713 of *LNCS*, pages 331–345, 1999.

[10] G. Pesant. A filtering algorithm for the stretch constraint. In *CP'01*, volume 2239 of *LNCS*, pages 183–195, 2001.

[11] G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *CP'04*, volume 3258 of *LNCS*, pages 482–495, 2004.

[12] M. Ågren, N. Beldiceanu, M. Carlsson, M. Sbihi, C. Truchet, and S. Zampelli. Six Ways of Integrating Symmetries within Non-Overlapping Constraints. In *CP-AI-OR'09*, volume 5547 of *LNCS*, pages 11–25, 2009.

Amélioration de l'apprentissage des clauses par symétrie dans les solveurs SAT

Belaïd Benhamou **Tarek Nabhani** **Richard Ostrowski** **Mohamed Réda Saïdi**[1]

[1] Université de Provence
Laboratoire des Sciences de l'Information et des Systèmes (LSIS)
Centre de Mathématiques et d'Informatique.
39, rue Joliot Curie - 13453 Marseille cedex 13, France.
{benhamou, nabhani, ostrowski, saidi}@cmi.univ-mrs.fr

Résumé

Le problème de satisfiabilité (SAT) est le premier problème de décision à avoir été montré NP-complet. Il est central en théorie de la complexité. Une formule mise sous forme CNF contient un nombre intéressant de symétries. En d'autres termes, la formule reste invariante si l'on permute quelques variables. De telles permutations sont les symétries de la formule et leurs éliminations peuvent conduire à une preuve plus courte pour la satisfiabilité. D'autre part, de nombreuses améliorations ont été apportées dans les solveurs actuels. Les solveurs de type CDCL sont aujourd'hui capables de résoudre de manière efficace des problèmes industriels de très grande taille (en nombre de variables et de clauses). Ces derniers utilisent des structures de données paresseuses, des politiques de redémarrage et apprennent de nouvelles clauses à chaque échec au cours de la recherche. Bien que l'utilisation des symétries et l'apprentissage de clauses s'avèrent être des principes puissants, la combinaison des deux n'a encore jamais été exploitée. Dans cet article, nous allons montrer comment la symétrie peut être utilisée afin d'améliorer l'apprentissage dans des solveurs de type CDCL. Nous avons mis en application l'apprentissage par symétries dans MiniSat et nous l'avons expérimenté sur différents problèmes. Nous avons comparé MiniSat avec et sans apprentissage par symétries. Les résultats obtenus sont très encourageants et montrent que l'utilisation des symétries dans l'apprentissage est profitable pour des solveurs à base de CDCL.

Abstract

The satisfiability problem (SAT) is shown to be the first decision NP-complete problem. It is central in complexity theory. A CNF formula usually contains an interesting number of symmetries. That is, the formula remains invariant under some variable permutations. Such permutations are the symmetries of the formula, their elimination can lead to make a short proof for a satisfiability proof procedure. On other hand, many improvements had been done in SAT solving, Conflict-Driven Clause Learning (CDCL) SAT solvers are now able to solve great size and industrial SAT instances efficiently. The main theoretical key behind these modern solvers is, they use lazy data structures, a restart policy and perform clause learning at each fail end point in the search tree. Although symmetry and clause learning are shown to be powerful principles for SAT solving, but their combination, as far as we now, is not investigated. In this paper, we will show how symmetry can be used to improve clause learning in CDCL SAT solvers. We implemented the symmetry clause learning approach on the MiniSat solver and experimented it on several SAT instances. We compared both MiniSat with and without symmetry and the results obtained are very promising and show that clause learning by symmetry is profitable for CDCL SAT solvers.

1 Introduction

Krishnamurthy a introduit dans [23] le principe de symétrie dans le calcul propositionnel et a montré que certains problèmes peuvent avoir des preuves plus courtes en augmentant la règle de résolution par les symétries. Les symétries pour les contraintes booléennes ont été étudiées de manière détaillée dans [7, 8, 9].

Les auteurs ont montré comment les détecter et ont prouvé que leur exploitation apporte une amélioration considérable sur plusieurs algorithmes de déduction au-

tomatique. Par la suite, de nombreux travaux sur les symétries ont été développés. Par exemple, l'approche statique utilisée par James Crawford et al. dans [10] pour des théories de logique propositionnelle consiste à ajouter des contraintes exprimant la *symétrie globale* du problème.

Cette technique a été améliorée dans [1] et étendue à la programmation logique en nombres entiers 0-1 dans [2]. Bien que la symétrie ait été introduite en logique propositionnelle, elle a été investie davantage en programmation par contraintes ces dernières années. La notion d'interchangeabilité dans les CSPs est introduite dans [14] et la symétrie pour les CSPs est étudiée plus tôt dans[26, 6].

Puisqu'un grand nombre de contraintes pouvait être ajouté dans l'approche statique, en CSPs, certains chercheurs ont proposé d'ajouter ces contraintes au cours de la recherche. Dans[4, 16, 17], les auteurs enregistrent quelques contraintes conditionnelles qui retirent la symétrie de l'interprétation partielle en cas de retour arrière. Dans[13, 12, 27, 15], les auteurs proposent d'utiliser chaque sous-arbre comme nogood afin d'éviter l'exploration d'interprétations symétriques. Le groupe d'arbres d'équivalence conceptuelle pour l'élimination de valeurs symétriques est introduit dans [28].

Après cela, T. Walsh étudia dans [30] divers propagateurs afin de casser des symétries notamment ceux agissant simultanément sur les variables et les valeurs.

Le problème de satisfiabilité est générique. De nombreux problèmes dans d'autres domaines peuvent se réduire à un test de satisfiabilité par exemple la déduction automatique, la configuration, la planification, l'ordonnancement, etc. . Plusieurs méthodes d'élimination de symétries sont introduites. Parmi elles, les approches statiques[10, 1, 2] qui éliminent les *symétries globales* du problème initial et les approches dynamiques[7, 8, 9] qui détectent et cassent les *symétries locales* à chaque noeud de l'arbre pendant la recherche. Ces deux approches se sont avérées profitables pour la résolution du problème SAT.

D'autre part, de nombreuses améliorations sont apparues ces dernières années autour des solveurs SAT. Ces derniers utilisent de meilleures structures de données, l'apprentissage de clauses [29, 31], le retour arrière non chronologique [29, 21] et différentes politiques de redémarrage [18, 20, 3]. Il a été montré que l'apprentissage est une notion importante [5, 19, 24, 25] qui améliore grandement lâefficacité des solveurs. L'utilisation des symétries est tout aussi utile pour les solveurs SAT mais n'a jamais été utilisée dans l'apprentissage de clauses.

Dans cet article, nous présentons une nouvelle approche pour l'apprentissage qui utilise les symétries du problème. Cette méthode consiste, dans un premier temps, à détecter toutes les symétries globales du problème et de les utiliser lorsqu'une nouvelle clause (clause assertive) est déduite au cours de la recherche. Cela nous permet de déduire toutes les clauses assertives symétriques du problème. L'apprentissage par symétrie est différent des approches consistant à éliminer les symétries globales du problème. Ces dernières sont des méthodes statiques qui ajoutent en prétraitement des clauses afin d'éliminer les interprétations symétriques du problème. L'approche que nous proposons ici est dynamique. Elle génère, au cours de la recherche, toutes les clauses symétriques des différentes clauses assertives afin d'éviter l'exploration de sous-espaces isomorphes. L'avantage de cette méthode est qu'elle n'élimine pas les modèles symétriques comme pour les méthodes statiques. Elle évite d'explorer des sous-espaces correspondant aux no-goods symétriques de l'interprétation partielle courante.

Cet article est organisé de la façon suivante : dans la deuxième partie, nous présentons le contexte des permutations et du problème de satisfiabilité. La troisième partie définit la symétrie et donne les résultats théoriques sur la symétrie que nous utilisons pour l'apprentissage. Nous décrivons ensuite comment utiliser la symétrie dans des solveurs de type CDCL. Nous l'avons ensuite implanté dans un solveur SAT moderne *MiniSat*[11] [1] et nous l'avons évalué sur différents problèmes. La sixième partie synthétise les résultats : nous comparons *MiniSat* avec et sans symétrie lors de l'apprentissage. Enfin, nous apportons une conclusion et présentons quelques-unes des perspectives de ces travaux.

2 Contexte et définitions

2.1 Logique propositionnelle

Nous supposons que le lecteur est familier avec le calcul propositionnel. Nous donnons ici une courte description. Soit V un ensemble de *variables propositionnelles*. Les variables propositionnelles seront distinguées des *littéraux* qui sont des variables propositionnelles avec une valeur d'affection 1 ou 0 qui signifie respectivement $Vrai$ ou $Faux$. Pour une variable propositionnelle ℓ, il y a deux littéraux : le littéral positif ℓ et le négatif $\neg\ell$.

Une $clause$ est une disjonction de littéraux $\ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$. Une formule $\mathcal{F}$ est mise sous *forme normale conjonctive* (CNF) si et seulement si c'est une conjonction de clauses.

Une *interprétation* d'une CNF $\mathcal{F}$ est une correspondance I définie de l'ensemble des variables de $\mathcal{F}$ dans l'ensemble $\{Vrai, Faux\}$. Nous pouvons considérer I comme l'ensemble des littéraux interprétés. La valeur d'une clause $\ell_1 \vee \ell_2 \vee \ldots \vee \ell_n$ dans I est à $Vrai$, si la valeur $Vrai$ est affectée à au moins un de ses littéraux dans I ; sinon, elle est à $Faux$. Par convention, nous définissons la valeur de la *clause vide* ($n = 0$) à $Faux$.

1. L'apprentissage par symétrie est générique. Il peut être utilisé dans n'importe quel solveur qui fait de l'apprentissage

La valeur $I[\mathcal{F}]$ est $Vrai$ si la valeur de chaque clause de $\mathcal{F}$ est à $Vrai$, $Faux$ sinon. Une formule CNF $\mathcal{F}$ est *satisfiable* s'il existe une interprétation I qui affecte la valeur $Vrai$ à $\mathcal{F}$, sinon elle est $insatisfiable$. Dans le premier cas, I est appelé *modèle* de $\mathcal{F}$. Une formule $\mathcal{G}$ est une *conséquence logique* d'une formule $\mathcal{F}$ si $\mathcal{F}$ implique $\mathcal{G}$ et est notée $\mathcal{F} \models \mathcal{G}$. Si une interprétation I est définie seulement sur un sous-ensemble de variables de $\mathcal{F}$, alors elle est appelée *interprétation partielle*. Elle est appelée un *no-good* si $I[\mathcal{F}] = Faux$. Généralement, les solveurs SAT manipulent une interprétation partielle I qui est constituée d'un sous-ensemble de littéraux de décisions D et d'un sous-ensemble de littéraux propagés P par propagation unitaire ($I = D \cup P$). Plus précisement, une interprétation partielle est un produit I=$\prod_{i=1}^{m} \langle (x_i), y_{i,1}, y_{i,2}, \ldots, y_{i,k_i} \rangle$ formé par m littéraux de décisions x_i et tous leurs littéraux propagés ordonnés $y_{i,1}, y_{i,2}, \ldots, y_{i,k_i}$. Le *niveau d'affectation* d'un littéral de décision x_i pour une interprétation I (noté $level(x_i)$) est son ordre d'affectation i dans I, et tous ses littéraux propagés $y_{i,1}, y_{i,2}, \ldots, y_{i,k_i}$ ont le même niveau d'affectation i.

2.2 Permutations

Soit $\Omega = \{1, 2, \ldots, N\}$ pour un entier N, où chaque entier peut représenter un littéral d'une formule CNF $\mathcal{F}$. Une permutation de Ω est une correspondante bijective σ de Ω à Ω qui est généralement représentée comme un produit de cycles de permutations. Nous dénotons par $Perm(\Omega)$ l'ensemble de toutes les permutations de Ω et $\circ$ la composition de la permutation de $Perm(\Omega)$. La paire $(Perm(\Omega), \circ)$ forme le groupe de permutation de Ω. En d'autres termes, $\circ$ est fermé et associatif, l'inverse d'une permutation est une permutation et la permutation identique est l'élément neutre. Une paire $(T, \circ)$ forme un sous-groupe de $(S, \circ)$ si et seulement si T est un sous-ensemble de S et constitue un groupe sous l'opération $\circ$.

L'orbite $\omega^{Perm(\Omega)}$ d'un élément ω de Ω sur lequel le groupe $Perm(\Omega)$ agit est $\omega^{Perm(\Omega)}$=$\{\omega^{\sigma} : \omega^{\sigma} = \sigma(\omega), \sigma \in Perm(\Omega)\}$.

Un ensemble générateur du groupe $Perm(\Omega)$ est un sous-ensemble $Gen(\Omega)$ de $Perm(\Omega)$ tel que chaque élément de $Perm(\Omega)$ peut être écrit comme une composition d'éléments de $Gen(\Omega)$. Nous écrivons $Perm(\Omega)$=$< Gen(\Omega) >$. Un élément de $Gen(\Omega)$ est appelé un générateur. L'orbite de $\omega \in \Omega$ peut être calculée à partir de l'ensemble des générateurs $Gen(\Omega)$.

3 Symétrie et apprentissage

3.1 Symétrie

Nous rappelons la définition de la symétrie qui est donnée dans [7, 8]

Définition 1 *Soit $\mathcal{F}$ une formule propositionnelle sous forme CNF et $L_{\mathcal{F}}$ l'ensemble de ses littéraux.* [2]*. Une symétrie de $\mathcal{F}$ est une permutation σ définie sur $L_{\mathcal{F}}$ telle que :*

1. $\forall \ell \in L_{\mathcal{F}}, \sigma(\neg \ell) = \neg \sigma(\ell)$,
2. $\sigma(\mathcal{F}) = \mathcal{F}$

En d'autres termes, une symétrie d'une formule est une permutation des littéraux de cette formule la laissant invariante. Si nous notons par $Perm(L_{\mathcal{F}})$ le groupe de permutations de $L_{\mathcal{F}}$ et par $Sym(L_{\mathcal{F}}) \subset Perm(L_{\mathcal{F}})$ le sous-ensemble de permutations de $L_{\mathcal{F}}$ qui sont les symétries de $\mathcal{F}$, alors $Sym(L_{\mathcal{F}})$ est trivialement un sous-groupe de $Perm(L_{\mathcal{F}})$.

Définition 2 *Soit $\mathcal{F}$ une formule, l'orbite d'un littéral $\ell \in L_{\mathcal{F}}$ sur lequel le groupe de symétries $Sym(L_{\mathcal{F}})$ agit est $\ell^{Sym(L_{\mathcal{F}})}$=$\{\sigma(\ell) : \sigma \in Sym(L_{\mathcal{F}})\}$*

Exemple 1 *Soit $\mathcal{F}$ l'ensemble des clauses suivantes :*
$\mathcal{F}$=$\{a \vee b \vee c, \neg a \vee b, \neg b \vee c, \neg c \vee a, \neg a \vee \neg b \vee \neg c\}$
,σ_1 et σ_2 deux permutations définies sur l'ensemble complet $L_{\mathcal{F}}$ de littéraux apparaissant dans $\mathcal{F}$ comme suit :

σ_1=$(a, b, c)(\neg a, \neg b, \neg c)$

σ_2=$(a, \neg a)(b, \neg c)(c, \neg b)$

σ_1 et σ_2 sont deux symétries de $\mathcal{F}$, puisque $\sigma_1(\mathcal{F})$=$\mathcal{F}$=$\sigma_2(\mathcal{F})$. L'orbite du littéral a est $a^{Sym(L_{\mathcal{F}})}$= $\{a, b, c, \neg a, \neg b, \neg c\}$. Nous voyons que tous les littéraux sont dans la même orbite. Par conséquent, ils sont tous symétriques.

La propriété principale d'une symétrie $\sigma \in Sym(L_{\mathcal{F}})$ d'une formule $\mathcal{F}$, est qu'elle conserve l'ensemble de ses modèles. Formellement, B. Benhamou et al [7, 8] ont introduit la propriété suivante :

Proposition 1 *Étant données une formule $\mathcal{F}$ et une symétrie $\sigma \in Sym(L_{\mathcal{F}})$, si I est un modèle de $\mathcal{F}$, alors $\sigma(I)$ est un modèle de $\mathcal{F}$.*

De plus, une symétrie σ transforme chaque no-good I de $\mathcal{F}$ en un no-good $\sigma(I)$.

3.1.1 Détection de symétries

Nous avons utilisé Bliss [22] afin de détecter les symétries d'une formule CNF $\mathcal{F}$. Bliss est un outil pour calculer le groupe d'automorphisme d'un graphe.

Dans [10, 1, 2], les auteurs montrent que chaque formule CNF $\mathcal{F}$ peut être représentée par un graphe $G_{\mathcal{F}}$ construit de la manière suivante :

2. L'ensemble des littéraux L_F contient chaque littéral et sa négation

– chaque variable booléenne est représentée par deux sommets (sommets littéraux) dans $G_{\mathcal{F}}$: le littéral positif et le littéral négatif. Ces deux sommets sont connectés par une arête dans le graphe $G_{\mathcal{F}}$.
– chaque clause dont la longueur est supérieure à deux est représentée par un sommet(un sommet clause). Une arête connecte ce sommet à chaque sommet représentant un littéral de la clause.
– chaque clause binaire est représentée par une arête qui connecte les deux sommets littéraux de cette clause. Nous n'avons pas besoin d'ajouter de sommet clause pour les clauses binaires.

Une propriété importante du graphe $G_{\mathcal{F}}$ est qu'elle préserve le groupe des symétries de $\mathcal{F}$. En d'autres termes, le groupe des symétries $Sym(L_F)$ de la formule $\mathcal{F}$ est identique au groupe d'automorphisme $Aut(G_{\mathcal{F}})$ de sa représentation graphique $G_{\mathcal{F}}$. Pour cela, nous utilisons Bliss sur $G_{\mathcal{F}}$ afin de détecter le groupe de symétries $Sym(L_F)$ de $\mathcal{F}$. Bliss retourne un ensemble de générateurs $Gen(Aut(G_{\mathcal{F}}))$ du groupe d'automorphisme duquel nous pouvons déduire toutes les symétries de $\mathcal{F}$. Bliss offre la possibilité de colorer les sommets du graphe. Un sommet peut être permuté avec un autre sommet de la même couleur. Deux couleurs sont utilisées dans $G_{\mathcal{F}}$, une pour les sommets correspondant aux clauses de $\mathcal{F}$ et une autre pour les littéraux de $L_{\mathcal{F}}$. Cela permet de distinguer les sommets clauses des sommets littéraux et de prévenir ainsi des permutations entre des clauses et des littéraux. Les sources de Bliss sont disponibles à l'adresse (http ://www.tcs.hut.fi/Software/bliss/index.html).

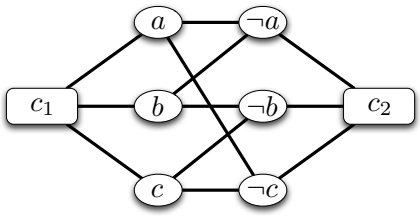


FIGURE 1 – Le graphe $G_{\mathcal{F}}$ correspondant à $\mathcal{F}$

Considérons par exemple la formule $\mathcal{F}$ donnée dans l'exemple 1. Son graphe associé $G_{\mathcal{F}}$ est donné à la figure 1. Les sommets clauses sont représentés par des boîtes et les sommets littéraux par des ellipses. Le groupe d'automorphismes de $G_{\mathcal{F}}$ est identique au groupe des symétries de $\mathcal{F}$. Nous pouvons noter que les deux symétries σ_1 et σ_2 de la formule $\mathcal{F}$ de l'exemple 1 peuvent être vues comme des automorphismes du graphe $G_{\mathcal{F}}$ correspondant.

3.2 Apprentissage par symétries (SLS)

La clé principale des solveurs SAT est la *propagation unitaire*. Étant donné une formule $\mathcal{F}$, nous écrivons $\mathcal{F} \vdash_U \ell$ pour indiquer que le littéral ℓ peut être dérivé de $\mathcal{F}$ en utilisant la *propagation unitaire*. Une formule CNF $\mathcal{F}$ est U-inconsistant si et seulement si $\mathcal{F} \vdash_U Faux$, sinon $\mathcal{F}$ est U-consistent. Maintenant, nous pouvons résumer les définitions d'une clause assertive, d'un littéral assertif et le niveau assertif d'une clause.

Définition 3 *Soit $\mathcal{F}$ une formule CNF, $c = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k \vee x$ une clause de $\mathcal{F}$, et I une interprétation partielle de $\mathcal{F}$ avec m littéraux de décision. La clause c est une clause assertive si et seulement si $I[c] = Faux$, $level(x) = m$ et $\forall i \in \{1, \ldots, k\}$, $level(\ell_i) < m$. Le littéral x est le littéral assertif de c et le niveau assertif de c est le plus haut niveau de ses autres littéraux (le littéral assertif n'est pas inclus).*

Nous introduisons le lemme suivant que nous utiliserons afin de prouver nos résultats sur les clauses assertives symétriques.

Lemme 1 *Soit $\mathcal{F}$ une formule CNF et σ une symétrie de $\mathcal{F}$. Si I=$\langle (x), y_1, y_2, \ldots, y_n \rangle$ est une interprétation partielle constitué par le seul littéral de décision x et tous ses littéraux propagés ordonnés $y_1, y_2, \ldots, y_n$, alors $\sigma(I)$=$\langle (\sigma(x)), \sigma(y_1), \sigma(y_2), \ldots, \sigma(y_n) \rangle$, $level(x)$ dans I est identique à $level(\sigma(x))$ dans $\sigma(I)$, et $\forall i \in \{1, \ldots, n\}$, $level(y_i)$ dans I est identique à $level(\sigma(y_i))$ dans $\sigma(I)$.*

***Preuve* 1** *Pour montrer ceci, nous devons prouver que si $\mathcal{F} \wedge x \vdash_U y_i$,*

alors nous avons $\mathcal{F} \wedge \sigma(x) \vdash_U \sigma(y_i)$ pour tout $i \in \{1, \ldots, n\}$.

Nous le prouvons par induction sur i. Pour $i = 1$ nous devons montrer que si

$\mathcal{F} \wedge x \vdash_U y_1$ alors $\mathcal{F} \wedge \sigma(x) \vdash_U \sigma(y_1)$.

Nous avons $\mathcal{F} \wedge x \vdash_U y_1$ par hypothèse. Cela signifie qu'il existe une clause $c \in \mathcal{F}$

tel que $c = \neg x \vee y_1$. Comme σ est une symétrie de $\mathcal{F}$, alors $\sigma(c) = \neg\sigma(x) \vee \sigma(y_1)$ est une clause de $\mathcal{F}$. Cela implique que $\mathcal{F} \wedge \sigma(x) \vdash_U \sigma(y_1)$.

Supposons maintenant que la propriété est vraie jusque $i - 1$, nous devons montrer qu'elle est vraie pour i. Nous supposons que $\mathcal{F} \wedge x \vdash_U y_i$, et devons montrer que $\mathcal{F} \wedge \sigma(x) \vdash_U \sigma(y_i)$.

De $\mathcal{F} \wedge x \vdash_U y_i$ nous déduisons qu'il existe une clause $c \in \mathcal{F}$ telle que $c = \alpha \vee y_i$ où $\alpha \subseteq \neg x \vee \neg y_1 \vee \neg y_2 \vee \cdots \vee \neg y_{i-1}$. Par conséquent $\sigma(c) = \sigma(\alpha) \vee \sigma(y_i)$ est une clause de $\mathcal{F}$ telle que $\sigma(\alpha) \subseteq \neg\sigma(x) \vee \neg\sigma(y_1) \vee \neg\sigma(y_2) \vee \cdots \vee \neg\sigma(y_{i-1})$. Comme $\mathcal{F} \wedge x \vdash_U y_i$, alors $\mathcal{F} \wedge x \vdash_U y_j, \forall j \in \{1, ;i - 1\}$. Par hypothèse, nous avons $\mathcal{F} \wedge \sigma(x) \vdash_U \sigma(y_j), \forall j \in \{1, \ldots, i - 1\}$. Il est maintenant trivial que $\mathcal{F} \wedge \sigma(x) \vdash_U \sigma(y_i)$. Il est aussi évident que $level(x)$ dans I est identique à $level(\sigma(x))$ dans $\sigma(I)$, et $\forall i \in \{1, \ldots, n\}$, $level(y_i)$ dans I est identique à $level(\sigma(y_i))$ dans $\sigma(I)$.

Du lemme précédent, nous pouvons déduire la proposition suivante.

Proposition 2 *Étant données une formule CNF $\mathcal{F}$ et une symétrie σ de $\mathcal{F}$, si I=$\prod_{i=1}^{m}\langle(x_i), y_{i,1}, y_{i,2}, \ldots, y_{i,k_i}\rangle$ est une interprétation partielle composée de m littéraux de décisions x_i et tous les littéraux ordonnés associés propagés $y_{i,1}, y_{i,2}, \ldots, y_{i,k_i}$ par propagation unitaire, nous avons alors $\sigma(I)$=$\prod_{i=1}^{m}\langle(\sigma(x_i)), \sigma(y_{i,1}), \sigma(y_{i,2}), \ldots, \sigma(y_{i,k_i})\rangle$, et $\forall x \in I, level(x)$ in I est identique à $level(\sigma(x))$ dans $\sigma(I)$.*

***Preuve* 2** *La preuve peut être dérivée par application du précédent lemme de manière récursive sur les littéraux de décision x_i et leurs littéraux propagés associés $y_{i,1}, y_{i,2}, \ldots, y_{i,k_i}$.*

Maintenant, nous introduisons la principale propriété sur la symétrie que nous utiliserons pour améliorer l'apprentissage dans les solveurs SAT de type CDCL.

Proposition 3 *Étant données une CNF $\mathcal{F}$, une symétrie σ de $\mathcal{F}$, et une interprétation partielle I=$\prod_{i=1}^{m}\langle(x_i), y_{i,1}, y_{i,2}, \ldots, y_{i,k_i}\rangle$ constituée de m littéraux de décision x_i et leurs littéraux ordonnés propagés $y_{i,1}, y_{i,2}, \ldots, y_{i,k_i}$. Si $c = \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k \vee x$ est une clause assertive correspondant à I avec x son littéral assertif, alors $\sigma(c)$ est une clause assertive correspondant à l'interprétation partielle symétrique $\sigma(I)$ avec $\sigma(x)$ comme littéral assertif.*

***Preuve* 3** *Nous devons montrer que $\sigma(I)[\sigma(c)] = Faux$, $level(\sigma(x)) = m$ et $\forall i \in \{1, \ldots, k\}$, $level(\sigma(\ell_i)) < m$ dans $\sigma(I)$. Par hypothèse, c est une clause assertive correspondant à I, donc $I[c] = Faux$. Comme σ est une symétrie, on a $\sigma(I)[\sigma(c)] = Faux$. De la proposition 2, nous pouvons vérifier la condition du niveau : $level(\sigma(x)) = m$ et $\forall i \in \{1, \ldots, k\}$ $level(\sigma(\ell_i)) < m$ dans $\sigma(I)$.*

Cette propriété permet aux solvers CDCL d'ajouter en même temps la clause assertive par rapport à I et toutes les clauses assertives symétriques $\sigma(c)$. Cela permet d'éviter de parcourir des sous-espaces isomorphiques correspondant à l'interprétation partielle symétrique $\sigma(I)$ de I. Les expérimentations qui vont suivre montreront que cette propriété améliore considérablement l'efficacité des solveurs CDCL.

Le cas le plus intéressant est lorsque la clause assertive est réduite à un mono-littéral. Nous pouvons déduire par symétrie tous les littéraux appartenant à l'orbite de ce mono-littéral et ainsi propager directement tous les littéraux opposés de cet orbite. Formellement, nous avons la propriété suivante :

Proposition 4 *Étant données une CNF $\mathcal{F}$ et une interprétation partielle I, si ℓ est une clause assertive unitaire, alors pour tout $\ell' \in \ell^{Sym(L_{\mathcal{F}})}$, ℓ' est une clause assertive unitaire et $\mathcal{F}$ est satisfiable si et seulement si $(\mathcal{F} \wedge \neg\ell \bigwedge_{\ell_i \in \ell^{Sym(L_{\mathcal{F}})}} \neg\ell_i)$ est satisfiable.*

***Preuve* 4** *La proposition est un cas particulier de la proposition 2.*

Le schéma d'apprentissage par symétrie (SLS) est alors un schéma classique d'apprentissage (SL) incluant les deux propriétés sur les symétries qui permettent d'inférer des clauses assertives symétriques afin de booster l'apprentissage. Dans ce qui suit, nous montrons comment ce schéma est implanté dans un solveur CDCL.

4 Avantages de la symétrie dans des algorithmes de recherche

Dans cette partie, nous présentons la façon dont le nouveau schéma d'apprentissage par symétrie (SLS) peut être ajouté aux solveurs CDCL. Ce nouveau solveur moderne est basé sur la propagation unitaire, l'apprentissage de clauses [29, 31] par symétrie, une politique de redémarrage [18] et du retour arrière non-chronologique [29, 21]. Dans l'algorithme 1 présenté ci-dessous, nous décrivons cette procédure avec l'ajout de la symétrie. Le pseudo-code de cette procédure (appelé SCLR) est basé sur celui présenté dans [25].

Tout le background théorique de base, par exemple, la séquence des points de décicion D, le niveau d'assertion, le retour arrière non-chronologique et les redémarrages sont les mêmes que les algorithmes classiques à base de CDCL [25]. Si $I = D \cup P$ est une interprétation partielle de la formule $\mathcal{F}$, alors l'état correspondant à I dans l'arbre de recherche du solveur SAT considéré est donné par $S = (\mathcal{F}, \Gamma, D)$ où $D = (\ell_1, \ell_2, \ldots, \ell_k)$ est l'ensemble ordonné des littéraux de décision de I et ℓ_i est le littéral de décision au niveau i. Γ est une CNF telle que $\mathcal{F} \models \Gamma$.

Un état $S = (\mathcal{F}, \Gamma, D)$ est U-inconsistant, respectivement U-consistant, si et seulement si $\mathcal{F} \wedge \Gamma \wedge D$ est U-inconsistant, respectivement U-consistant. Les clauses de Γ sont des clauses assertives qui sont ajoutées à la formule initiale $\mathcal{F}$ exprimant les interprétations partielles générées qui sont des no-goods.

La principale différence entre la procédure SCLR que nous proposons et les procédures classiques de type CDCL et l'implantation des deux propositions 3 et 4 dans SCLR (lignes 10 à 14). En effet, lorsqu'il y a une interprétation partielle conflictuelle I et que l'ensemble des points de décision D est non vide, une clause assertive c est déduite et deux cas sont testés : si la clause assertive c est unitaire(ligne 10), alors tous les littéraux symétriques (littéraux de son orbite, ligne 11) sont alors ajoutés à Γ (ligne 14) et leur négation est propagée à la racine de l'arbre de recherche. Si la clause assertive c n'est pas unitaire

(ligne 12), alors c et toutes les clauses symétriques $\sigma(c)$ induites par les générateurs $\sigma \in Gen(Sym(L_F))$ (ligne 13) sont ajoutées à Γ (ligne 14). Pour des raisons d'efficacité, dans notre implantation, nous limitons la génération de clauses symétriques (non unitaires) à celle produites par l'ensemble des générateurs du groupe de symétries de la formule.

Algorithm 1: SCLR : solveurs SAT avec apprentissage par symétries et redémarrages

entrée: Une formule CNF $\mathcal{F}$
sortie : Une solution de $\mathcal{F}$ ou unsat si $\mathcal{F}$ n'est pas satisfiable

1 $D \longleftarrow \langle\rangle$ // Littéraux de décision;
2 $\Gamma \longleftarrow vrai$ // Clauses apprises;
3 **while** $vrai$ **do**
4 **if** $S = (\mathcal{F}, \Gamma, D)$ *est* $U - inconsistant$ **then**
5 // Il y a un conflit.
6 **if** $D = \langle\rangle$ **then**
7 **return** *unsat*
8 $c \longleftarrow$ une clause assertive de S
9 $m \longleftarrow$ le niveau assertif de c
10 **if** c *est unitaire* **then**
11 $A \longleftarrow \{\ell \mid \ell \in c^{Sym(L_{\mathcal{F}})}\}$
12 **else**
13 $A \longleftarrow \{\sigma(c) | \sigma \in Gen(Sym(L_{\mathcal{F}}))\}$
14 $\Gamma \longleftarrow \Gamma \bigwedge_{c \in A} c$
15 $D \longleftarrow D_m$ // les m premières décisions
16 **else**
17 // Pas de conflit.
18 **if** *redémmarage* **then**
19 $D \longleftarrow \langle\rangle$
20 $S = (\mathcal{F}, \Gamma, D)$
21 Choisir un littéral ℓ tel que $S \nvdash \ell$ et $S \nvdash \neg\ell$
22 **if** $\ell = null$ **then**
23 **return** D // satisfiable
24 $D \longleftarrow D, \ell$

Nous avons choisi pour notre implantation le solveur *MiniSat*[11] auquel nous avons ajouté l'apprentissage par symétrie. Cependant, notre approche est générique et peut donc être utilisée dans d'autres solveurs de type CDCL. Nous présentons les résultats expérimentaux dans la partie suivante.

5 Expérimentation

Nous étudions les performances de notre approche par une analyse expérimentale. Nous avons pour cela choisi certaines instances afin de tester notre méthode d'apprentissage par symétrie. Nous nous attendons à ce que l'apprentissage par symétrie soit profitable aux solveurs pour des problèmes réels. Nous avons testé et comparé *MiniSat* et *MiniSat* avec apprentissage par symétrie (*MiniSat+SymCDCL*) sur plusieurs instances.

		Minisat		Minisat+SymCDCL	
Instance	$\#V : \#C$	*Noeuds*	*Temps*	*Noeuds*	*Temps*
chnl10_12	240 : 1344	2009561	67.26	28407	1.58
chnl10_13	260 : 1586	3140061	128.68	29788	1.92
chnl11_12	264 : 1476	− − −	>1200	246518	18.44
chnl11_13	286 : 1742	− − −	>1200	185417	15.20
chnl11_20	440 : 4220	− − −	>1200	61287	7.84
fpga10_8_sat	120 : 448	264	0.00	463	0.00
fpga10_9_sat	135 : 549	250	0.00	494	0.01
fpga12_11_sat	198 : 968	421	0.00	982	0.02
fpga12_12_sat	216 : 1128	403	0.00	343	0.00
fpga12_8_sat	144 : 560	390	0.00	568	0.01
fpga12_9_sat	162 : 684	383	0.00	1089	0.03
fpga13_10_sat	195 : 905	499	0.00	2311	0.06
fpga13_12_sat	234 : 1242	335	0.00	336	0.00
fpga13_9_sat	176 : 759	408	0.00	603	0.01
hole7	56 : 204	10123	0.08	233	0.00
hole8	72 : 297	40554	0.37	8323	0.15
hole9	90 : 415	202160	2.69	15184	0.35
hole10	110 : 561	1437244	27.54	73844	2.10
hole11	132 : 738	23096626	778.46	249897	9.49
hole12	156 : 949	− − −	>1200	837072	43.23
Urq3_5	46 : 470	9403639	79.09	1830	0.04
Urq4_5	74 : 694	− − −	>1200	18442	0.61
Urq5_5	121 : 1210	− − −	>1200	1798674	167

TABLE 1 – Résultats sur quelques instances SAT

D'abord, nous avons exécuté les deux méthodes sur différentes instances comme les $FPGA$ (Field Programmable Gate Array), $Chnl$ (Allocation de fréquences), Urq (Urquhart) and $Hole$ (Pigeon) qui possèdent des symétries. Ensuite, nous testons ces deux méthodes sur des problèmes de coloration de graphes difficiles. Pour finir, différents problèmes des dernières compétitions SAT sont testés. Nous comparons les performances en nombre de noeuds et en temps CPU. Le temps nécessaire pour le calcul des symétries du problème initial est ajouté au temps CPU total pour *MiniSat+SymCDCL*. Le code est écrit en C++ et compilé sur un Pentium 4, 2.8 GHZ avec 1 Gb de RAM.

5.1 Résultats sur différents problèmes symétriques

Le tableau 1 montre la comparaison de notre approche avec *MiniSat* sur des problèmes SAT connus. Les colonnes nous donnent le nom de l'instance, la taille ($\#V/\#C$), le nombre de noeuds de l'arbre de recherche et le temps CPU utilisé pour trouver la solution.

Le tableau 1 montre que notre approche *MiniSat+SymCDCL* est en général meilleure que *MiniSat* tant en temps qu'en nombre de noeuds sur les problèmes $Hole$ et $Chnl$. Pour les instances $FPGA$, nous obtenons des résultats similaires en temps. Ceci est dû au fait que ces pro-

blèmes sont satisfiables et *MiniSat* arrive à trouver une solution rapidement. Les problèmes Urq sont plus difficiles que les problèmes $FPGA$ ou les problèmes $Chnl$. Nous voyons que *MiniSat* n'est capable de résoudre que le premier problème alors que notre approche permet de les résoudre tous efficacement.

5.2 Résultats sur les problèmes de coloration de graphe

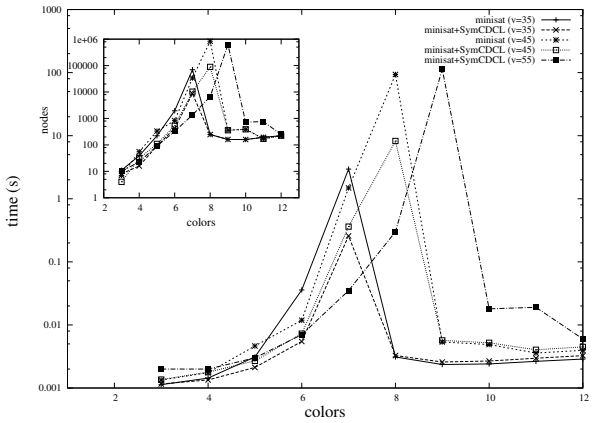


FIGURE 2 – Courbes des noeuds et temps CPU de *MiniSat* et *MiniSat+SymCDCL* sur des problèmes de coloration de graphe générés aléatoirement pour $n = \{35, 45, 55\}$ et $d = 0.5$

Les problèmes de coloration de graphe aléatoires sont générés selon les paramètres suivants :

(1) le nombre de sommets du graphe (n), (2) le nombre de couleurs ($Colors$) et (3) la densité du graphe (d), qui est un réel compris entre 0 et 1 correspondant au ratio du nombre de contraintes du graphe sur le nombre total de contraintes possibles. Les paramètres n, $Colors$ et d étant fixés, nous générons 100 problèmes aléatoirement et les résultats (temps CPU moyen , nombre de noeuds moyen) sont reportés. Le temps limite est fixé à 800 secondes pour résoudre l'instance.

Nous reportons à la figure 2 les résultats pratiques de *MiniSat*, et *MiniSat+SymCDCL*, sur trois classes de problèmes où le nombre de variables (les sommets du graphe) est fixé à 35, 45 et 55 respectivement et où la densité est fixée à ($d = 0.5$). La grande courbe de la figure 2 représente le temps CPU moyen en fonction du nombre de couleurs pour les deux méthodes et pour chaque classe de problème. La petite courbe reporte le nombre moyen de noeuds.

Des deux courbes, nous remarquons qu'en moyenne *MiniSat+SymCDCL* explore moins de noeuds que *MiniSat* et les résout plus rapidement. L'apprentissage de clauses par symétrie est profitable autour de la région difficile de ces problèmes et notre méthode *MiniSat+SymCDCL* peut résoudre des problèmes contenant 55 sommets dans le temps imparti, alors que *MiniSat* n'y arrive pas. Ceci explique que nous ne reportons pas les courbes de *MiniSat* (pour 55 sommets)à la figure 2.

5.3 Résultats sur des problèmes des dernières compétitions SAT

Nous présentons les résultats des deux méthodes (*MiniSat* et *MiniSat+SymCDCL*) sur 180 classes de problèmes issues des dernières compétitions SAT. Le temps CPU limite est fixé à 1200 secondes pour la résolution de chaque problème. Notre premier but est ici de trouver des problèmes contenant des symétries et de regarder le comportement de l'apprentissage par symétrie sur ces derniers. La sélection des problèmes s'est faite en recherchant si le problème comporte des symétries globales. Si c'est le cas, nous la sélectionnons. Sinon nous utilisons le prétraitement de *MiniSat* et nous testons la présence de symétrie sur le problème simplifié. La figure 3 montre la comparaison de *MiniSat+SymCDCL* et *MiniSat* en nombre de noeuds (figure du bas) et en temps CPU (figure du haut) sur les 180 problèmes selectionnés. Sur les deux figures,

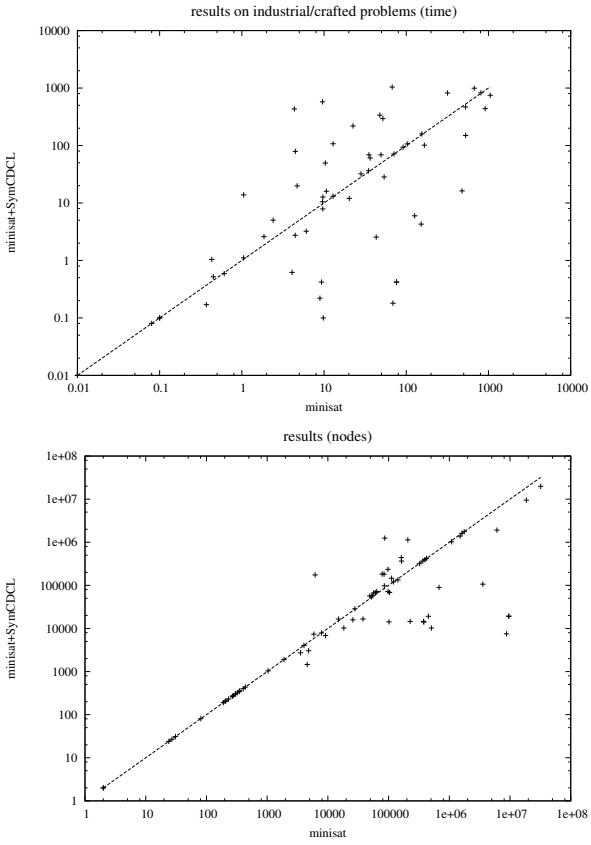


FIGURE 3 – Temps CPU et nombre de noeuds sur quelques problèmes SAT

		Minisat		Minisat+SymCDCL	
Instance	*#V : #C*	*Noeuds*	*Temps*	*Noeuds*	*Temps*
cmu-bmc-barrel6	2306 : 8931	101246	4.09	14256	0.62
counting-easier-php-012-010	120 : 672	3565803	151.89	106726	4.28
gus-md5-04	68679 : 223994	3542	6.08	2724	3.22
gus-md5-05	68827 : 224473	4811	20.29	3044	12.00
gus-md5-06	68953 : 224868	18311	53.74	10256	28.35
gus-md5-07	69097 : 225325	37867	165.98	16583	101.22
gus-md5-09	69487 : 226581	103672	1050.09	68221	747.90
gus-md5-10	69503 : 226618	− − −	>1200	97146	1143.27
mod2-3cage-9-12	87 : 232	9533790	76.14	19204	0.42
mod2-3cage-9-14	87 : 232	8754084	68.79	7490	0.18
pmg-11-UNSAT	169 : 562	18562286	911.78	9474789	439.57
Q32inK09	36 : 7938	452083	43.08	19195	2.55
Q32inK10	45 : 38430	380172	126.41	14059	5.98
Q32inK11	55 : 139590	376321	477.15	14543	16.20

TABLE 2 – Résultats sur certains problèmes

l'axe des y (resp. l'axe des x) représente les résultats de *MiniSat+SymCDCL* (resp. *MiniSat*). Un point sur la figure du haut (respectivement figure du bas) représente le temps CPU (respectivement le nombre de noeuds) des deux solveurs pour un problème donné. La projection du point sur l'axe des y (resp. l'axe des x) représente la performance de *MiniSat+SymCDCL* (resp. *MiniSat*) en temps CPU pour la figure du haut et en nombre de noeuds pour la figure du bas. Les points en dessous de la diagonale indiquent que *MiniSat+SymCDCL* est meilleur que *MiniSat*. Les points aux alentours de la diagonale montrent des résultats similaires et les points au dessus indiquent que *MiniSat* est meilleur. Nous voyons à la figure 3 qu'il existe de nombreux problèmes où *MiniSat+SymCDCL* est meilleur que *MiniSat*. Sur certains problèmes, *MiniSat* est meilleur que *MiniSat+SymCDCL* en temps CPU. Cela arrive généralement lorsque le problème admet une solution. *MiniSAT* trouve rapidement celle-ci.

Le tableau 2 donne le détail des résultats des deux méthodes sur certains problèmes issus des dernières compétitions SAT pour lesquels *MiniSat+SymCDCL* est meilleur que *MiniSat*, tant en temps CPU qu'en nombre de noeuds. Nous remarquons que *MiniSat+SymCDCL* résout l'instance *gus-md5-10* alors que *MiniSat* n'y parvient pas dans le temps imparti.

6 Conclusion et perspectives

Dans cet article, nous augmentons l'apprentissage des clauses par symétries. Nous introduisons un nouveau schéma d'apprentissage (SLS) qui peut être utilisé par n'importe quel

solveur SAT à base de CDCL. L'apprentissage par symétrie permet, en plus de l'ajout de la clause assertive dans un noeud de l'arbre de recherche, d'ajouter toutes les clauses assertives symétriques. Considérer les clauses assertives symétriques permet d'éviter aux solveurs SAT d'explorer des sous-espaces isomorphiques. Nous avons implanté le schéma d'apprentissage par symétrie (SLS) dans *MiniSat* et avons expérimenté *MiniSat* et *MiniSat* avec le schéma SLS sur une grande variété de problèmes. Les premiers résultats expérimentaux sont très encourageants, et montrent que l'utilisation de la symétrie dans l'apprentissage est profitable sur la plupart des problèmes testés.

Comme amélioration future, nous chercherons à trouver une bonne stratégie de suppression de clauses qui préservera les clauses symétriques apprises pendant la recherche. Actuellement, *MiniSat* peut supprimer des clauses assertives symétriques qui peuvent être utiles pour couper des sous-espaces isomorphes.

Un autre point est que seules les symétries du problème initial (symétries globales) sont utilisées dans ce schéma d'apprentissage. Nous prévoyons d'étendre ce schéma d'apprentissage à l'exploitation des symétries locales, qui peuvent être détectées au cours de la recherche.

Références

[1] F. A. Aloul, A.Ramani, I. L. Markov, and K. A. Sakallak. Solving difficult sat instances in the presence of symmetry. *In IEEE Transaction on CAD, vol. 22(9)*, pages 1117–1137, 2003.

[2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallak. Symmetry breaking for pseudo-boolean satisfiabilty. *In ASPDAC'04*, pages 884–887, 2004.

[3] Gilles Audemard, Lucas Bordeaux, Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. A generalized framework for conflict analysis. In *SAT*, pages 21–27, 2008.

[4] R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *Principle and Practice of Constraint Programming - CP'99*, 1999.

[5] 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.

[6] B. Benhamou. Study of symmetry in constraint satisfaction problems. *In Proceedings of the 2nd International workshop on Principles and Practice of Constraint Programming - PPCP'94*, 1994.

[7] B. Benhamou and L. Sais. Theoretical study of symmetries in propositional calculus and application. *Eleventh International Conference on Automated Deduction, Saratoga Springs,NY, USA*, 1992.

[8] B. Benhamou and L. Sais. Tractability through symmetries in propositional calculus. *Journal of Automated Reasoning (JAR),*, 12 :89–102, 1994.

[9] B. Benhamou, L. Sais, and P. Siegel. Two proof procedures for a cardinality based language. *in proceedings of STACS'94, Caen France*, pages 71–82, 1994.

[10] James Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates

for search problems. In *KR'96 : Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, San Francisco, California, 1996.

[11] Niklas Eén and Niklas S´orensson. 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.

[12] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *International conference on constraint programming*, volume 2239 of *LNCS*, pages 93–108. Springer Verlag, 2001.

[13] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *International conference on constraint programming*, volume 2239 of *LNCS*, pages 77–82. Springer Verlag, 2001.

[14] E.C. Freuder. Eliminating interchangeable values in constraints satisfaction problems. *Proc AAAI-91*, pages 227–233, 1991.

[15] I. P. Gent, W. Hervey, T. Kesley, and S. Linton. Generic sbdd using computational group theory. In *Proceedings CP'2003*, 2003.

[16] I. P. Gent and B. M. Smith. Symmetry breaking during search in constraint programming. In *Proceedings ECAI'2000*, 2000.

[17] I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints : Symmetry breaking during search. In *International conference on constraint programming*, volume 2470 of *LNCS*, pages 415–430. Springer Verlag, 2002.

[18] Carla P. Gomes, Bart Selman, and Nuno Crato. Heavy-tailed distributions in combinatorial search. In *CP*, pages 121–135, 1997.

[19] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In *AAAI*, pages 283–290, 2008.

[20] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI'07 : Proceedings of the 20th international joint conference on Artifical intelligence*, pages 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.

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

[22] Tommi Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In David Applegate, Gerth Stølting Brodal, Daniel Panario, and Robert Sedgewick, editors, *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics*, pages 135–149. SIAM, 2007.

[23] B. Krishnamurty. Short proofs for tricky formulas. *Acta informatica*, (22) :253–275, 1985.

[24] Knot Pipatsrisawat and Adnan Darwiche. A new clause learning scheme for efficient unsatisfiability proofs. In *AAAI*, pages 1481–1484, 2008.

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

[26] J. F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *In J. Kamorowski and Z. W. Ras,editors, Proceedings of ISMIS'93, LNAI 689*, 1993.

[27] J.F. Puget. Symmetry breaking revisited. In *International conference on constraint programming*, volume 2470 of *LNCS*, pages 446–461. Springer Verlag, 2002.

[28] C. M. Roney-Dougal, I. P. Gent, T. Kelsey, and S. A. Linton. Tractable symmetry breaking using restricted search trees. *In, proceedings of ECAI'04*, pages 211–215, 2004.

[29] J. P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *ICCAD*, pages 220–227, 1996.

[30] T. Walsh. General symmetry breaking constraints. *In, proceedings of CP'06*, pages 650–664, 2006.

[31] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

Détection et élimination dynamique de la symétrie dans le problème de satisfiabilité

Belaïd Benhamou Tarek Nabhani Richard Ostrowski Mohamed Réda Saïdi[1]

[1] Université de Provence
Laboratoire des Sciences de l'Information et des Systèmes (LSIS)
Centre de Mathématiques et d'Informatique.
39, rue Joliot Curie - 13453 Marseille cedex 13, France.
{benhamou; nabhani; ostrowski; saidi}@cmi.univ-mrs.fr

Résumé

Le problème SAT est connu pour être le premier problème de décision de classe NP-complet (Cook,71). C'est un problème central dans la théorie de la complexité. Dans la dernière décennie, les procédures prouvant la satisfiabilité ont été améliorées par l'élimination de la symétrie. Une formule CNF contient usuellement un nombre intéressant de symétries. Il y a deux types d'exploitions des symétries. La première correspond à l'élimination des symétries globales, c'est à dire, seules les symétries initiales du problème (le problème à la racine de l'arbre de recherche) sont détectées et éliminées. La seconde exploite toutes les symétries locales qui apparaissent à chaque noeud de l'arbre de recherche. Les symétries locales doivent être détectées et éliminées dynamiquement durant la recherche. Exploiter ce genre de symétrie semble être une tâche difficile. Quasiment tous les travaux sur l'exploitation de la symétrie dans le problème de la satisfiabilité traitent uniquement le cas des symétries globales. En dépit de leur importance en pratique, seuls quelques travaux étudient les symétries locales.

Détecter et éliminer efficacement les symétries locales durant la recherche est un challenge important. Le travail que nous présentons ici est une contribution pour répondre à ce difficile challenge. Nous présentons une nouvelle méthode pour l'élimination des symétries locales qui consiste à réduire l'instance partielle SAT, non encore résolue correspondante à chaque noeud de l'arbre de recherche, à un graphe dont le groupe d'automorphismes est équivalent au groupe de symétries de l'instance partielle SAT.

Nous avons utilisé l'outil Saucy pour le calcul du groupe d'automorphisme et nous avons implémenté une technique de coupure de symétrie dans un solveur SAT. Nous avons expérimenté cette méthode sur plusieurs instances SAT et nous l'avons comparé à une méthode qui exploite les symétries globales. Les résultats obtenus sont prometteurs. L'exploitation des symétries locales améliore l'exploitation des seules symétries globales dans la résolution de nombreux problèmes difficiles et elles leurs sont complémentaires si nous combinons les deux techniques.

Abstract

The SAT problem is shown to be the first decision NP-complete problem (Cook,71). It is central in complexity theory. In the last decade, the satisfiability proof procedures are improved by symmetry elimination. A CNF formula usually contains an interesting number of symmetries. There are two kinds of symmetry exploitation. The first one corresponds to global symmetry breaking, that is, only the symmetries of the initial problem (the problem at the root of the search tree) are detected and eliminated. The second one deals with all local symmetries that appear at each node of the search tree. Local symmetry has to be detected and eliminated dynamically during the search. Exploiting such symmetries seems to be a hard task. Almost all of the known works on symmetry in satisfiability are on global symmetry. Only few works are carried on local symmetry, despite their importance in practice. An important challenge is then to detect and break local symmetries efficiently during the search. The work that we present here is a contribution towards an answer to this hard challenge. We present a new method for local symmetries breaking that consists in detecting dynamically local symmetries by reducing the remaining partial SAT instance at each node of the search tree to a graph that has an equivalent automorphism group than the symmetry group of the partial SAT instance. We used the software Saucy to compute the automorphism group and implemented a local symmetry cut in a SAT solver. We experimented

this method on several SAT instances and compared it with a method exploiting global symmetries. The results obtained are very promising. Local symmetry improves global symmetry on some hard instances and is complementary to global symmetry.

1 Introduction

Krishnamurthy dans [20] introduit le principe de la symétrie dans le calcul propositionnel et a montré que certaines formules difficiles à prouver peuvent avoir des preuves courtes si on ajoute au système de résolution la règle de symétrie. Les symétries ont été même utilisées bien avant, pour la résolution de nombreux problèmes comme par exemple le problème des huit reines [16]. Elles ont aussi été introduites dans la résolution des problèmes de satisfaction de contraintes [14, 25, 3], dans un intelligent algorithme de Backtracking [9] et dans la logique du premier ordre [11].

La symétrie est devenue une notion importante dans la programmation par contrainte. Durant la dernière décennie, plusieurs travaux sur l'exploitation des symétries dans la résolution des problèmes SAT et CSP sont apparues. Cependant, peu de travaux exploitent la détection et l'élimination dynamique des symétries [4, 5, 6, 7, 15]. La plupart des méthodes n'exploitent que les symétries globales [12, 1, 2], c'est à dire, les symétries du problème initial correspondant à la racine de l'arbre de recherche.

Une symétrie d'une formule logique est une permutation de littéraux qui laisse invariant la formule. Il existe de nombreux problèmes en intelligence artificielle qui contiennent un nombre important de symétries qui s'expriment sous forme de formules CNF.

L'importance de l'exploitation des symétries lors de la résolution peut être mise en évidence sur de nombreux problèmes que les méthodes de résolution classiques ont du mal à résoudre efficacement. Si on prend par exemple le problème des pigeons [8, 17], ou le problème de Ramsey [18]. Les deux problèmes sont connus pour être difficiles à résoudre pour les méthodes de résolution classiques et qu'ils sont représentés en logique du premier ordre par un petit ensemble de formules qui devient très large lorsque l'on tente de calculer toutes les instancions propositionnelles terminales. L'ensemble des clauses propositionnelles obtenues, contient un nombre important de symétries. C'est à dire que, l'ensemble des clauses reste invariant par rapport à plusieurs permutations de variables. Il en résulte que prouver la satisfiabilité par l'exploitation de ces permutations a une complexité polynômiale alors que l'on sait que les méthodes de résolution classiques sont tous de complexité exponentielle pour résoudre ces deux problèmes, si l'élimination de la symétrie n'est pas prise en compte.

Le problème de satisfiabilité est un problème générique, de nombreux problèmes provenant d'autres domaines peuvent être réduit au problème de satisfiabilité. Par exemple, la déduction automatique, la configuration, la planification, l'ordonnancement, etc. Plusieurs méthodes d'élimination de la symétrie pour le problème de satisfiabilité ont été introduites [12, 1, 2]. Cependant, pratiquement toutes ces méthodes exploitent uniquement les symétries globales et ignorent le traitement des symétries locales. Ceci est du à la difficulté de la détection et de l'élimination dynamique de ces symétries. Contrairement aux symétries globales qui peuvent être exploitées par des approches statiques faciles à implémenter.

Une approche qui détecte et élimine dynamiquement les symétries locales en logique propositionnelle est proposée dans [4, 5, 6]. Mais cette méthode est incomplète dans le sens où elle détecte qu'une partie des symétries locales, et non pas tout le groupe total de symétries locales. Une alternative à cette méthode est d'adapter et d'utiliser l'outil Saucy qui permet de calculer les automorphismes de graphes [1] afin de détecter les symétries locales durant la recherche, puisque le groupe d'automorphismes du graphe déduit à partir de l'instance SAT est identique au groupe de symétries de l'instance SAT.

Dans ce papier, nous présentons une méthode alternative qui élimine les symétries locales pour la résolution du problème SAT, qui exploite le groupe total de symétries. Cette méthode consiste à réduire incrémentalement la sous-formule logique définie à chaque noeud de l'arbre de recherche à un graphe sur lequel nous utilisons un outil de calcul d'automorphismes de graphes tel que Saucy [1]. L'élimination des symétries est implémentée dans un solveur SAT que nous avons expérimenté et comparé sur plusieurs instances SAT. Les résultats obtenus sont encourageants et montrent que l'exploitation des symétries locales donne de meilleurs résultats que l'exploitation des symétries globales sur certaines instances SAT et que la combinaison de l'exploitation des deux types de symétries sont complémentaires.

Le reste du papier est organisé comme suite : la section 2 rappelle les notions élémentaires sur le problème de satisfiabilité et les permutations. La section 3 définie le principe de la symétrie et donne quelques propriétés. La quatrième section décrit la nouvelle méthode de détection et d'élimination de la symétrie que nous proposons. La section 5 montre comment la coupure de symétrie est intégrée dans une procédure du type Davis et Putnam. Nous évaluons la méthode proposée dans la sixième section où plusieurs instances SAT sont testées et où une comparaison avec d'autres

méthodes est donnée. Finalement, nous concluons ce travail dans la section 7.

2 Quelques rappels en logique propositionnelle

2.1 La loogique propositionnelle

Nous supposons que le lecteur est familier avec le calcul propositionnel. Nous donnons ici, une courte description, une plus complète description peut être trouvée dans [21]. Soit V l'ensemble des variables propositionnelles que l'on peut appeler simplement, variables. Les variables doivent être distinguées des littéraux, qui sont en fait, les variables assignées à une des deux valeurs possibles, 1 ou 0, qui veut dire Vrai ou Faux respectivement(True ou False en Anglais). Cette distinction peut être ignorée si cela convient et ne porte pas à confusion. Pour une variable propositionnelle p, il y a deux littéraux : p le littéral positif et $\neg p$ le négatif.

Une clause est une disjonction de littéraux $\{p_1, p_2, \ldots, p_n\}$ tel qu'aucun littéral n'apparaît plus d'une fois, ni un littéral et sa négation en même temps. Cette clause est désignée par $p_1 \vee p_2 \vee \ldots \vee p_n$. Un système $\mathcal{F}$ de clauses est une conjonction de clauses. En d'autres mots, on dit que $\mathcal{F}$ est sous la forme normale conjonctive (CNF).

Un assignement qui vérifie un système de clauses $\mathcal{F}$ est une application I définit de l'ensemble des variables de $\mathcal{F}$ vers l'ensemble {Vrai, Faux}. Si $I[p]$ est la valeur du littéral positif p alors $I[\neg p] = 1 - I[p]$. La valeur de la clause $p_1 \vee p_2 \vee \ldots \vee p_n$ dans I est vraie, si la valeur vrai est assignée à au moins un de ses littéraux dans I, faux sinon. Par convention, nous définissons la valeur de la clause vide ($n = 0$) à faux.

La valeur $I[\mathcal{F}]$ du système de clauses est vrai si la valeur de chaque clause de $\mathcal{F}$ est vrai, faux, sinon. On dit que le système de clauses $\mathcal{F}$ est satisfaisable s'ils existent des assignements I qui vérifient le système de clauses et qui assignent la valeur vrai à $\mathcal{F}$, sinon il est insatisfaisable. Dans le premier cas I est appelé un modèle de $\mathcal{F}$. Notons que le système de clauses qui contient la clause vide est insatisfaisable.

On sait que [27] pour chaque formule propositionnelle $\mathcal{F}$ il existe une formule $\mathcal{F}'$ sous la forme normale conjonctive (CNF) telle que la taille de $\mathcal{F}'$ est au plus 3 fois plus longue que la formule $\mathcal{F}$ et que $\mathcal{F}'$ est satisfaisable si et seulement si $\mathcal{F}$ est satisfaisable.

Dans la suite nous allons assumer que les formules sont données sous la forme normale conjonctive.

2.2 Les permutations

Soit l'ensemble $\Omega = \{1, 2, \ldots, N\}$ pour un entier donné N, où chaque entier représente une variable propositionnelle. Une permutation de Ω est une application bijective σ de Ω à Ω qu'on représente usuellement comme un produit de cycles de permutations. On dénote par $Perm(\Omega)$ l'ensemble des toutes les permutations de Ω et $\circ$ la composition de permutations de $Perm(\Omega)$. La paire $(Perm(\Omega), \circ)$ forme le groupe de permutation de Ω. En effet, $\circ$ est close, associative, l'inverse d'une permutation est une permutation et la permutation identité est l'élément neutre.

Une paire $(T, \circ)$ forme un sous-groupe de $(S, \circ)$ si et seulement si T est un sous-ensemble de S et forme muni de l'opération $\circ$ un groupe.

L'orbite $\omega^{Perm(\Omega)}$ d'un élément ω de Ω sur lequel le groupe $Perm(\Omega)$ agit est $\omega^{Perm(\Omega)}=\{\omega^\sigma : \sigma \in Perm(\Omega)\}$.

Un ensemble de générateurs du groupe $Perm(\Omega)$ est un sous-ensemble Gen de $Perm(\Omega)$ tel que chaque élément de $Perm(\Omega)$ peut être écrit comme composition des éléments de Gen. Nous écrivons $Perm(\Omega)=< Gen >$. Un élément de Gen est appelé générateur. L'orbite de $\omega \in \Omega$ peut être calculée en utilisant uniquement l'ensemble des générateurs Gen.

3 La symétrie

Depuis la définition de symétrie de Krishnamurthy [20] dans la logique propositionnelle, plusieurs d'autres définitions ont été données par la communauté CP. Freuder dans son papier [14], a introduit les notions d'interchangeabilité globale et locale, où deux valeurs d'un domaine sont interchangeables dans un CSP, si elles peuvent être substituées l'une à l'autre sans aucun effet sur le CSP. En revanche Benhamou dans [3] a défini deux niveaux de symétrie sémantique et une notion de symétrie syntaxique. Il a également montré que l'interchangeabilité globale de Freuder est un cas particulier de symétrie sémantique et que l'interchangeabilité de voisinage (locale) est un cas particulier de la symétrie syntaxique.

Plus récemment, Cohen et al [10] ont fait un état de l'art sur les définitions de symétrie les plus connus dans les CSPs et ont les regrouper dans deux définitions : les symétries de solutions (sémantiques) et les symétries de contraintes (syntaxiques). Presque toutes ces définitions peuvent être identifiées comme appartenant à l'une des deux familles de symétrie : la symétrie syntaxique ou la symétrie sémantique. Nous allons définir dans ce qui suit les deux symétries sémantiques et syntaxiques dans la logique propositionnelle et nous allons montrer leur relation avec les symétries de so-

lutions ainsi que les symétries de contraintes dans les CSPs.

3.1 La symétrie dans la logique propositionnelle

Définition 1 (La symétrie sémantique) *Soit $\mathcal{F}$ une formule propositionnelle donnée sous la forme CNF et $L_\mathcal{F}$ son ensemble complet*[1] *de littéraux. Une symétrie sémantique de $\mathcal{F}$ est une permutation σ définie sur $L_\mathcal{F}$ telle que $\mathcal{F} \models \sigma(\mathcal{F})$ et $\sigma(\mathcal{F}) \models \mathcal{F}$.*

En d'autres mots, une symétrie sémantique d'une formule est une permutation. Nous rappelons dans la suite, la définition de la symétrie syntaxique donnée dans [4, 5].

Définition 2 (La symétrie syntaxique) *Soit $\mathcal{F}$ une formule propositionnelle donnée sous la forme CNF et $L_\mathcal{F}$ son ensemble complet de littéraux. Une symétrie syntaxique de $\mathcal{F}$ est une permutation σ définie sur $L_\mathcal{F}$ telle que la condition suivante soit vérifiée :*

1. $\forall \ell \in L_\mathcal{F}, \sigma(\neg\ell) = \neg\sigma(\ell)$,
2. $\sigma(\mathcal{F}) = \mathcal{F}$

En d'autres mots, une symétrie syntaxique d'une formule est une permutation de littéraux qui laisse invariant la formule. Si on dénote par $Perm(L_\mathcal{F})$ le groupe de permutations de $L_\mathcal{F}$ et par $Sym(L_\mathcal{F}) \subset Perm(L_\mathcal{F})$ le sous ensemble des permutations de $L_\mathcal{F}$ qui sont les symétries syntaxiques de $\mathcal{F}$, alors $Sym(L_\mathcal{F})$ est trivialement un sous groupe de $Perm(L_\mathcal{F})$.

Remarque 1 *Les définitions de symétrie introduites dans les CSPs [10] sont liées à celles introduites dans la logique propositionnelle. On considère par exemple l'encodage SAT $\mathcal{F}$ directement du CSP P [19] où une variable booléenne est introduite pour chaque paire variable-valeur du CSP, et où une clause interdisant chaque tuple rejeté par une contrainte spécifique est ajoutée ainsi qu'une autre clause garantissant que la valeur choisie pour chaque variable est préalablement dans son domaine. Il est trivial de voir que la symétrie de solutions du CSP P est équivalente à la symétrie sémantique (la définition 1) de son encodage SAT $\mathcal{F}$ et que la symétrie de contraintes de P est équivalente à la symétrie syntaxique (la définition 2) de $\mathcal{F}$.*

Théorème 1 *Chaque symétrie syntaxique d'une formule $\mathcal{F}$ est une symétrie sémantique de $\mathcal{F}$.*

1. L'ensemble des littéraux contenant chaque littéral de $\mathcal{F}$ et son négatif.

Preuve 1 *Il est trivial de voir que la symétrie syntaxique est une condition suffisante pour la symétrie sémantique. En effet, si σ est une symétrie syntaxique de $\mathcal{F}$, alors $\sigma(\mathcal{F}) = \mathcal{F}$, donc il en résulte que $\mathcal{F}$ et $\sigma(\mathcal{F})$ ont les mêmes modèles. Chaque symétrie syntaxique est une symétrie sémantique et en général, l'inverse est n'est pas vrai.*

Exemple 1 *Soit $\mathcal{F}$ l'ensemble des clauses suivantes : $\mathcal{F}$=$\{a \vee b \vee c, \neg a \vee b, \neg b \vee c, \neg c \vee a, \neg a \vee \neg b \vee \neg c\}$ et σ_1 et σ_2 deux permutations définies sur l'ensemble complet $L_\mathcal{F}$ des littéraux participant dans $\mathcal{F}$ comme suit :*
σ_1=$(a, b, c)(\neg a, \neg b, \neg c)$
σ_2=$(a, \neg a)(b, \neg c)(c, \neg b)$
Les deux σ_1 et σ_2 sont des symétries syntaxiques de $\mathcal{F}$, puisque $\sigma_1(\mathcal{F})$=$\mathcal{F}$=$\sigma_2(\mathcal{F})$.

Dans la suite, nous travaillerons uniquement sur les symétries syntaxiques, nous utiliserons donc uniquement le mot symétrie pour designer la symétrie syntaxique.

Définition 3 *Deux littéraux ℓ et ℓ' d'une formule $\mathcal{F}$ sont symétriques, s'il existe une symétrie σ de $\mathcal{F}$ telle que $\sigma(\ell) = \ell'$.*

Définition 4 *Soit $\mathcal{F}$ une formule, l'orbite d'un littéral $\ell \in L_\mathcal{F}$ sur lequel le groupe de symétries $Sym(L_\mathcal{F})$ opère est $\ell^{Sym(L_\mathcal{F})}$=$\{\sigma(\ell) : \sigma \in Sym(L_\mathcal{F})\}$.*

Proposition 1 *Tous les littéraux d'une orbite ℓ sont symétriques deux à deux.*

Preuve 2 *La preuve est une conséquence triviale des deux définitions précédentes.*

Exemple 2 *Dans l'exemple 1, l'orbite du littéral a est $a^{Sym(L_\mathcal{F})} = \{a, b, c, \neg a, \neg b, \neg c\}$. Nous pouvons voir que tous les littéraux sont dans la même orbite. Donc, ils sont tous symétriques.*

Si I est un modèle de $\mathcal{F}$ et σ une symétrie, nous pouvons avoir un autre modèle de $\mathcal{F}$ en appliquant σ sur les variables qui apparaissent dans I. Autrement dit, si I est un modèle de $\mathcal{F}$ alors $\sigma(I)$ est un modèle de $\mathcal{F}$. Une symétrie σ transforme chaque modèle en un autre modèle et chaque no-good en un autre no-good. Dans la proposition suivante, nous assumons que σ est une symétrie de l'ensemble des clauses $\mathcal{F}$.

Proposition 2 *Soit ℓ un littéral, σ une symétrie telle que $\ell' = \sigma(\ell)$ et $I' = \sigma(I)$. Si I est telle que $I[\ell] = Vrai$, alors I' est telle que $I'[\ell'] = Vrai$*

Preuve 3 *La preuve est triviale. En effet, si ℓ est vrai dans le modèle I alors $\sigma(\ell) = \ell'$ doit être vrai dans le modèle $\sigma(I)$=I'.*

Nous déduisons la proposition suivante.

Proposition 3 *Si un littéral ℓ a la valeur vrai dans un modèle de $\mathcal{F}$, alors $\sigma(\ell)$ doit avoir la valeur vrai dans un modèle de $\mathcal{F}$.*

Théorème 2 *Soient ℓ et ℓ' deux littéraux de $\mathcal{F}$ qui sont dans la même orbite en ce qui concerne le groupe de symétries $Sym(L_{\mathcal{F}})$, alors ℓ est vrai dans un modèle de $\mathcal{F}$ si et seulement si ℓ' est vrai dans un modèle de $\mathcal{F}$.*

Preuve 4 *Si ℓ est dans la même orbite que ℓ' alors il est symétrique avec ℓ' dans $\mathcal{F}$. Donc, il existe une symétrie σ de $\mathcal{F}$ tel que $\sigma(\ell) = \ell'$. Si I est un modèle de $\mathcal{F}$ alors $\sigma(I)$ est aussi un modèle de $\sigma(\mathcal{F}) = \mathcal{F}$, en plus, si $I[\ell] = true$ alors $\sigma(I[\ell']) = true$ (la proposition 2). Pour l'inverse, il faut considérer $\ell = \sigma^{-1}(\ell')$, et faire une preuve similaire.*

Corolaire 1 *Soit ℓ un littéral de $\mathcal{F}$, si ℓ n'est pas vrai dans aucun modèle de $\mathcal{F}$, alors chaque littéral $\ell' \in orbit^{L_{\mathcal{F}}}$ n'est pas vrai dans aucun modèle de $\mathcal{F}$.*

Preuve 5 *La preuve est une conséquence directe du théorème 2.*

Le corolaire 1 exprime une propriété importante que nous allons utiliser pour éliminer les symétries locales à chaque noeud de l'arbre de recherche. C'est à dire, si un échec est détecté après avoir affecté la valeur vrai au littéral courant ℓ, alors nous calculons l'orbite de ℓ et nous assignons la valeur faux à chaque littéral dans l'orbite, puisque par symétrie nous savons que la valeur vrai ne peut être assignée à ces littéraux sous peine d'avoir de produire un échec, donc les littéraux assignés à vrai de l'orbite ne participe dans aucun modèle de la sous formule considérée.

De nombreux problèmes difficiles pour la résolution ont été démontrés de complexités polynômiales si la symétrie est considérée. Par exemple, trouver des nombres de Ramsey ou résoudre le problème des pigeons sont connus pour être exponentielles pour la résolution classique, tandis que de courtes démonstrations peuvent être faites pour les deux lorsqu'on exploite la symétrie dans le système de résolution. Nous allons montrer maintenant comment détecter dynamiquement la symétrie locale.

4 La détection et l'élimination de la symétrie locale

Les symétries locales doivent être détectées dynamiquement à chaque noeud de l'arbre de recherche. La détection dynamique de la symétrie a été étudiée dans [4, 5] où une méthode de recherche des symétries syntaxiques locales a été donnée. Cependant, cette méthode n'est pas complète, elle détecte uniquement une symétrie σ à chaque noeud de l'arbre de recherche lors de l'échec dans l'affection du littéral courant ℓ. Une heuristique est utilisée sur les permutations de variables de σ à fin d'avoir le nombre maximal de littéraux dans le même cycle de permutation dans lequel apparait ℓ. En dépit de cette heuristique, cette méthode ne détecte pas tous les littéraux symétriques avec ℓ correspondant à l'orbite de ℓ, puisqu'il n'utilise pas toutes les symétries locales.

Comme alternative à cette méthode de recherche de symétrie incomplète, nous avons adapté Saucy [1] pour détecter toutes les symétries syntaxiques et nous montrons comment éliminer ces symétries durant la recherche. Saucy est un outil pour le calcul des automorphismes de graphes. Il existe d'autres outils comme Nauty [22] et plus récemment AUTOM [26] ou aussi celui décrit dans [23] qui peuvent être adaptés pour la détection des symétries locales. Il est montré dans [26] que AUTOM est l'un des meilleurs outils. Cependant, il n'est pas gratuit et comme depuis peu, une nouvelle version plus performante de Saucy vient de sortir, [13] ; nous avons choisit Saucy. Il est montré dans [12, 1, 2] que chaque formule CNF $\mathcal{F}$ peut être représentée par un graphe $G_{\mathcal{F}}$ qui est construit de la façon suivante :

- Chaque variable booléenne est représentée par deux sommets (sommet littéral) dans $G_{\mathcal{F}}$: le littéral positif et son négatif. Ces deux sommets sont connectés par une arête dans le graphe $G_{\mathcal{F}}$.
- chaque clause non binaire est représentée par un sommet (sommet clause). Une arête connecte ce sommet à chaque sommet représentant un littéral de la clause.
- Chaque clause binaire est représentée par une arête connectant les deux sommets représentant les deus littéraux de la clause. Les sommets correspondants aux clauses binaires ne sont pas ajoutés.

Une propriété importante du graphe $G_{\mathcal{F}}$ est qu'il préserve le groupe de symétries syntaxiques de $\mathcal{F}$. En effet, le groupe de symétries syntaxiques de la formule $\mathcal{F}$ est identique au groupe d'automorphismes de sa représentation graphique $G_{\mathcal{F}}$, donc nous utilisons Saucy sur $G_{\mathcal{F}}$ pour détecter le groupe de symétries syntaxiques de $\mathcal{F}$. Saucy retournes l'ensemble des générateurs *Gen* du groupe de symétries duquel on peut déduire chaque symétrie. Saucy offre la possibilité de colorer les sommets du graphe tels que, chaque sommet est autorisé à être permuté avec un autre sommet s'ils ont la même couleur. Ceci restreint les permutations aux noeuds ayant la même couleur. Deux couleurs sont utilisées dans $G_{\mathcal{F}}$, une pour les som-

mets correspondant aux clauses de $\mathcal{F}$ et l'autre couleur pour les sommets représentant les littéraux de $L_{\mathcal{F}}$. Ceci permet de distinguer les sommets clauses des sommets littéraux, et prévient donc la génération de symétries entre clauses et littéraux. Le code source de Saucy peut être trouvé à l'adresse (http ://vlsi-cad.eecs.umich.edu/BK/SAUCY/).

Exemple 3 *Soit $\mathcal{F}$ la formule CNF donnée dans l'exemple 1. Son graphe associé $G_{\mathcal{F}}$ est donné dans la figure 1.*

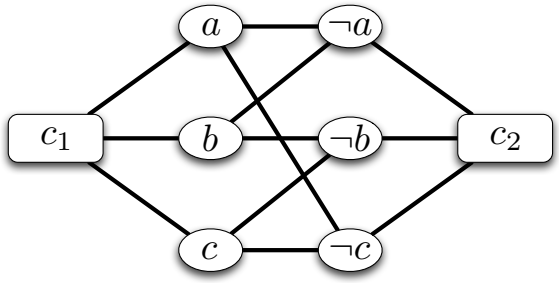


FIGURE 1 – Le graphe $G_{\mathcal{F}}$ correspondant à $\mathcal{F}$

4.0.1 Détection dynamique de la symétrie :

On considère la formule CNF $\mathcal{F}$, et un assignement partiel I de $\mathcal{F}$ où ℓ est le littéral courant en cours d'assignement. L'assignement I simplifie la formule donnée $\mathcal{F}$ en une sous formule $\mathcal{F}_I$ qui définit un état de l'espace de recherche correspondant au noeud courant n_I de l'arbre de recherche. L'idée, est de maintenir dynamiquement le graphe $G_{\mathcal{F}_I}$ de la sous formule $\mathcal{F}_I$ correspondant au sous problème local défini au noeud courant n_I, alors on colorie le graphe $G_{\mathcal{F}_I}$ et on calcule son groupe d'automorphismes $Aut(\mathcal{F}_I)$. La sous formule $\mathcal{F}_I$ peut être vu comme le sous problème restant correspondant à la partie non encore résolue. En appliquant Saucy sur ce graphe coloré, nous pouvons déduire l'ensemble des générateurs Gen du sous groupe de symétries existant entre les littéraux de $L_{\mathcal{F}_I}$ à partir desquels on peut calculer l'orbite du littéral courant ℓ qui sera utilisée pour faire des coupures de symétries.

4.0.2 Elimination des symétries :

Nous utilisons le corolaire 1 pour élaguer des espaces de recherche des méthodes de résolution. En effet, si l'assignement de la valeur vrai au littéral courant ℓ défini à un noeud donné n_I de l'arbre de recherche est montré qu'il mène à un échec, alors l'assignement de la valeur vrai à n'importe quel littéral de l'orbite de ℓ mènera aussi à un échec. Donc, la valeur $faux$ doit être assignée à chaque littéral de l'orbite de ℓ. Nous élaguons alors, le sous espace correspondant à l'assignement de la valeur alternative $vrai$ à ces littéraux dans l'arbre de recherche. C'est ce qu'on appelle les coupures de symétries.

5 Exploitation de la symétrie dans un algorithme de résolution

Maintenant nous allons montrer comment ces littéraux symétriques peuvent être utilisés pour augmenter l'efficacité des algorithmes de résolution SAT. Nous choisissons dans notre implémentation la procédure Davis Putnam (DP) comme méthode de base que nous souhaitons améliorer par l'exploitation de la symétrie.

Si I est une interprétation partielle inconsistante dans laquelle l'assignement de la valeur $vrai$ au littéral courant ℓ est montrée être en conflit, alors en accord avec le corolaire 1, tous les littéraux dans l'orbite de ℓ calculés en utilisant le groupe $Sym(\mathcal{F}_I)$ retourné par Saucy sont symétriques à ℓ. Ainsi, nous assignons la valeur faux à chaque littéral de $\ell^{Sym(L_{\mathcal{F}})}$ puisque la valeur vrai est montrée être contradictoire, et alors nous élaguons le sous espace correspondant aux assignements à la valeur vrai. La procédure résultante appelée Satisfiable est donnée dans l'algorithme 1.

La fonction $orbit(\ell, Gen)$ est élémentaire, elle permet de calculer l'orbite du littéral ℓ à partir de l'ensemble des générateurs Gen retourné par Saucy.

6 Expérimentations

Nous allons maintenant étudier les performances de notre méthode par le biais d'expérimentations. Nous choisissons pour notre étude des instances SAT pour mettre en évidence l'intérêt de l'exploitation de la symétrie dans la satisfiabilité. Nous posons que l'apport de la symétrie sera plus important dans des applications réelles. Ici, nous avons testé et comparé 4 méthodes :

1. **No-sym :** recherche sans élimination de symétries, c'est le solveur LSAT de base [24] ;
2. **Global-sym** recherche avec détection et élimination des symétries globales. Cette méthode exploite un programme nommé SHATTER, comme préprocesseur [1, 2] qui détecte et élimine les symétries globales de l'instance considérée en ajoutant à l'instance des clauses éliminant ces symétries. L'instance obtenue est alors résolu par le solveur LSAT. Le temps CPU de *Global-sym* dans la table 1 inclut le temps nécessaire à SHATTER pour le calcul et l'élimination des symétries globales.

Algorithm 1: La procédure Davis Putnam muni de l'élimination des symétries locales

Procedure: Satisfiable($\mathcal{F}$)

```
1  begin
2      if F = ∅ then F est satisfaisable
3      else si F contient la clause vide alors F est insatisfaisable
4      else begin
5          if il existe un mono-littéral ou un littéral monotone ℓ then
6          if Satisfiable(F_ℓ) then F est satisfaisable
7          else F est insatisfaisable
8          else begin
9              Choisir un littéral non assigné ℓ de F
10             if Satisfiable(F_ℓ) then F est satisfaisable
11             else begin
12                 Gen=Saucy(F);
13                 ℓ^{Sym(L_F)}=orbit(ℓ,Gen)={ℓ_1, ℓ_2, ..., ℓ_n};
14                 if Satisfiable(F_{¬ℓ_1∧¬ℓ_2∧...∧¬ℓ_n}) then F est satisfaisable else F est insatisfaisable
15             end
16         end
17     end
18 end
```

		No-sym		Global-sym		Local-sym		Global-Local-sym	
Instance	*Vars* : *clauses*	*Noeuds*	*Temps*	*Noeuds*	*Temps*	*Noeuds*	*Temps*	*Noeuds*	*Temps*
fpga10_8_SAT	120 : 448	6,637,776	44.41	449	0.02	9835	2.09	449	0.71
fpga10_9_SAT	135 : 549	-	>1,000	284	0.02	57080	20.37	284	0.53
fpga12_8_SAT	144 : 560	6,637,776	35.79	165	0.00	9835	2.14	165	0.32
fpga13_10_SAT	195 : 905	-	>1,000	4261	0.41	304,830	134.89	4261	14.08
Chnl10_11	220 : 1122	3,628,800	100.09	382	0.09	512	2.42	382	3.33
Chnl10_12_3	240 : 1344	3,628,800	120.72	322	0.10	512	2.63	322	3.41
Chnl11_12_3	264 : 1476	-	>1,000	1123	0.26	1024	6.28	1123	12.09
Chnl11_13	286 : 1742	-	>1,000	814	0.25	1024	7.38	814	10.96
Chnl11_20	440 : 4220	-	>1,000	523	0.38	1024	18.93	523	18.90
Urq3_5	46 : 470	-	>1,000	16384	0.16	30	0.09	15	0.00
Urq4_5	74 : 694	-	>1,000	-	>1,000	44	0.32	31	0.10
Urq5_5	121 : 1210	-	>1,000	-	>1,000	73	1.43	44	0.27
Urq6_5	180 : 1756	-	>1,000	-	>1,000	110	4.76	84	2.03
Urq7_5	240 : 2194	-	>1,000	-	>1,000	147	9.32	108	3.44
Urq8_5	327 : 3252	-	>1,000	-	>1,000	225	27.11	171	9.18

TABLE 1 – Quelques résultats sur des instances SAT

3. **Local-sym :** recherche avec détection et élimination des symétries locales. Cette méthode implémente dans LSAT la stratégie de détection et d'élimination dynamique décrite dans ce travail. Le temps CPU de *Local-sym* inclut le coût en temps de l'exploitation des symétries locales.
4. **Global-Local-sym :** recherche qui combine l'exploitation des symétries globales et locales. La méthode consiste à utiliser LSAT avec exploitation des symétries locale (à savoir donc la méthode *Local-sym*) sur les instances produites par l'utilisation de SHATTER comme préprocesseur.

sur de différentes instances SAT comme les FPGA (Field Programmable Gate Array), Chnl, Urquhart et quelques instances issus du problème de coloration de graphes. Nous rappelons que le point commun entre les différentes méthodes présentées précédemment est qu'elles ont toutes en commun la méthode LSAT comme méthode de base. Les indicateurs de complexité sont le nombre de noeuds ainsi que le temps (en secondes). Le temps nécessaire pour le calcul et l'exploitation des symétries globales et locales est ajouté au temps CPU total pour la résolution des instances. Le code source est en C, il est compilé et exécuté sur une machine équipée d'un Pentium 4, 2.8 GHZ et 1 Gb de RAM.

6.1 Résultats sur les instances SAT

La table 1 montre les premiers résultats obtenus des différentes méthodes sur les instances SAT choisit. Elle donne le nom des instances, la taille de celles-ci (*variables*/*clauses*), le nombre de noeuds et le temps CPU pour la résolution des instances pour chaque méthode.

La table 1 montre que *Global-sym* est en général meilleure que *Local-sym* et *No-sym* en nombre de noeuds et en temps CPU sur les problèmes $FPGA$ et $Chnl$, mais *Local-sym* est capable de les résoudre aussi. Ces problèmes contiennent un nombre très important de symétries globales, ceci explique qu'il est suffisant de les éliminer afin de résoudre efficacement ces instances. Eliminer les symétries locales sur ces problèmes peut parfois rendre la résolution plus lente. Les instances Urq sont connus pour être plus durs que les $FPGA$ et les $Chnl$, nous pouvons voir que *No-sym* n'est pas capable de les résoudre et que *Global-sym* n'a résolu que l'instance $Urq3\_5$ et n'a pas pu résoudre les autres dans la limite de temps imposée. La méthode *Local-sym* a résolu tous les instances Urq efficacement. L'élimination de la symétrie locale dans ce cas est plus avantageuse que l'élimination des symétries globales. Nous pouvons voir qu'en moyenne la méthode *Global-Local-sym* est meilleure que toutes les autres méthodes, puisqu'elle a résolu toutes les instances efficacement. Ses performances son comparable à la méthode *Global-sym* sur les instances $FPGA$ et $Chnl$ et à la méthode *Local-sym* sur les instances Urq. Il est donc plus avantageux de combiner les deux types d'élimination de la symétrie pour ces problèmes. Les résultats confirment que les deux méthodes *Global-sym* et *Local-sym* peuvent être complémentaires.

6.2 Les résultats sur les instances du problème de coloration de graphes

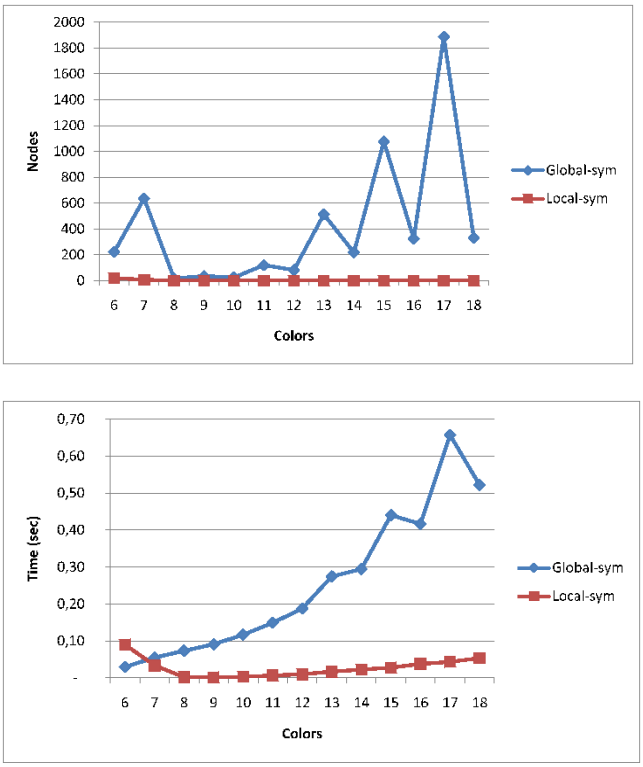


FIGURE 2 – Les courbes de noeuds et temps en moyenne pour les deux méthodes d'élimination de la symétrie sur le problème de coloration de graphes où $n = 30$ et $d = 0.5$

Les instances du problème de coloration de graphes sont générées en fonction des paramètres suivants : (1) n : le nombre de sommets, (2) $Colors$: le nombre de couleurs et (3) d : la densité qui est un nombre entre 0 et 1 qui exprime le ratio : le nombre de contraintes (le nombre d'arêtes dans le graphe) sur le nombre de toutes le contraintes possibles. Pour chaque test correspondant aux paramètres n, $Colors$ et d fixés, un échantillon de 100 instances est générée aléatoirement et les mesures (temps CPU, nombre de noeuds) sont prises en moyenne.

Nous avons reporté sur la figure 2 les résultats ob-

tenus à partir des méthodes testées : *Global-sym*, et *Local-sym*, sur les instances générées aléatoirement où nous avons fixé le nombre de variables à $n = 30$ et où la densité est fixée à $(d = 0.5)$. Les courbes donnent le nombre moyen de noeuds, respectivement, le temps CPU moyen en fonction du nombre de couleur pour chaque méthode.

Nous pouvons voir que les courbes représentant le nombre moyen de noeuds (les courbes du haut) que la méthode *Local-sym* détecte et élimine plus de symétries que la méthode *Global-sym* et que *Global-sym* n'est pas stable pour le problème de coloration de graphe. A partir des courbes de temps (les courbes du bas), nous pouvons voir que *Local-sym* est en moyenne plus rapide que *Global-sym* bien que Saucy est exécuté à chaque noeud pour la méthode *Local-sym*. L'élimination des symétries locale est plus avantageuse pour la résolution du problème de coloration de graphes que l'élimination des seules symétries globales sur ces problèmes.

7 Conclusion et perspectives

Ici, nous avons étendu le principe de détection et d'élimination des symétries aux symétries locales. En effet, les symétries de chaque sous formule CNF définie à chaque noeud de l'arbre de recherche et qui est dérivée de la formule initiale en considérant l'assignement partiel correspondant à ce noeud. Nous avons adapté Saucy pour calculer ces symétries locales en maintenant dynamiquement le graphe correspondant la sous formule définie à chaque noeud de l'arbre de recherche.

On donne à Saucy en entrée le graphe de la sous formule locale et il nous retourne alors l'ensemble des générateurs du groupe d'automorphismes du graphe donné, sachant que ce groupe est équivalent au groupe de symétries locale de la sous formule considérée. La technique de détection et d'élimination des symétries locales a été implémentée dans une méthode de résolution de problèmes SAT appelée *LSAT* afin d'améliorer ses performances. Les résultats expérimentaux confirment que l'élimination des symétries locales est d'un apport non négligeable pour la résolution des problèmes SAT et améliore les méthodes n'exploitant que l'élimination des symétries globales sur certains problèmes, et qu'elle peut être complémentaire à l'élimination des symétries globales si on les combine.

Comme travail futur, nous envisageons d'implémenter une version affaiblie des conditions de détection des symétries locales que nous supposons plus avantageuse pour détecter un plus grand nombre de symétrie, que nous comparerons ses résultats avec ceux qu'on vient de présenter.

Un autre point important est de tenter de détecter les symétries de variables locales et de poster dynamiquement des contraintes qui les éliminent. Il serait alors important de comparer les approches statiques qui détectent uniquement les symétries globales avec cette approche.

Références

[1] F. A. Aloul, A.Ramani, I. L. Markov, and K. A. Sakallak. Solving difficult sat instances in the presence of symmetry. *In IEEE Transaction on CAD, vol. 22(9)*, pages 1117–1137, 2003.

[2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallak. Symmetry breaking for pseudo-boolean satisfiabilty. *In ASPDAC'04*, pages 884–887, 2004.

[3] B. Benhamou. Study of symmetry in constraint satisfaction problems. *In Proceedings of the 2nd International workshop on Principles and Practice of Constraint Programming - PPCP'94*, 1994.

[4] B. Benhamou and L. Sais. Theoretical study of symmetries in propositional calculus and application. *Eleventh International Conference on Automated Deduction, Saratoga Springs,NY, USA*, 1992.

[5] B. Benhamou and L. Sais. Tractability through symmetries in propositional calculus. *Journal of Automated Reasoning (JAR),*, 12 :89–102, 1994.

[6] B. Benhamou, L. Sais, and P. Siegel. Two proof procedures for a cardinality based language. *in proceedings of STACS'94, Caen France*, pages 71–82, 1994.

[7] Belaïd Benhamou and Mohamed Réda Saïdi. Local symmetry breaking during search in csps. In Springer, editor, *The 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *LNCS*, pages 195–209, Providence, USA, 2007.

[8] W. Bibel. Short proofs of the pigeon hole formulas based on the connection method. *Automated reasoning*, (6) :287–297, 1990.

[9] Brown, C. A.Finkelstein, and L. P. W. Purdom. Bactrack searching in the presence of symmetry. *In t. Mora (ed), Applied algebra, algebraic algorithms and error-correcting codes, 6th International Conference. Springer-Verlag*, (6) :99–110, 1988.

[10] D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B. Smith. Symmetry definitions for constraint satisfaction problems. *In, proceedings of CP*, pages 17–31, 2005.

[11] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. *Workshop on Tractable Resonning, AAAI-92, San Jose*, July 1992.

[12] James Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *KR'96 : Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, San Francisco, California, 1996.

[13] P. T. Darga, K. A. Sakallah, and I. L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45st Design Automation Conference, Anaheim, California*, 2008.

[14] E.C. Freuder. Eliminating interchangeable values in constraints satisfaction problems. *Proc AAAI-91*, pages 227–233, 1991.

[15] Ian P. Gent, Tom Kelsey, S. A. Linton, J. Pearson, and Colva M. Roney-Dougal. Groupoids and conditional symmetry. In *CP*, pages 823–830, 2007.

[16] J. W. L. Glaisher. On the problem of the eight queens. *Philosophical Magazine*, 48(4) :457–467, 1874.

[17] A. Haken. The intractability of resolution. *Theorical Computer Science*, 39 :297–308, 1985.

[18] J.G. Kalbfleisch and R.G. Stanton. On the maximal triangle-free edge-chromatic graphs in three colors. *combinatorial theory*, (5) :9–20, 1969.

[19] J. De Kleer. A comparison of atms and csp techniques. *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'89)*, pages 290–296, 1989.

[20] B. Krishnamurty. Short proofs for tricky formulas. *Acta informatica*, (22) :253–275, 1985.

[21] R.C. Lyndon. *Notes of logic*. Van Nostrand Mathematical Studies, 1964.

[22] B McKay. Practical graph isomorphism. In *Congr. Numer. 30*, pages 45–87, 1981.

[23] C. Mears, M. Garcia de la Banda, and M. Wallace. On implementing symmetry detection. In *The CP 2006 Workshop on Symmetry and Constraint Satisfaction Problems (SymCon'06)*, pages 1–8, Cité des Congrès - Nantes, France, septembre 2006.

[24] R. Ostrowski, B. Mazure, and L. Sais. Lsat solver. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, 2002.

[25] J. F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *J. Kamorowski and Z. W. Ras,editors, Proceedings of ISMIS'93, LNAI 689*, 1993.

[26] J. F. Puget. Automatic detection of variable and value symmetries. In LNCS Springer, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP-2005)*, pages 474–488, Sitges, Spain, october 2005.

[27] P. Siegel. Representation et utilisation de la connaissance en calcul propositionnel, 1987. Thèse d'état, GIA - Luminy (Marseille).

Symétries dans les logiques non monotones

Belaïd Benhamou **Tarek Nabhani** **Pierre Siegel** [1]

[1] Université de Provence
Laboratoire des Sciences de l'Information et des Systèmes (LSIS)
Centre de Mathématiques et d'Informatique.
39, rue Joliot Curie - 13453 Marseille cedex 13, France.
{Belaid.Benhamou;Tarek.Nabhani;siegel}@cmi.univ-mrs.fr

Résumé

La symétrie a été bien étudiée dans les logiques classiques et dans la programmation par contraintes depuis une décennie. Toutefois, en Intelligence Artificielle, nous avons l'habitude de manipuler des informations incomplètes qui nécessitent d'inclure l'incertitude dans le raisonnement sur la connaissance avec exceptions et la non-monotonie. Plusieurs logiques non classiques sont mises en place à cet effet, mais, selon nos connaissances, la symétrie dans ces logiques n'a pas encore été étudiée. Ici, nous nous sommes intéressés à étendre la notion de la symétrie à des logiques non classiques telles que les logiques préférentielles, X-logiques et les logiques des défautss, puis donner des nouvelles règles d'inférence par symétrie pour les X-logiques et les logiques des défauts. Enfin, nous avons montré comment le raisonnement par symétrie est rentable pour ces logiques et comment elles gèrent certaines symétries qui n'existent pas dans des logiques classiques.

Abstract

Symmetry had been well studied in classical logics and constraint programming since a decade. However, in Artificial Intelligence, we usually manipulate incomplete information and need to include uncertainty to reason on knowledge with exceptions and non-monotonicity. Several non classic logics are introduced for that purpose, but as far as we know, symmetry for these frameworks had not been studied yet. Here, we are interested in extending the notion of symmetry to that non classical logics such as Preferential logics, X-logics and Default logics, then give new symmetry inference rules for the X-logics and the Default logics. Finally, we show how symmetry reasoning is profitable for these logics and how they handle some symmetries that do not exist in classical logics.

1 Introduction

La symétrie est par définition un concept multidisciplinaire. Il apparaît dans de nombreux domaines allant des mathématiques à l'intelligence artificielle, la chimie et la physique. En général, elle revient à une transformation, ce qui laisse invariant (ne modifie pas sa structure fondamentale et/ou ses propriétés) d'un objet (un chiffre, une molécule, un système physique, une formule ou d'un réseau de contraintes ...). Par exemple, la rotation d'un échiquier 180 degrés donne un état qui ne se distingue pas de celui d'origine. La symétrie est une propriété fondamentale qui peut être utilisée pour étudier ces différents objets, analyser finement ces systèmes complexes ou pour réduire la complexité de calcul lorsqu'il s'agit de problèmes combinatoires.

Le principe de la symétrie dans l'IA a été introduit par Krishnamurthy [19] pour améliorer la résolution en logique propositionnelle. Les symétries des contraintes booléennes sont étudiées en profondeur dans [5, 6, 7]. Les auteurs ont montré comment détecter les symétries et ont prouvé que leur exploitation est une réelle amélioration de l'efficacité des plusieurs algorithmes de déduction automatique. Ensuite, de nombreux travaux de recherche sur les symétries ont apparu. Par exemple, l'approche statique utilisée par James Crawford et al. [11] pour les théories de logiques propositionnelles consiste à ajouter des contraintes qui exprime les symétries globales du problème. Cette technique a été améliorée dans [1] et étendu à "0-1 Integer Logic Programming" dans [2]. La notion d'interchangeabilité dans les problèmes de satisfaction de contraintes (CSPs) est introduite dans [14] et la symétrie pour les CSPs est étudiée plus tôt dans [21, 4]. Comme il faut ajouter un grand nombre de contraintes dans l'approche statique dans le domaine de CSPs, certains chercheurs ont proposé d'ajou-

ter les contraintes lors de la recherche. Dans [3, 16, 17], les auteurs ajoutent certaines contraintes conditionnelles qui suppriment la symétrie de l'interprétation partielle en cas de retour arrière. Dans [13, 12, 22, 15], les auteurs ont proposé d'utiliser chaque sous-arbre comme un "no-good" pour éviter l'exploration de certaines interprétations symétriques [24]. Ensuite, Walsh, dans [27], a étudié des nouvelles propagateurs divers pour couper différentes symétries, parmi elles la symétrie agit simultanément sur les variables et les valeurs à la fois. En revanche, dans le cadre de l'intelligence artificielle, un paradigme important est de tenir compte des informations incomplètes (données incertaines, des informations révisables ...). Une composante essentielle de l'intelligence (qui est humaine, animale ou artificielle) est, en effet, liée à une certaine capacité d'adaptation du raisonnement. Contrairement à ce mode de raisonnement formalisé par une logique conventionnelle ou classique, le résultat déduit d'une information (à partir des connaissances, ou des croyances) n'est pas vrai, mais seulement probable dans le sens où il peut être invalide ou révisée lors de l'ajout d'une nouvelle information. Par exemple, il est admis qu'un oiseau normale vole. Ainsi, si l'on sait que Titi est un oiseau, alors on en conclut que, naturellement, Titi vole. Si on apprend par la suite que Titi est un pingouin, cette conclusion devra être révisée. Ceci est impossible dans une logique classique ayant la propriété de la monotonie : une information déduite d'une base de connaissances C, elle sera toujours vraie, si C est augmentée.

Pour gérer le problème des exceptions, plusieurs approches logiques ont été introduites dans l'intelligence artificielle. De nombreux formalismes non-monotones ont été présentés depuis une trentaine d'années, mais le problème de la symétrie dans ce cadre n'a pas été étudié. Le raisonnement en utilisant la symétrie est cependant pertinent pour la représentation des connaissances et le raisonnement non-monotone. Par exemple, dans l'exemple précédent, il est intéressant de considérer que les oiseaux normaux appartiennent à la même classe en respectant certaines propriétés de base, puis ils sont tous symétriques dans ce sens.

Dans ce travail, nous étudions la symétrie dans trois logiques non-classiques : la logique préférentielle [9, 10, 25, 8, 18], la X-logique [26] et la logique des défauts [23]. Le reste du papier est organisé comme suit : La section 2 donne les principales définitions de la symétrie dans la logique propositionnelle. Dans la section 3, on va étudier la symétrie dans les logiques préférentielles. Section 4 étend la symétrie au formalisme X-logique. Nous allons présenter dans la section 5 la symétrie dans la logique des défauts. Enfin, la section 6 conclut le travail et donne quelques perspectives.

2 Symétrie dans la logique propositionnelle

Tout d'abord, nous donnons la définition de symétrie sémantique dans la logique propositionnelle :

Définition 1 (Symétrie sémantique) *Soient F une formule propositionnelle sous la forme CNF et L_F son ensemble complet de littéraux*[1]*. Une symétrie sémantique de F est une permutation σ définie sur L_F telle que $F \models \sigma(F)$ et $\sigma(F) \models F$.*

En d'autres termes, une symétrie sémantique d'une formule est une permutation de littéraux qui conserve l'ensemble des modèles de la formule. Elle conserve également l'ensemble des no-goods (contre modèles). Maintenant, nous rappelons la définition d'une symétrie plus restreinte, qui est la symétrie syntaxique [5, 6] et qui peut être calculée de manière efficace.

Définition 2 (Symétrie syntaxique) *Soient F une formule propositionnelle sous la forme CNF et L_F son ensemble complet de littéraux. Une symétrie syntaxique de F est une permutation σ définie sur L_F telle que les conditions suivantes sont vérifiées :*

1. $\forall l \in L_F, \sigma(\neg l) = \neg\sigma(l)$,
2. $\sigma(F) = F$

Une symétrie syntaxique d'une formule est une permutation de variables qui laisse invariant la formule. Il est trivial de voir que chaque symétrie syntaxique est une symétrie sémantique mais, en général, l'inverse n'est pas vérifié.

En revanche, Krishnamurthy a introduit la règle de la symétrie ci-dessous pour augmenter le système de preuves par résolution.

Proposition 1 *Si L est la logique propositionnelle, A un ensemble de formules de L, B une formule de L et σ une symétrie syntaxique de A, alors la règle de la symétrie peut être définie comme suit :*

$$\frac{A \vdash B}{A \vdash \sigma(B)}$$

Plusieurs problèmes difficiles de la résolution ont été démontrés polynomiaux en utilisant la symétrie dans la résolution. Nous verrons dans la section 4 comment étendre cette règle à des logiques non-monotones.

Maintenant, nous présentons la contribution principale de ce travail qui consiste à étendre la symétrie aux logiques non-monotones.

1. L'ensemble complet des littéraux de F est un ensemble qui contient toutes les variables de F et leurs négations.

3 Symétrie dans la logique préférentielle

Ici, nous étendons la notion de la symétrie dans le cadre des logiques préférentielles.

Par exemple, il est admis qu'un étudiant normal est jeune. Ainsi, si l'on sait que John est un étudiant, on en conclut assez naturellement que John est jeune. Si on apprend par la suite que John est âgé de cinquante ans, la conclusion "John est jeune" sera révisée.

Il est donc important de considérer que les étudiants normaux appartiennent à la même classe, car ils sont tous symétriques par rapport à la propriété *normal*.

Au départ, le plus simple est de partir d'une approche préférentielle, telle qu'il a été initié par Bossu-Siegel [9, 10], repris aussi par Shoam [25] et Besnard-Siegel [8], puis par Kraus, Lehmann et Magidor dans [18]. Toutes ces approches sont construites sur une logique classique (calcul propositionnel, calcul des prédicats, logique modale), où la sémantique de l'inférence a été donnée par "une formule A implique une formule B si chaque modèle de A est un modèle de B". Cependant, une approche préférentielle, dans sa forme la plus générale, dit "A implique B si tous les modèles préférés de A sont des modèles de B". Les modèles préférés de A sont des modèles qui ont des propriétés utiles pour la gestion des exceptions. Ce concept de préférence peut être défini par une relation de préordre (une relation transitive et réflexive) sur les interprétations. Les modèles préférés étant les modèles minimaux pour cette relation. Pour notre exemple élémentaire, si I et J sont des interprétations et l'information pertinente est "jeune", alors la relation de préordre peut être définie par : $I \prec J$ si et seulement si toute personne jeune en J est jeune dans I.

Définition 3 *Soient L une logique classique et F l'ensemble des formules de L. Si A est un sous-ensemble de formules (ou une formule) de F, alors $\bar{A}$ est l'ensemble des formules logiquement impliquées par A. L'ensemble des formules A est déductivement clos si $A = \bar{A}$.*

Définition 4 *Une relation préférentielle $\prec$ est une relation de préordre (transitive et réflexive) sur les interprétations. Par ailleurs, si la relation $\prec$ est antisymétrique, alors $\prec$ devient un ordre.*

Intuitivement, on peut considérer les étudiants qui ne sont pas jeunes comme des exceptions (étudiants anormaux). Par conséquent, $I \prec J$ si l'ensemble des exceptions de I est inclus dans l'ensemble des exceptions de J.

Définition 5 *Si A est un ensemble de formules, un modèle minimal M de A est une interprétation qui satisfait A (ie $M \vdash A$) et qui est minimale par rapport à la relation $\prec$ définie sur l'ensemble des modèles de A. Autrement dit, si M' est un modèle de A tel que $M' \prec M$, alors $M \prec M'$ (ou de manière équivalente, $M' = M$ si $\prec$ est antisymétrique).*

Définition 6 *De façon classique, si $\prec$ est une relation préférentielle, on définit l'inférence logique de modèles préférentiels $\vdash_\prec$ comme suit : $A \vdash_\prec B$ si et seulement si chaque modèle minimal de A est un modèle de B.*

Nous pouvons trouver dans [26] la proposition suivante :

Proposition 2 *Si un langage L a un ensemble fini de variables, alors chacune de ses formules consistantes F a au moins un modèle minimal, et pour chaque modèle de M de F, il existe un modèle minimal M tel que $M' \prec M$.*

Remarque 1 *La logique propositionnelle satisfait les conditions de la Proposition 2. Chaque formule propositionnelle cohérente F admet au moins un modèle minimal, et pour chaque modèle de M de F, il existe un modèle minimal M' de F tel que $M' \prec M$.*

Exemple 1 *Pour représenter les énoncés "En général, les étudiants sont jeunes" et "Lea est un étudiant", nous pouvons utiliser une approche préférentielle, close pour la circonscription [20]. Un prédicat supplémentaire "anormale" est ajouté et notre premier énoncé est traduit en "Un étudiant qui n'est pas anormal est jeune". Maintenant, si les prédicats St, Ab, et Yo, indique respectivement "étudiants", "anormale" et "jeune", alors, dans la logique de premier ordre, on obtient l'ensemble des formules suivant :*

$$A \equiv \{St(Lea), \forall x(St(x) \wedge \neg Ab(x)) \rightarrow Yo(x)\}$$

Par instanciation de la variable x par la constante Lea, nous traduisons l'ensemble des formules en logique propositionnelle et on obtient l'ensemble suivant de formules :

$$A \equiv \{St(Lea), (St(Lea) \wedge \neg Ab(Lea) \rightarrow Yo(Lea)\}$$

Ainsi :

$$A \equiv \{St(Lea), Ab(Lea) \vee Yo(Lea)\}$$

L'ensemble A possède huit interprétations, parmi aux, les trois suivantes sont des modèles :

$$\begin{aligned} M_1 &= \{St(Lea), Ab(Lea), Yo(Lea)\} \\ M_2 &= \{St(Lea), \neg Ab(Lea), Yo(Lea)\} \\ M_3 &= \{St(Lea), Ab(Lea), \neg Yo(Lea)\} \end{aligned}$$

Dans une logique classique, il est impossible de déduire de A que Lea est jeune. En effet, A possède deux modèles dans lesquels Lea est jeune (M_1 et M_2), et des modèles dans lesquels Lea n'est pas jeune, en particulier ceux où Lea est anormal (M_3). Pour obtenir le résultat "Lea est jeune", on préférera les modèles qui ont moins d'étudiants anormaux. Ainsi, dans une approche de modèles préférentiels, la relation $\prec$ peut être définie comme : $M \prec M'$ si et seulement si "chaque individu qui est anormal dans M est anormal dans M'". Selon cette relation, nous obtenons les préférences suivantes entre les modèles de A : $M_1 \prec M_3$, $M_3 \prec M_1$, $M_2 \prec M_1$, $M_2 \prec M_3$. Par conséquent, A n'a qu'un seul modèle minimal qui est M_2, et

Lea est jeune dans ce modèle. On peut donc en déduire que Lea est jeune dans cette approche préférentielle. Il est alors important d'en déduire toutes les symétriques du littérale $Yo(Lea)$ par rapport à cette relation de préférence.

3.1 Symétrie

Maintenant, on étend la définition de la symétrie sémantique à la logique de modèles préférentiels et on montre comment les littéraux peuvent être symétriques dans cette logique non-classique, mais pas symétrique dans une logique classique.

Définition 7 (Symétrie préférentielle sémantique) *Si $\vdash_\prec$ est une inférence de modèles préférentiels, A est un ensemble de formules et σ une permutation définie sur les littéraux de A, alors σ est une symétrie de A, si et seulement si A et $\sigma(A)$ ont le même ensemble de modèles minimaux.*

Définition 8 *Deux littéraux ℓ et ℓ' sont symétriques dans A si et seulement si il existe une symétrie préférentielle sémantique σ de A tel que $\sigma(\ell) = \ell'$.*

Exemple 2 *Si on prend l'exemple précédent et on ajoute le fait que "John est un étudiant", alors on obtient la formule suivante : $A' = \{St(Lea), St(John), Ab(Lea) \vee Yo(Lea), Ab(John) \vee Yo(John)\}$ qui admet neuf modèles. Il est facile de voir que les personnes Lea et $John$ sont symétriques dans les deux logiques : classique et préférentielle. Cela dans le sens que chaque littéral, où Lea apparaît, est symétrique au littéral où l'on remplace $John$ par Lea. Si nous considérons les "anormaux" comme une information pertinente sur laquelle la préférence et la permutation $\sigma = (St(Lea), St(John))(Ab(Lea), Ab(John))(Yo(Lea), Yo(John))$ sont fondées, alors on peut facilement voir que σ est une symétrie préférentielle sémantique de la formule A'. En effet, il existe un modèle minimal qui est $M = \{St(Lea), St(John), \neg Ab(Lea), \neg Ab(John), Yo(Lea), Yo(John)\}$ qui est conservé par σ. Nous pouvons voir que les littéraux $Yo(Lea)$ et $Yo(John)$ sont symétriques ainsi que $Ab(Lea)$ et $Ab(John)$.*

Maintenant, si l'on ajoute à A' l'information "John n'est pas anormal", alors $Yo(Lea)$ et $Yo(John)$ ainsi que $\neg Ab(Lea)$ et $\neg Ab(John)$ restent symétriques deux à deux dans la logique préférentielle. Puisque M reste le seul modèle minimal de la théorie où tous les littéraux sont vraies. Cependant, les littéraux $Yo(Lea)$ et $Yo(John)$ ne sont pas symétriques dans la logique classique. En effet, la nouvelle formule contient les trois modèles étendus suivants :

$$\begin{aligned} M_1 &= \{St(Lea), St(John), Ab(Lea), Yo(Lea), \\ &\quad \neg Ab(John), Yo(John)\} \\ M_2 &= \{St(Lea), St(John), \neg Ab(Lea), Yo(Lea), \\ &\quad \neg Ab(John), Yo(John)\} \\ M_3 &= \{St(Lea), St(John), Ab(Lea), \neg Yo(Lea), \\ &\quad \neg Ab(John), Yo(John)\} \end{aligned}$$

où ces littéraux ne sont pas symétriques. Nous pouvons voir par exemple que le modèle M_3 ne reste pas un modèle si l'on permute $Yo(Lea)$ et $Yo(John)$.

Il est alors important de voir que certains littéraux pourraient être symétriques dans une approche préférentielle, mais non symétriques dans une logique classique. Cette idée est nouvelle et prometteuse pour le raisonnement en utilisant la symétrie dans des logiques non classiques.

Dans la section suivante, nous essayons d'étendre la notion de la symétrie syntaxique à des logiques non-classiques. Pour ce faire, nous avons choisi la X-logique [26] comme un cadre de référence.

4 Symétrie dans la X-Logique

Nous avons vu qu'il est facile d'étendre la notion de la symétrie sémantique aux logiques de modèles préférentiels, mais la définition de la symétrie syntaxique dans ces logiques semble ne pas être triviale. Pour ce faire, nous allons utiliser la X-logique [26] qui semble avoir quelques propriétés syntaxiques importantes que nous allons utiliser pour étendre la notion de la symétrie syntaxique aux logiques non-classiques.

Définition 9 (X-logique) *Soit X un ensemble de formules de la logique propositionnelle L (X n'est pas nécessairement déductivement clos). La relation d'inférence non-monotones $\vdash_X$ est définie par $A \vdash_X B$ si et seulement si $(\overline{A \cup B}) \cap X \subseteq \bar{A} \cap X$.*

Remarque 2
- *En d'autres termes, $A \vdash_X B$ si tout théorème de $(\overline{A \cup B})$, ce qui est dans X, est aussi un théorème de $(\overline{A})$. Cela veut dire, en ajoutant la connaissance B à A, l'ensemble des théorèmes qui sont en X n'augmente pas. Comme l'inférence logique classique $\vdash$ est monotone, alors nous avons également $\bar{A} \cap X \subseteq (\overline{A \cup B}) \cap X$, et par conséquent, il est possible de définir l'inférence dans la X-logique par $A \vdash_X B$ si et seulement si $(\overline{A \cup B}) \cap X = \bar{A} \cap X$.*
- *Pour le cas particulier où $X = F$ (l'ensemble de toutes les formules possibles de la logique L), l'inférence $\vdash_X$ est identique à l'inférence classique $\vdash$. En revanche, si X est vide, alors toute formule B peut être déduite par A.*

L'ensemble X peut être considérée comme un potentiomètre qui règle l'inférence. Intuitivement, si une formule

A encode une information (des connaissances, ou de certaines croyances ..), X peut être considéré comme l'ensemble des informations "pertinentes". L'ensemble A implique un ensemble d'information B, pour l'inférence dans la X-logique, si l'ajout de B à A ne produit pas de formules plus pertinentes que celles produites par A seul.

4.1 Symétrie

Maintenant, nous allons traiter la symétrie dans les X-logiques. Nous étendrons la définition de la symétrie syntaxique au cadre de la X-logique et donnerons une règle étendue de la symétrie qui peut être utilisée pour faire des courtes démonstrations en utilisant des formules symétriques dans ce cadre.

Définition 10 *Soient A un ensemble de formules de la logique propositionnelle, X le sous-ensemble de formules pertinentes sur lequel l'inférence $\vdash_X$ de la X-logique est construite et σ une permutation de littéraux. La permutation σ est une symétrie syntaxique de A dans la X-logique considérée, si les conditions suivantes sont vérifiées :*

1. $\sigma(A) = A$,
2. $\sigma(X) = X$

Maintenant, nous étendons la règle de la symétrie de Krishnamurthy au cadre de la X-logique.

Proposition 3 *Soient A et B deux formules ou deux ensembles de formules de la logique classique L et σ une symétrie syntaxique de A dans la X-logique considérée. Nous avons la règle suivante :*

$$\frac{A \vdash_X B}{A \vdash_X \sigma(B)}$$

***Preuve* 1** *Pour prouver que $A \vdash_X \sigma(B)$ nous allons prouver que $\overline{A \cup \sigma(B)} \cap X = \bar{A} \cap X$. Nous avons l'hypothèse $A \vdash_X B$, d'où on a $(\overline{A \cup B}) \cap X = \bar{A} \cap X$. Puisque σ est une symétrie syntaxique de A dans la X-logique, alors il préserve les théorèmes de la logique propositionnelle. Ainsi, nous avons $\sigma(\overline{A \cup B} \cap X) = \sigma(\bar{A} \cap X)$ ce qui équivaut à $\sigma(\overline{A \cup B}) \cap \sigma(X) = \sigma(\bar{A}) \cap \sigma(X)$. Cela donne $\overline{\sigma(A \cup B)} \cap \sigma(X) = \overline{\sigma(A)} \cap \sigma(X)$ ce qui équivaut aussi à $\overline{\sigma(A)} \cup \sigma(B) \cap \sigma(X) = \overline{\sigma(A)} \cap \sigma(X)$. Comme A et X sont invariantes par σ, alors on en déduit $\overline{A \cup \sigma(B)} \cap X = \bar{A} \cap \sigma(X)$. Par conséquent, $A \vdash_X \sigma(B)$.*

Exemple 3 *Prenons l'exemple des étudiants codés par l'ensemble de formules suivant :* $A' = \{St(Lea), St(John), Ab(Lea) \vee Yo(Lea), Ab(John) \vee Yo(John)\}$. *Considérons l'ensemble* $X = \{\neg Ab(Lea), \neg Ab(John)\}$ *et la permutation :* $\sigma = (St(Lea), St(John))(Ab(Lea), Ab(John))(Yo(Lea), Yo(John))$. *La permutation σ est une symétrie de la X-logique de A'. Nous avons $A' \vdash_X Yo(Lea)$, et par la symétrie, nous avons $A' \vdash_X Yo(John)$ puisque $\sigma(Yo(Lea)) = Yo(John)$.*

Cette règle pout être utilisée pour déduire toutes les formules symétriques d'une formule inférée dans la X-logique considérée. Sa mise en oeuvre dans un démonstrateur permettra de raccourcir les preuves d'un théorème.

Dans la prochaine section, nous étendons la symétrie à une logique plus générale, connue comme une logique non-monotone, qui est la logique des défauts introduit par Reiter [23].

5 Symétrie dans la logique des défauts

Nous étudions dans cette section la notion de la symétrie dans les logiques des défauts. Une logique des défauts est une logique non-monotone qui est introduite par Reiter [23] pour formaliser le raisonnement par des hypothèses de défaut. Avant de traiter la symétrie dans cette logique, nous introduisons cetaines notions préliminaires.

5.1 Préliminaires

Définition 11 *Une théorie des défauts T est une paire $\langle D, W \rangle$ où W est un ensemble de formules logiques de premier ordre, appelé* la théorie de base*, qui formalisent les faits qui sont connus avec certitude. D est un ensemble de* règles de défaut*, chacune étant de la forme :*

$$\frac{Prerequis : Justification_1, \ldots, Justification_n}{Conclusion}$$

Intuitivement, cela signifie que par défaut, si le $Prerequis$ est vrai, et chaque $Justification_i$ pour tout $i \in \{1, \ldots, n\}$ est consistante avec nos croyances actuelles, alors nous sommes amenés à croire que $Conclusion$ est vraie, et on l'infère (on l'ajoute à la théorie). Pour définir formellement le sens d'une règle de défaut, nous avons besoin d'introduire avant cela, la notion importante d'extensions dans une logique des défauts.

Une extension d'une théorie des défauts $T = \langle D, W \rangle$ est un ensemble de formules déductivement clos E contenant W et qui vérifie : si $\frac{\alpha : \beta_1, \ldots, \beta_n}{\gamma} \in D$ est un défaut tel que $\alpha \in E$ et $\forall i \in \{1, \ldots, n\}, \neg\beta_i \notin E$, alors $\gamma \in E$. Formellement :

Définition 12 *Si $Th(E)$ est l'ensemble des conséquences logiques de l'ensemble des formules E, alors E est une extension de la théorie des défauts $T = \langle D, W \rangle$ si et seulement si $E = \bigcup_{i=0,\ldots,\infty} E_i$ et les conditions suivantes sont vérifiées :* $E_0 = W$, *et*
$\forall i \geq 0, E_{i+1} = Th(E_i) \cup \{\gamma : \frac{\alpha : \beta_1, \ldots, \beta_n}{\gamma} \in D, \alpha \in E_i, \forall j \in \{1, \ldots, n\}, \neg\beta_j \notin E\}$

Une règle de défaut $\frac{\alpha:\beta_1,\ldots,\beta_n}{\gamma}$ peut être appliquée à une théorie donnée des défauts $T = \langle D, W\rangle$ si son prérequis α est dans W, et la négation de chacun de ses justifications $\neg\beta_j$ n'est pas dans l'extension E.

Remarque 3 *1. Lorsque tous les défauts de la théorie sont normaux (i.e. sous la forme $\frac{\alpha:\beta}{\beta}$), la condition de la justification $\neg\beta \notin E$ est simplifiée en $\neg\beta \notin E_i$. Nous obtenons, alors, une méthode plus simple et constructif pour construire l'extension E.*

2. *Les règles de défaut peuvent être appliquées dans un ordre différent et cela peut conduire à différentes extensions E pour la même théorie T.*
3. *Si un défaut contient des formules avec des variables libres, il est considéré comme représentant de l'ensemble de toutes les défauts obtenus en donnant une valeur à toutes ces variables.*

Exemple 4 *Prenons par exemple la règle de défaut "généralement, les oiseaux volent "qui est formalisé par le défaut suivant : $D = \left\{\frac{Bird(X):Flies(X)}{Flies(X)}\right\}$. Cette règle signifie que, si X est un oiseau et si l'on peut supposer qu'il vole, alors nous pouvons conclure qu'il vole. Une théorie de base contenant des faits sur les oiseaux est la suivante : $W = \{Bird(Condor),\ Bird(Penguin),\ \neg Flies(Penguin),\ Flies(Eagle)\}$. On obtient alors la théorie des défauts $T = \langle D, W\rangle$.*

Selon cette règle de défaut, un condor vole parce que $Bird(Condor)$ est vrai et la justification $Flies(Condor)$ n'est pas incompatible avec ce qui est actuellement connu. Au contraire, nous savons qu'un pingouin ne vole pas ($\neg Flies(Penguin)$). Ainsi, $Bird(Penguin)$ ne permet pas de conclure $Flies(Penguin)$, même si la prérequis $Bird(Penguin)$ est vrai, car la justification $Flies(Penguin)$ est incompatible avec ce qui est connu. Par conséquent, nous obtenons une extension unique pour la théorie T, ce qui est :
$E = \{Bird(Condor), Bird(Penguin), \neg Flies(Penguin), Flies(Eagle), Flies(Condor)\}$.

La notion la plus importante dans une logique des défauts est le calcul des extensions d'une théorie des défauts. Selon cet ensemble d'extensions, les chercheurs ont défini des sémantiques différentes pour la logique des défauts. L'implication d'une formule depuis d'une théorie des défauts $T = \langle D, W\rangle$ peut être défini en différentes façons :

Définition 13 *étant donné une théorie des défauts $T = \langle D, W\rangle$ et l'ensemble de tous ses extensions E_T.*

- **Approche sceptique :** *une formule f est impliqué de la théorie des défauts T si elle est impliquée de tous ses extensions, c'est-à-dire $T \vdash_S f$ si et seulement si $\forall E \in E_T, E \vdash f$.*
- **Approche crédule :** *une formule f est impliqué de la théorie des défauts T si elle est impliquée au moins d'une de ses extensions, c'est-à-dire $T \vdash_C f$ si et seulement si $\exists E \in E_T : E \vdash f$.*
- **Approche semi-crédule :** *une formule f est impliqué de la théorie des défauts T si elle est impliquée au moins d'une de ses extensions et tous ces extensions n'implique pas sa négation, c'est-à-dire $T \vdash_{SC} f$ si et seulement si $\exists E \in E_T : E \vdash f$ et $\forall E \in E_T, E \nvdash \neg f$.*

5.2 Symétrie

Maintenant, nous introduisons la notion de la symétrie dans la logique des défauts et on montre comment l'implication est améliorée par la propriété de la symétrie. Pour cette logique, nous distinguons deux niveaux de symétrie : la symétrie sémantique et la symétrie syntaxique. On va définir dans ce qui suit les deux symétries et on va étudier leur relation.

Définition 14 (Symétrie sémantique) *étant donné une théorie des défauts $T = \langle D, W\rangle$, L_T est l'ensemble de ses littéraux et E_T l'ensemble de toutes ses extensions. Une symétrie sémantique σ est une permutation de littéraux définie sur L_T telle que $E_T = E_{\sigma(T)}$.*

En d'autres termes, une symétrie sémantique d'une théorie des défauts T est une permutation de variables qui laisse invariant l'ensemble de ses extensions. Il résulte de là que chaque extension $E_i \in E_T$ est transformé par la symétrie σ à une autre extension $E_j = \sigma(E_i)$. Ces extensions sont ce que nous appelons *extensions symétriques*. Il est alors possible d'obtenir une famille des extensions symétriques, sans duplication des efforts, si nous savons que E_i est une extension et nous avons le groupe de symétrie de la théorie T. Malheureusement, le calcul de la symétrie sémantique est coûteux au niveau de temps, car il a besoin de calculer toutes les extensions. Dans ce qui suit, nous allons définir la symétrie syntaxique et on montre qu'il y a une condition suffisante pour cette symétrie et il peut être calculer d'une manière efficace.

Définition 15 (Symétrie syntaxique) *étant donné une théorie des défauts $T = \langle D, W\rangle$. Une symétrie syntaxique est une permutation σ définie sur l'ensemble L_T de littéraux de T, qui laisse la théorie T invariant. C'est-à-dire $\sigma(T) = T$, plus précisément, les conditions suivantes sont vérifiées : $\sigma(D) = D$ et $\sigma(W) = W$.*

Exemple 5 *Prenons l'exemple des étudiants discuté avant et considérer la théorie des défauts suivante $T = (\{\frac{St(X):\neg Ab(X)}{Yo(X)}\}, \{St(Lea), St(John)\})$. La permutation $\sigma = (St(Lea), St(John))(Ab(Lea), Ab(John))$ $(Yo(Lea), Yo(John))$ est une symétrie syntaxique, car*

il laisse invariant T. En considérant toutes les instanciations terminal des variables libres des défauts de la théorie T, nous pouvons voir que T a une extension $E = \{St(Lea), St(John), Yo(Lea), Yo(John)\}$ qui reste invariant sous σ. On peut voir que σ est aussi une symétrie sémantique.

Maintenant, si nous ajoutons $Yo(John)$ à l'ensemble des faits de la théorie T de l'exemple précédent, nous obtenons une nouvelle théorie T' pour laquelle σ n'est pas une symétrie syntaxique. Cependant, σ reste une symétrie sémantique de T, puisque la nouvelle théorie T' a la même extension E comme T. Cet exemple illustre que la symétrie sémantique contient la symétrie syntaxique.

Nous donnons dans le théorème suivant la relation entre la symétrie syntaxique et la symétrie sémantique d'une logique des défauts.

Théorème 1 *étant donné une théorie des défauts T. Si σ est une symétrie syntaxique de T, alors σ est une symétrie sémantique de T.*

***Preuve* 2** *La preuve est triviale. Puisque σ est une symétrie syntaxique, alors $\sigma(T) = T$. Donc, $E_T = E_{\sigma(T)}$.*

Maintenant, nous pouvons introduire la nouvelle règle d'inférence par symétrie dans les logiques des défauts. Prenons, par exemple, l'inférence sceptique $\vdash_S$.

Proposition 4 *Soient T une théorie des défauts, f une formule et σ une symétrie syntaxique de T, alors on a la règle suivante :*

$$\frac{T \vdash_S f}{T \vdash_S \sigma(f)}$$

***Preuve* 3** *Pour montrer que $T \vdash_S \sigma(f)$, on a besoin de prouver que $\forall E_k \in E_T, E_k \vdash \sigma(f)$. Soit $E_i \in E_T$, par la définition de la symétrie σ, il existe $E_j \in E_T$ telle que $E_i = \sigma(E_j)$. Par hypothèse, nous avons $T \vdash_S f$. Ainsi $E_j \vdash f$, alors $\sigma(E_j) \vdash \sigma(f)$. Par conséquent, $E_i \vdash \sigma(f)$. Nous concluons que $T \vdash_S \sigma(f)$.*

Remarque 4 *La règle précédente est également valable pour les crédules ($\vdash_C$) et les demi-crédules ($\vdash_{SC}$) inférences.*

Exemple 6 *Prenons l'exemple 5. Dans la théorie T, nous avons $T \vdash_S Yo(Lea)$. En impliquant la règle d'inférence par symétrie, nous pouvons faire l'implication $T \vdash_S \sigma(Yo(Lea))$. Ainsi, $T \vdash_S Yo(John)$.*

Maintenant, nous donnons une proposition importante qui utilise la symétrie pour calculer l'ensemble des extensions.

Proposition 5 *étant donné une théorie des défauts $T = \langle D, W\rangle$, un sous-ensemble de formules E et une symétrie syntaxique σ de T, alors E est une extension de T si et seulement si $\sigma(E)$ est une extension de T.*

***Preuve* 4** *Supposons que E est une extension. Comme la permutation σ est une symétrie syntaxique, alors, par le théorème 1, elle est une symétrie sémantique. Elle conserve alors l'ensemble E_T des extensions de T. Il résulte de là que $\sigma(E) \in E_T$. On prouve l'inverse de la même manière en considérant la symétrie inverse σ^{-1} of σ.*

Maintenant, nous discutons la relation entre les extensions symétriques d'une théorie des défauts et leurs sous-ensembles des défauts qui sont utilisés pour les construire.

Proposition 6 *étant donné une théorie des défauts $T = \langle D, W\rangle$, un sous-ensemble $D_1 \subset D$ et une symétrie syntaxique σ, alors il existe une extension E^{D_1} de T obtenue par l'application des défauts de D_1, si et seulement si, il existe une extension $E^{\sigma(D_1)}$ de T obtenue par l'application des défauts de $\sigma(D_1)$.*

***Preuve* 5** *Supposons que E^{D_1} est une extension de $T = \langle D, W\rangle$ obtenue par l'application des défauts de D_1. Nous allons montrer qu'il existe une extension $E^{\sigma(D_1)}$ de T qui peut être obtenue par l'application des défauts de $\sigma(D_1)$. Puisque E^{D_1} est une extension de T et à partir de la définition 12, nous avons ce qui suit : $E^{D_1} = \bigcup_{i=0,\ldots,\infty} E_i$ où : $E_0 = W$ et $\forall i \geq 0, E_{i+1} = Th(E_i) \cup \{\gamma : \frac{\alpha:\beta_1,\ldots,\beta_n}{\gamma} \in D_1, \alpha \in E_i, \forall j \in \{1,\ldots,n\}, \neg\beta_j \notin E^{D_1}\}$. Pour montrer l'existence de l'extension $E^{\sigma(D_1)}$ de T, on met $E' = \sigma(E^{D_1})$, $E'_0 = W$ et $\forall i \geq 0$, $E'_{i+1} = Th(E'_i) \cup \{\sigma(\gamma) : \frac{\sigma(\alpha):\sigma(\beta_1),\ldots,\sigma(\beta_n)}{\sigma(\gamma)} \in \sigma(D_1), \sigma(\alpha) \in E'_i, \forall j \in \{1,\ldots,n\}, \neg\sigma(\beta_j) \notin E'\}$ et on va prouver que E' est une extension de T et $E' = E^{\sigma(D_1)}$. Pour montrer que E' est une extension de T, nous devons montrer que $E' = \bigcup_{i=0,\ldots,\infty} E'_i$.*

Pour ce faire, nous devons d'abord montrer que $\forall i \geq 0$, $E'_i = \sigma(E_i)$. Nous démontrons cette propriété par récurrence sur i. Pour la première étape ($i = 0$), nous avons $E'_0 = W$, donc $E'_0 = \sigma(W)$ car σ est une symétrie syntaxique de T. Il en résulte que $E'_0 = \sigma(E_0)$. Maintenant, nous supposons que la propriété est vérifiée jusqu'à l'étape i, c'est-à-dire $E'_i = \sigma(E_i)$ et nous allons montrer que cette propriété est vérifiée à l'étape $i+1$, c'est-à-dire $E'_{i+1} = \sigma(E_{i+1})$. Par la définition de E'_{i+1}, nous avons $E'_{i+1} = Th(E'_i) \cup \{\sigma(\gamma) : \frac{\sigma(\alpha):\sigma(\beta_1),\ldots,\sigma(\beta_n)}{\sigma(\gamma)} \in \sigma(D_1), \sigma(\alpha) \in E'_i, \forall j \in \{1,\ldots,n\}, \neg\sigma(\beta_j) \notin E'\}$. Par l'hypothèse de récurrence et la définition de E', on peut réécrire E'_{i+1} comme : $E'_{i+1} = Th(\sigma(E_i)) \cup \{\sigma(\gamma) : \frac{\sigma(\alpha):\sigma(\beta_1),\ldots,\sigma(\beta_n)}{\sigma(\gamma)} \in \sigma(D_1), \sigma(\alpha) \in \sigma(E_i), \forall j \in \{1,\ldots,n\}, \neg\sigma(\beta_j) \notin \sigma(E^{D_1})\}$. En revanche, nous avons $\sigma(Th(E_i)) = Th(\sigma(E_i))$ puisque la symétrie σ vérifie le suivant : si $A \vdash B$, alors $\sigma(A) \vdash \sigma(B)$. Par conséquent, $E'_{i+1} = \sigma(Th(E_i)) \cup \sigma(\{\gamma : \frac{\alpha:\beta_1,\ldots,\beta_n}{\gamma} \in D_1, \alpha \in E_i, \forall j \in \{1,\ldots,n\}, \neg\beta_j \notin E^{D_1}\})$.

On en déduit que $E'_{i+1} = \sigma(Th(E_i) \cup \{\gamma : \frac{\alpha:\beta_1,\ldots,\beta_n}{\gamma} \in D_1, \alpha \in E_i, \forall j \in \{1,\ldots,n\}, \neg\beta_j \notin E^{D_1}\})$. Cela im-

plique que $E'_{i+1} = \sigma(E_{i+1})$ et alors, on a prouvé la propriété.

Maintenant, nous allons prouver que E' est une extension de T. Par définition, $E' = \sigma(E^{D_1})$, comme E^{D_1} est, par l'hypothèse, une extension de T ($E^{D_1} = \bigcup_{i=0,\ldots,\infty} E_i$), alors $E' = \sigma(\bigcup_{i=0,\ldots,\infty} E_i)$, donc $E' = \bigcup_{i=0,\ldots,\infty} \sigma(E_i)$. En revanche nous avons montré que $\forall i \geq 0$, $E'_i = \sigma(E_i)$, thus $E' = \bigcup_{i=0,\ldots,\infty} E'_i$, il en résulte que E' est une extension de T. Comme E' est une extension de T obtenue par l'application des défauts de $\sigma(D_1)$, on obtient $E' = E^{\sigma(D_1)}$. Enfin, nous concluons qu'il existe une extension $E^{\sigma(D_1)}$ de T qui peut être obtenue par l'application des défauts de $\sigma(D_1)$.

L'inverse peut être établi de la même manière en considérant la symétrie inverse σ^{-1} de σ.

Maintenant, nous allons montrer que si l'application d'un défaut conduit à une extension de la théorie considérée, alors l'application de chacun de ses défauts symétriques conduit aussi à une extension de la même théorie.

Proposition 7 *Si $T = \langle D, W\rangle$ est une théorie des défauts et σ un de ses symétries syntaxiques. Il existe une extension de T où le défaut d est appliqué, si et seulement si, il existe une extension où le défaut symétrique $\sigma(d)$ est appliqué.*

***Preuve* 6** *Soit E une extension de T où d est appliqué. Cela implique qu'il existe un sous-ensemble $D_1 \subset D$ tel que $d \in D_1$ et $E = E^{D_1}$. Par la proposition 6, on en déduit que $E^{\sigma(D_1)}$ est une extension de T, où $\sigma(d) \in \sigma(D_1)$ est appliqué.*

De la proposition précédente, on déduit la propriété suivante :

Corollaire 1 *étant donné une théorie des défauts $T = \langle D, W\rangle$, un défaut $d \in D$ et une symétrie syntaxique σ de T, alors il n'existe pas une extension de T où le défaut d est appliqué, si et seulement si, il n'existe pas une extension où le défaut symétrique $\sigma(d)$ est appliqué.*

Dans la logique des défauts, nous avons l'avantage, car les deux symétries : syntaxiques et sémantiques, sont définies. La logique des défauts est augmentée avec une règle d'inférence par symétrie qui peut être utilisée pour raccourcir les preuves. Par ailleurs, dans la logique des défauts, la symétrie peut être utilisée pour calculer l'ensemble des extensions. En effet, lorsque l'extension est identifiée lors de l'énumération, on peut déduire toutes ses extensions symétriques, sans effort supplémentaire (Proposition 5). Autrement dit, les branches isomorphes dans l'arbre de recherche correspondant à l'application des sous-ensembles symétriques des défauts conduit à des extensions symétriques, alors nous avons besoin d'explorer une seule branche et élaguer l'espace de recherche correspondant aux autres. En cas d'échec à obtenir une extension lors de l'application de défaut d, le corollaire 1 nous permet de tailler tous les sous-espaces de recherche correspondant aux défaut symétrique $\sigma(d)$, car ils ne conduisent pas à des extensions.

6 Conclusion et perspectives

Le But principal de ces travaux est d'étendre la notion de la symétrie aux formalismes logiques non-classiques. Nous avons défini la notion de la symétrie sémantique dans les logiques préférentielles et avons montré comment certaine information peut être déduite à l'aide de cette définition, alors que ces déductions ne peuvent pas être faites dans une logique classique. L'autre point étudié ici est l'extension de la définition de la symétrie syntaxique dans le cadre de la X-logique où une nouvelle règle de déduction a été introduite. Enfin, nous avons défini à la fois, la symétrie sémantique et syntaxique dans le cadre le plus général de la logique des défauts. Nous avons montré comment la symétrie peut être utilisée pour améliorer la recherche des extensions et nous avons introduit une nouvelle règle d'inférence par symétrie qui peut être utilisée pour faire de courtes démonstrations. En perspectives, on peut essayer d'étudier en profondeur la relation entre la symétrie sémantique en logique préférentielle et la symétrie syntaxique définie dans les X-logiques et d'établir leur relation, en utilisant les définitions de la symétrie introduites dans les logiques des défauts. Un autre point que nous voulons étudier est d'inclure la symétrie dans des algorithmes d'énumération des extensions de la logique des défauts pour les accélérer, en utilisant l'avantage de la nouvelle règle de la symétrie. Enfin, nous cherchons à étudier la relation entre la logique des défauts et les ASPs (Answer Set Programming) afin de calculer les extensions symétriques par des ASs symétriques et des modèles stables.

Références

[1] F. A. Aloul, A.Ramani, I. L. Markov, and K. A. Sakallak. Solving difficult sat instances in the presence of symmetry. *In IEEE Transaction on CAD, vol. 22(9)*, pages 1117–1137, 2003.

[2] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallak. Symmetry breaking for pseudo-boolean satisfiabilty. *In ASPDAC'04*, pages 884–887, 2004.

[3] R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *Principle and Practice of Constraint Programming - CP'99*, 1999.

[4] B. Benhamou. Study of symmetry in constraint satisfaction problems. *In Proceedings of the 2nd International workshop on Principles and Practice of Constraint Programming - PPCP'94*, 1994.

[5] B. Benhamou and L. Sais. Theoretical study of symmetries in propositional calculus and application. *Ele-*

venth International Conference on Automated Deduction, Saratoga Springs,NY, USA, 1992.

[6] B. Benhamou and L. Sais. Tractability through symmetries in propositional calculus. *Journal of Automated Reasoning (JAR),*, 12 :89–102, 1994.

[7] B. Benhamou, L. Sais, and P. Siegel. Two proof procedures for a cardinality based language. *in proceedings of STACS'94, Caen France*, pages 71–82, 1994.

[8] P. Besnard and P. Siegel. The preferential-models approach in nonmonotonic logics - in non-standard logic for automated reasoning. In *Academic Press*, pages 137–156. ed. P. Smets, 1988.

[9] Genevieve Bossu and Pierre Siegel. Nonmonotonic reasoning and databases. In *Advances in Data Base Theory*, pages 239–284, 1982.

[10] Genevieve Bossu and Pierre Siegel. Saturation, nonmonotonic reasoning and the closed-world assumption. *Artif. Intell.*, 25(1) :13–63, 1985.

[11] James Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *KR'96 : Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, San Francisco, California, 1996.

[12] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *International conference on constraint programming*, volume 2239 of *LNCS*, pages 93–108. Springer Verlag, 2001.

[13] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *International conference on constraint programming*, volume 2239 of *LNCS*, pages 77–82. Springer Verlag, 2001.

[14] E.C. Freuder. Eliminating interchangeable values in constraints satisfaction problems. *Proc AAAI-91*, pages 227–233, 1991.

[15] I. P. Gent, W. Hervey, T. Kesley, and S. Linton. Generic sbdd using computational group theory. In *Proceedings CP'2003*, 2003.

[16] I. P. Gent and B. M. Smith. Symmetry breaking during search in constraint programming. In *Proceedings ECAI'2000*, 2000.

[17] I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints : Symmetry breaking during search. In *International conference on constraint programming*, volume 2470 of *LNCS*, pages 415–430. Springer Verlag, 2002.

[18] Sarit Kraus, Daniel J. Lehmann, and Menachem Magidor. Nonmonotonic reasoning, preferential models and cumulative logics. *Artificial Intelligence*, 44(1-2) :167–207, 1990.

[19] B. Krishnamurty. Short proofs for tricky formulas. *Acta informatica*, (22) :253–275, 1985.

[20] J. McCarthy. Circumscription-a form of nonmonotonic reasoning. *Artificial Intelligence*, 13.

[21] J. F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *In J. Kamorowski and Z. W. Ras,editors, Proceedings of ISMIS'93, LNAI 689*, 1993.

[22] J.F. Puget. Symmetry breaking revisited. In *International conference on constraint programming*, volume 2470 of *LNCS*, pages 446–461. Springer Verlag, 2002.

[23] Ray Reiter. A logic for default reasoning. *Artificial Intelligence*, pages 81–132, 1980.

[24] C. M. Roney-Dougal, I. P. Gent, T. Kelsey, and S. A. Linton. Tractable symmetry breaking using restricted search trees. *In, proceedings of ECAI'04*, pages 211–215, 2004.

[25] Yoav Shoam. A semantical approach to nonmonotonic logic. In *IJCAI*, pages 388–392, 1987.

[26] P. Siegel, L. Forget, and V. Risch. Preferential logics are x-logics. *Journal of Logic and Computation*, 11(1) :71–83, 2001.

[27] T. Walsh. General symmetry breaking constraints. *In, proceedings of CP'06*, pages 650–664, 2006.

Caractérisation de souches bactériennes à l'aide de la logique propositionnelle

Fabien Chhel Frédéric Lardeux Frédéric Saubion

LERIA, Université d'Angers
{nom}@info.univ-angers.fr

Ces dernières années ont vu le développement d'importantes collaborations entre les biologistes et les informaticiens qui ont ainsi mis en lumière de nouveaux problèmes fondamentaux, en particulier dans le domaine de l'optimisation combinatoire, comme le problème d'alignement de séquences ou de reconstruction phylogénétique, et dont la résolution requière la conception d'algorithmes performants. En effet, dès lors que l'acquisition de données a fait d'énormes progrès, notamment en terme de séquençage ou d'utilisation de puces à ADN, les volumes de données expérimentales et de mesures ont été considérablement accrus et leur traitement effectif nécessite le développement de nouveaux outils et techniques. Soulignons d'ailleurs au passage qu'une utilisation pratique de ces données se fonde souvent sur une chaîne de compétences scientifiques allant du traitement d'images ou du signal à l'algorithmique en passant par l'utilisation d'outils statistiques d'analyse de données.

Si nous ne nous plaçons pas ici directement dans le champs académique, désormais bien reconnu, de la bioinformatique, l'objectif de cet article est de présenter une application des techniques de modélisation et de résolution en logique propositionnelle dans le domaine de la biologie végétale. Le problème de caractérisation que nous abordons pourra sans nul doute trouver d'autres applications, dont nous avons déjà identifié certaines, en particulier dans le domaine du diagnostic biologique.

La caractérisation précise de collections de souches bactériennes représente un enjeu scientifique majeur puisque les bactéries phytopahogènes sont en effet responsables d'importants dégâts sur les cultures et certaines sont recensées sur des listes d'organismes de quarantaine et font l'objet d'une lutte officielle (Directive 2000/29/CE). Le développement de tests de diagnostic permettant d'identifier des souches de ces espèces s'avère dès lors nécessaire.

Dans ce contexte nous nous intéressons aux souches bactériennes du genre *Xanthomonas*. La notion de pathovar est une subdivision de l'espèce bactérienne phytopathogène qui regroupes des souches responsables d'un même symptôme sur une espèce végétale ou une gamme d'espèces végétales. En particulier les Xanthomonas sont un modèle d'études puisqu'elles présentent une centaine de pathovars différents. Par exemple, le *Xanthmonas axonopodis* se décline en *pathovar citri* qui occasionne le chancre citrique des agrumes mais également en *pathovar vesicatoria* qui est responsable de la gale bactérienne du poivron. Toutefois, la phylogénie des souches ne suffit pas à expliquer la spécifié d'hôte, c'est à dire l'espèce végétale attaquée par la souche. En particulier, certains pathovars proches génétiquement peuvent avoir des spécificités d'hôtes très éloignées et vice versa. L'approche consiste alors à identifier pour les souches des répertoires de gènes pertinents (gènes de virulence) et à analyser la corrélation entre la présence/absence de ces gènes et la spécificité d'hôtes des pathovars (groupes de souches bactériennes) [3].

Récemment, la description de répertoires de 35 gènes de virulence dans une collection de 132 souches de Xanthomonas réparties dans 21 groupes et présentant des spécificités d'hôte différentes a permis de montrer qu'il existe une corrélation entre le répertoire de gènes de virulence d'une souche de Xanthomonas et sa spécificité d'hôte. Au sein du genre Xanthomonas, d'autres espèces sont inscrites sur les listes de quarantaine, ou même sur les listes de bioterrorisme, et font ainsi l'objet de réglementations strictes.

Le problème de caractérisation se pose alors comme l'identification d'une famille de souches par rapport aux autres familles en fonction de la présence ou de l'absence de certains gènes. Une souche sera donc un vecteur de valeurs binaires qui rendent compte de la présence (valeur 1) ou de l'absence (valeur 0) d'un caractère.

Plus concrètement, une instance de problème possédant 5 souches, réparties en 3 groupes et basée sur un ensemble de 4 gènes peut être illustrée par la figure .

Résoudre ce problème revient à caractériser chaque

groupe. Il faut donc, pour chaque groupe, trouver une combinaison de présences ou d'absences de gènes valide pour toutes les souches du groupe et non valide pour toutes les autres souches des autres groupes. Dans l'exemple de la figure , le groupe 1 est caractérisé par la présence conjointe des gènes grisés.

Souche	Groupe	Gènes			
		x1	x2	x3	x4
e1	g1	1	1	1	0
e2	g1	1	1	1	1
e3	g2	0	0	1	0
e4	g2	0	1	1	1
e5	g3	1	1	0	0

FIGURE 1 – Exemple de caractérisation

Une fois posé le problème de la caractérisation, naturellement diverses méthodes peuvent être envisagées. En particulier, la méthode des CCD (Coefficient de Capacité de Diagnostique [1]), dans laquelle, par le biais d'une étude statistique, les gènes sont triés par pertinence de caractérisation. L'atout d'une telle approche est la simplicité du calcul mais elle ne permet de traiter que la caractérisation sur un seul gène. Il existe alors un réel besoin de développer de nouvelles approches pour fournir des formules de caractérisation combinant plusieurs gènes. De plus, les biologistes sont intéressés par deux propriétés spécifiques des solutions :

- Une solution qui minimise le nombre de caractères utilisés. Ceci est d'autant plus important que, dans l'optique de la fabrication de tests de diagnostic basé sur des puces à ADN [5], le nombre de gènes à observer doit être minimisé à la fois pour des raisons de coûts, de temps d'expérience et de fiabilité. Enfin, un autre critère lié à la présence plutôt que leur absence des gènes peut intervenir, car il est plus facile d'observer expérimentalement l'existence d'un gène à l'aide de marqueurs que de vérifier son absence.
- Le calcul de toutes les solutions : il peut s'avèrer utile, du point de vue de l'interprétation biologique, de disposer d'une représentation de toutes les solutions possibles car elles pourraient faire apparaître des relations particulières entre les gènes permettant d'expliquer certaines spécificités fonctionnelles des bactéries.

D'un point de vue formel, nous pouvons considérer que les absences ou présences de gènes peuvent être vue comme les valeurs de vérités de variables booléennes et que la caractérisation d'un groupe revient alors à trouver une formule booléenne qui est satisfaite par les affectations correspondant à ce groupe et falsifiée par les autres. La relation avec l'apprentissage de fonction booléenne surgit alors naturellement. L'apprentissage automatique de fonction booléenne à partir d'exemples est un problème qui a été très largement étudié depuis de nombreuses années, depuis les travaux initiaux de Valiant [7] puis de Natarajan [4], jusqu'à des travaux plus récents comme Gavalda et al. [2]. Toutefois, le problème est ici différent. La caractérisation doit être faite pour chaque groupe par rapport aux autres de manière exacte. Enfin, l'objectif est de minimiser le nombre de littéraux utilisées pour cette caractérisation croisée. L'aspect combinatoire de cette caractérisation de chaque groupe par rapport aux autres induit d'ailleurs une complexité importante du problème.

Dans ce travail, nous modélisons problème de caractérisation comme la recherche d'ensembles de formules en logique propositionnelle, ce qui nous permet ensuite de mettre en évidence que sa classe de complexité est Σ_2^p-complet, puisqu'il revient à minimiser des formules DNF [6]. Nous avons développé un algorithme de caractérisation incluant des techniques de simplification qui nous a permis d'obtenir des résultats, ayant déjà été valorisés par la mise au point de tests de diagnostic collaboration avec nos collègues de l'UMR PAVE de l'INRA d'Angers.

Références

[1] P. Descamps and M. Véron. Une méthode de choix des caractères d'identification basée sur le théorème de bayes et la mesure de l'information. *Ann. Microbiol. (Paris)*, 132B, 1981.

[2] Ricard Gavaldà and Denis Thérien. An algebraic perspective on boolean function learning. In *Algorithmic Learning Theory, 20th International Conference, ALT 2009, Porto, Portugal, October 3-5, 2009. Proceedings*, volume 5809 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2009.

[3] Ahmed Hajri, Chrystelle Brin, Gilles Hunault, Frédéric Lardeux, Christophe Lemaire, Charles Manceau, Tristan Boureau, and Stéphane Poussier. A "repertoire for repertoire" hypothesis : Repertoires of type three effectors are candidate determinants of host specificity in xanthomonas. *PLoS ONE*, 4(8) :e6632, 08 2009.

[4] B. K. Natarajan. On learning boolean functions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 296–304. ACM, 1987.

[5] M. Schena, D. Shalon, R.W. Davis, and P.O. Brown. Quantitative monitoring of gene expression patterns with a complementary dna microarray. *Science*, 270(5235) :467–470., 2005.

[6] Christopher Umans. The minimum equivalent DNF problem and shortest implicants. *J. Comput. Syst. Sci.*, 63(4) :597–611, 2001.

[7] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11) :1134–1142, 1984.

Une approche syntaxique pour le problème de la fusion de réseaux de contraintes qualitatives

Jean-François Condotta, Souhila Kaci, Pierre Marquis et Nicolas Schwind

Université Lille-Nord de France, Artois, F-62307 Lens
CRIL, F-62307 Lens
CNRS UMR 8188, F-62307 Lens
{condotta,kaci,marquis,schwind}@cril.univ-artois.fr

Résumé

Dans cet article, nous nous intéressons au problème de la fusion de réseaux de contraintes qualitatives (RCQ) représentant des croyances ou des préférences locales sur les positions relatives d'entités spatiales ou temporelles. Nous définissons deux classes d'opérateurs de fusion Δ_1 et Δ_2 qui, à un ensemble de RCQ définis sur le même formalisme qualitatif et le même ensemble d'entités, associent un ensemble cohérent de configurations qualitatives représentant une vision globale de ces RCQ. Ces opérateurs sont paramétrés par une distance entre relations du formalisme qualitatif considéré et par des fonctions d'agrégation. Contrairement aux précédents opérateurs proposées pour la fusion de RCQ, nous optons pour une approche syntaxique, où chacune des contraintes des RCQ fournis a une influence sur le résultat de la fusion. Nous étudions les propriétés logiques des opérateurs de fusion définis et montrons leur équivalence sous certaines restrictions. Nous montrons que le résultat fourni par l'opérateur Δ_2 correspond à l'ensemble des solutions optimales d'un RCQ pondéré particulier. Afin de calculer ces solutions, un algorithme basé sur la méthode de fermeture par faible composition étendu au cas des RCQ pondérés est proposé.

1 Introduction

La représentation qualitative du temps et de l'espace intervient dans de nombreux domaines de l'Intelligence Artificielle comme le traitement du langage naturel, la conception assistée par ordinateur (CAO) ou l'ordonnancement d'activités. Etant donné un ensemble d'entités spatiales ou temporelles, exprimer de manière explicite l'ensemble des configurations globales plausibles ou préférées sur ces entités est une tâche ardue pour des raisons liées à son élicitation. En effet, on est généralement plus disposé à exprimer des relations locales sur ces entités à partir desquelles les configurations globales sous-jacentes peuvent être déduites.Considérons par exemple un étudiant, Jérome, qui exprime ses préférences sur l'emploi du temps de quatre matières (algèbre, analyse, algorithmique et anglais) se déroulant sur un intervalle de temps. Jérome préfère apprendre l'algorithmique après l'analyse. Une difficulté est que l'expression des préférences locales entre les choix est susceptible de provoquer une incohérence (ou conflit). Ainsi, si Jérome souhaite aussi que le cours d'anglais se déroule après celui d'algorithmique, et voudrait commencer à suivre le cours d'anglais avant la fin du cours d'analyse, alors aucune programmation possible de ces cours ne peut satisfaire toutes ses préférences. L'incohérence peut aussi être due à la présence de plusieurs agents. Dans notre exemple, il s'agit d'agréger les préférences de plusieurs étudiants sur la programmation des cours.

Dans cet article, les croyances ou les préférences de chaque agent sur les positions relatives d'entités spatiales ou temporelles sont représentées par un réseau de contraintes qualitatives (RCQ). Une méthode de fusion de RCQ a été proposée dans [5] qui est directement adaptée d'une méthode de fusion en logique propositionnelle. Elle consiste à définir un opérateur de fusion qui prend en entrée un ensemble fini de RCQ et retourne comme résultat un ensemble cohérent d'informations spatiales ou temporelles représentant une vision globale de ces RCQ. Néanmoins cette méthode présente deux inconvénients : le premier est qu'elle ne considère pas les croyances ou les préférences locales explicitement fournies par les RCQ, car chacun d'entre eux est interprété comme l'ensemble des configurations qualitatives cohérentes qu'il représente ; le second est sa complexité calculatoire élevée. Nous proposons dans cet article une approche syntaxique pour la fusion de RCQ. Dans cette approche, toutes les contraintes des RCQ fournis ont une influence sur le résultat de la fusion. Basées sur ce principe, nous définissons deux classes d'opérateurs de

fusion Δ_1 et Δ_2 qui, à un ensemble de RCQ définis sur le même formalisme qualitatif et le même ensemble d'entités, associent un ensemble de configurations qualitatives des entités représentant une vision globale de ces RCQ. Ces opérateurs sont paramétrés par une distance entre relations du formalisme qualitatif considéré et par deux fonctions d'agrégation de distances.

La suite de l'article est organisée de la manière suivante. La section suivante introduit des préliminaires sur les réseaux de contraintes qualitatives, les distances sur les relations d'un formalisme qualitatif et les fonctions d'agrégation. Dans la section 3, nous présentons le problème de la fusion de RCQ à travers un exemple, nous définissons les classes d'opérateurs de fusion de RCQ Δ_1 et Δ_2, nous étudions leurs propriétés logiques, les comparons avec les opérateurs de fusion proposés récemment dans la littérature et montrons l'équivalence entre Δ_1 et Δ_2 sous certaines conditions. Dans la section 4, nous introduisons les RCQ pondérés, directement adaptés des problèmes de satisfaction de contraintes pondérés, nous montrons que le résultat renvoyé par l'opérateur Δ_2 correspond à l'ensemble des solutions optimales d'un RCQ pondéré particulier, et nous proposons un algorithme pour calculer ces solutions. Nous concluons dans la dernière section et présentons des perspectives pour un travail futur.

2 Préliminaires

2.1 Formalismes qualitatifs et réseaux de contraintes qualitatives

Un formalisme qualitatif considère un ensemble fini B de relations de base binaires définies sur un domaine D. Les éléments de D représentent les entités temporelles ou spatiales considérées. Chaque relation de base $b \in \mathsf{B}$ représente une position relative particulière pouvant être satisfaite par deux éléments de D. L'ensemble B satisfait également les trois propriétés suivantes : (i) B forme une partition de $\mathsf{D} \times \mathsf{D}$, autrement dit tout couple de $\mathsf{D} \times \mathsf{D}$ satisfait une et une seule relation de base de B ; (ii) la relation identité sur D, notée eq dans la suite, appartient à l'ensemble B ; (iii) pour toute relation de base $b \in \mathsf{B}$, la relation inverse de b, notée b^{-1}, appartient également à l'ensemble B.

À titre d'illustration considérons le formalisme qualitatif appelé l'Algèbre des Intervalles. Ce formalisme, introduit par Allen [1], considère un ensemble $\mathsf{B}_{\mathsf{int}}$ de treize relations de base défini sur le domaine des intervalles de la droite des nombres rationnels : $\mathsf{D}_{\mathsf{int}} = \{(x^-, x^+) \in \mathbb{Q} \times \mathbb{Q} : x^- < x^+\}$. Ces intervalles représentent des entités temporelles non ponctuelles. Les relations de base de $\mathsf{B}_{\mathsf{int}} = \{eq, p, pi, m, mi, o, oi, s, si, d, di, f, fi\}$ sont illustrées à la figure 1. Chacune d'entre elles représente une situation particulière entre deux intervalles. Par exemple, la relation $m = \{((x^-, x^+), (y^-, y^+)) \in \mathsf{D}_{\mathsf{int}} \times \mathsf{D}_{\mathsf{int}} : x^+ = y^-\}$ représente le cas où la borne supérieure du premier intervalle et la borne inférieure du second coïncident.

À partir d'un ensemble B de relations de base sont définies les relations « complexes ». Une relation complexe est l'union de relations de base, que l'on représente par l'ensemble des relations de base qu'elle contient. Dans la suite nous omettons le qualificatif « complexe ». En considérant l'Algèbre des Intervalles, l'ensemble $\{m, d\}$ représente la relation issue de l'union des relations de base m et d. Ainsi 2^{B}, l'ensemble des sous-ensembles de B, représente l'ensemble des relations. 2^{B} est muni des opérations ensemblistes habituelles que sont l'union ($\cup$) et l'intersection ($\cap$), ainsi que de l'opération inverse ($^{-1}$) définie par : $\forall r \in 2^{\mathsf{B}}, r^{-1} = \{b \in \mathsf{B} \mid b^{-1} \in r\}$, et de l'opération de faible composition ($\diamond$) définie par : $\forall a, b \in \mathsf{B}, a \diamond b = \{c \in \mathsf{B} : \exists x, y, z \in \mathsf{D} \mid x\ a\ z \wedge z\ b\ y \wedge x\ c\ y\}$; $\forall r, s \in 2^{\mathsf{B}}, r \diamond s = \bigcup_{a \in r, b \in s} \{a \diamond b\}$. Notons que $r \diamond s$ est la plus petite relation de 2^{B} (pour l'inclusion ensembliste) contenant la composition relationnelle $r \circ s = \{(x, y) \in \mathsf{D} \times \mathsf{D} : \exists z \in \mathsf{D} \mid x\ r\ z \wedge z\ s\ y\}$. Pour certains formalismes qualitatifs, en particulier l'Algèbre des Intervalles, $r \circ s$ et $r \diamond s$ sont identiques.

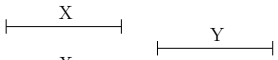
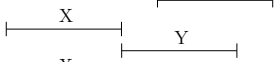
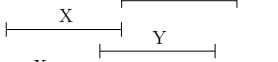
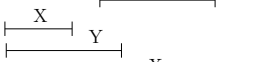
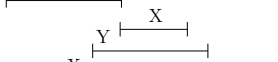
Relation	Symbole	Inverse	Illustration
precedes	p	pi	
meets	m	mi	
overlaps	o	oi	
starts	s	si	
during	d	di	
finishes	f	fi	
equals	eq	eq	

FIGURE 1 – Les relations de base de l'Algèbre des Intervalles.

Il est habituel de représenter un ensemble de configurations qualitatives préférées ou possibles d'un ensemble d'entités à l'aide de réseaux de contraintes qualitatives (RCQ en abrégé). Formellement, un RCQ (sur B) est défini de la manière suivante :

Définition 1 (Réseau de contraintes qualitatives). *Un* RCQ N *est un couple* (V, C) *où :*

- $V = \{v_1, \ldots, v_n\}$ *est un ensemble fini de variables représentant les entités,*
- C *est une application qui associe à tout couple de variables* $(v_i, v_j) \in V \times V$ *une relation* $N[i, j]$ *appartenant à* 2^{B}. C *est telle que* $N[i, i] = \{eq\}$ *et* $N[i, j] = N[j, i]^{-1}$*, pour toute paire* $(v_i, v_j) \in V$.

Étant donné un RCQ $N = (V, C)$, une *solution partielle* de N sur $V' \subseteq V$ est une application α de V' sur D telle que pour chaque couple (v_i, v_j) de variables de V', $(\alpha(v_i), \alpha(v_j))$ satisfait $N[i, j]$, *i.e.,* il existe une relation de base $b \in N[i, j]$ telle que $(\alpha(v_i), \alpha(v_j)) \in b$ pour

tout $v_i, v_j \in V$. Une *solution* de N est une solution partielle de N sur V. N est *cohérent* ssi il admet une solution. Un *sous*-RCQ N' de N est un RCQ (V, C') tel que $N'[i,j] \subseteq N[i,j]$, pour chaque paire de variables (v_i, v_j). Un *scénario* σ est un RCQ tel que chaque contrainte est définie par une relation singleton de 2^{B}, *i.e.,* une relation ne contenant qu'une seule relation de base. Pour un scénario σ donné, la relation de base définissant la contrainte entre deux variables v_i et v_j est notée σ_{ij}. Un scénario σ de N est un sous-RCQ de N. Dans la suite de l'article, $\langle N \rangle$ dénote l'ensemble des scénarios de N et $[N]$ l'ensemble de ses scénarios cohérents. Deux RCQ N et N' sont dits *équivalents*, noté $N \equiv N'$, ssi $[N] = [N']$. N^V_{All} dénote le RCQ particulier sur V vérifiant pour tout couple de variables (v_i, v_j), $N^V_{All}[i,j] = \{eq\}$ si $v_i = v_j$, $N^V_{All}[i,j] = \mathsf{B}$ sinon. N^V_{All} représente l'absence totale d'information sur la configuration des variables de V.

Un RCQ $N = (V, C)$ est dit *fermé par faible composition* ssi $N[i,j] \subseteq N[i,k] \diamond N[k,j]$ pour tout $v_i, v_j, v_k \in V$. La méthode de la fermeture (notée dans la suite $\diamond$-fermeture) par faible composition est la méthode qui consiste à itérer l'opération de triangulation : $N[i,j] \leftarrow N[i,j] \cap (N[i,k] \diamond N[k,j])$ jusqu'à ce qu'un point fixe soit obtenu. Le RCQ obtenu est un sous-RCQ équivalent au réseau initial. Cette méthode a une complexité polynomiale et est habituellement utilisée comme méthode de filtrage lors d'une recherche de scénarios cohérents d'un RCQ. Elle est complète pour le problème de déterminer si un RCQ est cohérent ou non lorsqu'on se restreint à des sous-ensembles particuliers de 2^{B} appelés *classes traitables*, comme par exemple l'ensemble des relations singletons de l'Algèbre des Intervalles.

Les figures 2(a), 2(b) et 2(c) représentent respectivement un RCQ N de l'Algèbre des Intervalles défini sur l'ensemble des variables $V = \{v_1, v_2, v_3, v_4\}$, un scénario non cohérent σ de N et un scénario cohérent σ' de N. Une solution α de σ' est représentée à la figure 2(d). Pour tout couple (v_i, v_j) de variables, on ne représente pas graphiquement la relation $N[i,j]$ lorsque $N[i,j] = \mathsf{B}$, lorsque $N[j,i]$ est présente, ou lorsque $i = j$.

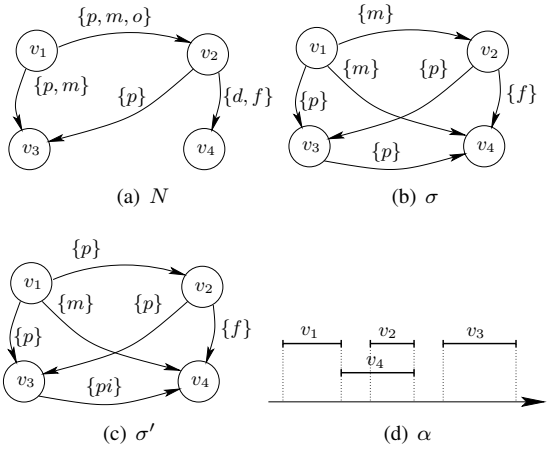


FIGURE 2 – Un RCQ N, un scénario non cohérent σ de N, un scénario cohérent σ' de N et une solution de σ'.

2.2 Distances de base et fonctions d'agrégation

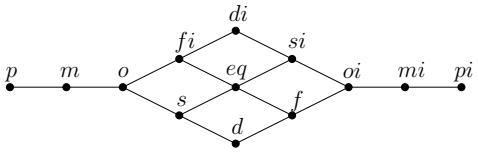
Dans la suite, nous considérons deux classes d'opérateurs de fusion de RCQ, toutes deux paramétrées par une distance entre relations de base de B, appelée distance de base, et par des fonctions d'agrégation. Nous définissons formellement la notion de distance de base, d'une part, de fonction d'agrégation d'autre part.

Distances de base. Une distance de base associe à un couple de relations de base de B une valeur numérique représentant leur degré de proximité [5].

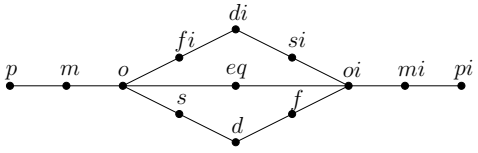
Définition 2 (Distance de base). *Une distance de base d_{B} est une application de $\mathsf{B} \times \mathsf{B}$ vers $\mathbb{R}^+$ telle que $\forall b, b' \in \mathsf{B}$ nous avons :*

$$\begin{cases} d_{\mathsf{B}}(b,b') = d_{\mathsf{B}}(b',b) & \textit{(symétrie)} \\ d_{\mathsf{B}}(b,b') = 0 \textit{ ssi } b = b' & \textit{(séparation)} \\ d_{\mathsf{B}}(b,b') = d_{\mathsf{B}}(b^{-1},(b')^{-1}). \end{cases}$$

Par exemple, une distance de base est la distance drastique d_D, égale à 1 pour tout couple de relations de base distinctes, 0 sinon. Cependant, dans le cadre des algèbres qualitatives, deux relations de base distinctes peuvent être considérées « plus ou moins » distantes. Cette intuition a donné lieu à des études dans le passé sur la notion de proximité entre relations de base. Nous pouvons par exemple citer les travaux de Freksa [6] qui définit une notion de voisinage conceptuel entre relations de base de l'algèbre des intervalles. En généralisant cette définition, deux relations de base $b, b' \in \mathsf{B}$ sont conceptuellement voisines si par transformation continue sur les éléments du domaine nous pouvons à partir d'une configuration entre deux entités satisfaisant la relation b obtenir une configuration satisfaisant b' sans qu'aucune autre relation n'ait été satisfaite au cours de la transformation. La transformation utilisée définit une relation de voisinage particulière sur B. Cette relation peut être représentée à l'aide d'un graphe de voisinage conceptuel, *i.e.,* un graphe connexe non orienté dont chaque noeud représente un élément de B. Dans un tel graphe, une arête relie deux noeuds correspondant à deux relations de base voisines. Prenons pour exemple les relations de base de l'algèbre des intervalles. Dans un contexte où une transformation continue entre deux intervalles correspond au déplacement d'une des quatre bornes considérées, nous obtenons le graphe de voisinage conceptuel $\mathsf{GB}_{\mathsf{int}}$ représenté à la figure 3(a). Dans d'autres circonstances, une transformation peut correspondre au décalage des intervalles sans faire varier leur durée, et nous obtenons dans ce cas le graphe de voisinage conceptuel $\mathsf{GB}'_{\mathsf{int}}$ illustré à la figure 3(b).



(a) $\mathsf{GB}_{\mathsf{int}}$



(b) $\mathsf{GB}'_{\mathsf{int}}$

FIGURE 3 – Deux graphes de voisinage conceptuels $\mathsf{GB}_{\mathsf{int}}$ (a) et $\mathsf{GB}'_{\mathsf{int}}$ (b) de l'Algèbre des Intervalles.

Une distance entre relations de base spécifique au cadre des RCQ et utilisant les graphes de voisinage conceptuel a été proposée dans [5], appelée *distance de voisinage conceptuel*. Elle est définie formellement comme suit :

Définition 3 (Distance de voisinage conceptuel). *Soit* GB *un graphe de voisinage conceptuel sur* B. *La distance de voisinage conceptuel $d_{\mathsf{GB}}(a, b)$ entre deux relations de base a, b de* B *est la longueur de la chaîne la plus courte entre les deux noeuds n_a et n_b représentant respectivement a et b dans* GB.

Dans les exemples qui suivent, nous utilisons la distance de voisinage conceptuel $d_{\mathsf{GB}_{\mathsf{int}}}$ définie à partir du graphe conceptuel de voisinage $\mathsf{GB}_{\mathsf{int}}$. Par exemple, $d_{\mathsf{GB}_{\mathsf{int}}}(m, di) = 4$. Clairement, $d_{\mathsf{GB}_{\mathsf{int}}}$ est une distance de base (au sens de la définition 2) et peut donc être utilisée dans nos processus de fusion.

Fonctions d'agrégation. Typiquement, une fonction d'agrégation [10, 7] associe à un ensemble de valeurs numériques une valeur représentant cet ensemble dans sa globalité, selon certains critères.

Définition 4 (Fonction d'agrégation). *Une fonction d'agrégation f associe à un ensemble de nombres réels positifs un nombre réel positif et satisfait les propriétés suivantes :*

$$\begin{cases} \textit{si } x_1 \leq x'_1, \ldots, x_p \leq x'_p, \textit{ alors} \\ \qquad f(x_1, \ldots, x_p) \leq f(x'_1, \ldots, x'_p) & \textit{(monotonie)} \\ x_1 = \cdots = x_p = 0 \textit{ ssi } f(x_1, \ldots, x_p) = 0 & \textit{(minimalité)} \end{cases}$$

Nous présentons quelques propriétés supplémentaires sur les fonctions d'agrégation.

Définition 5 (Propriétés sur les fonctions d'agrégation). *Soient f et g deux fonctions d'agrégation.*

– *f est* associative *ssi*

$$f(f(x_1, \ldots, x_p), f(y_1, \ldots, y_{p'})) = f(x_1, \ldots, x_p, y_1, \ldots, y_{p'}).$$

– *f est* symétrique *ssi pour toute permutation τ de $\mathbb{R}^p$ dans $\mathbb{R}^p$, p étant un entier naturel strictement positif,*

$$f(x_1, \ldots, x_p) = f(\tau(x_1), \ldots, \tau(x_p)).$$

– *f est* strictement monotone *ssi*

$$\begin{array}{l} \textit{si } x_1 \leq x'_1, \ldots, x_p \leq x'_p \\ \textit{et } \exists i \in \{1, \ldots, p\}, x_i < x'_i, \\ \textit{alors } f(x_1, \ldots, x_p) < f(x'_1, \ldots, x'_p). \end{array}$$

– *f* commute *sur g (ou f et g sont* inter-commutatives*) ssi*

$$\begin{array}{l} f(g(x_{1,1}, \ldots, x_{1,q}), \ldots, g(x_{p,1}, \ldots, x_{p,q})) \\ = g(f(x_{1,1}, \ldots, x_{p,1}), \ldots, f(x_{1,q}, \ldots, x_{p,q})). \end{array}$$

Dans le reste de l'article, toute fonction d'agrégation est supposée associative et symétrique. L'associativité impose que l'agrégation de valeurs peut etre décomposée par des agrégations partielles, la symétrie signifie que l'ordre des arguments passés en paramètres de la fonction d'agrégation n'a pas d'importance. Dans [12], il est souligné que la propriété d'inter-commutativité entre des fonctions d'agrégation joue un rôle important dans un processus de fusion d'information en plusieurs étapes dans lequel le résultat ne doit pas dépendre de l'ordre dans lequel les étapes sont traitées. Cette assertion sera discutée dans notre cadre (cf. section 3.4). Par exemple la fonction d'agrégation MAX renvoyant le maximum de ses arguments est associative, symétrique, et commute sur elle-même. Il en est de même pour la fonction $\sum$ qui effectue la somme de ses arguments, et qui de plus est strictement monotone.

3 Fusion de RCQ

3.1 Problème et exemple

Nous définissons dans cette section deux classes d'opérateurs de fusion de croyances ou de préférences émanant d'un ensemble d'agents $\{Agent_1, \ldots, Agent_m\}$ concernant la configuration d'un ensemble d'entités représentées par un ensemble de variables V. À chaque agent $Agent_k$ exprimant ses croyances ou ses préférences locales sur les positions relatives de chaque couple d'entités est associé un RCQ $N^k = (V, C^k)$. Une contrainte $N^k[i, j]$ correspond pour l'agent $Agent_k$ à l'ensemble des relations de base représentant ses préférences ou ses croyances pour le couple (v_i, v_j). Dans un tel cadre, deux types d'incohérence sont suceptibles d'apparaître. D'une part, pour tout RCQ N^k, $k \in \{1, \ldots, m\}$, N^k n'est pas nécessairement cohérent. En effet, les informations fournies permettant de définir N^k sont données sur des paires de variables et sont donc locales, ce qui peut généralement provoquer l'apparition d'un conflit. D'autre part, un autre type d'incohérence peut survenir lorsque les différents RCQ fournis sont combinés entre eux. Par exemple, dans le cas des préférences, un simple conflit d'intérêt entre deux agents à propos d'un

même couple de variables suffit à rendre l'ensemble des RCQ conflictuels.

Nos deux classes d'opérateurs de fusion considèrent en entrée un multi-ensemble $\mathcal{N} = \{N^1, \ldots, N^m\}$ de RCQ définis sur V, tel que tout RCQ $N^k = (V, C^k)$ représente les croyances ou les préférences de l'agent $Agent_k$. Un tel multi-ensemble est appelé *profil*, ou *profil de* RCQ. De manière naturelle, nous imposons qu'aucune des contraintes des RCQ ne soit définie par la relation vide, ainsi $N^k[i,j] \neq \emptyset$ pour tout $k \in \{1, \ldots, m\}$. Si une contradiction au niveau d'un agent et d'une paire de variables apparaît, nous définissons la contrainte correspondante par la relation B exprimant l'absence totale d'information.

Exemple. Considérons un ensemble de quatre cours communs pour un ensemble d'étudiants donnés, nous restreignons notre exemple à un ensemble de trois étudiants Pierre, Paul et Jacques. Chacun d'entre eux peut exprimer ses préférences sur l'emploi du temps en fournissant un ensemble de relations binaires entre ces cours. Nous considérons alors quatre variables temporelles, chacune d'elles représentant un cours se déroulant sur un intervalle de temps. Les quatre variables v_1, v_2, v_3, v_4 représentent respectivement les cours d'algèbre, analyse, algorithmique et anglais, et forment l'ensemble V. Nous considérons l'algèbre des intervalles pour représenter des relations qualitatives entre ces cours. Par exemple, Pierre préfère commencer à étudier l'algèbre avant le début du cours d'analyse et voudrait également terminer l'algèbre avant de terminer l'analyse. Ceci peut être exprimé par la relation $v_1\{p, m, o\}v_2$. Ainsi, Pierre, Paul et Jacques fournissent respectivement les RCQ N^1, N^2, N^3 représentés à la figure 4, représentant leurs préférences sur les relations binaires entre ces cours et formant le profil $\mathcal{N}$. Remarquons que l'exemple de conflit décrit dans l'introduction intervient dans le RCQ N^3 ; en effet il n'existe pas de solution partielle de N^3 sur $\{v_2, v_3, v_4\}$. Nous nous appuyons sur cet exemple dans le reste de l'article.

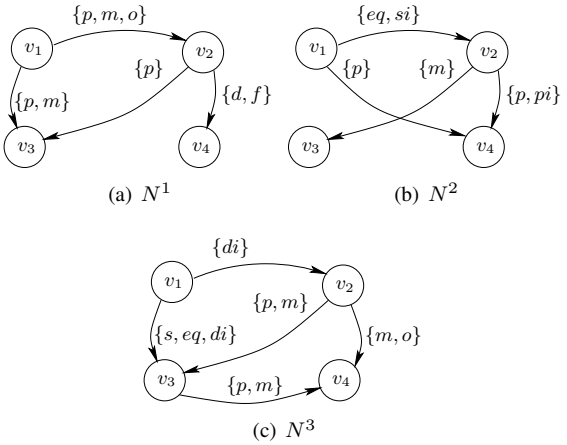


FIGURE 4 – Trois RCQ N^1, N^2 et N^3 à fusionner.

3.2 Postulats de rationalité pour les opérateurs de fusion de RCQ

Étant donné un profil $\mathcal{N} = \{N_1, \ldots, N_m\}$ de RCQ sur V représentant les croyances ou les préférences d'un ensemble d'agents, nous désirons obtenir comme résultat de la fusion un ensemble d'informations *cohérentes* représentant $\mathcal{N}$ dans sa globalité. Dans [3] des postulats de rationalité ont été proposés pour les opérateurs de fusion de RCQ. Les auteurs considèrent un cadre très général où un opérateur de fusion de RCQ est fondamentalement une application qui, à un profil de RCQ, associe un ensemble de RCQ (éventuellement incohérents). Notre but ici est de proposer un compromis *cohérent* pour la fusion de RCQ :

Définition 6 (**Opérateur** Δ)**.** *Un opérateur Δ est une application qui, à un profil $\mathcal{N}$ de* RCQ*, associe un ensemble $\Delta(\mathcal{N})$ de scénarios cohérents.*

Nous adaptons à cette définition les postulats proposés dans [3] pour définir un opérateur de fusion de RCQ.

Définition 7 (**Opérateur de fusion de** RCQ)**.** *Soit $\mathcal{N}$ un profil de* RCQ*. Δ est un opérateur de fusion de* RCQ *ssi il satisfait les postulats suivants :*

(N1) $\Delta(\mathcal{N}) \neq \emptyset$.
(N2) *Si* $\bigcap\{[N^k] \mid N^k \in \mathcal{N}\} \neq \emptyset$*, alors* $\Delta(\mathcal{N}) = \bigcap\{[N^k] \mid N^k \in \mathcal{N}\}$.

(N1) assure que la cohérence de $\Delta(\mathcal{N})$. (N2) impose pour $\Delta(\mathcal{N})$ de correspondre à l'ensemble des scénarios cohérents partagés par chacun des RCQ de $\mathcal{N}$, lorsque cet ensemble est non vide.

Nous présentons un ensemble de postulats supplémentaires pour les opérateurs Δ. Avant de les présenter, nous définissons la notion d'équivalence entre profils de RCQ. Deux profils $\mathcal{N}$ et $\mathcal{N}'$ sont dits *équivalents*, noté $\mathcal{N} \equiv \mathcal{N}'$, ssi il existe une bijection f de $\mathcal{N}$ dans $\mathcal{N}'$ telle que $\forall N \in \mathcal{N}$, $N \equiv f(N)$. $\sqcup$ est l'opérateur d'union pour les multi-ensembles.

Définition 8 (**Postulats (N3) - (N6)**)**.** *Soient $\mathcal{N}, \mathcal{N}_1$ et $\mathcal{N}_2$ trois profils de* RCQ*, et soient N, N' deux* RCQ *cohérents.*

(N3) *Si $\mathcal{N}_1 \equiv \mathcal{N}_2$, alors $\Delta(\mathcal{N}_1) \equiv \Delta(\mathcal{N}_2)$.*
(N4) *Si $\Delta(\{N, N'\}) \cap [N] \neq \emptyset$, alors $\Delta(\{N, N'\}) \cap [N'] \neq \emptyset$.*
(N5) $\Delta(\mathcal{N}_1) \cap \Delta(\mathcal{N}_2) \subseteq \Delta(\mathcal{N}_1 \sqcup \mathcal{N}_2)$.
(N6) *Si $\Delta(\mathcal{N}_1) \cap \Delta(\mathcal{N}_2) \neq \emptyset$, alors $\Delta(\mathcal{N}_1 \sqcup \mathcal{N}_2) \subseteq \Delta(\mathcal{N}_1) \cap \Delta(\mathcal{N}_2)$.*

(N3) applique le principe d'indépendance à la syntaxe des RCQ de $\mathcal{N}$. (N4) est le postulat d'équité, il impose que le résultat de la fusion de deux RCQ cohérents ne doit pas préférer l'un d'entre eux. (N5) indique qu'un scénario cohérent appartenant à la fusion d'un premier profil et à la fusion d'un second profil doit nécessairement appartenir à la fusion des deux profils joints. (N5) et (N6), ensemble,

assurent que si deux profils fusionnés de manière indépendante s'accordent sur un ensemble non vide E de scénarios cohérents, alors la fusion des deux profils joints doit être cet ensemble E.

3.3 Deux classes d'opérateurs de fusion

Dans cette section, nous définissons de manière constructive deux classes d'opérateurs de fusion de RCQ. Les opérateurs de la première et de la seconde classe sont respectivement notés Δ_1 et Δ_2. Ces opérateurs associent à un profil $\mathcal{N}$ un ensemble de scénarios cohérents les plus « proches » de $\mathcal{N}$ en terme de « distance ». C'est dans la définition de cette distance que réside leur différence. Pour $i \in \{1, 2\}$, un opérateur de fusion Δ_i est caractérisé par un triplet $(d_{\mathsf{B}}, f_i, g_i)$ où d_{B} est une distance de base sur B et f_i et g_i sont deux fonctions d'agrégations symétriques et associatives. L'opérateur Δ_i caractérisé par $(d_{\mathsf{B}}, f_i, g_i)$ est noté $\Delta_i^{d_{\mathsf{B}}, f_i, g_i}$. L'ensemble des scénarios cohérents $\Delta_i^{d_{\mathsf{B}}, f_i, g_i}(\mathcal{N})$ est défini par le résultat d'un processus en deux étapes.

Les opérateurs Δ_1 La première étape consiste à calculer une *distance locale* d_{f_1} entre tout scénario cohérent sur l'ensemble de variables V, *i.e.,* tout scénario cohérent de N^V_{All} et chacun des RCQ du profil $\mathcal{N}$. À cette fin, la distance de base d_{B} et la fonction d'agrégation f_1 sont utilisées pour définir la distance d_{f_1} entre deux scénarios σ et σ' de N^V_{All} de la manière suivante :

$$d_{f_1}(\sigma, \sigma') = f_1\{d_{\mathsf{B}}(\sigma_{ij}, \sigma'_{ij}) \mid v_i, v_j \in V, i < j\}.$$

La distance entre deux scénarios résulte donc de l'agrégation de distances au niveau de leurs contraintes au moyen d'une fonction d'agrégation. Le choix de la fonction d'agrégation f_1 dépend du contexte. Par exemple $f_1 = MAX$ est un choix approprié lorsque seule la distance la plus grande pour une paire de contraintes entre un scénario et un RCQ est importante, quelque soit le nombre de contraintes. Pour $f_1 = \sum$, les distances d_{B} sur toutes les contraintes sont sommées, et donc chacune d'entre elles est prise en compte.

La définition de d_{f_1} est étendue afin de calculer une distance entre un scénario cohérent de N^V_{All} et tout RCQ N^k de $\mathcal{N}$ de la manière suivante :

$$d_{f_1}(\sigma, N^k) = \min\{d_{f_1}(\sigma, \sigma') \mid \sigma' \in \langle N^k \rangle\}.$$

La distance entre un scénario σ et un RCQ N est donc la plus petite des distances (pour d_{f_1}) entre σ et l'ensemble des scénarios de N.

Exemple (suite). Par souci de concision de représentation, nous définissons un scénario comme l'ensemble de ses relations ordonné selon l'ordre lexicographique sur les couples de variables $(v_i, v_j), i < j$. Ainsi, le scénario cohérent σ_1 représenté à la figure 5(a) est désigné par l'ensemble de relations $\{\{fi\}, \{m\}, \{p\}, \{m\}, \{p\}, \{m\}\}$.

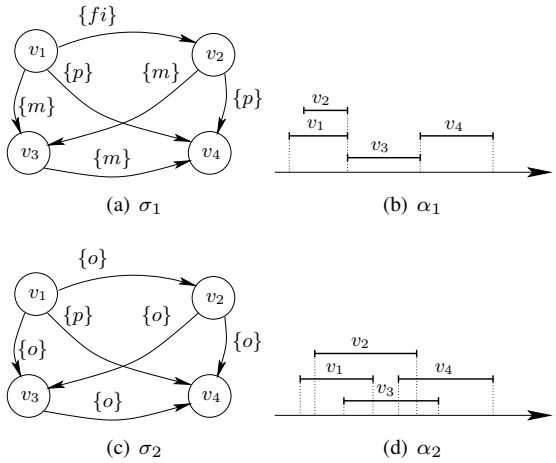


FIGURE 5 – Deux scénarios cohérents σ_1 et σ_2 de N^V_{All}, et deux instantiations cohérentes α_1 et α_2 de σ_1 et de σ_2.

Soit σ'' le scénario du RCQ N^1 (cf. figure 4(a)) défini par $\{\{o\}, \{m\}, \{p\}, \{p\}, \{d\}, \{m\}\}$. Nous utilisons ici la distance de base $d_{\mathsf{GB}_{\mathsf{int}}}$, de même que pour les exemples qui suivent. Nous choisissons pour l'exemple $f_1 = \sum$. On a alors :

$$\begin{aligned} & d_{\sum}(\sigma_1, N^1) = \min\{d_{\sum}(\sigma_1, \sigma') \mid \sigma' \in \langle N^k \rangle\} \\ & = d_{\sum}(\sigma_1, \sigma'') \\ & = \sum\{d_{\mathsf{GB}_{\mathsf{int}}}(fi, o), d_{\mathsf{GB}_{\mathsf{int}}}(m, m), d_{\mathsf{GB}_{\mathsf{int}}}(p, p), \\ & \qquad d_{\mathsf{GB}_{\mathsf{int}}}(m, p), d_{\mathsf{GB}_{\mathsf{int}}}(p, d), d_{\mathsf{GB}_{\mathsf{int}}}(m, m)\} \\ & = 1 + 0 + 0 + 1 + 4 + 0 = 6. \end{aligned}$$

De même, on a $d_{\sum}(\sigma_1, N^2) = 1$ et $d_{\sum}(\sigma_1, N^3) = 4$.

La deuxième étape du processus de fusion consiste, pour un scénario cohérent σ de N^V_{All}, à utiliser la fonction d'agrégation g_1 sur les distances $d_{f_1}(\sigma, N^k)$ calculées à l'étape précédente pour tout $N^k \in \mathcal{N}$ afin de calculer une *distance globale* d_{g_1} entre σ et le profil $\mathcal{N}$. Cette distance est définie comme suit :

$$d_{g_1}(\sigma, \mathcal{N}) = g_1\{d_{f_1}(\sigma, N^k) \mid N^k \in \mathcal{N}\}.$$

Plusieurs fonctions d'agrégations ont été étudiées dans la litérature dans le cadre de la résolution de conflits entre plusieurs agents [11, 9]. Pour $g_1 = \sum$, la distance globale reflète le point de vue de la majorité des sources [9] ; pour l'opérateur d'arbitrage $g_1 = MAX$, elle représentera une valeur plus consensuelle [11].

Exemple (suite). Considérons ici $g_1 = MAX$. On a alors :

$$\begin{aligned} & d_{MAX}(\sigma_1, \mathcal{N}) = \max\{d_{\sum}(\sigma_1, N^k) \mid N^k \in \mathcal{N}\} \\ & = \max\{6, 1, 4\} = 6. \end{aligned}$$

L'ensemble $\Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N})$ est l'ensemble des scénarios cohérents de N^V_{All} minimisant la distance globale d_{g_1}. Formellement,

$$\begin{aligned} \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}) = \{\sigma \in [N^V_{All}] \mid \nexists \sigma' \in [N^V_{All}], \\ d_{g_1}(\sigma', \mathcal{N}) < d_{g_1}(\sigma, \mathcal{N})\}. \end{aligned}$$

Exemple (suite). Considérons le scénario cohérent σ_2 représenté à la figure 5(c). On peut calculer sa distance globale de la même manière que pour σ_1. On a alors $d_{MAX}(\sigma_2, \mathcal{N}) = 5$. Puisque $d_{MAX}(\sigma_2, \mathcal{N}) < d_{MAX}(\sigma_1, \mathcal{N})$, le scénario cohérent σ_1 n'appartient pas à l'ensemble $\Delta_1^{d_{\mathsf{GB}_{\mathsf{int}}}, \sum, MAX}(\mathcal{N})$.

Proposition 1. *$\Delta_1^{d_{\mathsf{B}}, f_1, g_1}$ est un opérateur de fusion de* RCQ *au sens de la définition 7, c'est-à-dire qu'il satisfait les postulats **(N1)** et **(N2)**. De plus, $\Delta_1^{d_{\mathsf{B}}, f_1, g_1}$ satisfait **(N5)** et si g_2 satisfait la propriété de stricte monotonie, alors $\Delta_1^{d_{\mathsf{B}}, f_1, g_1}$ satisfait **(N6)**. Il ne satisfait pas **(N3)** et **(N4)** dans le cas général.*

Preuve. (N1) Par définition.
(N2) Soit $\sigma \in [N^V_{All}]$. D'après la propriété de séparation de d_{B}, on a $\forall \sigma' \in \langle N^V_{All} \rangle, \forall v_i, v_j, i < j, d_{\mathsf{B}}(\sigma_{ij}, \sigma'_{ij}) = 0$ ssi $\sigma_{ij} = \sigma'_{ij}$. Donc puisque f_1 satisfait la propriété la minimalité, on a $\forall \sigma' \in \langle N^V_{All} \rangle, d_{f_1}(\sigma, \sigma') = 0$ ssi $\sigma = \sigma'$. Donc $d_{f_1}(\sigma, N^k) = 0$ ssi $\sigma \in [N^k]$. Par minimalité de g_1, on a $d_{g_1}(\sigma, \mathcal{N}) = 0$ ssi $\sigma \in [N^k] \forall N^k \in \mathcal{N}$. Or $\bigcap\{[N^k] \mid N^k \in \mathcal{N}\} \neq \emptyset$. Donc par définition de $\Delta_1^{d_{\mathsf{B}}, f_1, g_1}$, $\sigma \in \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N})$ ssi $\sigma \in \bigcap\{[N^k] \mid N^k \in \mathcal{N}\}$.
(N5) Soient $\mathcal{N}_1$ et $\mathcal{N}_2$ deux profils. Soit $\sigma \in \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_1) \cap \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_2)$ et soit $\sigma' \in [N^V_{All}]$. On a $d_{g_1}(\sigma, \mathcal{N}_1) \leq d_{g_1}(\sigma', \mathcal{N}_1)$ et $d_{g_1}(\sigma, \mathcal{N}_2) \leq d_{g_1}(\sigma', \mathcal{N}_2)$, ou encore $g_1\{d_{f_1}(\sigma, N^k) \mid N^k \in \mathcal{N}_1\} \leq g_1\{d_{f_1}(\sigma', N^k) \mid N^k \in \mathcal{N}_1\}$ et $g_1\{d_{f_1}(\sigma, N^k) \mid N^k \in \mathcal{N}_2\} \leq g_1\{d_{f_1}(\sigma', N^k) \mid N^k \in \mathcal{N}_2\}$. Donc par monotonie et associativité de g_1, on a $g_1\{d_{f_1}(\sigma, N^k) \mid N^k \in \mathcal{N}_1 \sqcup \mathcal{N}_2\} \leq g_1\{d_{f_1}(\sigma', N^k) \mid N^k \in \mathcal{N}_1 \sqcup \mathcal{N}_2\}$. Donc $\sigma \in \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_1 \sqcup \mathcal{N}_2)$.
(N6) Soient $\mathcal{N}_1$ et $\mathcal{N}_2$ deux profils. Soit $\sigma \in \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_1 \sqcup \mathcal{N}_2)$. Par l'absurde, supposons que $\sigma \notin \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_1) \cap \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_2)$. Soit $\sigma \notin \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_1)$ (la preuve est similaire si on suppose $\sigma \notin \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_2)$). Puisque $\Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_1) \cap \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_2) \neq \emptyset$, par minimalité de g_1, $\exists \sigma' \in [N^V_{All}]$ tel que $d_{g_1}(\sigma', \mathcal{N}_1) < d_{g_1}(\sigma, \mathcal{N}_1)$ et $d_{g_1}(\sigma', \mathcal{N}_2) \leq d_{g_1}(\sigma, \mathcal{N}_2)$, ou encore $g_1\{d_{f_1}(\sigma', N^k) \mid N^k \in \mathcal{N}_1\} < g_1\{d_{f_1}(\sigma, N^k) \mid N^k \in \mathcal{N}_1\}$ et $g_1\{d_{f_1}(\sigma', N^k) \mid N^k \in \mathcal{N}_2\} \leq g_1\{d_{f_1}(\sigma, N^k) \mid N^k \in \mathcal{N}_2\}$. Dans par monotonie stricte et associativité de f_1, on a $g_1\{d_{f_1}(\sigma', N^k) \mid N^k \in \mathcal{N}_1 \sqcup \mathcal{N}_2\} < g_1\{d_{f_1}(\sigma, N^k) \mid N^k \in \mathcal{N}_1 \sqcup \mathcal{N}_2\}$, ce qui contredit $\sigma \in \Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}_1 \sqcup \mathcal{N}_2)$. ⊣

Les opérateurs Δ_2 Un opérateur Δ_2 est défini en deux étapes de la manière suivante. La première étape consiste, pour tout couple (v_i, v_j), $i < j$, à calculer une *distance locale* d_{f_2} entre toute relation de base de B et l'ensemble $\mathcal{N}[i,j] = \{N^k[i,j] \mid N^k \in \mathcal{N}\}$. La définition de la distance de base d_{B} entre deux relations de base de B est étendue à celle de la distance de base entre une relation de base $b \in \mathsf{B}$ et une relation $R \in 2^{\mathsf{B}}$, $R \neq \emptyset$. Elle correspond à la distance de base minimale entre b et chacune des relations de base de R et est définie formellement comme suit :

$$d_{\mathsf{B}}(b, R) = \min\{d_{\mathsf{B}}(b, b') \mid b' \in R\}.$$

La fonction d'agrégation f_2 permet alors de calculer la distance locale d_{f_2} entre toute relation de base de B et l'ensemble des contraintes $\mathcal{N}[i,j] = \{N^k[i,j] \mid N^k \in \mathcal{N}\}$ de la manière suivante :

$$d_{f_2}(b, \mathcal{N}[i,j]) = f_2\{d_{\mathsf{B}}(b, N^k[i,j]) \mid N^k[i,j] \in \mathcal{N}[i,j]\}.$$

Le choix de la fonction d'agrégation f_2 est motivé de la même manière que pour celui de la fonction d'agrégation g_1 des opérateurs Δ_1. En fonction du contexte, on choisira selon le cas une fonction majoritaire $\sum$ [9] ou une fonction d'arbitrage MAX [11]. En effet l'agrégation ici concerne pour chaque couple (v_i, v_j) les contraintes $N^k[i,j]$ dans les différents RCQ N^k du profil $\mathcal{N}$.

Exemple (suite). Considérons l'ensemble $\mathcal{N}[1,2] = \{\{p, m, o\}, \{eq, si\}, \{di\}\}$ (cf. figure 4) et choisissons $d_{\mathsf{GB}_{\mathsf{int}}}$ pour la distance de base et $f_2 = MAX$. La distance entre la relation de base fi et l'ensemble $\mathcal{N}[1,2]$ est alors définie par :

$$\begin{aligned} d_{MAX}(fi, \mathcal{N}[1,2]) &= \max\{d_{\mathsf{GB}_{\mathsf{int}}}(fi, \{p, m, o\}), \\ &\quad d_{\mathsf{GB}_{\mathsf{int}}}(fi, \{eq, si\}), d_{\mathsf{GB}_{\mathsf{int}}}(fi, \{di\})\} \\ &= \max\{d_{\mathsf{GB}_{\mathsf{int}}}(fi, o), d_{\mathsf{GB}_{\mathsf{int}}}(fi, eq), d_{\mathsf{GB}_{\mathsf{int}}}(fi, di)\} \\ &= \max\{1, 1, 1\} = 1. \end{aligned}$$

La deuxième étape consiste, pour toute relation de base de B, à agréger sur l'ensemble des couples de variables (v_i, v_j), $i < j$, les distances locales calculées à l'étape précédente, afin d'obtenir une *distance globale* d_{g_2} entre un scénario σ de N^V_{All} ainsi généré et le profil $\mathcal{N}$. Cette distance utilise la fonction d'agrégation g_2 et est définie comme suit :

$$d_{g_2}(\sigma, \mathcal{N}) = g_2\{d_{f_2}(\sigma_{ij}, \mathcal{N}[i,j]) \mid v_i, v_j \in V, i < j\}.$$

Le choix de la fonction d'agrégation g_2 sur l'ensemble des paires de variables (v_i, v_j) du scénario σ est ici motivé de manière identique à celui de la fonction d'agrégation f_1 pour les opérateur Δ_1.

Exemple (suite). Considérons de nouveau le scénario cohérent σ_1 (cf. figure 5(a)) et choisissons $g_2 = \sum$. On a alors :

$$\begin{aligned} d_{\sum}(\sigma_1, \mathcal{N}) &= \sum\{d_{MAX}(\sigma_1(1,2), \mathcal{N}[1,2]), \ldots, \\ &\quad d_{MAX}(\sigma_1(3,4), \mathcal{N}[3,4])\} \\ &= 1 + 2 + 0 + 1 + 4 + 0 = 8. \end{aligned}$$

De manière similaire aux opérateurs Δ_1, le résultat de la fusion du profil $\mathcal{N}$ par l'opérateur $\Delta_2^{d_{\mathsf{B}}, f_2, g_2}$ correspond à l'ensemble des scénarios *cohérents* de N^V_{All} qui minimisent la distance globale d_{g_2}. Formellement,

$$\begin{aligned} \Delta_2^{d_{\mathsf{B}}, f_2, g_2}(\mathcal{N}) = \{\sigma \in [N^V_{All}] \mid \nexists \sigma' \in [N^V_{All}], \\ d_{g_2}(\sigma', \mathcal{N}) < d_{g_2}(\sigma, \mathcal{N})\}. \end{aligned}$$

Exemple (suite). Considérons de nouveau le scénario cohérent σ_2 représenté à la figure 5(c). Sa distance globale, calculée de manière similaire à celle de σ_1, est définie par $d_{\sum}(\sigma_2, \mathcal{N}) = 8$. Remarquons alors que les deux scénarios cohérents σ_1 et σ_2 ont une même distance globale, ce qui implique que $\sigma_1 \in \Delta_2^{d_{\mathsf{GB}_{\mathsf{int}}}, MAX, \sum}(\mathcal{N})$ ssi $\sigma_2 \in \Delta_2^{d_{\mathsf{GB}_{\mathsf{int}}}, MAX, \sum}(\mathcal{N})$.

Proposition 2. *$\Delta_2^{d_{\mathsf{B}}, f_2, g_2}$ est un opérateur de fusion de* RCQ *au sens de la définition 7, c'est-à-dire qu'il satisfait les postulats **(N1) et (N2)**. Les postulats **(N3) - (N6)** ne sont pas vérifiés dans le cas général.*

Preuve. (N1) Par définition. (N2) D'après la propriété de séparation de d_{B}, on a $\forall b \in \mathsf{B}, \forall R \in 2^{\mathsf{B}}, d_{\mathsf{B}}(b, R) = 0$ ssi $b \in R$. Soit $\mathcal{R}$ un multi-ensemble de relations de 2^{B}. D'après la minimalité de f_2, on a $d_{f_2}(b, \mathcal{R}) = 0$ ssi $b \in R \ \forall R \in \mathcal{R}$. Soit $\sigma \in [N^V_{All}]$. D'après la minimalité de g_2, on a $d_{g_2}(\sigma, \mathcal{N}) = 0$ ssi $\forall v_i, v_j, i < j, \sigma_{ij} \in N^k[i,j] \ \forall N^k \in \mathcal{N}$. Ceci signifie que $\sigma \in \Delta_2^{d_{\mathsf{B}}, f_2, g_2}(\mathcal{N})$ ssi $\sigma \in \bigcap\{[N^k] \mid N^k \in \mathcal{N}\}$. $\dashv$

3.4 Comparaison des opérateurs Δ_1 et Δ_2

Etant donné un profil $\mathcal{N}$, on choisira un opérateur Δ_1 lorsque les sources sont indépendantes, *i.e.,* lorsque l'information fournie par chaque RCQ du profil doit être traitée indépendamment. En effet, la première étape d'agrégation est « locale » à un RCQ particulier, la deuxième étape consiste alors à combiner les sources entre elles. Dans cet esprit, les opérateurs Δ_1 sont définis en s'inspirant des opérateurs de fusion de RCQ récemment proposés dans [5] et aux opérateurs de fusion de croyances $\mathbf{DA}^2$ [7] définis dans le cadre de la logique propositionelle. Dans [5] les opérateurs de fusion de RCQ, notés Θ, considèrent de même que les opérateurs Δ_1 un profil $\mathcal{N}$ de RCQ et renvoient en résultat un ensemble de scénarios cohérents suivant un processus similaire en deux étapes, en considérant systématiquement $f_1 = \sum$. Cependant alors que les opérateurs Δ_1 considèrent l'ensemble des scénarios des RCQ de $\mathcal{N}$ dans le calcul de la distance locale d_{f_1}, un opérateur Θ considère l'ensemble de leurs scénarios *cohérents*. Ainsi, à la différence d'un opérateur Δ_1, les RCQ incohérents de $\mathcal{N}$ ne sont pas considérés par un opérateur Θ, de même que les relations de base contenues dans les contraintes d'un RCQ qui ne participent à aucun scénario cohérent de ce RCQ. Dans [7] les auteurs définissent une classe d'opérateurs de fusion de bases de croyances propositionnelles nommée $\mathbf{DA}^2$, paramétrée par une distance entre interprétations et deux fonctions d'agrégation. Un profil correspond alors à un ensemble de bases de croyances, chacune étant représentée par un ensemble de formules propositionelles. Une première étape consiste alors à calculer une distance entre une interprétation et une base de croyances au moyen d'une première fonction d'agrégation sur l'ensemble des formules propositionnelles constituant la base de croyances, puis d'utiliser une deuxième fonction d'agrégation pour combiner les différentes bases de croyance du profil. Dans le cadre des RCQ, les opérateurs Δ_1 suivent typiquement le même principe de fusion.

Les opérateurs Δ_2 sont adaptés dans contexte où une décision globale doit être prise *a priori*, pour chaque paire de variables (v_i, v_j), de manière indépendante. Dans un tel cas une décision commune doit être prise en premier lieu pour chaque paire de variables, l'ensemble des paires de variables étant considéré comme un ensemble de « critères » indépendants. Les opérateurs Δ_2 considèrent une distance locale d_{f_2} qui coïncide avec celle proposée dans [4, 2]. Dans ces travaux, les auteurs utilisent cette distance locale d_{f_2} pour définir un *opérateur de fusion de contraintes*. Un tel opérateur associe à un ensemble de relations l'ensemble des relations de base minimisant la distance d_{f_2} à cet ensemble. Dans un tel cadre, un opérateur de fusion de RCQ, noté Ω, associe à un profil $\mathcal{N}$ de RCQ un RCQ unique $\Omega(\mathcal{N})$. Tout comme un opérateur Δ_2, un opérateur Ω prend en compte des RCQ éventuellement incohérents et considère l'ensemble des relations de base fournies dans chaque contrainte pour chaque RCQ de $\mathcal{N}$, même si ces dernières ne participent à aucun scénario cohérent du RCQ en question. Un ordre total $<_V$ sur les paires des variables (v_i, v_j) est supposé fixé au départ. En suivant l'ordre $<_V$, chaque contrainte portant sur (v_i, v_j) du RCQ $\Omega(\mathcal{N})$ est affectée à une relation en appliquant l'opérateur de fusion des contraintes des RCQ de $\mathcal{N}$ portant sur (v_i, v_j). A chaque étape la cohérence du RCQ $\Omega(\mathcal{N})$ est préservée. Bien que le calcul de $\Omega(\mathcal{N})$ soit efficace en pratique, il renvoit un résultat sous-optimal dans le cas général. Un opérateur Δ_2 suit un même objectif de fusion d'un profil $\mathcal{N}$, en garantissant toutefois un résultat cohérent optimal.

Aucun des opérateurs Δ_1 et Δ_2 ne satisfont les postulats d'indépendance syntaxique (N3) et d'équité (N4) (cf. propositions 1 et 2). Ceci est dû en particulier à la prise en compte par ces opérateurs des RCQ incohérents du profil. Les propositions 1 et 2 montrent également que les opérateurs Δ_1 satisfont un ensemble de propriétés logiques plus important que les opérateurs Δ_2. Par conséquent, ces opérateurs renvoient des résultats différents dans le cas général. Leur principale différence réside dans l'inversion des deux étapes d'agrégation. La proposition suivante exprime le fait que sous certaines conditions sur les fonctions d'agrégations utilisées, les opérateurs Δ_1 et Δ_2 sont équivalents.

Proposition 3. *Si $f_1 = g_2$, $f_2 = g_1$ et f_1 et f_2 sont intercommutatives, alors $\Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N}) = \Delta_2^{d_{\mathsf{B}}, f_2, g_2}(\mathcal{N})$.*

Preuve. Il suffit de montrer que les distances globales du calcul de $\Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N})$ et $\Delta_2^{d_{\mathsf{B}}, f_2, g_2}(\mathcal{N})$ coïncident. Soient $N^k \in \mathcal{N}$ et $\sigma \in [N^V_{All}]$. Soit h une fonction d'agrégation, notons $\alpha^k(h) = d_h(\sigma, N^k) = \min\{h\{d_{\mathsf{B}}(\sigma_{ij}, \sigma'_{ij}) \mid v_i, v_j \in V, i < j\} \mid \sigma' \in \langle N^k \rangle\}$ et $\beta^k(h) = h\{\min\{d_{\mathsf{B}}(\sigma_{ij}, b) \mid b \in N^k[i,j]\} \mid v_i, v_j \in V, i < j\}$. Montrons d'abord que $\alpha^k(f_1) = \beta^k(f_1)$.

Par monotonie de f_1, on a clairement $\alpha^k(f_1) \geq \beta^k(f_1)$, donc montrons $\alpha^k(f_1) \leq \beta^k(f_1)$. Soit σ' un scénario cohérent défini $\forall v_i, v_j$ par $\sigma'_{ij} = b$ tel que $b \in N^k[i,j]$ et $d_{\mathsf{B}}(\sigma_{ij}, b) = \min\{d_{\mathsf{B}}(\sigma_{ij}, b') \mid b' \in N^k[i,j]\}$. Par définition, $\sigma' \in \langle N^k \rangle$, donc $\beta^k(f_1) = f_1\{d_{\mathsf{B}}(\sigma_{ij}, \sigma'_{ij}) \mid v_i, v_j \in V, i < j\}$. Donc $\alpha^k(f_1) \leq \beta^k(f_1)$. La distance globale d_{g_1} entre σ et $\mathcal{N}$ dans le processus de calcul de $\Delta_1^{d_{\mathsf{B}}, f_1, g_1}(\mathcal{N})$ est définie comme étant $d_{g_1}(\sigma, \mathcal{N}) = g_1\{\alpha^k(f_1) \mid N^k \in \mathcal{N}\} = g_1\{\beta^k(f_1) \mid N^k \in \mathcal{N}\}$. Comme $f_1 = g_2$ et $f_2 = g_1$, on a $d_{g_1}(\sigma, \mathcal{N}) = f_2\{\beta^k(g_2) \mid N^k \in \mathcal{N}\}$. d_{g_2} est la distance globale dans le processus de calcul de $\Delta_1^{d_{\mathsf{B}}, f_2, g_2}(\mathcal{N})$; puisque f_2 et g_2 sont inter-commutatives, on a $d_{g_1}(\sigma, \mathcal{N}) = d_{g_2}(\sigma, \mathcal{N})$. ⊣

Par conséquent, cette proposition s'applique lorsque $f_1 = g_2$, $f_2 = g_1$ et par exemple lorsque $(f_1, f_2) \in \{(\sum, \sum), (MAX, MAX)\}$. Notons cependant que $\sum$ et MAX ne sont pas inter-commutatives, ainsi la proposition ne s'applique pas lorsque $(f_1, f_2) \in \{(\sum, MAX), (MAX, \sum)\}$.

Dans la section suivante, nous proposons une méthode pratique pour le calcul de $\Delta_2^{d_{\mathsf{B}}, f_2, g_2}(\mathcal{N})$.

4 Les réseaux de contraintes qualitatives pondérés

En s'inspirant de l'approche proposée dans le cadre des CSP discrets pondérés [13, 8], nous définissons les réseaux de contraintes qualitatives pondérés (RCQP en abrégé). Un RCQP est un RCQ particulier où est associé à chaque relation de base de chacune des contraintes une valeur numérique. Chacune de ces valeurs correspond à un coût : le coût associé à une relation de base pour une contrainte donnée est d'autant plus faible que cette relation de base est préférée pour cette contrainte. À un RCQP est également associée une fonction d'agrégation permettant d'évaluer le coût d'un scénario cohérent et d'ainsi caractériser ses solutions optimales.

Définition 9 (Réseau de contraintes qualitatives pondérés). *Un* RCQP *défini sur* 2^{B} *est un quadruplet* $(V, C, \pi, \oplus)$ *où*

- (V, C) *est un* RCQ N *défini sur* 2^{B},
- π *une est application associant à chaque couple* $(v_i, v_j) \in V \times V, i < j$ *une fonction qui associe à chaque élément* $b \in N[i,j]$ *une valeur réelle positive notée* $\pi_{ij}(b)$,
- $\oplus$ *est une fonction d'agrégation.*

Soit $P = (V, C, \pi, \oplus)$ un RCQP défini sur 2^{B}. Un scénario de P est un scénario du RCQ (V, C). Une solution de P est un scénario cohérent de P. $[P]$ dénote l'ensemble des solutions de P. Soit σ un scénario de P. Le coût de σ, noté $\mathcal{V}(\sigma)$, est défini par $\oplus\{\pi_{ij}(\sigma_{ij}) \mid v_i, v_j \in V, i < j\}$. L'ensemble des solutions optimales de P est noté $\mathsf{Sols}(P)$.

La procédure $SearchSols$ permet de calculer l'ensemble des solutions optimales d'un RCQP $P = (V, C, \pi, \oplus)$ à l'aide d'une recherche de type *Branch and Bound*. N est le RCQ (V, C). À chaque point de la recherche, l'ensemble Sols contient l'ensemble des solutions de meilleur coût trouvées jusqu'à présent. Ce coût est stocké par la variable UB. Cette dernière doit être initialisée avec une valeur strictement supérieure au coût de toute solution de P. La valeur $1 + \oplus\{\max\{\pi_{ij}(b) : b \in N[i,j]\} : v_i, v_j \in V, i < j\}$ peut être par exemple utilisée. Lorsque N est un scénario, un test de cohérence de N est réalisé à l'aide de la methode de la fermeture par faible composition (ligne 2) pour vérifier que N est solution de P. Notons que toute autre méthode peut être utilisée pourvue que celle-ci soit complète pour la problème de la cohérence d'un scénario sur 2^{B}. De manière générique, nous utilisons la fermeture par faible composition du fait que pour la plupart des formalismes qualitatifs cette méthode est complète pour les scénarios. Le coût de N est forcément inférieur ou égal à UB. En effet, soit le RCQP P dont est issu N est le RCQP initial sur lequel a été appelée la procédure $SearchSols$, soit P est issu d'un appel récursif (ligne 16) qui n'a lieu que dans le cas où $\oplus\{\min\{\pi_{ij}(b) : b \in N[i,j]\} : v_i, v_j \in V, i < j\} \leq \mathsf{UB}$. Puisque N est un scénario, dans les deux cas nous avons $\mathcal{V}(N) \leq \mathsf{UB}$. Il s'ensuit que N peut être rajoutée à l'ensemble des meilleures solutions. Le cas où N n'est pas un scénario est pris en compte à la ligne 9. Une contrainte $N[i,j]$ non singleton est alors sélectionnée. La contrainte $N[i,j]$ est alors substituée successivement par chacune des relations de base la composant (lignes 12–16). La méthode de filtrage qu'est la fermeture par faible composition est alors réalisée sur N. Dans le cas où N n'a pas été détecté non cohérent, une mise à jour de la fonction de coût π est réalisée afin de prendre en compte les relations de base supprimées sur N par le filtrage précédent. Un appel récursif de $SearchSols$ sur P est alors réalisé.

Proposition 4. *Si la méthode de la fermeture par faible composition est complète pour les scénarios définis sur* 2^{B} *alors la procédure* $SearchSols$ *est saine et complète pour le problème de trouver l'ensemble des solutions optimales d'un* RCQP *sur* 2^{B}.

Nous montrons maintenant que le résultat d'un opérateur de fusion de la classe Δ_2 sur un profil $\mathcal{N}$ peut être représenté au travers des solutions d'un RCQP particulier. On définit l'application RCQP_{Δ_2} qui associe à un opérateur de Δ_2 et un profil $\mathcal{N}$ un RCQP particulier, comme suit :

Définition 10. *Soit* $\Delta_2^{d_{\mathsf{B}}, f_2, g_2}$ *un opérateur de fusion de la classe* Δ_2 *et un profil* $\mathcal{N}$ *de* RCQ *définis sur* V. $\mathsf{RCQP}_{\Delta_2}(\Delta_2^{d_{\mathsf{B}}, f_2, g_2}, \mathcal{N})$ *est le* RCQP $(V', C, \pi, \oplus)$ *défini par :*

- $V' = V$,
- $\forall v_i, v_j \in V$, $N[i,j] = \mathsf{B}$, *avec* $(V, C) = N$,
- $\forall v_i, v_j \in V$, $i < j$, $\forall b \in B$, $\pi_{ij}(b) = f_2\{d_{\mathsf{B}}(b, N^k[i,j]) \mid N^k[i,j] \in \mathcal{N}[i,j]\}$,
- $\oplus = g_2$.

Procedure `SearchSols(`P`)`

Entrée: Un RCQP $P = (V, C, \pi, \oplus)$
// N dénote le RCQ (V, C)
Sortie: SolsOpts est l'ensemble $\mathsf{Sols}(P)$

```
1  if N est un scénario then
2      if ⋄-fermeture(N) then
3          if V(N) < UB then
4              UB ← V(N)
5              Sols ← ∅
6          Sols ← Sols ∪ {N}
7
8  else
9      P' ← P
10     Selectionner N[i,j] telle que |N[i,j]| ≠ 1
11     foreach b ∈ N[i,j] do
12         N[i,j] ← {b}
13         if ⋄-fermeture(N) then
14             Mettre à jour π à partir de N
15             if ⊕{min{π_ij(b) : b ∈ N[i,j]} : v_i, v_j ∈
               V, i < j} ≤ UB then
16                 SearchSols(P)
17
18         P ← P'
```

Le résultat du processus de fusion sur un ensemble de RCQ à l'aide d'un opérateur de fusion de la classe Δ_2 peut être obtenu en résolvant le RCQP associé par l'application RCQP_{Δ_2}. En effet, nous avons la proposition suivante :

Proposition 5. *Soit $\Delta_2^{d_B, f_2, g_2}$ un opérateur de fusion appartenant à Δ_2 et un profil $\mathcal{N} = \{N^1, \ldots, N^m\}$. Nous avons* $\mathsf{Sols}(\mathsf{RCQP}_{\Delta_2}(\Delta_2^{d_B, f_2, g_2}, \mathcal{N})) = \Delta_2^{d_B, f_2, g_2}(\mathcal{N})$.

Preuve. Evident par définition. ⊣

Une conséquence des propositions 3 et 5 est que sous les conditions de la proposition 3, la procédure $SearchSols$ permet également de calculer les scénarios cohérents de $\Delta_1^{d_B, f_1, g_1}(\mathcal{N})$.

5 Conclusion

Dans cet article, nous avons défini deux classes d'opérateurs de fusion de réseaux de contraintes qualitatives (RCQ) définis sur une même algèbre qualitative. A la différence des approches de fusion de RCQ existant dans la littérature, ces opérateurs considèrent l'information fournie par chacune des contraintes des RCQ fournis en entrée, permettant ainsi de considérer des RCQ représentant des croyances ou des préférences locales sur les positions relatives d'entités spatiales ou temporelles. Nous avons étudié les propriétés logiques de chacune de ces deux classes d'opérateurs. Nous avons enfin introduit les RCQ pondérés, montré que les solutions d'un RCQ pondéré particulier correspond au résultat renvoyé par l'un de ces opérateurs, et proposé un algorithme pour calculer ces solutions.

Ce travail donne lieu à des perspectives multiples. Il serait d'abord intéressant de définir des méthodes de propagation de contraintes pour les RCQ pondérés, en s'inspirant notamment des travaux réalisés dans le cadre des CSP pondérés. Pour de nombreuses algèbres qualitatives des classes traitables ont été identifiées dans la littérature pour résoudre le problème de la cohérence des RCQ. Un travail futur est alors d'exploiter ces classes traitables pour la résolution des RCQ pondérés. Par ailleurs, une étude théorique et expérimentale doit être menée afin de comparer la complexité de mise en œuvre des différents opérateurs de fusion de RCQ proposés à ce jour. La plupart de ces opérateurs de fusion renvoient un résultat de taille exponentielle au nombre de variables, les opérateurs définis dans cet article n'en faisant pas exception. Une autre perspective de recherche est alors de caractériser la taille de l'ensemble résultant en fonction des RCQ fournis et de l'opérateur choisi.

Références

[1] J-F. Allen. An interval-based representation of temporal knowledge. In *IJCAI'81*, pages 221–226.

[2] J-F. Condotta, S. Kaci, P. Marquis, and N. Schwind. Merging qualitative constraint networks in a piecewise fashion. In *ICTAI'09*, pages 605–608.

[3] J-F. Condotta, S. Kaci, P. Marquis, and N. Schwind. Merging qualitative constraints networks using propositional logic. In *ECSQARU'09*, pages 347–358.

[4] J-F. Condotta, S. Kaci, P. Marquis, and N. Schwind. Fusion de réseaux de contraintes qualitatives par morceaux. In *JFPC'09*, 2009.

[5] J-F. Condotta, S. Kaci, and N. Schwind. A Framework for Merging Qualitative Constraints Networks. In *FLAIRS'08*, pages 586–591.

[6] Christian Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54(1) :199–227, 1992.

[7] S. Konieczny, J. Lang, and P. Marquis. $\mathbf{DA}^2$ merging operators. *Artificial Intelligence*, 157(1-2) :49–79, 2004.

[8] J. Larrosa and T. Schiex. Solving weighted CSP by maintaining arc consistency. *Artificial Intelligence*, 159(1-2) :1–26, 2004.

[9] J. Lin. Integration of weighted knowledge bases. *Artificial Intelligence*, 83(2) :363–378, 1996.

[10] J.-L. Marichal. *Aggregation Operators for Multicriteria Decision Aid*. PhD thesis, Institute of Mathematics, University of Liège, Liège, Belgium, 1998.

[11] P. Z. Revesz. On the Semantics of Arbitration. *Journal of Algebra and Computation*, 7 (2) :133–160, 1997.

[12] S. Saminger-Platz, R. Mesiar, and D. Dubois. Aggregation operators and commuting. *IEEE T. Fuzzy Systems*, 15(6) :1032–1045, 2007.

[13] T. Schiex. Arc consistency for soft constraints. In *CP'00*, pages 411–424.

Une nouvelle architecture parallèle pour le problème de validité des QBF

Benoit Da Mota, Pascal Nicolas, Igor Stéphan

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

Résumé

Dans ce papier, nous présentons une nouvelle architecture parallèle ouverte pour répondre à différents problèmes portant sur les formules booléennes quantifiées, dont la validité. La principale caractéristique de notre approche est qu'elle est basée sur un découpage syntaxique de la formule pour le traitement de sous-formules indépendantes. Elle est liée au choix de traiter des formules booléennes quantifiées sans restriction syntaxique, comme la forme prénexe ou la forme normale conjonctive. Dans ce cadre parallèle général ouvert, nous sommes capables d'introduire différents oracles, sous la forme d'algorithmes séquentiels pour l'élimination de quantificateurs. Nous présentons nos premières expérimentations en instanciant un unique oracle et en rapportons les résultats.

1 Introduction

Le problème de validité pour les formules booléennes quantifiées (QBF) est une généralisation du problème de satisfiabilité pour les formules booléennes. Tandis que décider de la satisfiabilité des formules booléennes est NP-complet, décider de la validité des QBF est PSPACE-complet. C'est le prix à payer pour une représentation plus concise pour de très nombreuses classes de formules. Une multitude d'importants problèmes de décision parmi des champs très divers ont des transformations polynomiales vers le problème de validité des QBF : planification [22, 2], construction de modèles bornés [2], vérification formelle (voir [4] pour une vue d'ensemble).

La plupart des procédures récentes et efficaces pour décider de la validité des QBF, nécessitent d'avoir en entrée un formule sous forme normale négative ou dans une forme encore plus restrictive comme la forme normale conjonctive. Mais il est rare que les problèmes s'expriment directement sous ces formes qui détruisent complètement la structure originale des problèmes. Il est plus naturel d'utiliser toute l'expressivité du langage QBF : tous les connecteurs logiques usuels, y compris l'implication, la bi-implication et le ou-exclusif, ainsi que la possibilité d'utiliser des quantificateurs à l'intérieur de la formule. Aussi, notre premier objectif est de développer un solveur QBF sans restriction sur les formules en entrée.

Depuis quelques années, la fréquence du ou des cœurs des processeurs ne progresse plus, voire même diminue. En contrepartie, le nombre de cœurs par processeur augmente et le domaine de la programmation parallèle est en pleine effervescence : multithread, multi-CPU, GPU computing, HPC, grid computing, cloud computing, etc. Certaines procédures de résolution profitent déjà de ce nouveau potentiel et les futurs solveurs devront en faire de même pour rester compétitifs. Le second objectif de notre travail est d'exploiter les opportunités qu'offre la programmation parallèle pour s'attaquer au problème de validité des QBF en réutilisant, adaptant, améliorant ou combinant des algorithmes séquentiels pour QBF à l'intérieur d'une architecture parallèle.

Par ailleurs, au delà du problème de validité, nous nous intéressons aussi à d'autres problèmes en rapport avec QBF, tels celui de leur compilation pour une représentation plus compacte et un accès de complexité moindre aux solutions. Dans ce cas, une procédure de décision ne suffit plus (la réponse du système n'étant plus simplement « oui » ou « non »), mais un ensemble de formules décrivant l'espace des solutions pour les symboles propositionnels existentiellement quantifiés. Aussi dans ce but, notre objectif à long terme est de réaliser une architecture aussi ouverte que possible pour offrir à l'utilisateur final un ensemble d'outils en rapport avec les QBF.

Ce présent article est structuré ainsi : après un rap-

pel des notions fondamentales concernant la logique propositionnelle, le problème SAT et la définition de la syntaxe et de la sémantique des QBF, la section 3 dresse un court panorama des procédures de décision pour le problème de validité des QBF dans le cadre des solveurs séquentiels puis parallèles. La section 4 décrit notre nouvelle architecture parallèle pour le problème de validité des QBF comme une architecture maître/esclave dont le maître assure la répartition de tâches aux esclaves obtenues par un découpage syntaxique de la formule initiale. La section 5 propose une instance de notre nouvelle architecture selon un mode client/serveur : le client est une implantation de l'esclave basée sur la procédure de décision par oracle `QSAT` ; cette section détaille les choix techniques ainsi que les résultats préliminaires de notre approche. La section 6 dresse une conclusion et des perspectives à ce travail.

2 Préliminaires

2.1 Logique propositionnelle

L'ensemble des valeurs booléennes **vrai** et **faux** est noté **BOOL**. L'ensemble des symboles propositionnels est noté $\mathcal{SP}$. Les symboles $\top$ et $\bot$ sont les constantes booléennes. Le symbole $\wedge$ est utilisé pour la conjonction, $\vee$ pour la disjonction, $\neg$ pour la négation, $\rightarrow$ pour l'implication, $\leftrightarrow$ pour la bi-implication et $\oplus$ pour le ou-exclusif. L'ensemble des opérateurs binaires $\{\wedge, \vee, \rightarrow, \leftrightarrow, \oplus\}$ est noté $\mathcal{O}$. L'ensemble **PROP** des formules propositionnelles est défini inductivement ainsi : tout symbole propositionnel ou constante propositionnelle est élément de **PROP** ; si F est élément de **PROP** alors $\neg F$ est élément de **PROP** ; si F et G sont éléments de **PROP** et $\circ$ est élément de $\mathcal{O}$ alors $(F \circ G)$ est élément de **PROP**. Un littéral est un symbole propositionnel ou la négation de celui-ci. Une clause est une disjonction de littéraux. Une formule propositionnelle est sous forme normale négative (FNN) si la formule n'est constituée exclusivement que de conjonctions, disjonctions et littéraux. Une formule propositionnelle est sous forme normale conjonctive (FNC) si c'est une conjonction de disjonctions de littéraux. Toute formule sous FNC est une formule sous FNN. Une valuation v est une fonction de $\mathcal{SP}$ dans **BOOL** (l'ensemble des valuation est noté **VAL**).

2.2 Syntaxe des formules booléennes quantifiées

Le symbole $\exists$ est utilisé pour la quantification existentielle et $\forall$ pour la quantification universelle (q est utilisé pour noter un quantificateur quelconque). L'ensemble **QBF** des formules booléennes quantifiées est défini inductivement ainsi : si F est un élément de **PROP** alors c'est un élément de **QBF** ; si F est un élément de **QBF** et x est un symbole propositionnel alors $(\exists x\ F)$ et $(\forall x\ F)$ sont des éléments de **QBF** ; si F est élément de **QBF** alors $\neg F$ est élément de **QBF** ; si F et G sont éléments de **QBF** et $\circ$ est élément de $\mathcal{O}$ alors $(F \circ G)$ est élément de **QBF**. Un symbole propositionnel x est libre s'il n'apparaît pas sous la portée d'un quantificateur $\exists x$ ou $\forall x$. Une QBF est close si elle ne contient pas de symbole propositionnel libre. Une substitution est une fonction de l'ensemble des symboles propositionnels dans l'ensemble des formules (quantifiées ou non). Nous définissons la substitution de x par F dans G, notée $[x \leftarrow F](G)$, comme étant la formule obtenue de G en remplaçant toutes les occurrences du symbole propositionnel x par la formule F sauf pour les occurrences de x sous la portée d'un quantificateur portant sur x. Un lieur est une chaîne de caractères $q_1 x_1 \dots q_n x_n$ avec $x_1, \dots, x_n$ des symboles propositionnels distincts et $q_1, \dots, q_n$ des quantificateurs. Une QBF QM est sous forme prénexe si Q est un lieur et M est une formule booléenne, appelée matrice. Une QBF prénexe QM est sous forme normale négative (resp. conjonctive) si M est une formule booléenne en FNN (resp. FNC).

2.3 Sémantique des formules booléennes quantifiées

La sémantique des QBF présentée fait appel à la sémantique des (constantes et) opérateurs booléens qui est définie de manière habituelle, en particulier, à chaque (constante et) opérateur (resp. $\top$, $\bot$, $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, $\oplus$) est associée une fonction booléenne (resp. $i_\top, i_\bot :\rightarrow$ **BOOL**, $i_\neg$: **BOOL** $\rightarrow$ **BOOL**, $i_\wedge, i_\vee, i_\rightarrow, i_\leftrightarrow, i_\oplus$: **BOOL** $\times$ **BOOL** $\rightarrow$ **BOOL**) qui en définit sa sémantique ; la sémantique des QBF fait aussi appel à une sémantique pour les quantificateurs :

$$I^x_\exists, I^x_\forall : (\mathbf{VAL} \rightarrow \mathbf{BOOL}) \times \mathbf{VAL} \rightarrow \mathbf{BOOL}$$

$$I^x_\exists(f)(v) = i_\vee(f(v[x := \mathbf{vrai}]), f(v[x := \mathbf{faux}]))$$

$$I^x_\forall(f)(v) = i_\wedge(f(v[x := \mathbf{vrai}]), f(v[x := \mathbf{faux}]))$$

enfin elle est définie inductivement :

- $[[\bot]](v) = i_\bot$;
- $[[\top]](v) = i_\top$;
- $[[x]](v) = v(x)$ si $x \in \mathcal{SP}$;
- $[[(F \circ G)]](v) = i_\circ([[F]](v), [[G]](v))$ si $F, G \in$ **QBF** et $\circ \in \mathcal{O}$;
- $[[\neg F]](v) = i_\neg([[F]](v))$ si $F \in$ **QBF** ;
- $[[(\exists x\ F)]](v) = I^x_\exists([[F]])(v)$ si $F \in$ **QBF** ;
- $[[(\forall x\ F)]](v) = I^x_\forall([[F]])(v)$ si $F \in$ **QBF**.

Une QBF close F est valide si $[[F]](v) =$ **vrai** pour toute valuation v. Par exemple la QBF

$\exists a \exists b \forall c((a \vee b) \leftrightarrow c)$ n'est pas valide tandis que la QBF $\forall c \exists a \exists b((a \vee b) \leftrightarrow c)$ l'est. Cet exemple montre que l'ordre des quantificateurs est crucial pour décider de la validité d'une QBF.

Comme dans le cas propositionnel, une relation d'équivalence notée $\equiv$ est définie pour les QBF par $F \equiv G$ si $[[F]](v) = [[G]](v)$ pour toute valuation v. En lien avec l'exemple qui précède, $\exists x \exists y F \equiv \exists y \exists x F$ et $\forall x \forall y F \equiv \forall y \forall x F$ mais $\exists a \exists b \forall c((a \vee b) \leftrightarrow c) \not\equiv \forall c \exists a \exists b((a \vee b) \leftrightarrow c)$.

Enfin, rappelons que le problème (SAT) consistant à décider si une formule booléenne est satisfiable ou non est le problème canonique de la classe NP-complet. De son côté, le problème consistant à décider si une formule booléenne quantifiée est valide ou non est le problème canonique de la classe PSPACE-complet [28].

3 État de l'art des solveurs sequentiels/parallèles pour QBF

État de l'art des procédures de décision pour le problème de validité des QBF. Puisque la vérification d'un modèle pour une QBF prénexe est co-NP-complet [16], il y a très peu de procédures incomplètes basées sur des métaheuristiques. Pour autant que nous sachions, il y en a deux qui sont basées sur la recherche locale : `WalkQSAT` [12] (basé sur `WalkSAT` [25]) et `QBDD(LS)` [1]. Ainsi, la plupart des procédures pour le problème de validité des QBF sont des procédures de décision et elles peuvent être partitionnées en trois catégories : les procédures dites « monolithiques » qui se suffisent à elles-mêmes, les procédures qui transforment le problème dans un autre formalisme possédant une procédure de décision et les procédures qui exploitent un oracle.

Dans la première catégorie, les procédures sont basées sur la résolution telles que `QKN` [15], sont des procédures par élimination de symboles propositionnels à la Fourier-Motzkin telles que `quantor` [5] (pour des QBF FNC) ou `Nenofex` [19] (pour des QBF FNN) comme extension de l'algorithme de Davis et Putnam [8], ou sont des procédures de recherche par élimination des quantificateurs les plus externes telles que `Evaluate` [6], `decide` [23], `QUBE` [14] ou `QSOLVE` [10] (toutes pour des QBF FNC) ou `qpro` [9] (pour des QBF FNN) comme extension de l'algorithme de Davis, Logemann et Loveland [7].

Une procédure basée sur une transformation traduit la QBF dans un formalisme qui dispose déjà d'une procédure de décision efficace : SAT (en FNC) pour `sKizzo` [3] ou ASP [27] (dans ces deux cas avec une croissance potentiellement exponentielle de la formule).

Les procédures avec oracle reviennent à l'idée initiale de la « hiérarchie polynomiale » [20] en ce qu'un oracle capable de résoudre un sous-problème avec une complexité moindre est nécessaire : `QBDD(DLL)` [1] utilise un oracle NP-complet et `QSAT` [21] utilise deux oracles dont l'un est NP-complet et l'autre co-NP-complet. Cette dernière procédure est détaillée au paragraphe 5 car l'implantation de notre modèle parallèle s'instancie grâce à `QSAT`.

État de l'art pour les solveurs parallèles. Autant que nous le sachions, il existe trois implantations de solveurs parallèles pour le problème de validité des QBF : `PQSOLVE` [10], `PaQube` [17] et `QMiraXT` [18]. Nous faisons deux remarques importantes :

- Ces procédures sont toutes dédiées aux QBF sous FNC prénexes.
- Ces procédures sont toutes basées sur un algorithme de recherche séquentiel par élimination du quantificateur le plus externe vers le plus interne (`QSOLVE` [10] pour `PQSOLVE`, `QUBE` [14] pour `PaQube` et `PaMiraXT` [24], un solveur SAT parallèle, pour `QMiraXT`) et ainsi appliquent une *stratégie de partitionnement sémantique* qui choisit un symbole propositionnel du bloc de quantificateurs le plus externe et applique la sémantique des quantificateurs pour partitionner le problème et distribuer les tâches : si la QBF dont on désire décider de la validité est $qxQM$ alors les deux tâches $Q[x \leftarrow \top](M)$ et $Q[x \leftarrow \bot](M)$ sont distribuées.

La procédure `PQSOLVE` est un solveur distribué qui utilise les techniques de la parallélisation des programmes d'échec [10]. Il instancie un modèle pair-à-pair : un processus inactif demande du travail à un processus choisi au hasard et en devient l'esclave pour une tâche. Chaque processus a une pile de tâches à réaliser qui est augmentée par le partitionnement sémantique de celles qui sont estimées trop complexes ; le maître envoie une de ces tâches à son esclave momentané qui a sollicité une tâche à réaliser. Un esclave peut devenir lui-même le maître d'un autre processus.

La procédure `QMiraXT` est dédiée à la prise en compte du potentiel de performance des architectures modernes multi-cœur et/ou processeurs multithreadés. En utilisant un solveur threadé à mémoire partagée, les clauses apprises par conflit [29] sont partagées en les différents espaces de recherche. Il n'y a pas de processus maître mais à la place un « Master Control Object » (MCO) qui permet aux threads de communiquer via des messages asynchrones pour des évènements globaux (par exemple, si un sous-problème est valide ou non). Le MCO prend en charge aussi la stratégie de partitionnement sémantique, appelée dans ce cadre « Single Quantification Level Scheduling » (ou SQLS)

et distribue les tâches.

La procédure `PaQube` est conçue selon le modèle maître/esclave où un processus est dédié au maître (ce qui ne nécessite pas de CPU dédiée) et les autres aux esclaves qui réalisent en fait la recherche. `PaQube` est un solveur QBF parallèle basé sur MPI. Au travers de messages, les esclaves partagent certaines des clauses et cubes issus respectivement des conflits et solutions appris. Ainsi, chaque esclave cumule localement toute l'expertise obtenue par l'ensemble des autres esclaves. Enfin, le travail principal du maître est de réaliser, comme pour `QMiraXT`, la SQLS.

De par leur modèle, `PQSOLVE` et `PaQube` sont plus extensibles aux clusters et grids que `QMiraXT` mais ce dernier solveur semble tenir plus compte de l'évolution du matériel que les deux premiers.

4 La parallélisation syntaxique

Les solveurs parallèles présentés dans l'état de l'art de la section précédente sont tous basés sur un partitionnement sémantique des tâches distribuées selon une « architecture de parallélisation sémantique ». Nous proposons une nouvelle approche : l'« architecture de parallélisation syntaxique ». Celle-ci n'est pas basée sur le partitionnement de l'espace de recherche selon la sémantique des quantificateurs mais selon la localité de l'information dans les QBF non prénexes (et non FNC) représentée par des sous-formules quantifiées à symboles propositionnels libres.

4.1 L'extraction syntaxique de sous-problèmes

L'extraction syntaxique de sous-problèmes consiste à chercher des sous-formules que l'on peut traiter de manière quasiment indépendante, c-à-d. partageant peu de symboles propositionnels libres avec le reste de la QBF. Une sous-formule extraite est réduite par élimination des quantificateurs grâce à un oracle qui peut prendre des formes diverses selon les instanciations de l'architecture (comme nous le verrons à la section suivante où la procédure de décision `QSAT` est choisie) et qui en calcule une formule équivalente.

Par exemple, pour la QBF

$$(\exists a\ (a{\rightarrow}(\forall b\ (b{\rightarrow}((\forall c\ (c{\vee}b)){\wedge}(\exists d\ (a{\wedge}{\neg}d)))))))$$

l'extraction syntaxique calcule l'ensemble $\{(\forall c\ \ (c{\vee}b)), (\exists d\ \ (a{\wedge}{\neg}d))\}$ comme étant les sous-problèmes qui peuvent être traités séparément. Nous appelons *largeur syntaxique* la taille de l'ensemble obtenu par cette extraction syntaxique (celle-ci ne variant pas d'une exécution à l'autre, ceci permet de limiter les hypothèses lors de l'étude des résultats expérimentaux). Les oracles invoqués rendent les formules $b \equiv (\forall c\ (c{\vee}b))$ et $a \equiv (\exists d\ (a{\wedge}{\neg}d))$; ces formules sont réintroduites dans la QBF originale pour obtenir $(\exists a\ \ (a{\rightarrow}(\forall b\ \ (b{\rightarrow}(b{\wedge}a)))))$; à nouveau l'extraction syntaxique calcule un ensemble $\{(\forall b\ \ (b{\rightarrow}(b{\wedge}a)))\}$; l'oracle invoqué rend la formule $a \equiv (\forall b\ (b{\rightarrow}(b{\wedge}a)))$; finalement cette formule est réinjectée dans la formule intermédiaire pour obtenir la QBF $(\exists a\ (a{\rightarrow}a))$ qui est alors démontrée comme étant valide car équivalent à $\top$.

4.2 Un modèle maître/esclave

Une manière d'instancier l'architecture de parallèlisation syntaxique est de choisir un modèle maître/esclave. Ainsi le maître, dont le fonctionnement est illustré dans la figure 1 qui décrit la boucle générale d'exécution, est chargé d'assurer la répartition de tâches et la complétude de la recherche. Il lit la formule originale, extrait syntaxiquement des sous-problèmes, les distribue aux esclaves, puis attend les réponses et réinsère les résultats dans la formule d'origine. Pour le problème de validité des QBF, il suffit de répéter l'opération jusqu'à obtenir $\top$ ou $\bot$. Le rôle d'un esclave est alors d'accepter une QBF prénexe non FNC avec des symboles propositionnels libres et de répondre par un formule booléenne non quantifiée équivalente, uniquement composée des symboles propositionnels libres.

A chaque cycle d'extraction de sous-formules nous pouvons attribuer une *largeur syntaxique*. Nous appelons *largeur syntaxique maximale*, la plus grande valeur de largeur atteinte. Ce nombre traduit le besoin maximal en esclaves pour une formule et un découpage syntaxique donné. De même, nous appelons *largeur syntaxique minimale*, la plus petite valeur de largeur rencontrée. Selon le découpage syntaxique choisi, l'approche parallèle pourrait très vite être limitée. Tout esclave au delà de la largeur syntaxique maximale serait inutile. Or, de nombreux problèmes possèdent une largeur maximale de 1 ou 2 avec le découpage présenté dans la section 4.1.

Avant d'appliquer une méthode de résolution, le client utilise un heuristique de découpage (client splitting heuristic). Il a la responsabilité de créer des sous-problèmes si le problème qu'il reçoit lui parait trop difficile. Le client peut par exemple effectuer un découpage sémantique sur les symboles propositionnels libres et renvoyer un ensemble de tâches au maître. Si tous les clients sont occupés et que le problème est difficile, le client rajoute des tâches en file d'attente, si le problème semble facile, il le résout a priori rapidement et attend une nouvelle tâche. Si des esclaves sont en attente de travail et que le problème est difficile, le nouveau découpage va permettre au maître de

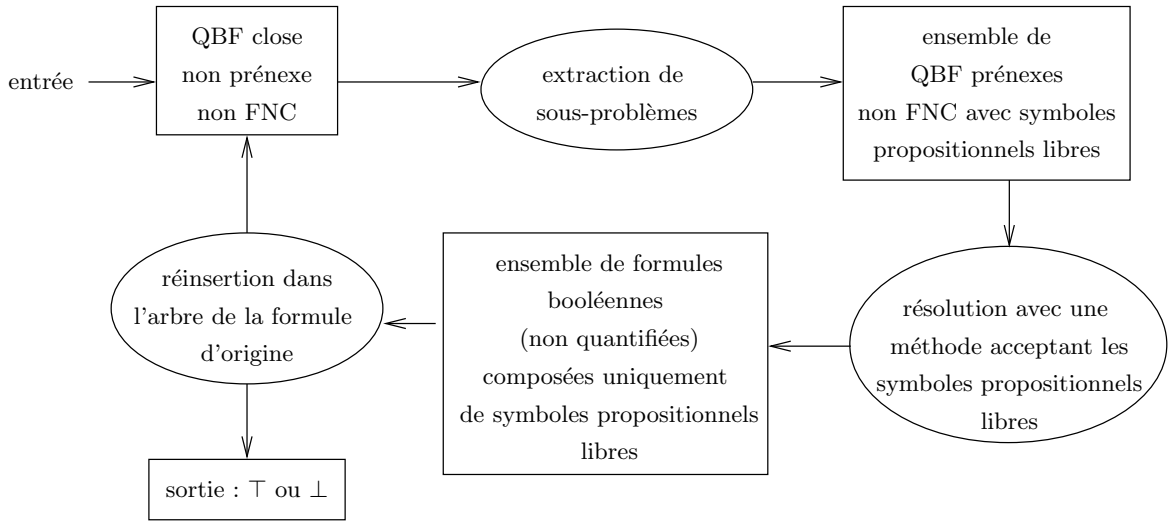


FIG. 1 – Boucle générale d'exécution.

redonner des tâches.

5 Une première instance de l'architecture de parallélisation syntaxique

Dans l'architecture présentée dans cet article, la tâche des esclaves est de calculer pour une QBF, une formule propositionnelle équivalente au sens de la préservation des modèles ; la QBF étant une sous-formule d'une QBF plus large, celle là est remplacée dans cette dernière par la proposition générée. Nous avons choisi pour instancier notre architecture parallèle un principe client/serveur sur un cluster de nœuds de calcul[1] et une procédure de l'état de l'art qui applique un partitionnement syntaxique de l'espace de recherche avec un oracle : la procédure `QSAT` [21]. Nous décrivons tout d'abord la procédure `QSAT` puis les résultats expérimentaux obtenus.

5.1 La procédure `QSAT`

`QSAT` [21] est une procédure de décision pour les QBF par élimination de quantificateurs, des plus internes vers les plus externes. Cette procédure opère itérativement sur une formule $Q(\exists x\ (F \wedge G))$ telle que x n'apparaît pas dans F ; l'équivalence $(\exists x\ (F \wedge G)) \equiv (F \wedge (\exists x\ G))$ permet d'isoler la sous formule $(\exists x\ G)$ qui va être mise en équivalence logique par une procédure *simp* avec une formule G' ne contenant pas le symbole propositionnel x ; par substitution des équivalents $Q(\exists x\ (F \wedge G)) \equiv Q(F \wedge G')$. Le procédé est similaire avec une quantification universelle. Le processus est itéré jusqu'à élimination de tous les quantificateurs. La procédure *simp* a besoin d'une procédure de décision pour le problème SAT et d'une procédure de décision pour le problème TAUT pour être effective. Cette procédure construit une FNC G' sur les symboles propositionnels libres de G telle que $G' \equiv (\exists x\ G)$. Elle opère comme un Davis-Logemann-Loveland par séparation de l'espace de recherche sur un symbole propositionnel libre y de G par appel récursif sur $[y \leftarrow \top](G)$ et $[y \leftarrow \bot](G)$. Si $[y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)$ est insatisfiable alors $simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)))$ retourne la clause $((y_1 \oplus C_1) \vee \dots \vee (y_n \oplus C_n))$; sinon si $[y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)$ est une tautologie alors $simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)))$ retourne $\top$; sinon si $(\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G))$ ne contient plus de symbole propositionnel libre alors soit cette formule est insatisfiable (et ce cas a déjà été traité) soit elle est satisfiable et $simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n](G)))$ retourne $\top$; sinon un nouveau symbole propositionnel libre y_{n+1} est considéré et $G_\top = simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n][y_{n+1} \leftarrow \top](G)))$ et $G_\bot = simp((\exists x\ [y_1 \leftarrow C_1] \dots [y_n \leftarrow C_n][y_{n+1} \leftarrow \bot](G)))$ sont calculés, $(G_\top \wedge G_\bot)$ est retourné. L'algorithme de la procédure *simp* est basé sur la construction classique de la FNC comme étant la conjonction de la négation des lignes de la table de vérité (considérées comme des cubes) qui falsifient la négation de la formule. Seule l'implantation d'une double restriction de `QSAT` est décrite dans [21] : restrictions à SAT et aux formules sous FNC. Dans le cadre de cette dernière restriction, la recherche d'une forme $Q(\exists x\ (F \wedge G))$ telle que x n'apparaît pas dans

[1] Ainsi, par la suite, nous utiliserons de manière indifférenciée les termes « serveur » ou « maître » et « client » ou « esclave ».

F est triviale ; seul le choix de x demeure et permet l'édification de stratégies.

5.2 Description du client

D'un point de vue général, un client reçoit une QBF quelconque avec des symboles propositionnels libres et doit renvoyer une formule booléenne non quantifiée composée uniquement de ces symboles propositionnels libres. La procédure QSAT, décrite dans le paragraphe précédent, s'y prête bien. De plus, il est possible d'éliminer plusieurs quantificateurs consécutifs du même type. Pour une QBF prénexe, il faudra autant d'itérations de la procédure QSAT que d'alternances de quantificateurs. La procédure QSAT possède un inconvénient majeur, elle est efficace sur une certaine catégorie de formules dites longues et fines (long and thin [21]). C'est pourquoi, d'une manière générale, notre architecture prévoit qu'un client puisse renoncer à résoudre une tâche qu'il juge trop difficile. La figure 2 décrit le fonctionnement d'un client.

Tout d'abord le client démarre et se met en attente d'une tâche. Puis il reçoit `MSG_JOB` suivi d'une tâche : une formule accompagnée d'une affectation partielle ou `MSG_JOB_SAME` suivi uniquement d'une affectation partielle. Le client appelle une fonction heuristique tentant d'évaluer rapidement la difficulté de la tâche reçue :

- La tâche est considérée comme abordable : la tâche est exécutée et un message est retourné :
  - `MSG_CONST` avec $\top$ ou $\bot$,
  - ou `MSG_OP` avec une formule propositionnelle seulement constituée des symboles propositionnels libres,
  - ou `MSG_CNF` avec une formule comme pour `MSG_OP` mais en FNC.
- La tache est considérée trop difficile a résoudre : alors un découpage sémantique est appliqué à la formule et le message `MSG_SPLIT` est envoyé avec un ensemble d'affectations partielles. Puis est envoyé le message
  - `MSG_STRING` plus "no change"
  - ou `MSG_QUANT` avec la formule partiellement traitée.

Notre fonction heuristique pour estimer la difficulté d'une tâche est très simple :

- Si une tâche a déjà été découpée par cette heuristique, elle est abordable,
- sinon, si le nombre de symboles propositionnels libres est inférieur à une constante, la tâche est abordable,
- sinon la tâche n'est pas abordable.

De même notre procédure de découpage sémantique est très simple. A la lecture de la formule, les symboles propositionnels libres sont empilés. Si un découpage est nécessaire, on calcule le nombre de symboles propositionnels libres à affecter sans toutefois dépasser une autre constante définissant le plus fin découpage autorisé. Ensuite, est dépilé le nombre de symboles propositionnels et est généré l'ensemble des affectations partielles possibles. Cet ensemble sera envoyé après `MSG_SPLIT`.

Les plus grandes améliorations que l'on puisse apporter au client sont au niveau de la fonction heuristique qui évalue la difficulté d'une tâche et la façon de choisir les symboles propositionnels du découpage sémantique. L'avantage de notre implantation est le comportement déterministe de ces deux composants, qui permet de limiter le nombre de variations lors de l'exécution. Ainsi, un client découpe toujours une même tâche de la même manière.

Actuellement, un client finit toujours un travail de découpage en retournant `MSG_STRING` suivi de "no change", mais dans le futur, en couplant une estimation de la difficulté et une résolution partielle, nous envisageons de pouvoir retourner des affectations partielles accompagnées du message `MSG_QUANT` suivies par la formule partiellement traitée.

5.3 Choix techniques

Une structure de données composite pour représenter les QBF. Afin de s'affranchir des problèmes liées à la mise sous forme prénexe et à la mise sous forme normale conjonctive, notre procédure devra travailler sur des QBF non prénexes non FNC closes. De plus nous voulons pouvoir traiter les implications, les bi-implications et les ou-exclusifs. Nous avons choisi comme format d'entrée QBF1.0 que nous avons étendu avec des symboles pour l'implication, la bi-implication et le ou-exclusif.

Notre structure de données n'a pas pour objectif d'être performante ou économe en espace. Nous cherchons un maximum d'expressivité et d'extensibilité. Afin de préserver l'information présente dans la formulation d'origine, une QBF est représentée par un ensemble d'éléments abstraits de formule, liés les uns aux autres sous forme d'un arbre. Un élément abstrait de formule impose uniquement deux actions aux éléments concrets : savoir démarrer un visiteur abstrait et savoir se sérialiser/désérialiser. Parmi les éléments concrets d'une formule, il y a les nœuds de l'arbre : les opérateurs logiques (binaires et unaires) et les quantificateurs. Puis il y a les feuilles de l'arbre : les littéraux et les constantes propositionnelles. Notre structure de données est donc un ensemble d'éléments composites héritant tous de l'élément abstrait de formule. Les visiteurs sont donc des traitements externes qui s'adaptent automatiquement à l'élément concret rencontré (design pattern composite + visitor) [11].

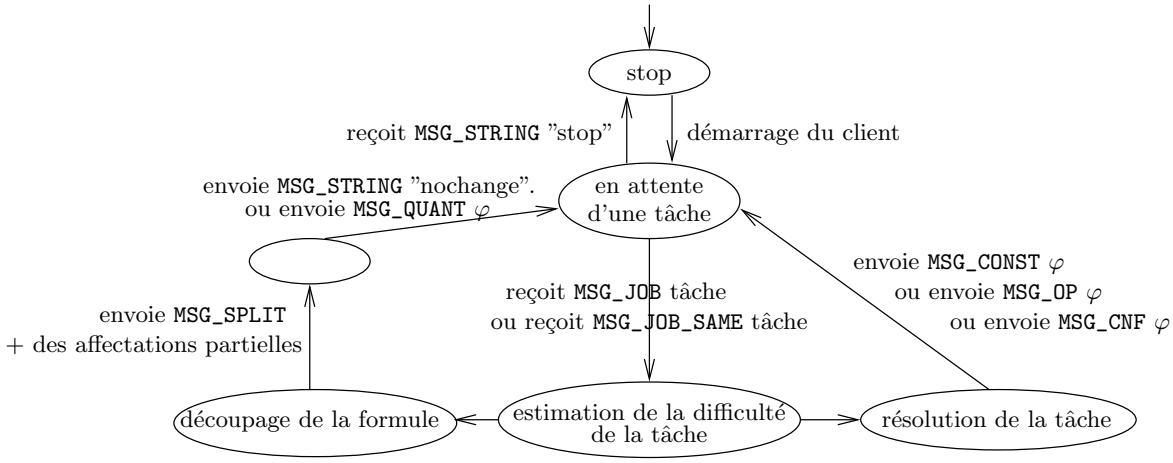


FIG. 2 – Automate du client

Une formule concrète est sérialisable afin de la transformer en flux, puis désérialisable afin de la restaurer en mémoire. Afin de stocker plus efficacement les formules possédant une partie sous forme normale conjonctive, nous avons défini un élément concret feuille FNC, stockant directement une matrice de littéraux. Il est alors facile de représenter des formules QBF sous FNC à l'aide uniquement des nœuds de quantificateurs et d'une feuille FNC.

MPI. Toute la partie communication est réalisée à l'aide du standard Message Passing Interface (MPI) [26]. Nous avons choisi comme implantation Open MPI. Pour notre approche avec une structure de données composite, MPI possède une lacune : il n'est pas possible simplement d'envoyer et recevoir des types de données complexes de façon native. La bibliothèque C++ Boost.MPI est une sur-couche répondant à cette contrainte à l'aide de Boost.Serialization. Toutes nos communications sont synchrones et utilisent le send/recv de Boost.MPI, faisant appel à MPI_Send et à MPI_Recv d'Open MPI.

MiniSat. Pour utiliser la méthode de la section 5, il faut pourvoir répondre à deux questions : la formule est-elle une tautologie ? Sinon, la formule est-elle insatisfiable ? Pour répondre, nous avons intégré la procédure de décision de `MiniSat`. Outre le fait que cette procédure soit très performante, ses sources sont disponibles et distribuées librement sous licence MIT. Les structures de données dans `MiniSat` sont très différentes des nôtres et sont orientées pour la performance. Nous avons donc implémenté une interface (un visiteur) permettant de formuler nos requêtes dans le format de données de `MiniSat`. A l'inverse, nous pouvons interpréter les réponses de `MiniSat` pour les intégrer directement dans notre structure de données. Il est ainsi possible à partir de notre structure de données expressive et extensible, de faire des traitements dans un modèle performant, le tout à l'aide de traitement en temps polynomial (linéaire) par rapport à la taille de la formule (nombre de noeuds).

Le caching. Afin de réduire le besoin en bande passante, chaque nœud de calcul garde en cache l'objet solveur Minisat de la dernière sous-formule traitée. Si une nouvelle tâche sur la même sous-formule arrive, le premier gain, est l'économie du transfert de la formule. Mais ce n'est pas tout : Minisat utilise le clause learning. L'instance déjà entraînée de Minisat peut possiblement répondre plus vite à une nouvelle question. L'idéal serait que les différents nœuds de calcul partage l'apprentissage réalisé par chacun, mais ce n'est pas le cas pour l'instant. De plus, il faudrait étudier en contre partie, le surplus de trafic généré par l'actualisation de ces informations.

5.4 Résultats expérimentaux

Afin d'évaluer notre architecture, nous avons effectué quelques tests préliminaires sur quelques instances de *qbflib* [13] et sur quelques problèmes que nous avons générés. Le découpage sémantique est strictement identique quel que soit le nombre de clients de calcul. Les tests ont été réalisés sur une machine hautes performances (HPC) composée de 12 serveurs de calcul Bull Novascale R422, connectés entre eux par 2 réseaux Ethernet Gigabit. Chaque serveur possède 2 nœuds de calcul de type 2x Intel(R) Xeon(R) E5440 à 2.83GHz et possède 16Go de mémoire locale. Les

	c8_8 ($\exists^{72}\forall^{64}$)		c8_16 ($\exists^{136}\forall^{128}$)		r4_5 ($\exists^{42}\forall^{35}$)		s5_4 ($\exists^{70}\forall^{56}$)	
1	105	1	7264	1	18483	1	58812	1
2	52	2.0	6627	1.1	11215	1.6	27741	2.1
4	28	3.8	4560	1.6	3601	5.1	19938	2.9
8	15	7.0	2437	3.0	4940	3.7	5537	10.6
16	11	9.5	3247	2.2	2268	8.1	4320	13.6
32	7	15.0	967	7.5	857	21.6	1027	57.2
64	6	17.5	3950	1.8	247	74.8	582	101.1
128	7	15.0	3548	2.0	128	144.4	780	75.4

TAB. 1 – Résultats obtenus sans prétraitement

machines tournent avec une version 2.6.18 du noyau Linux en version 64 bits.

Nous avons choisi quelques résultats pour illustrer différents cas. La première colonne de nos tables représente le nombre de clients de calcul auxquels il faut ajouter le processus maître tournant sur un processeur dédié. Les résultats sont sous la forme d'un couple *(temps en secondes/accélération)*.

La table 1 présente les résultats de 4 instances : counter8_8, counter8_16, ring4_5 et semaphore5_4. Elles font partie de la suite QBF1.0 disponible sur *qbflib*. Toutes ces formules sont prénexes, leur largeur syntaxique maximale est donc 1. Le seul choix que nous ayons pour traiter ces formules en parallèle est l'utilisation de l'extraction sémantique de sous-problèmes décrite dans la sous-section 5.2. Le premier problème, counter8_8, est simple ; l'augmentation du nombre de clients est vraiment efficace jusqu'à 8. L'explication est simple : plus nous utilisons de processus de calcul, plus le temps d'initialisation lié à MPI est long. Par exemple, l'initialisation pour 128 processus prend 5 secondes.

Pour évaluer notre approche avec beaucoup de ressources de calcul, il nous faut des problèmes plus conséquents. Le second problème, counter8_16, satisfait cette contrainte, malgré cela les temps sont irréguliers et l'accélération est assez mauvaise. Deux phénomènes surviennent ici. Tout d'abord, certains sous-problèmes sont très difficiles à résoudre. Par exemple, certaines tâches requièrent plusieurs centaines de secondes. Une tâche qui monopolise une ressource pendant 10% du temps, va limiter à 10 l'accélération maximale dans le meilleur des cas, c'est à dire en commençant la recherche par cette tâche. Le second phénomène, est lié à l'utilisation de `MiniSat` comme oracle. Chacun des 4 problèmes ne possède qu'une alternance de quantificateurs. C'est pourquoi, chaque nœud de calcul reçoit la formule une seule fois et crée une unique instance de la procédure `MiniSat` Toutes les autres tâches utiliseront cette instance en cache et tireront des bénéfices de l'apprentissage de clauses déjà réalisé localement lors des tâches précédentes. Le propre de notre exécution parallèle, c'est l'impossibilité de prévoir la distribution des tâches. Par extension, chaque sous-problème sémantique entraîne une instance de `MiniSat` et cet apprentissage est imprévisible et différent à chaque exécution. Plus le nombre de nœuds de calcul augmente, plus la probabilité d'apprendre localement une information intéressante avant un sous-problème difficile diminue. Avoir plus de nœuds de calcul, n'implique pas nécessairement une résolution plus rapide s'il n'est pas possible de partager des informations.

Pour le problème ring4_5, nous observons une accélération super-linéaire avec 64 et 128 processus de calcul. Comme pour le problème précédent, nous pensons que l'apprentissage de `MiniSat` a un effet, positif cette fois ci. Il est possible que certains sous-problèmes entraînent efficacement la plupart des instances de `MiniSat` Avec l'accroissement du nombre de nœuds de calcul, chaque solveur reçoit moins de sous-problèmes à résoudre. Peut-être cela leur permet-il de garder plus longtemps des informations plus pertinentes. Nous remarquons aussi qu'aucune tâche ne monopolise beaucoup de ressources. De part la nature aléatoire des exécutions parallèles, il faudrait multiplier les exécutions pour consolider nos hypothèses.

Nous observons une accélération super-linéaire pour le problème semaphore5_4 avec 32 et 64 clients. par contre, pour 128 le gain est inférieur à celui pour 64. Contrairement à ring5_4, certaines tâches sont très longues et peuvent représenter plus de 50% du temps total d'exécution pour 64 clients ou plus. Comme pour counter8_16, plus l'apprentissage est réparti sur différents clients plus les tâches longues sont pénalisantes.

La table 2 présente les résultats pour les mêmes problèmes mais avec l'application de quelques optimisations sur la formule en entrée. Le nœud maître, après lecture de la formule, applique récursivement une propagation naïve comparable à la propagation unitaire et cherche les littéraux monotones (n'apparaissant que dans une seule polarité). Ces améliorations simples montrent qu'il sera possible d'améliorer les performances de notre procédure de résolution en appliquant les techniques de l'état de l'art. Ces améliorations pourront être appliquées aussi sur les

	c8_8 ($\exists^{72}\forall^{64}$)		c8_16 ($\exists^{136}\forall^{128}$)		r4_5 ($\exists^{42}\forall^{35}$)		s5_4 ($\exists^{70}\forall^{56}$)	
1	52	1	6069	1	8718	1	25647	1
2	27	1.9	2959	2.1	3536	2.5	6530	3.9
4	14	3.7	1855	3.3	1597	5.4	1855	13.8
8	10	5.2	3338	1.8	914	9.5	2226	11.5
16	6	8.7	1114	5.4	349	25.0	1993	12.9
32	5	10.4	1527	4.0	202	43.2	778	33.0
64	5	10.4	1249	4.9	89	98.0	240	106.9
128	5	10.4	1750	3.5	38	229.4	588	43.6

TAB. 2 – Résultats obtenus avec un prétraitement

	adder_6 (non prénexe)		chaine_30 (non prénexe)	
1	3479	1	68343	1
2	1377	2.5	33068	2.1
4	995	3.5	16181	4.2
8	1788	1.9	7504	9.1
16	708	4.9	3780	18.1
32	1242	2.8	2043	33.5
64	1238	2.8	1023	66.8
128	1007	3.5	438	156.0

TAB. 3 – Résultats pour adder_6 et chaine_30

sous-problèmes. Nous relevons que pour ring4_5 les temps obtenus correspondent à une accélération superlinéaire et pour 128 clients le gain est de 229,4.

Contrairement aux problèmes précédents, adder_6 et chaine_30 ne sont pas prénexes et possèdent des bi-implications et/ou des ou exclusifs. Le problème adder_6 possède une largeur syntaxique de 2, sauf lors de la dernière itération. Le problème chaine_30 possède une largeur syntaxique de 30 lors de la première itération, puis de 1 ensuite. La table 3 résume les différents résultats. Pour adder_6, le gain est bon jusqu'à 4 clients. Pour ce problème la dernière itération est la tâche la plus longue, or cette tâche ne possède aucun symbole propositionnel libre : seul un client est actif. Pour chaine_30, nous observons une accélération super-linéaire. Dans un premier temps, le nœud maître distribue 30 sous-problèmes qui sont simplifiés très rapidement, puis la largeur syntaxique passe à 1. L'extraction sémantique de sous-problèmes prend le relais. Comme constaté pour ring5_4, les tâches sont de longueurs régulières.

6 Conclusion

Nous avons présenté dans cet article une nouvelle architecture pour la parallélisation du problème de validité des QBF dite « architecture de parallélisation syntaxique » par opposition aux architectures de parallélisation basée sur la sémantique des quantificateurs. Nous avons choisi d'implanter notre architecture selon un modèle maître/esclave avec pour oracle la procédure de décision `QSAT`. Cette implantation est déjà opérationnelle et la seule à notre connaissance à traiter en parallèle des QBF non FNC non prénexes. Le cadre proposé est suffisamment général pour intégrer d'autres procédures du moment même où elles réalisent une élimination des quantificateurs d'une QBF à symboles propositionnels libres.

Références

[1] G. Audemard and L. Sais. A Symbolic Search Based Approach for Quantified Boolean Formulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, 2005.

[2] A. Ayari and D. Basin. Qubos : Deciding Quantified Boolean Logic using Propositional Satisfiability Solvers. In *Formal Methods in Computer-Aided Design, Fourth International Conference, FMCAD 2002*. Springer-Verlag, 2002.

[3] M. Benedetti. skizzo : a suite to evaluate and certify QBFs. In *Proceedings of the 20th International Conference on Automated Deduction (CADE'05)*, pages 369–376, 2005.

[4] M. Benedetti and H. Mangassarian. Experience and Perspectives in QBF-Based Formal Verification. *Journal on Satisfiability, Boolean Modeling and Computation*, 2008.

[5] A. Biere. Resolve and Expand. In *Proceedings of the 7th International Confrerence on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 59–70, 2004.

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

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

[8] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3) :201–215, July 1960.

[9] U. Egly, M. Seidl, and S. Woltran. A Solver for QBFs in Nonprenex Form. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 477–481, 2006.

[10] R. Feldmann, B. Monien, and S. Schamberger. A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI'00) and 12th Conference on Innovative Applications of Artificial Intelligence (IAAI'00)*, 2000.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Assison-Wesley Professional Computing Series, 1994.

[12] I.P. Gent, H.H. Hoos, A.G.D. Rowley, and K. Smyth. Unsing Stochastic Local Search to Solve Quantified Boolean Formulae. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, 2003.

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

[14] E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE : A System for Deciding Quantified Boolean Formulas Satisfiability. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR'01)*, pages 364–369, 2001.

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

[16] H. Kleine Büning and X. Zhao. On Models for Quantified Boolean Formulas. In *Logic versus Approximation, In Lecture Notes in Computer Science 3075*, 2004.

[17] M. Lewis, P. Marin, T. Schubert, M. Narizzano, B. Becker, and E. Giunchiglia. PaQuBE : Distributed QBF Solving with Advanced Knowledge Sharing. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT'09)*, pages 509–523, 2009.

[18] M. Lewis, T. Schubert, and B. Becker. QMiraXT - A Multithreaded QBF Solver. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'09)*, pages 7–16, 2009.

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

[20] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (SWAT'72)*, pages 125–129, 1972.

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

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

[23] J. Rintanen. Improvements to the Evaluation of Quantified Boolean Formulae. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 1192–1197, 1999.

[24] T. Schubert, M. Lewis, and B. Becker. PaMiraXT : Parallel SAT Solving with Threads and Message Passing. *Journal on Satisfiability, Boolean Modeling and Computation*, 6 :203–222, 2009.

[25] B. Selman, H.A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1993.

[26] M. Snir, S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman. *MPI : The Complete Reference*. MIT Press, Cambridge, 1995.

[27] I. Stéphan, B. Da Mota, and P. Nicolas. From (Quantified) Boolean Formulas to Answer Set Programming. *Journal of logic and computation*, 19(4) :565–590, 2009.

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

[29] L. Zhang and S. Malik. Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In *International Conference on Computer Aided Design (ICCAD'02)*, 2002.

Une modélisation en CSP des grammaires de propriétés

Denys Duchier, Thi-Bich-Hanh Dao, Yannick Parmentier, Willy Lesaint

Laboratoire d'Informatique Fondamentale d'Orléans – Université d'Orléans
Bâtiment 3IA, Rue Léonard de Vinci – 45067 Orléans Cedex 2
prenom.nom@univ-orleans.fr

Résumé

Les Grammaires de Propriétés (GP) constituent un formalisme à base de contraintes capable de décrire à la fois des énoncés bien-formés et des énoncés agrammaticaux, ce qui en fait un formalisme particulièrement intéressant pour traiter de la gradience de grammaticalité, comme l'a démontré Prost [12]. Duchier *et al* [7] ont défini une sémantique des grammaires de propriétés en théorie des modèles. Cet article poursuit ce travail en montrant comment, à partir de cette sémantique, définir l'analyse syntaxique en GP sous la forme d'un Problème de Satisfaction de Contraintes (CSP), traitable au moyen de la programmation par contraintes.

Abstract

Property grammars offer a constraint-based formalism capable of handling both well-formed and deviant utterances. It is thus well-suited for addressing issues of gradience of grammaticality, as demonstrated by Prost [12]. Duchier *et al* [7] contributed precise model-theoretic semantics for property grammars. The present article follows up on that work and explains how to turn such a formalization into a concrete constraint satisfaction problem (CSP), solvable using constraint programming.

1 Introduction

Dans ce travail, nous nous intéressons à une tâche spécifique du domaine du Traitement Automatique des Langues, à savoir l'analyse syntaxique. Cette tâche a pour but de construire, à partir d'une description formelle de la syntaxe de la langue naturelle (*i.e.*, une grammaire), une structure exprimant les relations entre les divers constituants d'un énoncé. Cette structure prend généralement une forme arborescente, on parle alors d'arbre syntaxique.

De nombreux formalismes grammaticaux ont été proposés pour décrire la syntaxe de la langue naturelle, généralement en se basant sur un système de réécriture (*e.g.* réécriture de chaînes pour les grammaires hors-contexte [4], ou encore réécriture d'arbres pour les grammaires d'arbres adjoints [10]). Un problème majeur de ces formalismes, dans un contexte de Traitement Automatique de la Langue, est leur manque de robustesse. En effet, ils ne permettent pas d'analyser des énoncés qui sont mal-formés, même lorsqu'il s'agit d'erreurs mineures.

Comme l'ont démontré Pullum et Scholz [13], les grammaires formelles de type *syntaxe générative-énumérative*, dans la mesure où elles se concentrent sur la génération de modèles bien formés, sont intrinsèquement inadaptées au traitement d'énoncés agrammaticaux. Par contre, des grammaires formelles de type *syntaxe fondée sur la théorie des modèles*, qui se concentrent sur une validation de modèles en termes de contraintes satisfaites, sont naturellement adaptées au traitement de *quasi-expressions*.

Blache [2, 3] a proposé le formalisme des Grammaires de Propriétés (GP), comme un formalisme à base de contraintes, permettant d'analyser à la fois des énoncés grammaticaux et agrammaticaux. Prost [12] a développé une technique d'analyse syntaxique utilisant GP, et permettant de produire une structure syntaxique pour tout type d'énoncé, tout en y associant un jugement précis sur la grammaticalité de l'énoncé en question. Duchier *et al* [7] ont fourni une sémantique en théorie des modèles pour GP, ainsi qu'une définition logique formelle des travaux d'analyse syntaxique menés par Prost. Dans cet article, nous montrons comment une telle formalisation de la sémantique de GP, peut être convertie en un CSP, ouvrant la voie à l'implantation d'un analyseur syntaxique à

base de contraintes, qui calcule des arbres syntaxiques optimaux (en termes de grammaticalité d'énoncé), au moyen d'une recherche de type *branch-and-bound*.

L'article est structuré comme suit. En section 2, nous introduisons le formalisme des grammaires de propriétés. En section 3, nous définissons une sémantique des grammaires de propriétés basée sur la théorie des modèles. Cette sémantique va servir de cadre à la définition de la tâche d'analyse syntaxique en termes de problème de satisfaction de contraintes. Cette définition va reposer sur deux types de contraintes permettant de construire l'arbre syntaxique d'un énoncé. Le premier type de contraintes, les contraintes de structure d'arbre, sera présenté en section 4, le second type de contraintes, les contraintes de propriétés, en section 5. Cette définition à base de contraintes de l'analyse syntaxique pour les grammaires de propriétés mènera naturellement à une implantation en programmation par contraintes, qui sera introduite en section 6. Enfin, en section 7, nous comparons notre travail avec les approches existantes, et en section 8, nous concluons en présentant les travaux futurs.

2 Grammaires de propriétés

Les grammaires de propriétés [2, 3] constituent un formalisme permettant de décrire la langue naturelle en termes de contraintes locales, appelées *propriétés*. Ces propriétés peuvent être violées indépendemment les unes des autres, ce qui rend possible la description d'énoncés agrammaticaux (*i.e.*, qui ne vérifieraient pas l'ensemble des propriétés de la langue), et également de définir une notion de grammaticalité (ratio entre propriétés violées et propriétés vérifiées).

Les propriétés utilisées pour décrire la langue se basent sur des observations linguistiques : ordre (partiel) entre les mots, exclusion mutuelle entre certains mots dans un contexte proche, cooccurrence systématique de certains mots dans un contexte proche, non-répétition de certains mots dans un contexte proche, présence facultative de certains mots, *etc.*

Chaque propriété a la forme $A : \psi$, où A représente la catégorie (*i.e.*, l'étiquette) d'un nœud dans un arbre syntaxique, et ψ la contrainte qui s'applique sur les nœuds fils de celui-ci. L'ensemble des propriétés auxquelles nous nous intéressons est le suivant :

obligation $A : \triangle B$, pour tout nœud d'étiquette A, présence d'au moins un nœud fils étiqueté B,

unicité $A : B!$, pour tout nœud d'étiquette A, présence d'au plus un nœud fils étiqueté B,

linéarité $A : B \prec C$, pour tout nœud d'étiquette A, en cas de présence de nœuds fils étiquetés B et C, B précède C,

exigence [1] $A : B \Rightarrow C$, pour tout nœud d'étiquette A, la présence d'un nœud fils étiqueté B implique la présence d'un nœud fils étiqueté C,

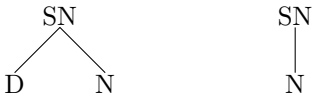
exclusion $A : B \nRightarrow C$, pour tout nœud d'étiquette A, des nœuds fils d'étiquettes B et C sont mutuellement exclusifs,

constituence $A : S$, pour tout nœud d'étiquette A, les étiquettes des nœuds fils appartiennent à l'ensemble S.

A partir de cet ensemble de propriétés, il est possible d'énoncer un certain nombre de règles grammaticales décrivant la syntaxe de la langue naturelle. Par exemple, une règle grammaticale stipulant qu'un groupe nominal contient exactement un nom et que ce nom peut être précédé d'un déterminant, pourrait être représentée par l'ensemble de propriétés ci-dessous :

$$_{(1)}\texttt{SN} : \{\texttt{D}, \texttt{N}\},\ _{(2)}\texttt{SN} : \triangle\texttt{N},\ _{(3)}\texttt{SN} : \texttt{N}!,\ _{(4)}\texttt{SN} : \texttt{D}!,$$
$$_{(5)}\texttt{SN} : \texttt{D} \prec \texttt{N},\ _{(6)}\texttt{D} : \{\},\ _{(7)}\texttt{N} : \{\}$$

où SN réfère à l'étiquette *Syntagme Nominal* (groupe nominal), D à *Déterminant*, et N à *Nom*. La propriété (1) ($\texttt{SN} : \{\texttt{D}, \texttt{N}\}$) indique que, dans un arbre syntaxique, les nœuds fils du nœud représentant le groupe nominal doivent être étiquetés D ou N. (2) ($\texttt{SN} : \triangle\texttt{N}$) indique qu'un nœud SN domine directement au moins un nœud N. (3) ($\texttt{SN} : \texttt{N}!$) indique qu'un nœud SN domine directement au plus un nœud N. (4) ($\texttt{SN} : \texttt{D}!$) indique qu'un nœud SN domine directement au plus un nœud D. (5) ($\texttt{SN} : \texttt{D} \prec \texttt{N}$) indique que, pour tout nœud d'étiquette SN, si ce nœud comporte deux nœuds fils d'étiquettes respectives D et N, alors le nœud d'étiquette D précède celui d'étiquette N. Enfin, (6) et (7) permettent de s'assurer que les nœuds d'étiquettes D et N sont des nœuds feuilles (seuls les mots du lexique peuvent être situés en dessous de ces nœuds). L'ensemble de ces propriétés a exactement deux modèles d'arbre syntaxique ayant la racine SN :



Les étiquettes introduites dans cet exemple sont appelées *catégories syntaxiques* et renseignent sur le type d'un constituant de la phrase. Une grammaire de propriétés définit donc des contraintes sur les relations entre les différents types de constituants apparaissant dans une phrase. Pour faire le lien avec le lexique, on définit une association entre mots et catégories syntaxiques. Par exemple, on peut spécifier que le mot *pomme* est un nom (catégorie N) via :

$$\mathsf{cat}(\mathit{pomme}) = \texttt{N}$$

[1] Également appelée *cooccurrence* dans la littérature.

Notons qu'il est possible qu'un même mot soit associé à plusieurs catégories (on parle alors d'ambiguïté lexicale). C'est le cas par exemple du mot *ferme* qui peut correspondre à un nom, un adjectif ou encore un verbe (forme conjuguée du verbe *fermer*).

3 Sémantique des grammaires de propriétés en théorie des modèles

En section précédente, nous avons présenté le formalisme des grammaires de propriétés. Ici, nous allons voir comment définir une sémantique à ce formalisme dans le cadre de la théorie des modèles. Cette sémantique a été introduite par Duchier *et al* [7]. Elle nous permettra de traduire la tâche de l'analyse syntaxique en GP sous la forme d'un problème de satisfaction de contraintes.

Sémantique forte Nous interprétons les grammaires de propriétés sous forme d'arbres syntaxiques. Un arbre syntaxique τ est un modèle *fort* d'une grammaire $\mathcal{G}$, si et seulement si, pour chaque nœud n de τ, et pour chaque propriété p de $\mathcal{G}$, si la propriété p en question est *pertinente* au nœud n, alors elle y est également *satisfaite*.

Cette interprétation sous forme d'arbre syntaxique nous conduit à considérer des *instances* de propriété. Une instance s'applique sur un nœud d'un arbre syntaxique candidat.

Pour illustrer cela, considérons la propriété de linéarité $\texttt{SN} : \texttt{D} \prec \texttt{N}$ vue à la section précédente. Lors de la recherche d'un modèle d'arbre syntaxique, cette propriété est *instanciée* à chaque nœud de l'arbre syntaxique candidat. Plus précisément, pour chaque nœud n de cet arbre, nous allons considérer toutes les paires possibles de nœuds fils (n_1, n_2). Nous allons, pour cette configuration (n, n_1, n_2), évaluer le fait que la propriété soit pertinente (*i.e.*, qu'elle doive s'appliquer) et également qu'elle soit satisfaite. Nous notons cette instance de la propriété de linéarité sous la forme suivante :

$$\texttt{SN} : \texttt{D} \prec \texttt{N} \ @ \ \langle n, n_1, n_2 \rangle$$

Pour que cette instance de propriété soit pertinente, il faut que n soit étiqueté SN, et n_1 (respectivement n_2) soit étiqueté D (resp. N). Pour être satisfaite, il faut en plus que le nœud n_1 précède n_2.

Considérons un autre type de propriété, à savoir la constituence, illustrée par $\texttt{SN} : \{\texttt{D}, \texttt{N}\}$. Cette propriété est, elle aussi, instanciée à chaque nœud de l'arbre syntaxique candidat. Ici, pour pouvoir vérifier si cette propriété est pertinente à un nœud donné n, il nous suffit de considérer un seul de ses nœuds fils, appelons le n_1. Ainsi, nous notons une instance de la propriété de constituence sous la forme suivante :

$$\texttt{SN} : \{\texttt{D}, \texttt{N}\} \ @ \ \langle n, n_1 \rangle$$

Une telle instance est pertinente lorsque n est étiqueté SN, et satisfaite lorsque n_1 est de plus étiqueté D ou N.

Nous avons donc différents types de propriétés selon le nombre de nœuds impliqués dans son instanciation. Nous avons vu que les propriétés de linéarité s'instancient sur un triplet de nœuds[2], celles de constituence sur un couple de nœuds. Un dernier type de propriété correspond à celles dont l'instanciation s'applique sur un nœud, ce qui est le cas de l'obligation. Pour ce type de propriété, une instance de propriété a la forme suivante :

$$\texttt{SN} : \triangle\texttt{N} \ @ \ \langle n \rangle$$

Pour tout nœud n de l'arbre syntaxique candidat, si ce nœud est étiqueté SN alors l'instance de propriété est pertinente, et s'il a en outre au moins un nœud fils étiqueté N, alors elle est satisfaite.

Dans un modèle fort, toutes les instances de propriété qui sont pertinentes doivent obligatoirement être satisfaites.

Sémantique relâchée Un arbre syntaxique τ est un modèle *relâché* d'une grammaire de propriétés $\mathcal{G}$, si son *score d'adéquation* est maximal. Par score d'adéquation, nous entendons le ratio entre instances de propriétés satisfaites parmi l'ensemble des instances de propriétés pertinentes.

Dans un modèle relâché, il est possible que certaines instances de propriétés, bien que pertinentes, ne soient pas satisfaites. Le fait de pouvoir violer certaines propriétés permet de calculer des modèles d'arbres syntaxiques correspondant à des énoncés agrammaticaux. Le score d'adéquation nous permet ainsi de chiffrer l'adéquation syntaxique du modèle.

Exemple Pour illustrer ces deux sémantiques de GP, considérons la grammaire jouet $\mathcal{G}$ suivante :

$$_{(1)}\texttt{P} : \{\texttt{SN}, \texttt{VP}\}, \ _{(2)}\texttt{SN} : \{\texttt{D}, \texttt{N}\}, \ _{(3)}\texttt{VP} : \{\texttt{V}, \texttt{SN}\},$$

$$_{(4)}\texttt{P} : \triangle\texttt{VP}, \ _{(5)}\texttt{P} : \texttt{VP}!, \ _{(6)}\texttt{P} : \triangle\texttt{SN}, \ _{(7)}\texttt{P} : \texttt{SN}!,$$

$$_{(8)}\texttt{P} : \texttt{SN} \prec \texttt{VP}, \ _{(9)}\texttt{VP} : \triangle\texttt{V}, \ _{(10)}\texttt{VP} : \texttt{V}!, \ _{(11)}\texttt{VP} : \texttt{SN}!,$$

$$_{(12)}\texttt{SN} : \texttt{D}!, \ _{(13)}\texttt{SN} : \triangle\texttt{N}, \ _{(14)}\texttt{SN} : \texttt{D} \Rightarrow \texttt{N}, \ _{(15)}\texttt{SN} : \texttt{D} \prec \texttt{N},$$

$$_{(16)}\mathsf{cat}(\textit{mange}) = \texttt{V}, \ _{(17)}\mathsf{cat}(\textit{la}) = \texttt{D},$$

$$_{(18)}\mathsf{cat}(\textit{Pierre}) = \texttt{N}, \ _{(19)}\mathsf{cat}(\textit{pomme}) = \texttt{N}$$

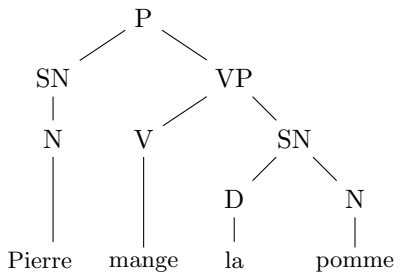
$\mathcal{G}$ permet de construire un arbre syntaxique pour la phrase "Pierre mange la pomme"[3]. Les propriétés (16)

[2] Ce qui est également le cas de l'unicité, de l'exigence, et de l'exclusion.

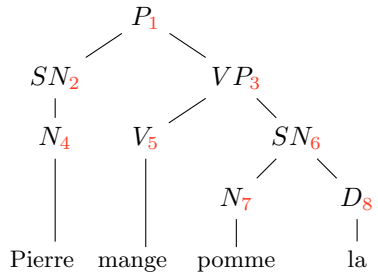
[3] Il est possible qu'une grammaire décrive plusieurs arbres syntaxiques pour un même énoncé, on parle alors d'ambiguïté syntaxique.

à (19) forcent la catégorie des nœuds feuilles de notre arbre syntaxique, elles ne peuvent être violées.

À partir de cette grammaire et de la phrase "Pierre mange la pomme", nous pouvons construire l'arbre syntaxique (*i.e.*, l'unique modèle ici en sémantique forte, qui est également le modèle de score d'adéquation maximal[4] en sémantique relâchée) suivant :



Si nous considérons dans un deuxième temps l'énoncé agrammatical "Pierre mange pomme la", il n'existe plus d'arbre syntaxique en sémantique forte. Cependant, modulo la violation de la propriété de linéarité fixant l'ordre entre les nœuds d'étiquettes `D` et `N`, il est possible d'obtenir un modèle d'arbre syntaxique de score d'adéquation maximal 14/15, représenté ci-dessous[5] :



Ce score d'adéquation correspond au ratio entre instances de propriétés pertinentes et instances pertinentes et satisfaites. Par exemple, la propriété de consituence (1)$\mathtt{P} : \{\mathtt{SN}, \mathtt{VP}\}$ est instanciée, pertinente et satisfaite à deux reprises (pour chacun des fils du nœud 1) :

$$_{(1)}\mathtt{P} : \{\mathtt{SN}, \mathtt{VP}\} \ @ \ \langle 1, 2\rangle \qquad _{(1)}\mathtt{P} : \{\mathtt{SN}, \mathtt{VP}\} \ @ \ \langle 1, 3\rangle$$

Les autres propriétés instanciées et pertinentes sont les suivantes (la seule propriété non satisfaite est précédée du symbole $\times$)[6] :

$$_{(2)}\mathtt{SN} : \{\mathtt{D}, \mathtt{N}\} \ @ \ \langle 2, 4\rangle$$

$$_{(2)}\mathtt{SN} : \{\mathtt{D}, \mathtt{N}\} \ @ \ \langle 6, 7\rangle \qquad _{(2)}\mathtt{SN} : \{\mathtt{D}, \mathtt{N}\} \ @ \ \langle 6, 8\rangle$$
$$_{(3)}\mathtt{VP} : \{\mathtt{V}, \mathtt{SN}\} \ @ \ \langle 3, 5\rangle \qquad _{(3)}\mathtt{VP} : \{\mathtt{V}, \mathtt{SN}\} \ @ \ \langle 3, 6\rangle$$
$$_{(4)}\mathtt{P} : \triangle\mathtt{VP} \ @ \ \langle 1\rangle \qquad _{(6)}\mathtt{P} : \triangle\mathtt{SN} \ @ \ \langle 1\rangle$$
$$_{(8)}\mathtt{P} : \mathtt{SN} \prec \mathtt{VP} \ @ \ \langle 1, 2, 3\rangle \qquad _{(9)}\mathtt{VP} : \triangle\mathtt{V} \ @ \ \langle 3\rangle$$
$$_{(13)}\mathtt{SN} : \triangle\mathtt{N} \ @ \ \langle 2\rangle \qquad _{(13)}\mathtt{SN} : \triangle\mathtt{N} \ @ \ \langle 6\rangle$$
$$_{(14)}\mathtt{SN} : \mathtt{D} \Rightarrow \mathtt{N} \ @ \ \langle 6, 8, 7\rangle \qquad \times \ _{(15)}\mathtt{SN} : \mathtt{D} \prec \mathtt{N} \ @ \ \langle 6, 8, 7\rangle$$

Ce qui nous donne 14 instances de propriété pertinentes et satisfaites sur 15 instances pertinentes.

4 Contraintes de structure d'arbre

Étant données une grammaire de propriétés et une expression (qui peut éventuellement être une quasi-expresssion), l'objectif de notre approche est de trouver tous les quasi-modèles qui peuvent être candidats pour une analyse syntaxique et de retenir ceux qui maximisent un score défini. Un arbre d'analyse syntaxique est un arbre dont les feuilles correspondent aux mots de l'expression, et dont les nœuds sont étiquetés par des catégories de la grammaire. Pour définir notre CSP, nous devons spécifier les variables utilisées pour modéliser un arbre syntaxique, or nous ne connaissons pas *a priori* le nombre de nœuds que contient un tel arbre. Nous choisissons donc de borner ce nombre en considérant une grille, sur laquelle seront placés les nœuds d'un arbre syntaxique.

Soit m le nombre de mots de l'expression à analyser. Chaque arbre syntaxique a m feuilles (les grammaires de propriétés n'utilisent pas de nœuds "vides" ϵ). Nous notons n la profondeur maximale d'un arbre syntaxique (n est un paramètre du problème). Nous allons chercher les arbres syntaxiques qui peuvent être placés sur un sous-ensemble de nœuds d'une grille de taille $n \times m$. Dans cette section, nous présentons la structure d'arbre rectangulaire, les contraintes permettant de définir un arbre rectangulaire sur une grille $n \times m$ et une réalisation de ces contraintes.

4.1 Arbres rectangulaires

Nous référons à un nœud d'un arbre syntaxique par sa position sur une grille de taille $n \times m$, dont les lignes sont numérotées de 1 à n et les colonnes de 1 à m, la ligne du bas portant le numéro 1.

Un arbre est caractérisé par un ensemble de nœuds, une racine et pour chaque nœud, un nœud père. De façon duale, à chaque nœud nous pouvons associer un ensemble des nœuds fils. Chaque nœud de l'arbre se plaçant sur un point de la grille, l'arbre s'étale sur un sous-ensemble des points de la grille.

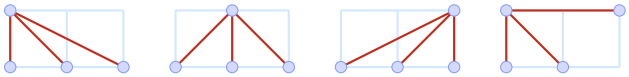
Cependant, il existe plusieurs façons d'étaler un même arbre sur une grille, si l'on ne tient pas compte

[4] Sur cet arbre, toutes les propriétés pertinentes sont satisfaites, et donc ce score d'adéquation vaut 1.

[5] Les numéros annotant les nœuds servent uniquement à référer au nœud dans l'explication du score.

[6] La sémantique donnée à l'unicité par Duchier *et al* [7] est qu'elle n'est pertinente que lorsqu'elle est violée, elle n'apparaît donc pas ici.

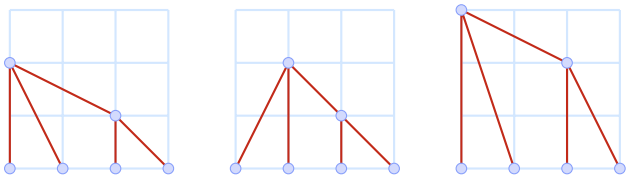
des symétries. Par exemple, sur une grille de taille 2×3, un arbre de 4 nœuds peut entre autres avoir les représentations équivalentes suivantes :



Afin d'arriver à une représentation unique d'un arbre sur une grille, nous restreignons les arbres à être *rectangulaires*. Ces arbres satisfont les conditions suivantes :

1. les feuilles se situent toutes au niveau le plus bas de la grille (la ligne numéro 1) ;
2. chaque nœud de l'arbre doit se situer dans la même colonne que la feuille la plus à gauche de son sous-arbre (ceci implique que le sous-arbre à partir d'un nœud s'étale sur le rectangle en bas à droite du nœud) ;
3. chaque nœud doit se situer à un niveau plus haut que le niveau des nœuds de son sous-arbre (ceci implique que la racine se situe au niveau le plus haut de l'arbre) ;
4. chaque nœud qui n'est pas une feuille doit avoir au moins un fils au niveau inférieur immédiat (ceci implique qu'il n'y a pas de niveau intermédiaire non occupé par l'arbre).

Par exemple, parmi les arbres suivants, seul le premier est un arbre rectangulaire, les deux autres ne le sont pas car la condition 2 n'est pas vérifiée pour le second et la condition 4 n'est pas vérifiée pour le dernier :



Ce problème de reconnaissance d'arbres sur une grille est à distinguer de celui de la reconnaissance d'arbres sur un graphe, pour lequel il existe plusieurs travaux (*e.g.* Beldiceanu et al. [1], Prosser et Unsworth [11]). En effet, ces derniers visent à calculer un arbre ou une forêt d'arbres couvrants dans un graphe, en cherchant le nœud racine. Dans notre cas, nous ne connaissons pas *a priori* les nœuds de notre arbre, nous savons uniquement qu'ils peuvent occuper certaines positions sur une grille de taille fixée (grille qui ne correspond pas à des arêtes d'un graphe).

4.2 Contraintes de structure d'arbre

Arbre Nous écrivons w_{ij}, avec $1 \leq i \leq n$ et $1 \leq j \leq m$, pour les nœuds de la grille. Soit V l'ensemble des nœuds de la grille. L'arbre s'étale sur un sous-ensemble de V. Un nœud est appelé *actif* s'il est utilisé par l'arbre et *inactif* sinon. Soient V^+ l'ensemble des nœuds actifs et V^- l'ensemble des autres nœuds. Une contrainte relie donc ces deux ensembles avec V :

$$V = V^+ \uplus V^-$$

où $\uplus$ représente "l'union disjointe". Utilisant la technique de modélisation dans [6], pour chaque nœud w, nous écrivons $\downarrow w$ pour l'ensemble des nœuds fils de w, $\downarrow^+ w$ pour ses descendants et $\downarrow^* w$ pour w et ses descendants. D'une façon duale, nous écrivons $\uparrow w$ pour l'ensemble des parents de w, $\uparrow^+ w$ pour ses ancêtres et $\uparrow^* w$ pour w et ses ancêtres. Les contraintes reliants ces ensembles sont :

$$\downarrow^+ w = \uplus\{\downarrow^* w' \mid w' \in \downarrow w\} \qquad \downarrow^* w = \{w\} \uplus \downarrow^+ w$$
$$\uparrow^+ w = \uplus\{\uparrow^* w' \mid w' \in \uparrow w\} \qquad \uparrow^* w = \{w\} \uplus \uparrow^+ w$$

L'utilisation de l'union disjointe ici est justifiée par le fait que dans un arbre, chaque nœud ne peut appartenir à deux sous-arbres de même niveau. La contrainte suivante renforce la dualité :

$$w \in \uparrow w' \;\Leftrightarrow\; w' \in \downarrow w$$

Le fait que dans un arbre chaque nœud a au plus un parent est représenté par la contrainte suivante :

$$|\uparrow w'| \leq 1$$

Un nœud inactif n'appartient pas à l'arbre, il n'a donc ni parents ni enfants :

$$w \in V^- \Rightarrow \downarrow w = \uparrow w = \emptyset$$

Nous écrivons R pour l'ensemble des nœuds qui sont racines de l'arbre. Un arbre doit avoir une seule racine :

$$|R| = 1$$

et une racine ne peut être enfant d'un autre nœud :

$$V^+ = R \uplus (\uplus\{\downarrow w \mid w \in V\})$$

Arbre rectangulaire Pour chaque nœud w, nous écrivons $\Downarrow w$ pour l'ensemble des colonnes occupées par le sous-arbre ancré au nœud w. Nous écrivons $c(w)$ pour la colonne du nœud w et $\ell(w)$ pour sa ligne :

$$c(w_{ij}) = j \qquad \ell(w_{ij}) = i$$

La première condition d'un arbre rectangulaire concerne les feuilles. Les feuilles de l'arbre correspondent aux mots de l'expression. Elles doivent se

situer sur la ligne numéro 1 de la grille. Comme le nombre de colonnes de la grille est égal au nombre de mots, chaque nœud de cette ligne doit être actif :

$$\{w_{1j} \mid 1 \leq j \leq m\} \subseteq V^+$$

La seconde condition indique que chaque nœud actif doit se situer dans la même colonne que la feuille la plus à gauche de son sous-arbre. Elle est imposée par la contrainte suivante :

$$w_{ij} \in V^+ \quad \Leftrightarrow \quad c(w_{ij}) = \min \Downarrow w_{ij}$$

Par la troisième condition, chaque nœud de l'arbre doit se situer à un niveau plus haut que le niveau des nœuds de son sous-arbre. Cette condition jointe à une conséquence de la seconde condition est représentée par les contraintes de domaine suivantes. Pour chaque nœud w_{ij} :

- les nœuds descendants sont dans le rectangle en bas à droit du nœud, la colonne du nœud incluse

$$\downarrow^+ w_{ij} \subseteq \{w_{lk} \mid 1 \leq l < i,\ j \leq k \leq m\}$$

- les nœuds ascendants sont dans le rectangle en haut à gauche du nœud, la colonne du nœud incluse

$$\uparrow^+ w_{ij} \subseteq \{w_{lk} \mid i < l \leq n,\ 1 \leq k \leq j\}$$

La dernière condition indique que chaque nœud actif qui n'est pas une feuille doit avoir au moins un fils à la ligne inférieure immédiate. Elle est imposée par la contrainte suivante pour chaque nœud w_{ij}, avec $1 < i \leq n$:

$$w_{ij} \in V^+ \quad \Leftrightarrow \quad i - 1 \in \{\ell(w) \mid w \in \downarrow w_{ij}\}$$

Projection Les feuilles de l'arbre doivent se situer toutes à la première ligne de la grille, chacune d'elles occupe donc une seule colonne :

$$\Downarrow w_{1j} = \{j\}$$

Pour les nœuds de niveau supérieur, la projection correspond à une union disjointe des projections des fils :

$$\Downarrow w_{ij} = \uplus\{\Downarrow w \mid w \in \downarrow w_{ij}\} \qquad 1 < j \leq m$$

La projection de chaque nœud ne doit pas contenir de trou (les arbres sont projectifs)[7] :

$$\mathsf{convex}(\Downarrow w) \qquad \forall w \in V$$

[7] Cette contrainte de convexité d'un ensemble d'entiers est par exemple implantée dans la librairie Gecode.

Catégories Dans un arbre syntaxique, chaque nœud actif doit être étiqueté par une catégorie syntaxique. L'ensemble des catégories possibles pour chaque nœud sera déterminé par les contraintes de propriétés présentées dans la section 5. En ce qui concerne les feuilles, chacune d'elles sera étiquetée par la catégorie du mot correspondant :

$$\mathsf{cat}(w_{1j}) = \mathsf{cat}(\mathrm{mots}_j)$$

Les nœuds non actifs sont les seuls étiquetés par la catégorie `none` :

$$\mathsf{cat}(w) = \texttt{none} \quad \Leftrightarrow \quad w \in V^-$$

4.3 Réalisation des contraintes

Pour déterminer un arbre, les éléments à calculer sont V^+, V^-, R, les fonctions $\downarrow, \downarrow^+, \downarrow^*$ et $\uparrow, \uparrow^+, \uparrow^*$, la fonction $\Downarrow$ et la fonction cat.

Chaque nœud (i, j) de la grille est identifié par le numéro $(i-1)m + j$. L'ensemble V des nœuds de la grille est donc l'ensemble des entiers de 1 à nm. Pour désigner les ensembles V^+, V^- et R, nous utilisons des variables ensemblistes dont les éléments sont dans V. Les contraintes reliant ces ensembles sont facilement traduites par des contraintes ensemblistes.

Concernant les fonctions $\downarrow, \downarrow^+, \downarrow^*$ et $\uparrow, \uparrow^+, \uparrow^*$, chacune d'elles est représentée par un tableau indexé sur les éléments de V, où le i-ème élément du tableau est une variable ensembliste dont les éléments sont dans V. Par exemple le fait que le 9ème élément du tableau $\downarrow$ est l'ensemble $\{1, 4, 6\}$ indique que les enfants du nœud numéro 9 sont les nœuds numéro 1, 4 et 6. Quant à la fonction $\Downarrow$, elle est représentée par un tableau également indexé sur les éléments de V. Chaque i-ème élément du tableau est une variable ensembliste dont les éléments sont dans l'intervalle $[1, m]$. Cette variable désigne l'ensemble des colonnes possibles pour le nœud numéro i.

Les contraintes concernant ces ensembles sont posées à l'aide des contraintes `element`, `convex`, `min`, d'union disjointe, de domaine et de cardinalité. Les contraintes faisant intervenir les connecteurs logiques implication et équivalence sont réalisées à l'aide des contraintes réifiées et des variables booléennes. Nous présentons en particulier la contrainte :

$$\downarrow^+ w = \uplus\{\downarrow^* w' \mid w' \in \downarrow w\}$$

Soit i le numéro du nœud w. Cette contrainte est réalisée par une contrainte de sélection introduite dans [6] $A = \uplus\langle B_1, \ldots, B_n\rangle[S]$ qui signifie $A = \uplus_{i \in S} B_i$. Ici la suite $\langle B_1, \ldots, B_n\rangle$ est le tableau réalisant la fonction $\downarrow^*$ et l'ensemble S est le i-ème élément du tableau réalisant la fonction $\downarrow$.

La dernière fonction cat est également représentée par un tableau indexé sur les éléments de V. Chaque i-ème élément du tableau est une variable entière, indiquant la catégorie étiquetant le nœud numéro i.

5 Contraintes pour les propriétés

Après avoir construit la structure d'arbre, il faut déterminer le degré de satisfaction des propriétés de l'arbre. Nous présentons dans cette section les contraintes réalisant les propriétés des grammaires de propriétés et une contrainte qui optimise le degré de satisfaction des propriétés.

5.1 Instance de propriété

Dans un premier temps, revenons sur la notion d'instance de propriété introduite en section 3.

Les propriétés correspondent à des contraintes, qui doivent être satisfaites par un modèle. Une contrainte s'applique sur certains paramètres (des variables du modèle). Dans notre cas, nous appliquons une contrainte à des nœuds d'une grille. Suivant le type de contrainte, cette application peut concerner un, deux ou trois nœuds. Une contrainte de propriété appliquée à un n-uplet de nœuds de la grille est ce que nous appelons une instance de propriété. Chaque instance de propriété dépend uniquement de la grammaire et de la grille. La détermination d'un modèle d'arbre syntaxique passe par l'évaluation de la pertinence et de la satisfaction de ces instances de propriétés.

5.2 Contraintes pour les propriétés

Pour chaque instance I, on définit deux variables booléennes $P(I)$ indiquant sa pertinence et $S(I)$ indiquant sa pertinence et sa satisfaction.

Linéarité La propriété $A : B \prec C$ nécessite les instances I de la forme :

$$A : B \prec C \ @ \ \langle w_{i_0j_0}, w_{i_1j_1}, w_{i_2j_2} \rangle$$

où $w_{i_1j_1}$ et $w_{i_2j_2}$ sont des fils de $w_{i_0j_0}$ et $w_{i_1j_1} \neq w_{i_2j_2}$. L'instance I est pertinente si le nœud $w_{i_0j_0}$ est actif, si les nœuds $w_{i_1j_1}$ et $w_{i_2j_2}$ sont ses enfants, et chaque nœud est étiqueté par la catégorie demandée :

$$P(I) \Leftrightarrow \left(\begin{array}{l} w_{i_0j_0} \in V^+ \wedge \mathsf{cat}(w_{i_0j_0}) = A \wedge \\ w_{i_1j_1} \in \downarrow w_{i_0j_0} \wedge w_{i_2j_2} \in \downarrow w_{i_0j_0} \wedge \\ \mathsf{cat}(w_{i_1j_1}) = B \wedge \mathsf{cat}(w_{i_2j_2}) = C \end{array} \right)$$

Sa satisfaction est définie par $j_1 < j_2$, la variable $S(I)$ est donc définie par

$$S(I) \quad \Leftrightarrow \quad P(I) \wedge j_1 < j_2$$

Obligation La propriété $A : \triangle B$ indique que chaque nœud d'étiquette A de l'arbre doit avoir au moins un nœud fils étiqueté B. Cette propriété nécessite donc les instances I de la forme :

$$A : \triangle B \ @ \ \langle w_{i_0j_0} \rangle$$

Une instance est pertinente si $w_{i_0j_0}$ est un nœud actif étiqueté par la catégorie A :

$$P(I) \quad \Leftrightarrow \quad w_{i_0j_0} \in V^+ \wedge cat(w_{i_0j_0}) = A$$

Elle est satisfaite si au moins un nœud fils est étiqueté par B :

$$S(I) \quad \Leftrightarrow \quad P(I) \wedge \bigvee_{w_{ij} \in \downarrow w_{i_0j_0}} cat(w_{ij}) = B$$

Unicité La propriété $A : B!$ indique que chaque nœud d'étiquette A de l'arbre doit avoir au plus un fils étiqueté B, l'existence est cependant optionnelle. Cette propriété nécessite les instances I de la forme :

$$A : B! \ @ \ \langle w_{i_0j_0}, w_{i_1j_1}, w_{i_2j_2} \rangle$$

où $w_{i_1j_1}$ et $w_{i_2j_2}$ sont des fils de $w_{i_0j_0}$ et $w_{i_1j_1} \neq w_{i_2j_2}$. L'instance I est pertinente si le nœud $w_{i_0j_0}$ est actif et étiqueté par A et les nœuds $w_{i_1j_1}$ et $w_{i_2j_2}$ sont ses enfants et sont étiquetés par B :

$$P(I) \Leftrightarrow \left(\begin{array}{l} w_{i_0j_0} \in V^+ \wedge \mathsf{cat}(w_{i_0j_0}) = A \wedge \\ w_{i_1j_1} \in \downarrow w_{i_0j_0} \wedge w_{i_2j_2} \in \downarrow w_{i_0j_0} \wedge \\ \mathsf{cat}(w_{i_1j_1}) = B \wedge \mathsf{cat}(w_{i_2j_2}) = B \end{array} \right)$$

Elle est satisfaite si $w_{i_1j_1}$ et $w_{i_2j_2}$ sont le même nœud :

$$S(I) \quad \Leftrightarrow \quad P(I) \wedge w_{i_1j_1} = w_{i_2j_2}$$

Exigence La propriété $A : B \Rightarrow C$ indique que pour tout nœud d'étiquette A de l'arbre, la présence d'un nœud fils étiqueté B implique la présence d'un nœud fils étiqueté C. Cette propriété nécessite donc les instances I de la forme :

$$A : B \Rightarrow C \ @ \ \langle w_{i_0j_0}, w_{i_1j_1} \rangle$$

Une instance est pertinente si le nœud $w_{i_0j_0}$ est actif et étiqueté par A et le nœud $w_{i_1j_1}$ étiqueté par B est un de ses fils :

$$P(I) \Leftrightarrow \left(\begin{array}{l} w_{i_0j_0} \in V^+ \wedge w_{i_1j_1} \in \downarrow w_{i_0j_0} \wedge \\ \mathsf{cat}(w_{i_0j_0}) = A \wedge \mathsf{cat}(w_{i_1j_1}) = B \end{array} \right)$$

Elle est satisfaite si un nœud fils de $w_{i_0j_0}$ est étiqueté par la catégorie C :

$$S(I) \quad \Leftrightarrow \quad P(I) \wedge \bigvee_{w_{ij} \in \downarrow w_{i_0j_0}} \mathsf{cat}(w_{ij}) = C$$

Exclusion La propriété $A : B \not\Leftarrow C$ indique qu'aucun nœud d'étiquette A de l'arbre ne peut avoir à la fois un nœud fils étiqueté B et un nœud fils étiqueté C. Cette propriété nécessite alors les instances I de la forme :

$$A : B \not\Leftarrow C \ @ \ \langle w_{i_0j_0}, w_{i_1j_1}, w_{i_2j_2} \rangle$$

où $w_{i_1j_1}$ et $w_{i_2j_2}$ sont des fils de $w_{i_0j_0}$ et $w_{i_1j_1} \neq w_{i_2j_2}$. L'instance I est pertinente si le nœud $w_{i_0j_0}$ est actif et étiqueté par A et les nœuds $w_{i_1j_1}$ et $w_{i_2j_2}$ sont ses enfants avec soit $w_{i_1j_1}$ est étiqueté par B soit $w_{i_2j_2}$ est étiqueté par C :

$$P(I) \Leftrightarrow \left(\begin{array}{l} w_{i_0j_0} \in V^+ \wedge \mathsf{cat}(w_{i_0j_0}) = A \wedge \\ w_{i_1j_1} \in \downarrow w_{i_0j_0} \wedge w_{i_2j_2} \in \downarrow w_{i_0j_0} \wedge \\ (\mathsf{cat}(w_{i_1j_1}) = B \vee \mathsf{cat}(w_{i_2j_2}) = C) \end{array} \right)$$

Sa pertinence et satisfaction est définie par :

$$S(I) \Leftrightarrow P(I) \wedge (\mathsf{cat}(w_{i_1j_1}) \neq B \vee \mathsf{cat}(w_{i_2j_2}) \neq C)$$

Constituance La propriété $A : S$ indique que pour tout nœud d'étiquette A de l'arbre, les étiquettes des nœuds fils appartiennent à l'ensemble S. Cette propriété nécessite les instances I de la forme :

$$A : S \ @ \ \langle w_{i_0j_0}, w_{i_1j_1} \rangle$$

Une instance est pertinente si le nœud $w_{i_0j_0}$ est actif et étiqueté par A et le nœud $w_{i_1j_1}$ est un de ses enfants :

$$P(I) \Leftrightarrow \left(\begin{array}{l} w_{i_0j_0} \in V^+ \wedge w_{i_1j_1} \in \downarrow w_{i_0j_0} \wedge \\ \mathsf{cat}(w_{i_0j_0}) = A \end{array} \right)$$

Elle est satisfaite si la catégorie étiquetant le nœud $w_{i_1j_1}$ est dans S :

$$S(I) \quad \Leftrightarrow \quad P(I) \wedge \mathsf{cat}(w_{i_1j_1}) \in S$$

Contrainte d'optimalité Une instance est comptée si elle est pertinente. Elle est comptée comme positive si elle est pertinente et satisfaite, et comme négative sinon. Soient $\mathcal{I}$ l'ensemble de tous les instances, $\mathcal{I}^0$ l'ensemble des instances pertinentes et $\mathcal{I}^+$ l'ensemble des instances positives. Le degré de satisfaction des propriétés pour l'arbre est donc déterminé par le ratio $|\mathcal{I}^+|/|\mathcal{I}^0|$. Dans le cadre de la sémantique forte, les modèles sont ceux dont ce ratio vaut 1, car toute instance pertinente est satisfaite. Dans le cadre de la sémantique relâchée, il est possible qu'il existe des instances pertinentes mais non satisfaites. Dans ce cadre, les quasi-modèles sont ceux qui maximisent ce ratio.

5.3 Réalisation des contraintes

Nous considérons toujours une grille de taille $n \times m$. Pour chaque propriété nous considérons toutes les instances possibles à partir de chaque point de la grille. Puisque les arbres sont rectangulaires, pour chaque nœud $w_{i_0j_0}$ les nœuds fils sont dans le rectangle $DR(w_{i_0j_0})$ en dessous à droite de $w_{i_0j_0}$ sur la grille. Soient i le numéro de ligne de $w_{i_0j_0}$ et j son numéro de colonne. Le rectangle $DR(w_{i_0j_0})$ est formé par les points (i', j') avec $1 \leq i' < i$ et $j \leq j' \leq m$. Pour les instances de la forme $A : \psi \ @ \ \langle w_{i_0j_0}, w_{i_1j_1} \rangle$, le couple $\langle w_{i_0j_0}, w_{i_1j_1} \rangle$ se forme donc par $w_{i_0j_0}$ et un nœud $w_{i_1j_1} \in DR(w_{i_0j_0})$. Et pour les instances de la forme $A : \psi \ @ \ \langle w_{i_0j_0}, w_{i_1j_1}, w_{i_2j_2} \rangle$, le triplet $\langle w_{i_0j_0}, w_{i_1j_1}, w_{i_2j_2} \rangle$ se forme par $w_{i_0j_0}$ et deux nœuds $w_{i_1j_1}, w_{i_2j_2} \in DR(w_{i_0j_0})$. Étant données une grille et une grammaire, nous pouvons donc calculer le nombre d'instances possibles. Soit $\mathcal{I}$ l'ensemble de ces instances. En considérant les propriétés dans un ordre fixé (par exemple l'ordre de leur apparition dans la grammaire), nous pouvons ordonner les instances de $\mathcal{I}$ et accéder à chacune par un indice. Remarquons également que le nombre d'instances ne dépend que de la grammaire et de la taille de la grille, il ne dépend pas de l'expression à analyser.

Les variables booléennes $P(I)$ et $S(I)$ sont donc organisées en deux tableaux P et S indexés sur les instances de $\mathcal{I}$. Les conditions de pertinence et de satisfaction sont pour la plupart des propriétés des conjonctions et/ou disjonctions des contraintes d'égalité ou d'appartenance. Les appartenances sont représentées par des variables booléennes, dont la valeur est définie par une contrainte réifiée. Les valeurs de $P(I)$ et $S(I)$ pour chaque instance I sont également reliées à ces conditions par des contraintes réifiées.

Dans le cadre de la sémantique relâchée, les modèles recherchés sont ceux qui maximisent le ratio $|\mathcal{I}^+|/|\mathcal{I}^0|$, où $\mathcal{I}^0$ est l'ensemble des instances pertinentes et $\mathcal{I}^+$ l'ensemble des instances pertinentes et satisfaites. Du fait que pour chaque instance I, les variables $P(I)$ et $S(I)$ sont booléennes, leur valeurs peuvent être considérées comme 0 ou 1. La cardinalité de ces ensembles peut être calculée par :

$$|\mathcal{I}^0| = \sum_{I \in \mathcal{I}} P(I) \qquad |\mathcal{I}^+| = \sum_{I \in \mathcal{I}} S(I)$$

Dans le cadre de la sémantique forte, du fait que chaque instance pertinente doit aussi être satisfaite, il suffit d'ajouter, pour chaque instance I, une contrainte d'implication $P(I) \Rightarrow S(I)$.

6 Implantation d'un prototype

L'approche à base de CSP décrite dans cet article a été implantée en utilisant la bibliothèque de programmation par contraintes Gecode [9].

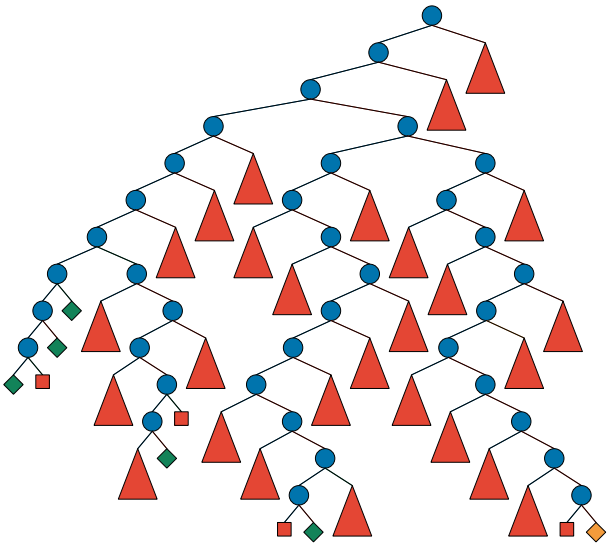


FIG. 1 – Arbre de recherche de l'analyse optimale de "Pierre mange la pomme"

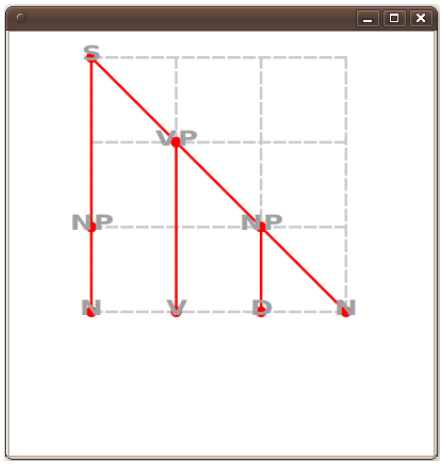


FIG. 2 – Arbre syntaxique optimal de "Pierre mange la pomme"

Notons que le but ici n'est pas tellement de développer un analyseur performant, en effet les analyseurs à base de modèles probabilistes sont actuellement bien plus rapides que les analyseurs à base de modèles symboliques. L'intérêt de notre approche est plutôt d'étudier les conséquences logiques d'une modélisation des grammaires de propriétés en théorie des modèles, et dans ce contexte, de pouvoir évaluer le degré de grammaticalité d'un énoncé.

Comme cela a été mentionné en section 5, les définitions de la pertinence $P(I)$ et satisfaction $S(I)$ d'instance de propriété sont représentées au moyen de contraintes réifiées. La recherche d'un arbre syntaxique optimal est réalisée en utilisant une stratégie de recherche de type *branch-and-bound* maximisant le ratio $|\mathcal{I}^+|/|\mathcal{I}^0|$.

La figure 1 donne un ordre d'idée de la taille de l'arbre de recherche de l'analyse syntaxique optimale pour l'exemple de la section 3 (grammaire de propriétés $\mathcal{G}$ et phrase "Pierre mange la pomme").[8] Cet arbre comporte près de 450 000 nœuds, correspondant à des points de choix lors de la recherche des valeurs à assigner aux variables de notre CSP. Parmi ces 450 000 nœuds, 6 sont des solutions (représentées sous forme de losanges dans l'arbre de recherche). La solution optimale est colorée en orange et correspond à l'arbre syntaxique de la figure 2.

Ce prototype est encore en cours de développement, en particulier il ne bénéficie pas encore d'une interface graphique d'entrée, ni d'un éditeur de grammaires de propriétés. Cependant, il est disponible sous licence GPL sur demande.

Actuellement, il est utilisé avec une grammaire jouet, néanmoins nous prévoyons de l'utiliser avec la grammaire du français de Prost [12].

7 Comparaison avec les travaux existants

Les grammaires de propriétés sont un formalisme relativement jeune. Néanmoins, nous ne sommes pas les premiers à nous être intéressés au problème de l'analyse syntaxique pour ce formalisme.

En particulier, nous pouvons citer les travaux de van Rullen [14], ou encore ceux cités précédemment de Prost [12]. La différence majeure avec notre approche réside dans le fait que nous nous basons sur une sémantique formelle des grammaires de propriétés en théorie des contraintes, permettant une modélisétion naturelle de l'analyse sous forme de CSP.

Cette analyse de GP sous forme de CSP n'est pas sans rappeler les travaux de Estratat et Henocque sur les grammaires de configuration [8]. Dans leurs travaux, les auteurs traduisent une grammaire de propriété décrivant un fragment de la langue naturelle, sous forme d'un modèle objet contraint, représenté avec le langage Z. La description grammaticale résultante est ensuite passée à un configurateur, dont le rôle est de calculer les analyses syntaxiques. Il convient de noter que les auteurs proposent un cadre grammatical général censé permettre l'analyse syntaxique pour plusieurs formalismes grammaticaux, pas uniquement les grammaires de propriétés.

La différence principale entre l'approche d'Estratat et Henocque et la notre, est que le passage par

[8] Cette représentation graphique de l'arbre de recherche est obtenue au moyen de l'outil *Gist* intégré à Gecode.

un configurateur traitant une description grammaticale permet uniquement l'analyse d'énoncés grammaticaux. L'un des atouts des grammaires de propriétés résidant dans la possibilité d'associer à un énoncé agrammatical une valeur de grammaticalité, il est préférable de pouvoir utiliser cette caractéristique du formalisme. La sémantique formelle de Duchier *et al* [7] à base de contraintes relâchées permet cela.

8 Conclusion

Duchier *et al* [7] ont défini formellement une sémantique en théorie des modèles pour les grammaires de propriétés. Dans cet article, nous avons présenté une modélisation en CSP de cette formalisation. Cette modélisation ouvre la voie à l'implantation d'un analyseur syntaxique à base de contraintes, calculant des arbres syntaxiques optimaux (en termes de grammaticalité d'énoncé). Un prototype d'analyseur a été développé, et a permis de commencer à expérimenter l'analyse d'énoncés grammaticaux et agrammaticaux.

Dans sa version actuelle, l'analyseur décrit ici manipule un grand nombre d'instances de propriétés, même sur des énoncés de taille réduite, ce qui se traduit par un arbre de recherche de grande taille. Dans ce contexte, nous souhaitons optimiser la recherche des solutions. L'une des pistes dans ce sens, correspond à paralléliser l'exploration de l'arbre de recherche des solutions du CSP, en utilisant par exemple les travaux de [5]. Une autre piste consisterait à coupler un reconnaisseur de constituants probabliste à l'analyseur pour réduire le nombre de constituants manipulés et ainsi le nombre de contraintes.

Enfin, nous allons travailler au développement d'un moteur de recherche de solutions dédié, car le moteur utilisé actuellement ne permet de conserver que la meilleure solution du CSP, or nous souhaiterions conserver toutes les meilleures solutions lorsqu'il y a plusieurs modèles de même score.

Remerciements

Merci à Sylvie Billot, Mathieu Lopez, Jean-Philippe Prost et Isabelle Tellier pour les interactions fructueuses sur ce travail.

Références

[1] Nicolas Beldiceanu, Pierre Flener, and Xavier Lorca. The tree constraint. In *CPAIOR*, pages 64–78, 2005.

[2] Philippe Blache. *Les Grammaires de Propriétés : des contraintes pour le traitement automatique des langues naturelles*. Hermès Sciences, 2001.

[3] Philippe Blache. Property Grammars : a Fully Constraint-Based Theory. In J. Villadsen H. Christiansen, P. Rossen Skadhauge, editor, *Constraint Solving and Language Processing*, volume 3438 of *Lecture Notes in Artificial Intelligence*, pages 1–16. Springer, 2004.

[4] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory IT-2*, 2(3) :113–124, 1956.

[5] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In *15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *LNCS*, pages 226–241, Lisbon, Portugal, 2009. Springer.

[6] Denys Duchier. Configuration of labeled trees under lexicalized constraints and principles. *Journal of Research on Language and Computation*, 1(3/4), September 2003.

[7] Denys Duchier, Jean-Philippe Prost, and Thi-Bich-Hanh Dao. A model-theoretic framework for grammaticality judgements. In *Conference on Formal Grammar (FG2009)*, Bordeaux, France, June 2009.

[8] Mathieu Estratat and Laurent Henocque. Les grammaires de configuration : un cadre grammatical moderne. In *Troisièmes Journées Francophones de Programmation par Contraintes (JFPC07)*, Rocquencourt, France, 2007.

[9] Gecode Team. Gecode : Generic constraint development environment, 2010. Available from `http ://www.gecode.org`.

[10] Aravind Joshi, Leon Levy, and Masako Takahashi. Tree adjunct grammars. *Journal of Computer and System Sciences*, 10(1) :136–163, 1975.

[11] Patrick Prosser and Chris Unsworth. Rooted tree and spanning tree constraints. In *17th ECAI Workshop on Modelling and Solving Problems with Constraints*, 2006.

[12] Jean-Philippe Prost. *Modelling Syntactic Gradience with Loose Constraint-based Parsing*. Cotutelle Ph.D. Thesis, Macquarie University, Sydney, Australia, and Université de Provence, Aix-en-Provence, France, December 2008.

[13] Geoffrey Pullum and Barbara Scholz. On the Distinction Between Model-Theoretic and Generative-Enumerative Syntactic Frameworks. In *Logical Aspects of Computational Linguistics : 4th International Conference*, volume 2099 of *LNAI*, pages 17–43. Springer, 2001.

[14] Tristan van Rullen. *Vers une analyse syntaxique à granularité variable*. PhD thesis, Université de Provence, Aix-Marseille 1, France, 2005.

Vérification de consistance pour la contrainte de bin packing revisitée

Julien Dupuis[1] **Pierre Schaus**[2] **Yves Deville**[1]

[1] Departement d'ingénierie informatique, UCLouvain, Belgique
[2] Dynadec Europe, Belgique
{julien.dupuis,yves.deville}@uclouvain.be pschaus@dynadec.com

Résumé

En plus d'un algorithme de filtrage, la contrainte `Pack` pour le bin packing à une dimension introduite par Shaw [Shaw04] utilise un algorithme de détection d'inconsistance. Ce test se base sur une réduction de la solution partielle à un problème de bin packing et sur le calcul d'une borne inférieure du nombre de boîtes sur le problème réduit. Ce papier propose deux nouveaux algorithmes de rédution et prouve que l'un d'eux domine théoriquement les autres. Les résultats expérimentaux montrent qu'une combinaison de ces réductions améliore la qualité du filtrage.

1 Introduction

Le problème de bin packing (BP) à une dimension consiste en la recherche du nombre minimal de boîtes nécessaires pour placer un ensemble d'objets de sorte que la taille totale des objets dans chaque boîte ne dépasse pas la capacité C des boîtes. La capacité est commune à toutes les boîtes.

Ce problème peut être résolu en programmation par contraintes (CP) en introduisant une variable de placement x_i pour chaque objet et une variable de charge l_j pour chaque boîte.

La contrainte `Pack` introduite par Shaw [9] lie les variables de placement $x_1, \dots, x_n$ de n objets ayant les poids $w_1, \dots, w_n$ avec les variables de charge de m boîtes $l_1, \dots, l_m$ ayant pour domaines $\{0, \dots, C\}$. Plus précisément, la contrainte s'assure que $\forall j \in \{1, \dots, m\} : l_j = \sum_{i=1}^{n} (x_i = j) \cdot w_i$ où $x_i = j$ est réifiée à 1 en cas d'égalité et à 0 autrement. La contrainte `Pack` a été utilisée dans diverses applications, comme les problèmes d'équilibrage de lignes d'assemblage [5], les problèmes de Steel Mill Slab [2] et les problèmes de Nurse Rostering [6].

En plus des contraintes de décomposition $\forall j \in \{1, \dots, m\} : l_j = \sum_{i=1}^{n} (x_i = j) \cdot w_i$ et de la contrainte redondante $\sum_{i=1}^{n} w_i = \sum_{j=1}^{n} l_j$, Shaw a introduit :

1. un algorithme de filtrage basé sur des raisonnements de sac à dos à l'intérieur de chaque boîte, et
2. un algorithme de détection d'inconsistances basé sur une réduction de la solution partielle à un problème de bin packing.

Le présent travail se concentre sur l'amélioration de l'algorithme de détection d'inconsistances.

2 Reductions à des problèmes de bin packing

Shaw [9] décrit un algorithme rapide de détection d'inconsistances pour la contrainte `Pack` utilisant une borne inférieure sur nombre de boîtes (bin packing lower bound : BPLB). L'idée est de réduire l'assignement partiel courant des variables (c'est-à-dire que certains objets sont déjà assignés à une boîte) de la contrainte `Pack` à un problème de bin packing. Ensuite, une inconsistance est détectée si la borne inférieure est plus grande que le nombre m de boîtes disponibles.

Nous proposons deux nouvelles réductions de la solution partielle à un problème de bin packing. La première peut dans certains cas dominer la réduction de Shaw, tandis que la seconde domine en théorie les deux autres.

Réduction de Paul Shaw : R0 La réduction de Shaw consiste à créer un problème de bin packing avec les propriétés suivantes : la capacité des boîtes est la plus grande borne supérieure des variables de charge,

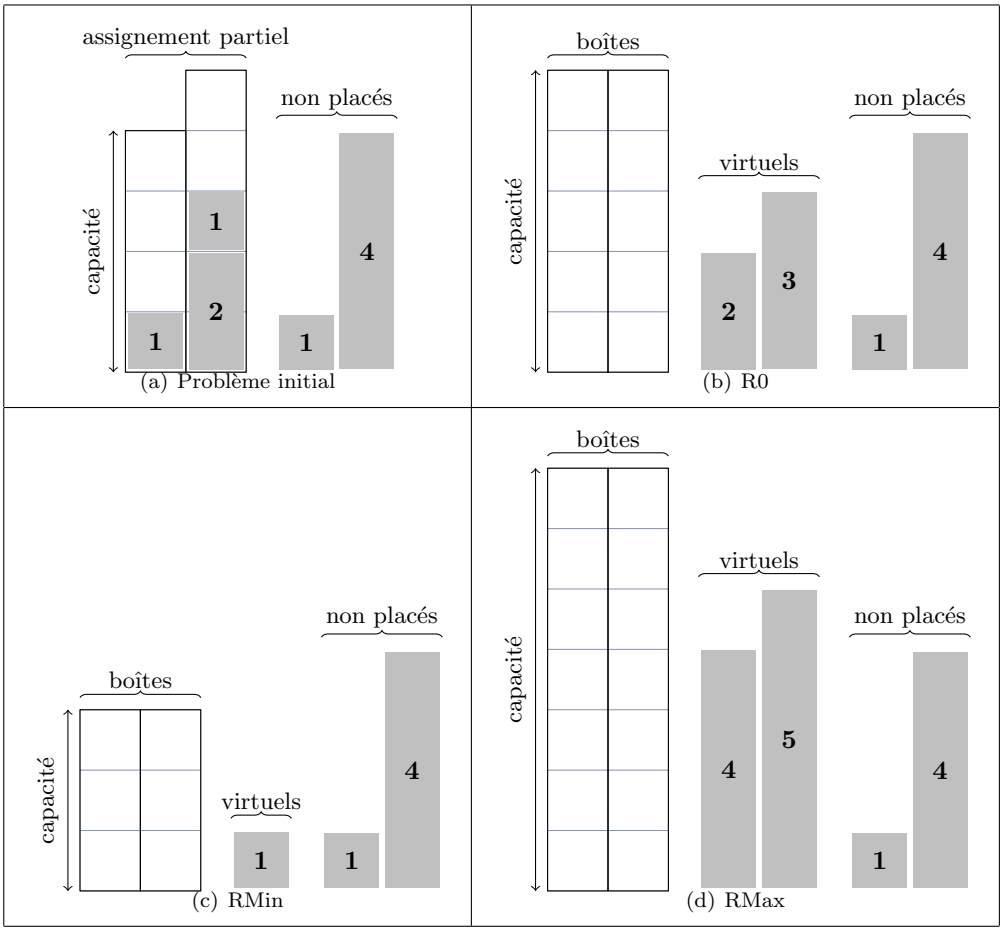


FIG. 1 – Exemple des trois réductions pour le problème de bin packing

c'est-à-dire $c = \max_{j\in\{1,\ldots,m\}}(l_j^{\max})$. Tous les objets qui ne sont pas assignés à une boîte font partie des objets du problème réduit. De plus, pour chaque boîte, un objet virtuel est ajouté au problème réduit pour représenter (1) la dissimilarité de borne supérieure des variables de charge et (2) les objets déjà placés. Plus précisément, la taille de l'objet virtuel d'une boîte j est $(c - l_j^{\max} + \sum_{\{i|x_i=j\}} w_i)$, c'est-à-dire la capacité c réduite de la capacité réelle de la boîte, plus la taille totale des objets qui sont déjà placés dans cette boîte. Un exemple est montré à la figure 1(b).

RMin Nous introduisons RMin qui est obtenu à partir de R0 en réduisant la capacité des boîtes et la taille de tous les objets virtuels de la taille du plus petit objet virtuel. Les objets virtuels ont pour taille $(c - l_j^{\max} + \sum_{\{i|x_i=j\}} w_i - min_k(c - l_k^{\max} + \sum_{\{i|x_i=k\}} w_i))$. Cette réduction est illustrée à la figure 1(c).

RMax Nous proposons RMax qui consiste à augmenter la capacité et la taille des objets virtuels par une unique quantité, de sorte que, lorsque les objets sont placés par un algorithme de bin packing, il soit garanti que les objets virtuels occupent chacun une boîte différente. Afin que ceci soit vérifié, la taille de chaque objet virtuel doit être plus grande que la moitié de la capacité des boîtes.

Dans R0, appelons p la taille du plus petit objet virtuel, et c la capacité des boîtes. La taille des objets virtuels et la capacité doivent être augmentées de $(c - 2p + 1)$. Le plus petit objet virtuel aura ainsi une taille de $s = (c - p - 1)$ et la capacité des boîtes sera $(2c - 2p + 1) = 2s - 1$. On peut facilement remarquer que le plus petit objet virtuel a une taille plus grande que la moitié de la capacité. Si $c = 2p - 1$, cette rédution est équivalente à celle de Shaw. Notons que si $c < 2p-1$, la capacité et la taille des objets virtuels seront réduites.

Les objects virtuels ont une taille de $(2c - 2p + 1 - l_j^{\max} + \sum_{\{i|x_i=j\}} w_i)$. Cette réduction est illustrée à la figure 1(d).

Réduction générique : Rδ Toutes ces réductions sont un cas particulier d'une réduction générique (Rδ) qui, à partir de R0, consiste à ajouter un delta (δ) positif ou négatif à la capacité et à la taille des objets

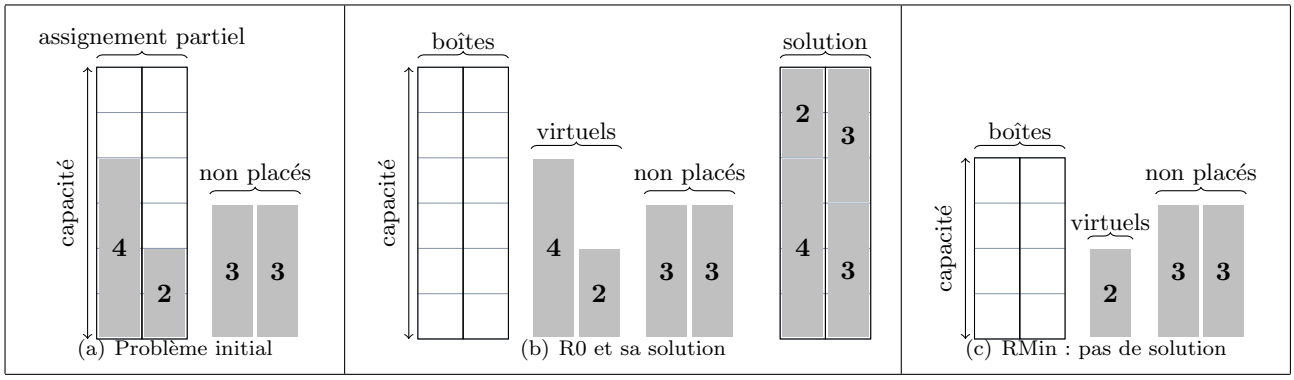


FIG. 2 – Instance de bin packing où R0 ne peut détecter l'inconsistance

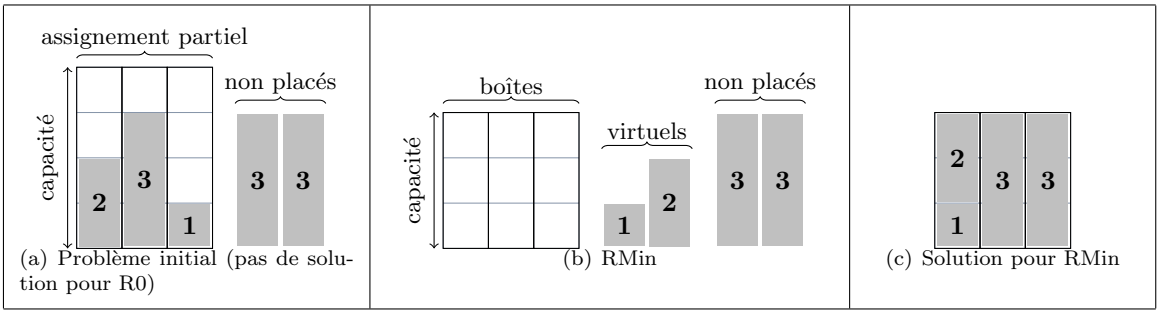


FIG. 3 – Instance de bin packing où RMin ne peut détecter l'inconsistance

virtuels.

Pour R0, $\delta = 0$. Pour RMin, δ est la plus petite valeur gardant toutes les tailles positives. Un plus petit δ créerait une inconsistance, puisque le plus petit objet virtuel aurait une taille négative. δ_{RMin} est toujours négatif ou nul. Pour RMax, δ est la plus petite valeur garantissant que les objets virtuels ne s'empilent pas. Notons que dans certains cas, δ_{RMin} ou δ_{RMax} peuvent être nuls. Notons aussi que δ_{R0} peut être plus grand que les deux autres.

3 Comparaison théorique des trois réductions

Définition Soit A et B deux réductions de la contrainte `Pack` à un problème de bin packing. On dit que A *domine* B si, pour toute instance de la contrainte `Pack`, le nombre de boîtes requises dans A est plus grand que le nombre de boîtes requises dans B.

Théorème Rδ est une relaxation du problème de tester la consistance de la contrainte `Pack`.

Démonstration. Si une solution partielle de la contrainte `Pack` peut être étendue à une solution où tous les objets sont placés, alors Rδ a également une solution : si chaque objet virtuel est placé dans sa boîte initiale, l'espace libre de chaque boîte est égal à l'espace libre dans la solution partielle, et les objets non placés peuvent donc être placés dans la même boîte que dans la solution étendue de l'assignement partiel. □

Théorème Ni R0 ni RMin ne domine l'autre.

Démonstration. La figure 2 montre une instance où R0 a une solution et pas RMin. La figure 3 montre une instance où RMin a une solution et pas R0. □

Théorème RMax est équivalent au problème de tester la consistence de la contrainte `Pack`.

Démonstration. Par le théorème 3, RMax est une relaxation de la solution partielle du problème de bin packing. Il reste à montrer que s'il existe une solution pour RMax, alors la solution partielle peut être étendue à une solution complète de la contrainte `Pack`. Appelons v la boîte d'où vient l'objet virtuel v. Il est garanti, par la taille des objets virtuels, que ceux-ci seront chacun placés dans une boîte b_v différente. L'espace restant dans chaque boîte b_v correspond à l'espace libre de la boîte v dans le problème original. Une solution étendue de la contrainte `Pack` est obtenue en plaçant dans v tous les objets se trouvant dans b_v. □

Corollaire RMax domine R0 et RMin.

TAB. 1 – Comparaison du nombre d'inconsistances détectées avec différents réductions

Instances	Nombre d'inconsistances détectées(%)					
	RMin	R25	R50	R75	RMax	R0
Inst1	74.16	78.87	86.40	89.53	**99.58**	74.79
Inst2	**99.93**	86.75	87.03	87.8	87.15	**99.93**
Inst3	80.64	86.55	93.37	97.75	**99.39**	98.52

D'un point de vue théorique, la réduction RMax est toujours meilleure ou équivalente à R0, RMin et toute autre instance de $R\delta$. En pratique, cependant, ce n'est pas toujours le cas, comme nous le montrons dans la section suivante.

4 Experimental comparison

Le test d'inconsistance de Shaw [9] utilise l'algorithme de borne inférieure de bin packing $\mathcal{L}_2$ de Martello et Toth [4], qui peut être calculé en temps linéaire. Récemment, il a été prouvé [1] que l'algorithme de borne inférieure $\mathcal{L}_3$ de Labbé [3] donne toujours une borne plus grande ou égale à $\mathcal{L}_2$, et bénéficie d'une meilleure performance asymptotique (3/4 pour $\mathcal{L}_3$ [1] et 2/3 pour $\mathcal{L}_2$ [4]), tout en ayant une complexité temporelle linéaire. Les expériences montrent que $\mathcal{L}_3$ permet de détecter environ 20% d'inconsistances en plus que $\mathcal{L}_2$.

Dans les expériences ci-dessous, l'algorithme $\mathcal{L}_3$ est utilisé. Tous les programmes pour les expériences ont été implémentés dans le langage Comet.

Bien qu'en théorie, RMax est toujours meilleur que R0 et RMin, les résultats pratiques sont moins systématiques. Cela est dû au fait que $\mathcal{L}_3$ (tout autant que $\mathcal{L}_2$) n'est pas monotone, ce qui veut dire qu'une instance de bin packing nécessitant un plus grand nombre de boîtes qu'une autre instance peut produire une borne inférieure plus petite que la seconde. En fait, $\mathcal{L}_3$ est mieux adapté aux instances où la plupart des objets ont une taille plus grande que le tiers de la capacité des boîtes. RMax augmente la capacité, rendant ainsi les objets proportionnellement plus petits. Pour chacune des réductions R0, RMin et RMax, il y a des instances pour lesquelles ils contribuent à détecter une inconsistance, alors que les deux autres ne le permettent pas.

La table 1 présente les performances de la détection d'inconsistances en utilisant chacune des réductions. Elle montre le rapport du nombre d'inconsistances détectées en utilisant chacune des réductions sur le nombre total d'inconsistances détectées par au moins un des filtres. Des réductions supplémentaires ont été expérimentées, avec δ étant placé entre δ_{RMin} et δ_{RMax} à 25%, 50% et 75%. Ces résultats ont été obtenus en générant plus de 1000 instances aléatoires et en calculant $\mathcal{L}_3$ sur chacune des réductions. Voici commment ces instances ont été produites :

Inst1 Le nombre de boîtes, le nombre d'objets et la capacité C sont choisis aléatoirement entre 30 et 50. Les boîtes sont déjà remplies jusque $1..C$. La taille des objets est choisie aléatoirement dans $\{1, \ldots, C\}$.

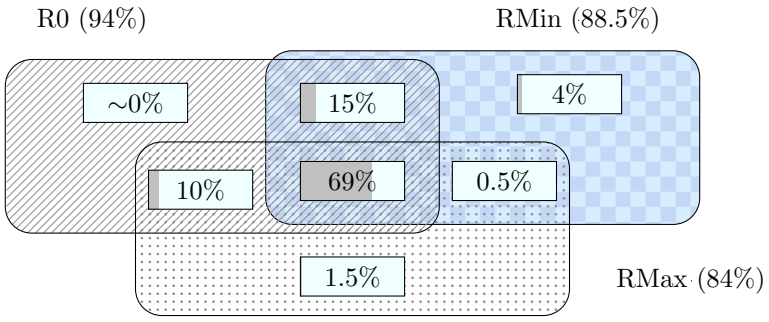
Inst2 Il y a 50 boîtes. La capacité est de 100. Le nombre d'objets est entre 100 et 200. La taille des objets suit une distribution normale ($\mu = 5000/n$, $\sigma \in \{3n, 2n, n, n/2, n/3\}$ où n est le nombre d'objets). Parmi ceux-ci, le pourcentages d'objets déjà placés $\in \{10\%, 20\%, 30\%, 40\%, 50\%\}$.

Inst3 Mêmes paramètres que pour la deuxième instance, mais le pourcentage d'objets déjà placés est de 90% ou 95%.

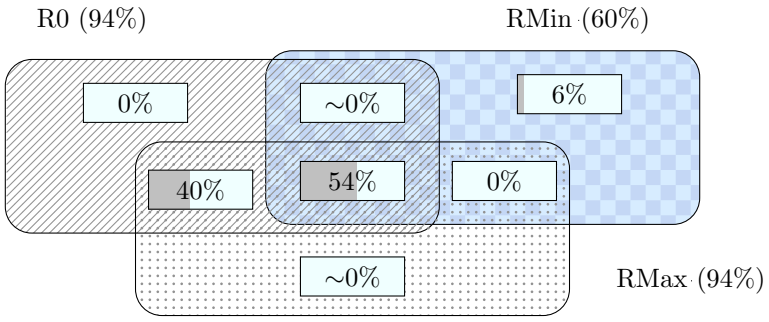
Ceci révèle que certains types d'instances sont mieux adaptées à R0 et RMin, tandis que d'autres sont mieux adaptées à RMax. Les réductions R25, R50 et R75 ne sont jamais meilleures, en moyenne, que RMin et RMax. C'est pourquoi ces réductions intermédiaires ne sont plus utilisées dans les expériences suivantes.

Comparaison sur des données de la litérature. Pour que l'analyse soit plus précise, nous comparons le comportement des trois réductions proposées sur des instances réelles. Des algorithmes CP ont été exécutés sur les instances SALBP-1 de Scholl [7] et sur les instances de bin packing de Scholl [8] (premier jeu de données avec n=50 et n=100), et à chaque changement du domaine des variables, la solution partielle courante a été extraite. 30 000 instances ont été sélectionnées aléatoirement parmi celles-ci, pour chaque jeu de données. Dans le second cas, seules des instances pour lesquelles au moins un des filtres détectait une inconsistance ont été sélectionnées. Les trois réductions ont été appliquées aux instances sélectionnées, avec $\mathcal{L}_3$. La figure 4 donne un schéma des résultats.

Ces résultats montrent que R0 détecte un plus grand nombre d'inconsistances. Mais (presque) toutes ces inconsistances sont également détectés par RMin ou RMax. On peut en conclure que combiner RMin et RMax est meilleur que R0 seul. Il est aussi inutile de combiner R0 avec RMin et RMax.



(a) Instance de SALBP-1



(b) Instances de bin packing

FIG. 4 – Proportions de détetion d'inconsistances en utilisant chaque réduction sur les instances de SALBP-1 (en haut) et les instances de bin packing (en bas)

TAB. 2 – Comparaison des réductions sur la résulution du problème de bin packing

	Pas de filtre	R0	RMin	RMax	RMin & RMax
Nombre de solutions optimales	281	317	315	309	**319**
Temps moyen (s)	5.39	1.88	1.60	3.50	**1.25**

Impact sur une recherche CP. Nous avons comparé l'effet d'appliquer l'algorithme de détection d'inconsistances dans une recherche CP sur les instances de bin packing de Scholl N1 et N2 (360 instances au total), en utilisant R0, RMin, RMax et la combinaison de RMin et RMax, avec une limite de temps de cinq minutes pour chaque instance. Le temps moyen d'exécution a été calculé pour les instances pour lesquelles toutes les réductions menaient à la même solution. Tous ces résultats sont montrés dans la table 2. On peut observer que RMin et RMax combinés trouvent plus de solutions optimales (bien que la différence ne soit pas significative), et mène plus rapidement vers une solution que les autres (accélération de 33% par rapport à R0).

5 Conclusion

Ce papier présentait deux nouvelles réductions d'une solution partielle de la contrainte `Pack` à un problème de bin packing. Lors d'une recherche CP, ces réductions sont soumises à un algorithme de borne inférieure du nombre de bin afin de détecter les inconsistances de la contrainte `Pack`, comme le suggère Shaw [9].

Nous prouvons que notre deuxième réduction (RMax) donne en théorie un meilleur filtrage que les autres, en supposant que l'algorithme de borne inférieure est parfait. Nous concluons que la meilleure stratégie est de considérer conjointement les filtres RMin et RMax lors d'une recherche CP.

Remerciements Le premier auteur est supporté par le FNRS belge (Fonds National de la Recherche Scientifique). Cette recherche est également partiellement supportée par le Programme d'Attraction Interuniversitaire et le projet FRFC 2.4504.10 du FNRS belge.

Références

[1] Jean-Marie Bourjolly and Vianney Rebetez. An analysis of lower bound procedures for the bin packing problem. *Comput. Oper. Res.*, 32(3) :395–405, 2005.

[2] P. Van Hentenryck and L. Michel. The steel mill slab design problem revisited. *CP'AI'OR-08, Paris, France*, 5015 :377–381, May 2008.

[3] Martine Labbé, Gilbert Laporte, and Hélène Mercure. Capacitated vehicle routing on trees. *Operations Research*, 39(4) :616–622, 1991.

[4] S. Martello and P. Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Appl. Math.*, 28(1) :59–70, 1990.

[5] P. Schaus and Y. Deville. A global constraint for bin-packing with precedences : Application to the assembly line balancing problem. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 369–374, Chicago, Illinois, USA, July 2008. AAAI Press.

[6] P. Schaus, P. Van Hentenryck, and J-C. Régin. Scalable load balancing in nurse to patient assignment problems. In *6th International Conference Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, Lecture Notes in Computer Science, Pittsburgh, Pennsylvania, USA, 2009. Springer.

[7] Armin Scholl. Data of assembly line balancing problems. *Technische Universität Darmstadt*, 93.

[8] Armin Scholl, Robert Klein, and Christian Jürgens. Bison : A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & Operations Research*, 24(7) :627 – 645, 1997.

[9] Paul Shaw. A constraint for bin packing. *Principles and Practice of Constraint Programming - CP 2004*, 3258 :648–662, 2004.

D'un plan optimal parallèle vers un plan optimal séquentiel

Stéphane Grandcolas et Cyril Pain-Barre

LSIS – UMR CNRS 6168
Domaine Universitaire de Saint-Jérôme
Avenue Escadrille Normandie-Niemen
F-13397 Marseille Cedex 20 - France
{stephane.grandcolas,cyril.pain-barre}@lsis.org

Résumé

En planification on distingue les plans optimaux en le nombre d'étapes (plans parallèles) et les plans optimaux en le nombre d'actions (plans séquentiels). Il est généralement admis que le calcul d'un plan séquentiel est plus couteux que le calcul d'un plan parallèle. Büttner et Rintanen ont proposé une procédure de recherche qui calcule des plans dont le nombre d'étapes est fixé et le nombre d'actions minimal. Cette procédure est utilisée pour le calcul d'un plan séquentiel optimal partant d'un plan parallèle optimal. Nous décrivons dans cet article une approche de ce type, développée à partir du système de planification FDP. L'idée consiste à maintenir deux structures, l'une représentant le plan parallèle et l'autre le plan séquentiel, en répercutant les choix effectués pendant la recherche simultanément dans les deux structures. Les techniques développées dans FDP pour le calcul de plans séquentiels ou de plans parallèles permettent de détecter les échecs dans les deux structures. Les résultats expérimentaux montrent que cette approche est très compétitive comparée au calcul de plans séquentiels optimaux obtenus avec FDP.

1 Introduction

Lors de la compétition de planificateurs IPC5 en 2006 un nouveau planificateur, FDP [4], est apparu. Ce planificateur a la particularité de produire des plans séquentiels optimaux. On admet généralement que le calcul de tels plans est plus coûteux que le calcul de plans parallèles optimaux. Les plans parallèles ont moins d'étapes, et sont donc plus *contraints*, ce qui tend à aider la recherche. Comme beaucoup d'autres planificateurs optimaux la preuve d'optimalité vient de la démonstration qu'il n'existe pas de plans en moins d'étapes ou d'actions. Ainsi le système cherche des plans en une étape, puis deux,... tant qu'un plan solution n'est pas trouvé et qu'une borne sur le nombre d'étapes ou d'actions n'est pas atteinte. La simplicité du planificateur FDP vient du fait qu'il implémente sa propre procédure de recherche, de type recherche en avant en profondeur : la procédure tente de toutes les façons possibles d'étendre le plan partiel courant. Le principe général est donc d'effectuer des recherches en profondeurs bornées en augmentant la borne itérativement (IDDFS [6] : Iterative Deepening Depth First Search).

FDP a obtenu des résultats honorables dans la catégorie des planificateurs optimaux (la plupart produisent des plans parallèles). Ce succès tient beaucoup à l'efficacité de la procédure de recherche qui utilise de nombreuses techniques pour éviter des traitements redondants de différents types. Notamment, un ordre *a priori* sur les actions, similaire à l'*élagage de la commutativité* [5], permet d'éviter la construction de séquences d'actions redondantes, ce qui est inutile lors de la recherche de plans parallèles. La procédure fait aussi appel à des évaluations grossières du nombre d'actions pour atteindre les buts qui sont utilisées pour abandonner la recherche de façon précoce. Enfin les états dont l'invalidité est démontrée sont mémorisés dans une table de hashage, afin de ne pas les traiter à nouveau lors d'une rencontre future.

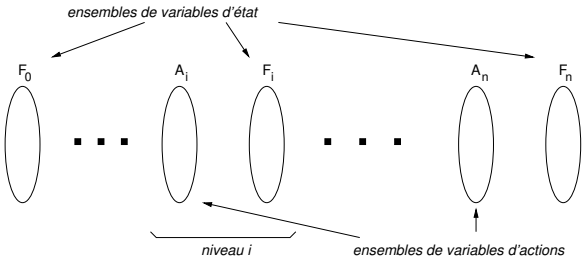
S'il existe peu de travaux sur la recherche de plans séquentiels optimaux, Büttner et Rintanen [2] ont développé un système pour ce calcul à partir d'un plan parallèle optimal. Leur méthode consiste à effectuer des recherches successives à partir de ce plan, en di-

minuant le nombre d'actions quite à accroître, s'il n'y a pas de solution, le nombre d'étapes. La recherche se termine lorsque le nombre d'étapes et le nombre d'actions sont égaux. On espère de cette façon accélérer la recherche du fait que le problème est très contraint d'une part, et du fait qu'on a à traiter des problèmes plutôt satisfiables d'autre part. La procédure de recherche qui est utilisée dans ce cadre détermine si il existe un plan solution en m étapes et contenant au plus n actions. Remarquons que cette procédure en soit est originale et permet par exemple de calculer des plans optimaux contraints sur leur nombre d'actions (en augmentant le nombre d'étapes tant qu'aucune solution n'est rencontrée) ou encore des plans optimaux contraints en leur nombre d'étapes (dans ce cas en augmentant le nombre d'actions). Nous proposons dans cet article, en nous appuyant sur les travaux de G. Gabriel, S. Grandcolas et C. Pain-Barre sur la recherche de plans séquentiels optimaux [4] et de plans parallèles optimaux [3], une approche originale pour le calcul de plans optimaux contraints en leur nombre d'actions ou d'étapes. Le principe consiste à maintenir parallèlement deux structures représentant le plan séquentiel en construction et le plan parallèle correspondant. Les techniques développées dans FDP sont utilisées ici dans les deux structures. Pendant la recherche l'une ou l'autre provoque des échecs. Nous avons implémenté cette approche et l'avons comparée aux résultats obtenus par FDP [1].

Dans la première partie nous revenons brièvement sur le planificateur FDP dans sa version séquentielle et sa version parallèle et sur les FDP-structures qu'il utilise. Ensuite nous présentons la procédure de recherche d'un plan de m étapes et n actions. La troisième partie est consacrée au calcul d'un plan séquentiel optimal à partir d'un plan parallèle optimal comme l'ont proposé Büttner et Rintanen. Enfin dans la quatrième partie sont présentés les résultats expérimentaux que nous avons obtenus.

2 FDP-structures

Un problème de planification $\mathcal{P}$ est un triplet (I, G, A) où I est l'état initial, G est l'ensemble des buts à satisfaire et A est l'ensemble des actions. De $\mathcal{P}$, on en déduit l'ensemble F des fluents du problème. Une action a de A a une précondition $pre(a)$ et des effets $\mathit{eff}(a)$. Une action ajoute un fluent f si $f \in \mathit{eff}(a)$. Elle le supprime si $\neg f \in \mathit{eff}(a)$.
Fondamentalement, le cœur de FDP est une fonction de décision répondant à la question : existe-t-il une solution de longueur n à un problème $\mathcal{P}$? En planification séquentielle, n est le nombre d'actions que doit comporter la solution. En planification parallèle, c'est son nombre d'étapes, où chaque étape comporte un ensemble non vide d'actions compatibles. Pour y répondre, FDP utilise une structure de type CSP de taille n, appelée *fdp-structure*, qui représente un ensemble de plans potentiellement valides de longueur n. La recherche consiste alors à supprimer ou fixer des actions dans la structure, jusqu'à ce qu'elle ne contienne plus qu'un plan valide satisfaisant G, ou qu'un échec soit démontré.
La *fdp-structure* est composée d'un ensemble de variables soumises à des contraintes, partitionnées en ensembles de variables d'état et en ensembles de variables d'actions. Une fdp-structure de longueur n comporte $n+1$ ensembles de variables d'état et n ensembles de variables d'action. Elle peut être vue comme un graphe nivelé, similaire à celui de Graphplan [1], où un niveau i comporte un ensemble de variables d'action A_i et un ensemble de variables d'état F_i :



L'ensemble F_i code l'état après i étapes : pour chaque fluent $f \in F$, une [2] variable d'état f_i indique s'il est vrai, faux ou indéfini. S'il est indéfini, l'état est lui-même partiellement défini. Lorsqu'aucune variable de F_i n'est indéfinie, alors F_i est un état. L'état initial est codé par F_0 et est totalement défini, selon l'hypothèse du monde clos. L'ensemble A_i code les actions de l'étape i. Il diffère selon que la planification est séquentielle ou parallèle. En planification séquentielle, une seule action est autorisée par étape : A_i se réduit alors à une seule variable a_i dont le domaine est l'ensemble A des actions du problème. En planification parallèle, plusieurs actions peuvent être exécutées dans une étape, et A_i contient autant de variables d'actions que d'actions, chacune indiquant si l'action correspondante est fixée, supprimée ou possible à cette étape. On définit ainsi deux fdp-structures : $S_{seq}(n) = (\{F_0^{seq}, \ldots, F_n^{seq}\}, \{A_1^{seq}, \ldots, A_n^{seq}\})$ pour la planification séquentielle et $S_{par}(n) = (\{F_0^{par}, \ldots, F_n^{par}\}, \{A_1^{par}, \ldots, A_n^{par}\})$ pour la planification parallèle.

1. Aucun exécutable de Büttner et Rintanen n'étant disponible, nous n'avons pas comparé notre méthode à la leur.

2. En réalité, nous utilisons deux variables : f_i pour le f et $\neg f_i$ pour $\neg f$. Nous ferons le plus possible abstraction de la seconde pour ne pas surcharger inutilement l'article.

Avant de commencer la recherche d'une solution de longueur n, toutes les variables d'état ont pour domaine $\{vrai, faux\}$ (le fluent correspondant est indéfini). En planification séquentielle, toutes les variables d'actions ont pour domaine l'ensemble des actions du problème. En planification parallèle, leur domaine est $\{vrai, faux\}$. Dans les deux cas, toute action est donc possible à chaque étape. Ensuite, les domaines des variables de F_0 sont réduits pour que F_0 corresponde à l'état initial I. De même, ceux de certaines variables de F_n sont réduits pour correspondre aux buts G. La réduction du domaine d'une variable d'état s'apparente à la suppression d'un littéral : enlever $vrai$ pour f_i signifie que f est supprimé de l'étape i, ne pouvant y être vrai. Par abus de langage, nous dirons que f_i est vrai (faux) si $vrai$ ($faux$) est la seule valeur de son domaine.
Ces "suppressions" rendent incohérentes des valeurs dans le domaine d'autres variables, par propagation. En effet, les variables d'une fdp-structure sont soumises à des contraintes implicites, inhérentes à la planification. Par exemple, si un littéral est supprimé à une étape, alors les actions qui l'ont en précondition peuvent être supprimées de l'étape qui suit. De même, les actions qui ajoutent ce littéral peuvent être supprimées de l'étape qui précède. Cette propagation, similaire au maintien de la consistance d'arc [7], est assurée par la fonction MakeConsistent de FDP. Elle est appelée à la suite de la réduction d'un domaine d'une variable : à l'initialisation de la structure, puis pendant la recherche lorsqu'un choix est opéré (ce qui se traduit par la réduction d'un domaine). Cette fonction rend cohérente une fdp-structure en supprimant des littéraux ou des actions lorsque ceux-ci ne sont plus cohérents, ainsi que les littéraux et les actions que ces suppressions ont rendu incohérents. Elle peut échouer, notamment lorsqu'un littéral et son opposé sont incohérents (tous les deux vrais ou tous les deux faux). Une cause d'échec en planification séquentielle est lorsqu'aucune action n'est possible à une étape. Dans ce cas, la procédure de recherche doit remettre en cause le choix qui a conduit à l'échec.

MakeConsistent utilise les règles suivantes pour déterminer si un littéral ou une action est incohérent. Une seule règle ne s'applique qu'à la planification parallèle et est marquée par (par). Une autre ne s'applique qu'à la planification séquentielle. Elle est marquée par (seq). Par simplicité, on notera A_i l'ensemble des actions possibles (et non choisies) à l'étape i, et X_i l'ensemble des actions fixées à l'étape i. En planification séquentielle, X_i est vide ou A_i est vide et X_i ne peut être qu'un singleton.

Soit un fluent $f \in F$ et f_i sa variable associée à l'étape i. La valeur $vrai$ (resp. $faux$) pour f_i est incohérente si l'une des conditions suivantes est satisfaite :
- $i > 0$ et $\neg f_i$ est vrai (resp. faux)
- $i > 0$ et $\exists a \in X_i$ t.q. $\neg f \in \mathit{eff}(a)$ (resp. $f \in \mathit{eff}(a)$)
- $i < n$ et $\exists a \in X_{i+1}$ t.q. $\neg f \in \mathit{pre}(a)$ (resp. $f \in \mathit{pre}(a)$)
- $i > 0, X_i = \emptyset$ et $\forall a \in A_i,\ \neg f \in \mathit{eff}(a)$ (resp. $f \in \mathit{eff}(a)$)
- $i < n, X_{i+1} = \emptyset$ et $\forall a \in A_{i+1},\ \neg f \in \mathit{pre}(a)$ (resp. $f \in \mathit{pre}(a)$)
- $i > 0, f_{i-1}$ est faux (resp. vrai) et $\forall a \in X_i \cup A_i,\ f \notin \mathit{eff}(a)$ (resp. $\neg f \notin \mathit{eff}(a)$)
- $i < n, f_{i+1}$ est faux (resp. vrai) et $\forall a \in X_{i+1} \cup A_{i+1},\ \neg f \notin \mathit{eff}(a)$ (resp. $f \notin \mathit{eff}(a)$)
- $i > 0, \exists f'_i$ t.q. f'_i est vrai et mutex(f_i, f'_i, i) (resp. mutex($\neg f_i$, f'_i, i)) ou f'_i est faux et mutex(f, $\neg f'_i$, i) (resp. mutex($\neg f_i$, $\neg f'_i$, i)

Une action a est incohérente à une étape $i > 0$ si l'une des conditions suivantes est satisfaite :
- $\exists f \in \mathit{pre}(a)$, t.q. f_{i-1} est faux
- $\exists f \in \mathit{eff}(a)$, t.q. f_{i+1} est faux
- (par) $\exists a' \in X_i$ t.q. mutex(a, a', i)
- (seq) $\exists f \in F$ t.q. f_i est faux (resp. vrai), f_{i+1} est vrai (resp. faux), et $f \notin \mathit{eff}(a)$ (resp. $\neg f \notin \mathit{eff}(a)$)
- $\exists f \in \mathit{pre}(a)$ t.q. $\neg f \notin \mathit{eff}(a)$ et f_{i+1} est faux

Cette dernière règle est nouvelle, ainsi que les règles concernant les *mutex*. Il faut noter que les mutex ne sont pas les mêmes selon que la planification est séquentielle ou parallèle. Pour la planification parallèle, nous avons adopté la définition de Graphplan [1] mais nous l'avons étendue aux littéraux négatifs : un littéral et un littéral négatif peuvent être marqués comme mutex à une étape, de même que deux littéraux négatifs. Pour la planification séquentielle, à notre connaissance, la notion de mutex n'avait pas encore été définie. Puisqu'il n'y a qu'une action par étape, toutes les actions sont mutex entre elles. Pour les littéraux, la définition est la suivante.

Deux littéraux l_1 et l_2 sont marqués mutex à l'étape i si l'une des conditions suivantes est satisfaite :
- l_1 et l_2 sont deux littéraux opposés
- l_1 et l_2 apparaissent pour la première fois à l'étape i et aucune action de cette étape ne les ajoute tous les deux
- l_1 apparaît pour la première fois à l'étape i et l_2 était apparu à une étape précédente : aucune action de l'étape i ne les ajoute tous les deux et toute action qui ajoute l_1 en i supprime l_2 ou requiert un littéral l où mutex(l, l_2, $i-1$)
- l_1 et l_2 sont déjà apparus à une étape précédente : mutex(l_1, l_2, $i-1$) et aucune action de i ne les ajoute tous les deux, et toute action de i qui ajoute l'un supprime l'autre ou requiert un littéral mutex avec l'autre à l'étape $i-1$

Function Search(S_{seq}, s, n, S_{par}, l, m)

Data: $S_{seq} = (A^q, F^q, n)$ a FDP-structure of length n, the first s steps are instanciated,
Data: $S_{par} = (A^p, F^p, m)$ a FDP-structure of length m, the first l steps are instanciated,
Result: $TRUE$ if there exists a plan in these structures, $FALSE$ in the other case.

```
begin
    if s > n or l > m then
        return TRUE;
    C := A_s^seq ∩ A_l^par;
    for a ∈ C do
        remove all actions from A_s^seq but a;
        if not MakeConsistent(S_seq) then
            goto REVERT_SEQ;
        if s + 1 ≤ n and ⟨F_{s+1}^seq, A_{s+1}^seq ∩ A_l^par, n − (s + 1), m − l⟩ ∈ H then
            goto REVERT_SEQ;
        fix the action a at step l in S_par;
        if MakeConsistent(S_par) then
            if Search(S_seq, s + 1, n, S_par, l, m) then
                return TRUE;
            if s + 1 ≤ n then
                H := ∪{⟨F_{s+1}^seq, A_{s+1}^seq ∩ A_l^par, n − (s + 1), m − l⟩};
        Revert(S_par);
     REVERT_SEQ :
        Revert(S_seq);
    if l < m then
        remove unfixed actions from A_l^par;
        remove C actions from A_s^seq;
        if MakeConsistent(S_par) and MakeConsistent(S_seq) then
            if Search(S_seq, s, n, S_par, l + 1, m) then
                return TRUE;
        Revert(S_seq), Revert(S_par);
    return FALSE;
end
```

3 Méthode de recherche d'un plan m étapes-n actions

La fonction Search détermine si il existe un plan avec n actions et m étapes en énumérant tous les plans possibles simultanément dans la structure séquentielle S_{seq} et dans la structure parallèle S_{par}. Cette dernière représente le plan séquentiel en construction, en regroupant dans chaque niveau les actions successives qui sont indépendantes. Toute suppression effectuée dans l'une des deux structures est répercutée dans l'autre. La recherche progresse de l'état initial vers l'état final, ce qui signifie qu'à tout moment les états précédent l'étape courante s dans la structure séquentielle sont complètement définis, de même que tous les états précédents le niveau l dans la structure parallèle.

La fonction est appelée initialement avec la valeur 0 pour s et l. Dans chaque appel toutes les actions qui apparaissent à la fois à l'étape s de la structure séquentielle et au niveau l de la structure parallèle sont choisies tour à tour afin de déterminer si le plan partiel courant peut être étendu en un plan valide avec cette action. Si aucune solution n'est produite avec ces actions, alors les actions qui ne figurent pas dans la structure parallèle sont considérées. Puisque elles n'apparaissent pas au niveau l de la structure S_{par} il faut les chercher au niveau $l + 1$. Ce changement de niveau signifie que toutes les actions indéfinies du niveau l doivent être supprimées. Chaque fois que des actions sont supprimées ou fixées dans une structure, les conséquences de ces modifications sont propagées à travers la structure par filtrage, avec l'appel de la fonction MakeConsistent qui a été présentée dans la section précédente. L'échec du choix de l'action a peut provenir de son inconsistance dans le plan séquentiel ou de son inconsistance dans le plan parallèle.

Mémorisation des échecs

Le planificateur FDP mémorise les échecs rencontrés pendant la recherche, afin de ne pas reproduire ces recherches inutiles par la suite. Chaque fois que le planificateur démontre qu'il n'est pas possible de satis-

faire les buts à partir de l'état courant F en le nombre d'étapes restant d, le couple $\langle F, d\rangle$ est enregistré, signifiant qu'il est inutile d'essayer d'atteindre les buts à partir de l'état F si il n'y a pas plus de d étapes.

Nous proposons de mémoriser les échecs de façon similaire. Chaque fois qu'il y a un changement de niveau dans la structure parallèle, le triplet $\langle F, na, ns\rangle$ est enregistré, dans lequel F est l'état courant (c'est le même dans les deux structures), na est le nombre d'actions potentielles et ns est le nombre d'étapes restant dans la structure parallèle. Cette technique n'est pas très efficace : de nombreux choix sont effectués sans qu'il y ait de changement de niveau dans la structure parallèle, peu d'états sont mémorisés, et il faut parfois attendre longtemps avant de changer de niveau et détecter un échec. La solution consiste à mémoriser les états à tout moment, c'est à dire chaque fois qu'une action est choisie dans la structure séquentielle. Mais cette mémorisation invalide la recherche. Supposons par exemple qu'en fixant l'action a dans la structure parallèle, une action b du même niveau soit supprimée du fait qu'une précondition de a figure parmi ses suppressions (voir figure 1). L'état F résultant de l'application de a dans la structure séquentielle (en jaune dans la figure) ne conduit pas aux buts, il est donc mémorisé. Puisque l'action b a été supprimée de la structure parallèle, elle ne pourra donc pas être choisie dans le plan séquentiel à l'étape $s+1$ bien qu'elle soit applicable dans F. Si une solution existe avec cette action elle ne sera jamais découverte.

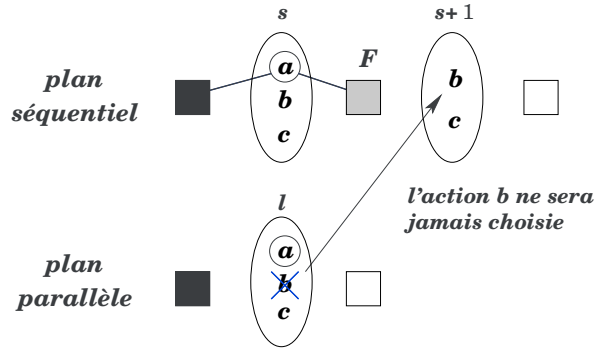


FIGURE 1 – mémorisation de l'état jaune

Pour remédier à ce problème une solution consiste à mémoriser avec l'état F les actions qui lui sont applicables dans la structure séquentielle et qui sont indéfinies dans la structure parallèle (voir figure 2), sous la forme d'un quadruplet de la forme $\langle F, U, na, ns\rangle$.

Les quadruplets représentant les échecs sont mémorisés dans la table H. Chaque fois qu'une action est choisie dans la structure séquentielle, avant de relancer la recherche récursivement, on vérifie que l'état courant n'apparaît pas déja dans la table H avec un

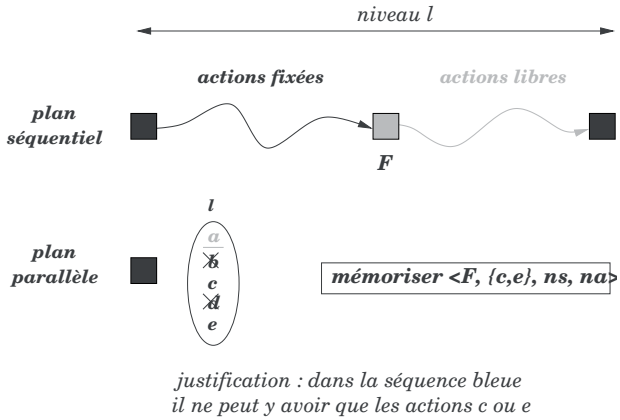


FIGURE 2 – mémorisation de l'état F et des actions indéfinies

ensemble d'actions contenant les actions applicables à l'étape suivante qui sont indéfinies dans la structure parallèle et des nombres d'étapes et d'actions au moins supérieurs, auquel cas il est inutile d'effectuer à nouveau la recherche de solutions à partir de cet état.

4 Recherche d'un plan séquentiel optimal

La procédure RecherchePlanOptimalSequentiel suit le schéma proposé par Büttner et Rintanen [2]. A partir d'un plan parallèle optimal, des plans avec moins d'actions sont recherchés, quite à augmenter le nombre d'étapes s'il n'y a pas de solution. Lorsque le nombre d'étapes atteint le nombre d'actions on ne pourra pas trouver de meilleur plan. Remarquons que l'utilité de la fonction Search n'est pas uniquement de calculer des plans séquentiels optimaux. Il peut s'avérer utile dans certains cas de chercher le nombre minimal d'actions pour un nombre d'étapes fixé, et en particulier on peut vouloir limiter le nombre d'actions pour un nombre d'étapes optimal.

Si l'objectif est uniquement la recherche d'un plan séquentiel optimal il est inutile de parcourir tous les plans optimaux en nombre d'actions en augmentant le nombre de niveaux au fur et à mesure. On peut simplement effectuer une sorte de recherche dichotomique sur le nombre d'étapes. En effet il arrive souvent que la solution optimale produite par la recherche d'un plan parallèle initial soit quasiment optimale en le nombre d'actions, ce qui conduit ensuite à faire un grand nombre de recherches infructeuses en incrémentant chaque fois le nombre de niveaux. La fonction RechercheDichotomiquePlanOptimalSequentiel permet de converger plus rapidement vers un nombre d'étapes minimal, en augmentant le nombre de niveaux plus rapidement : connaissant le nombre de niveaux minimal m_{min} pour avoir un plan

Function `RecherchePlanOptimalSequentiel(`P`)`

Data: P un problème de planification,
Result: n_{opt} la taille d'un plan séquentiel optimal pour le problème P.
begin
  soit π un plan parallèle optimal en m étapes et n actions pour le problème P;
  $n_{opt} := n$;
  $n := n - 1$;
  while $m < n$ **do**
    if *Search(S,* 0*,* n*,* 0*,* m*)* **then**
      $n_{opt} := n$;
      $n := n - 1$;
    else
      $m := m + 1$;
  return n_{opt};
end

Function `RechercheDichotomiquePlanOptimalSequentiel(`P`)`

Data: P un problème de planification,
Result: n_{opt} la taille d'un plan séquentiel optimal pour le problème P.
begin
  soit π un plan parallèle optimal en m étapes et n actions pour le problème P;
  $n_{opt} := n$;
  $n := n - 1$;
  $m_{min} := m$;
  while $m_{min} \leq n$ **do**
    $m = (m_{min} + n)/2$;
    if *Search(S,* 0*,* n*,* 0*,* m*)* **then**
      $n_{opt} := n$;
      $n := n - 1$;
    else
      $m_{min} := m + 1$;
  return n_{opt};
end

en n actions, on teste le problème avec un nombre de niveaux intermédiaire entre m_{min} et n. Si il y a une solution le nombre d'actions est décrémenté, dans le cas contraire on a démontré que le nombre de niveaux minimal pour une solution séquentielle avec n actions était supérieur à $(m_{min} + n)/2$. Avec cette méthode on ne parcourt pas tous les couples (m, n) où m est un nombre de niveaux et n le nombre minimal d'actions pour ce nombre de niveaux. En particulier le plan séquentiel optimal final n'a pas forcément un nombre de niveaux minimal.

5 Résultats expérimentaux

Nous avons implémenté la fonction de recherche d'un plan séquentiel présentée dans le paragraphe précédent, afin de la comparer avec la procédure de recherche de plans séquentiels FDP. La comparaison avec les résultats de Büttner et Rintanen [2] n'a pas été possible du fait que les résultats publiés concernent très peu de problèmes, et que ces problèmes sont bien adaptés aux approches de type *planning as satisfiability*, tandis que FDP a beaucoup de mal à les traiter. Les résultats apparaissent dans la figure 3. Nous avons comparé les deux procédures sur des machines identiques, avec des temps maximaux de calcul de 3000 secondes (timeout). Les problèmes sont issus des compétitions de planificateurs IPC-2, IPC-3, IPC-4 et IPC-5. Pour chaque problème nous avons indiqué le nombre d'actions, le nombre de faits, le nombre d'actions d'un plan séquentiel optimal et le nombre de niveaux d'un plan parallèle optimal.

Nous avons comparé différentes versions de la méthode présentée précédemment. La version standard, appelée SPS pour *search parallel sequential plans*, énumère les plans optimaux à partir d'un plan parallèle optimal. Chaque fois que le nombre d'actions est décrémenté on cherche le nombre minimal d'étapes pour avoir un plan (fonction `RecherchePlanOptimalSequentiel`). Ainsi la procédure produit tous les plans optimaux en leur nombre d'étapes qui ont un nombre d'actions compris entre la valeur optimale et le nombre d'actions du plan parallèle optimal de départ. La seconde version que nous proposons, notée SPS-DICHO, implémente la recherche dichotomique de la valeur minimale du plan séquentiel (fonction `RechercheDichotomiquePlanOptimalSequentiel`). Enfin nous avons testé aussi une procédure de recherche plus simple, notée SPS-CTR, qui consiste simplement à chercher des plans à partir du plan parallèle optimal en diminuant le nombre d'actions et en laissant libre le nombre d'étapes. De cette façon on trouve le plan séquentiel optimal, et on espère, si le nombre d'étapes est contraignant, rendre plus facile les premières recherches pour lesquelles le nombre d'actions est important et le problème soluble. Pour information nous avons fait figurer aussi les temps pour le calcul préliminaire d'un plan parallèle optimal, sous le sigle PFDP, et le temps de calcul pour le calcul d'un plan séquentiel optimal avec FDP.

Les temps de calcul indiqués comprennent la lecture du problème, le calcul des exclusions mutuelles et des séquences ordonnées d'actions, le calcul d'un plan parallèle optimal et enfin la procédure de recherche d'un plan séquentiel optimal à partir du plan parallèle.

Remarquons tout d'abord que pour beaucoup de problèmes le calcul d'un plan optimal parallèle est moins coûteux que le calcul d'un plan séquentiel (voir aussi [3]). Ceci est dû au fait que les plans parallèles sont plus courts, et donc l'espace de recherche est moins important : les états et les ensembles d'actions sont plus proches des buts et de la partie complètement définie du plan et donc plus sensibles au filtrage. Les cas où FDP est plus efficace correspondent souvent à des problèmes dans lesquels il n'y a qu'une action par étape dans le plan parallèle. Rappelons aussi que FDP utilise une heuristique efficace pour évaluer si les buts sont atteignables avec les actions restantes, et qu'il dispose de règles de consistance plus fortes.

Le calcul d'un plan séquentiel optimal à partir d'un plan parallèle optimal est rarement intéressant. En fait si l'objectif est de gagner du temps il faut faire une recherche dichotomique ou partir du plan parallèle optimal en décrémentant le nombre d'actions, sans utiliser la structure parallèle (procédure SPS-CTR). Il est naturel de penser que le coût de recherches successives qui terminent avec succès, ce qui est le cas de la recherche SPS-CTR, est moindre que le coût de recherches infructueuses, ce qui est le cas de FDP, puisque la première recherche qui trouve un plan produit le plan optimal.

On note dans certains cas une explosion combinatoire pour le calcul ou simplement la preuve de l'optimalité d'un plan séquentiel, comme pour les problèmes Free-Cell ou Storage. Dans d'autres cas, comme les problèmes PSR ou certains PipesWorld c'est l'inverse. On pourrait donc imaginer qu'il serait utile de lancer en parallèle des procédures de recherches suivant différentes stratégies, afin de minimiser les risques de s'engager dans des calculs interminables.

La figure 4 présente les résultats obtenus pour une sélection de problèmes Airport. Dans ce domaine particulier la procédure SPS-DICHO produit des résultats meilleurs que la procédure FDP dédiée au calcul de plans séquentiels. On remarque que plus les problèmes sont difficiles plus le gain est important. Ces problèmes ont la particularité d'avoir des solutions optimales avec beaucoup d'actions. D'autre part la recherche d'un plan parallèle optimal produit la plupart du temps

problem	act.	facts	act.	niv.	FDP	SPS	SPS-DICHO	SPS-CTR	PFDP
mprime-x-7	1728	426	5	5	11,4	19,43	19,37	19,36	19,16
mprime-x-9	1904	270	8	5	74,27	30,84	29,52	20,38	8,98
mprime-x-26	4594	287	6	5	61,52	61,34	61,33	61,4	58,27
mystery-x-2	3036	357	7	5	30,39	74,29	80,74	33,57	29,23
mystery-x-30	3357	408	9	6	87,48	113,49	109,87	79,18	71,06
Depot-7512	162	78	15	8	0,83	6,45	3,94	1,61	0,36
driverlog-2-2-3	108	57	19	9	8,23	25,87	15,04	6,06	0,08
driverlog-3-2-4	144	63	16	7	9,3	117	52,21	30,06	0,11
FreeCell3-4	1143	139	14	8	56,13	232,13	259,63	95,89	3,36
FreeCell4-4	1614	183	18	7	462,58	1813,01	2059,08	1260,91	5,79
FreeCell5-4	52	20	-	13	3022,96	3004,69	3005,11	3004,5	17,33
satellite-x-1	259	71	17	10	23,86	–	1043,52	343,86	249,33
Optical-P01-OPT2	418	282	36	13	49,57	156,89	60,23	13,05	5,79
Philosophers-P03-PHIL4	112	120	44	11	81,14	233,84	111,35	11,82	0,68
PSR-33	162	41	25	15	6,67	52,97	46,6	40,31	38,25
PSR-37	112	56	33	25	55,45	183,74	158,35	159,23	124,54
PSR-49	660	63	19	16	23,54	145,51	144,89	158,78	144,48
pipesworld-n1-14-6	632	139	13	8	77,36	496,96	438,75	175,43	20,48
pipesworld-n2-10-2	720	201	20	12	140,3	294	157,71	71,67	11,19
pipesworld-n3-12-2	1140	280	14	14	13,58	1475,4	1477,46	1478,81	1483,32
pipesworld-p04	656	154	11	6	41,73	67,14	49,47	20,76	3,61
pipesworld-p06	764	164	10	6	6,43	16,74	15,02	8,61	4,8
pipesworld-p07	2672	204	8	6	33,73	297,7	294,78	295,1	180,03
Storage-11	460	146	17	11	41,47	173,11	93,67	51,22	16,13
Storage-12	690	164	16	9	214,41	1705,17	877,45	329,35	45,87
Truck-7	1044	269	23	18	714,18	1600,07	1266	671,44	394,37

FIGURE 3 – temps CPU pour une série de problèmes sélectionnés (temps en secondes)

un plan optimal en le nombre d'actions. De ce fait la seul chose à vérifier est qu'il n'existe pas de plan avec une action en moins, quel que soit le nombre d'étapes. La procédure SPS-CTR et dans une moindre mesure la procédure SPS-DICHO n'ont pas à effectuer un grand nombre de recherches infructueuses comme la procédure SPS pour trouver un plan séquentiel optimal.

Le calcul d'un plan séquentiel optimal à partir d'un plan parallèle optimal est souvent plus rapide que le calcul direct avec FDP. C'est le cas en particulier sur la série de problèmes Airport ou Truck où certains problèmes n'ont pu être résolus avec FDP.

6 Conclusion

Nous avons présenté une adaptation du planificateur FDP pour la recherche de plans dont le nombre d'actions et le nombre de niveaux sont contraints. Cette procédure de recherche utilise deux fdp-structures, l'une pour le plan séquentiel, l'autre pour le plan parallèle. Nous avons proposé deux utilisations de cette procédure, d'une part pour parcourir tous les plans dont le nombre d'actions est optimal pour un nombre de niveaux variant entre le minimum et le nombre optimal d'actions, et d'autre part pour calculer des plans optimaux séquentiels à partir d'un plan parallèle optimal. Nous avons implémenté ces méthodes et les avons comparées avec FDP. Les résultats ne sont pas toujours meilleurs que le calcul direct d'un plan séquentiel mais ils restent compétitifs. L'intérêt de cette approche est aussi de produire des plans optimaux en le nombre d'actions (resp. le nombre de niveaux) en contraignant le nombre de niveaux (resp. le nombre d'actions). C'est la première fois que des résultats de ce type sont publiés pour les problèmes des compétitions de planification IPC les plus récentes (IPC-4 et IPC-5 notamment).

Références

[1] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[2] M. Büttner and J. Rintanen. Improving parallel planning with constraints on the number of operators. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling*, pages 292–299, 2005.

[3] Guillaume Gabriel and Stéphane Grandcolas. Searching optimal parallel plans : A filtering and decomposition approach. In *proceedings of the 21st International Conference on Tools with Artificial Intelligence*, pages 576–580, 2009.

[4] Stéphane Grandcolas and Cyril Pain-Barre. Filtering, decomposition and search space reduction for optimal sequential planning. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, pages 993–998, july 2007.

[5] Patrik Haslum and Hector Geffner. Admissible heuristics for optimal planning. In *AIPS*, pages 140–149, 2000.

problem	**act.**	**facts**	**act.**	**niv.**	FDP	SPS	SPS-DICHO	SPS-CTR	PFDP
Airport-1	15	80	8	8	0,08	0,09	0,09	0,09	0
Airport-2	23	81	9	9	0,08	0,11	0,1	0,1	0
Airport-3	38	131	17	9	0,33	0,41	0,38	0,37	0,3
Airport-4	23	156	-	500	0,31	0,23	0,23	0,22	0,2
Airport-5	54	197	21	21	2,85	2,87	2,88	2,86	2,8
Airport-6	77	291	41	21	6,68	7,9	6,99	6,68	6,4
Airport-7	77	291	41	21	6,68	7,87	6,97	6,67	6,4
Airport-8	131	412	62	26	74,78	117,94	45,68	22,2	13,3
Airport-9	143	483	71	27	578,53	1367,91	281,56	93,36	18,3
Airport-10	29	178	18	18	0,47	0,49	0,5	0,48	0,4
Airport-11	60	219	21	21	3,55	3,58	3,57	3,54	3,5
Airport-12	89	327	39	21	9,62	10,62	9,76	9,46	9,3
Airport-13	87	322	37	19	7,98	8,87	8,22	7,94	7,7
Airport-14	149	462	60	26	81,77	134,63	46,7	30,04	18,6
Airport-15	147	459	58	22	73,53	130,31	44,87	28,73	17,3
Airport-16	207	594	79	27	2990,31		1090,05	408,66	35,2
Airport-17	225	687	88	28				2795,95	77,1
Airport-18	283	811	107	31					888,7
Airport-19	229	755	90	30				1906,49	103,1
Airport-20	302	898	115	32					3219,4

FIGURE 4 – temps CPU pour une série de problèmes Airport (temps en secondes)

[6] Richard E. Korf. Depth-first iterative-deepening : An optimal admissible tree search. *Artificial Intelligence*, 27 :97–109, 1985.

[7] A.K. Mackworth. Consistency in networks of relations. In *Artificial Intelligence*, pages 8 :99–118, 1977.

Un codage SAT pour les problèmes de placement

Stéphane Grandcolas **Cédric Pinto**

Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen
13397 Marseille Cedex 20 (France)
{stephane.grandcolas, cedric.pinto}@lsis.org

Résumé

Le problème de placement orthogonal (OPP) consiste à déterminer si un ensemble d'objets peut etre placé dans un conteneur de taille connue. Ce problème est NP-complet [6]. Une modélisation de ce problème à base de graphes d'intervalles a été proposée par S. P. Fekete et al [3][5]. Cette modélisation permet de représenter des classes de placements équivalents, diminuant d'autant l'espace de recherche.

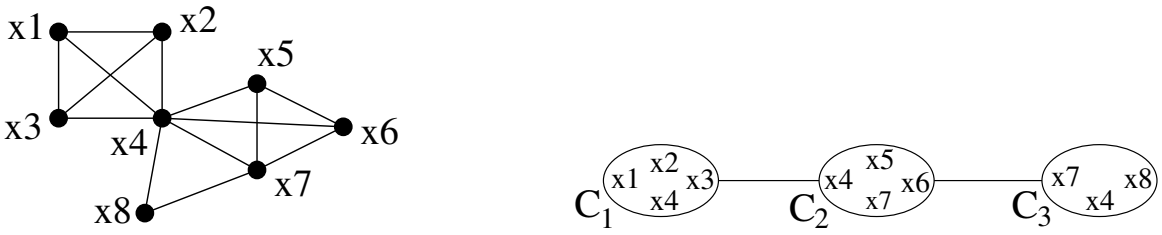
Dans cet article nous proposons de représenter par des formules de la logique propositionnelle la modélisation de S. P. Fekete et al. Nous avons implémenté cette approche en utilisant le solveur MiniSat, et nous l'avons comparée d'une part avec les résultats de S. P. Fekete et al. sur des problèmes classiques, et d'autre part avec l'approche de T. Soh et al. [12] basée aussi sur un codage SAT sur des problèmes de Strip Packing.

1 Introduction

Le problème de placement orthogonal en plusieurs dimensions (OPP) consiste à déterminer si un ensemble d'objets de tailles connues peut être placé dans un conteneur donné. Les objets doivent être placés parallèlement aux bords du conteneur et aucune rotation ne peut être effectuée. Bien que ce problème soit NP-complet [6], des algorithmes efficaces pour le résoudre sont essentiels du fait que OPP est souvent utilisé pour le traitement de problèmes d'optimisation comme les problèmes de *strip packing* ou de *bin-packing*.

Les graphes d'intervalles sont utilisés dans de nombreux domaines comme la génétique ou l'ordonnancement. Ils constituent une classe de graphes très particulière et possèdent des propriétés algorithmiques intéressantes [7]. S. P. Fekete et al. ont introduit une nouvelle caractérisation de OPP [3] basée sur les graphes d'intervalles. Pour chaque dimension de l'espace un graphe représente les intersections entre les objets dans cette dimension. Pour chaque graphe G_i correspondant à la dimension i, le poids d'un sommet est égal à la taille de l'objet correspondant dans la dimension i. Le problème de placement orthogonal en d dimensions est équivalent à trouver d graphes $G_1, \ldots, G_d$ tels que **(P1)** chaque graphe G_i est un graphe d'intervalles, **(P2)** dans chaque graphe G_i, tout stable est i-faisable, c'est-à-dire que la somme des poids de ses sommets est inférieure ou égale à la taille du conteneur dans la dimension i, et **(P3)** il n'y a pas d'arête qui apparaît dans chacun des d graphes. S. P. Fekete et al. proposent une procédure de recherche [5] complète qui consiste à énumerer tous les graphes d'intervalles possibles, en choisissant pour chaque arête de chaque graphe si elle appartient au graphe ou pas. Durant la recherche, la condition (P3) est toujours satisfaite en interdisant de fixer une arête apparaissant déjà dans d-1 graphes. Chaque fois qu'un graphe G_i est un graphe d'intervalles, la i-faisabilité de ses stables est vérifiée, en calculant son stable de poids maximum. Dès que les graphes $G_1, \ldots, G_d$ satisfont les trois conditions, la recherche s'arrête, les d graphes représentant alors une classe de solutions équivalentes pour le problème de placement. La figure 2 montre un exemple en deux dimensions de deux placements, parmi tant d'autres, correspondant à la même paire de graphes d'intervalles.

Il existe peu d'approches SAT pour les problèmes de placement. En 2008, T. Soh et al.[12] ont proposé un codage SAT pour le problème du *strip packing* en deux dimensions (SPP) [12]. Ce problème consiste à trouver la hauteur minimale d'une bande de largeur fixée contenant tous les objets. Pour ce calcul, ils effectuent des recherches successives avec différentes hauteurs (choisies par recherche dichotomique), en codant chaque fois le problème de décision sous la forme d'une formule CNF qui est résolue avec un solveur SAT externe (Minisat). Dans leur formula-



(a) (b)

FIGURE 1 – (a) Un graphe d'intervalles, (b) une suite ordonnée de cliques de ce graphe

tion, les positions exactes des objets dans le conteneur sont représentées par des variables booléennes, ainsi que les positions relatives des objets entre eux (à gauche, à droite, au dessus, au dessous). M. D. Moffit et al. [11] se sont intéressés à une représentation de type CSP très proche de celle de T. Soh et al., étendant les travaux de R. Korf [8] avec l'ajout de variables pour décrire les positions relatives des objets.

Dans cet article nous proposons une représentation de la caractérisation de S. P. Fekete et al. avec des formules de la logique propositionnelle. L'idée consiste à représenter les arêtes des graphes d'intervalles par des variables booléennes et à coder sous forme de clauses les stables infaisables. L'article est organisé de la façon suivante. Dans la partie 2 nous présentons la caractérisation de S. P. Fekete et al. après avoir rappelé quelques propriétés sur les graphes d'intervalles. Dans la partie 3 nous revenons sur les codages SAT et CSP connus. Dans la partie 4 nous définissons le codage SAT du problème OPP que nous proposons. Enfin dans la partie 5 nous exposons les résultats expérimentaux, et nous terminons avec la conclusion.

2 Une caractérisation de OPP avec des graphes d'intervalles

Les graphes d'intervalles constituent une classe très particulière de graphes possédant de nombreuses propriétés algorithmiques intéressantes : de nombreux problèmes NP-difficiles peuvent être résolus en temps polynomial dans le cas des graphes d'intervalles [7]. Ces graphes sont utilisés dans de nombreux domaines comme la génétique ou l'ordonnancement. De plus, ils peuvent être reconnus par des algorithmes de complexité linéaire [9] en le nombre de sommets et d'arêtes du graphe.

Définition 1 *Un graphe $G = (X, E)$ est un graphe d'intervalles si et seulement si à tout sommet $x \in X$ on peut associer un intervalle I_x de l'ensemble des réels tel que :*

$$\forall\, x,\ y, \in X,\ \{x, y\} \in E \Leftrightarrow I_x \cap I_y \neq \emptyset$$

Dans un graphe G, une clique est un ensemble de sommets tel que les sommets de cette clique sont tous reliés entre eux par une arête dans G. Inversement, un stable est un ensemble de sommets dans lequel les sommets ne sont reliés à aucun autre sommet du même stable. Calculer l'ensembles des cliques maximales est un problème NP-difficile dans le cas de graphes quelconques. Dans le cas des graphes d'intervalles, ce problème peut être résolu en temps polynomial [1].

Les graphes d'intervalles ont la propriété d'être des graphes triangulés [7]. Or, l'ensemble des cliques maximales d'un graphe triangulé peut se calculer en temps linéaire en le nombre de sommets et d'arêtes du graphe [7]. De plus, le nombre de cliques maximales d'un graphe triangulé ne peut dépasser le nombre de sommets n alors qu'il peut être exponentiel en n dans le cas des graphes quelconques. Puisque les graphes d'intervalles sont triangulés, le calcul des cliques maximales peut être fait en temps polynomial et le nombre de cliques maximales est majoré par le nombre de sommets. Nous avons également la propriété suivante.

Propriété 1 *G est un graphe d'intervalles si et seulement si il existe une suite de cliques maximales de G, notées $\mathcal{C} = (C_1, \ldots, C_k)$, telle que :*

$$\forall a,\ b,\ c, \qquad 1 \leq a \leq b \leq c \leq k, \qquad C_a \cap C_c \subseteq C_b$$

Autrement dit, un graphe G est un graphe d'intervalles si et seulement si les cliques maximales de G peuvent être ordonnées de sorte que tout sommet appartient à des cliques consécutives dans cet ordre. Nous appellerons *suite ordonnée de cliques* toute suite de cliques maximales vérifiant cette propriété. Nous montrons en figure 1 un graphe d'intervalles et une suite ordonnée de cliques de ce graphe. Pour un graphe d'intervalles donné, il peut y avoir plusieurs suites ordonnées de cliques tandis qu'une suite ordonnée de cliques ne correspond qu'à un seul graphe d'intervalles.

S. P. Fekete et al. ont proposé une caractérisation des placements pour le problème de packing othogonal à l'aide de graphes d'intervalles, un pour chaque dimension de l'espace [3]. Dans ce qui suit, O désigne un ensemble d'objets

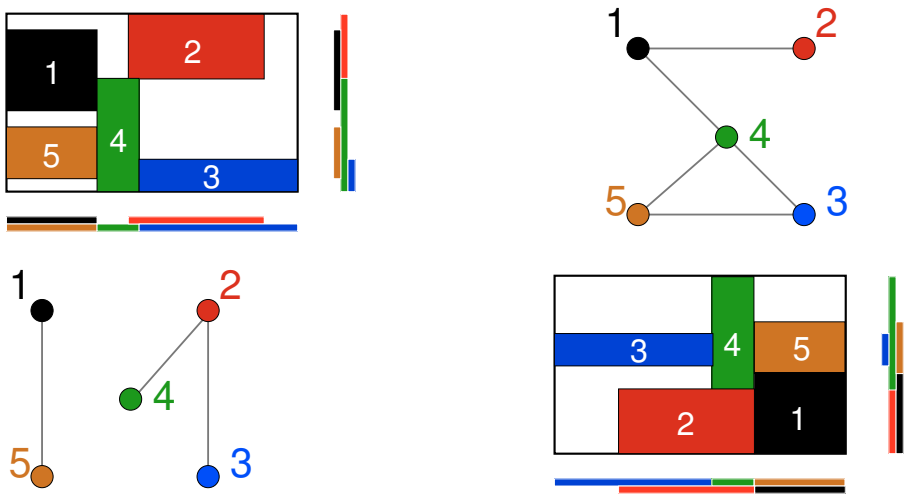


FIGURE 2 – Deux graphes d'intervalles et deux exemples de placements équivalents représentés par ces graphes.

dont les tailles sont connues. Si on note G_i le graphe associé à la dimension i, ses sommets correspondent aux objets de O et ses arêtes représentent les intersections entre les projections des objets sur la dimension i (voir figure 2). Le *poids* d'un sommet x dans un graphe G_i est égal à la taille de x dans la dimension i. Par exemple, considérons le cas d'un problème en deux dimensions. Dire que deux objets x et y s'intersectent dans la dimension horizontale signifie que x doit être au-dessus de y ou inversement. Remarquons que x et y ne peuvent s'intersecter dans toutes les dimensions à la fois auquel cas x et y se chevaucheraient. Afin de garantir la validité des placements représentés par les graphes d'intervalles, les auteurs ont ajouté deux propriétés, la première étant qu'*une arête ne peut être présente dans tous les graphes d'intervalles*. Cette propriété garantit que les objets ne se chevaucheront pas dans l'espace. La deuxième propriété devant être satisfaite par chaque graphe d'intervalles permet d'assurer que les objets seront placés à l'intérieur du conteneur. Pour cela, *dans chaque graphe d'intervalles G_i, chaque stable doit être i-faisable*. Un stable S est dit i-faisable si son poids (défini par la somme de poids des sommets qui composent S) est inférieur ou égal à la taille du conteneur dans la dimension i. Dans le cas avec deux dimensions, si un ensemble d'objets forment un stable dans le graphe associé à la dimension horizontale alors ces objets seront placés côte à côte horizontalement. Il faut donc s'assurer que la largeur nécessaire pour les placer côte à côte est inférieure ou égale à la largeur du conteneur, d'où la propriété ci-dessus.

L'intérêt de cette approche réside dans la capture d'une multitude de placements avec une seule famille de graphes d'intervalles. De plus, à partir d'une famille de graphes d'intervalles, nous pouvons facilement exhiber l'ensemble des placements représentés par cette famille. Il suffit de considérer le graphe complémentaire de chaque graphe d'intervalles. Ces graphes sont des graphes de comparabilité [7]. Dans chaque graphe de comparabilité, une orientation transitive des arêtes peut être calculée en temps linéaire en le nombre d'arêtes. Une telle orientation existe par définition des graphes de comparabilité. Chaque orientation transitive induit un ordre sur les objets. Cet ordre peut être utilisé pour en déduire un placement des objets. Si $n_1, \ldots, n_d$ désignent les nombres d'orientations transitives des graphes complémentaires $\overline{G_1}, \ldots, \overline{G_d}$ alors le nombre de placements représentés par $G_1, \ldots, G_d$ est égal à $\prod_{i=1}^{d} n_i$.

S. P. Fekete et al. ont donc démontré que résoudre OPP en d dimensions est équivalent à déterminer l'existence de d graphes tels que :

P1 : Chaque graphe est un graphe d'intervalles

P2 : Pour chaque graphe G_i, chaque stable est i-faisable

P3 : Chaque arête doit apparaître dans au plus d-1 graphes.

Dans [5], S. P. Fekete et al. définissent une méthode de résolution de OPP. Cette méthode consiste à énumérer toutes les familles de graphes d'intervalles. Pour faire cela, ils choisissent pour chaque graphe G_i et chaque arête de G_i si celle-ci est présente ou pas dans G_i. Initialement, tous les graphes sont vides, c'est-à-dire sans aucune arête. La propriété (P3) est vérifiée par construction, puisque si une arête est présente dans d-1 graphes alors celle-ci est immédiatement retirée dans le dernier graphe. Chaque fois qu'une arête est ajoutée ou retirée dans un graphe G_i, la méthode consiste à vérifier si G_i est un graphe d'intervalles. Si ce n'est pas le cas alors une autre arête est choisie (éventuellement dans un autre graphe). Si G_i est un graphe d'intervalles, un stable de poids maximum dans G_i est calculé et sa i-faisabilité est vérifiée. Si le stable calculé n'est pas i-faisable alors les arêtes reliant deux sommets

de ce stable seront choisies en priorité par la suite. Si tous les graphes sont des graphes d'intervalles et que chaque stable de poids maximum est faisable alors d graphes d'intervalles vérifiant (P1), (P2) et (P3) ont été trouvés. Dans ce cas, le problème est consistant et un placement peut facilement être exhibé en calculant une orientation transitive dans chaque graphe complémentaire. Enfin, notons que S. P. Fekete et al. ont également montré l'intérêt de leur approche en pratique [5].

3 Approches SAT et CSP pour OPP

Peu d'approches SAT existent pour les problèmes de placement. En 2008, T. Soh et al. [12] ont proposé un codage SAT pour le problème du *strip packing* en deux dimensions (SPP) [12]. Ce problème consiste à trouver la hauteur minimale d'une bande de largeur fixée contenant tous les objets. Pour ce faire, ils effectuent des recherches successives avec différentes hauteurs (choisies par recherche dichotomique). Pour chaque recherche, le problème de décision est codé sous la forme d'une formule CNF qui est ensuite résolue avec un solveur SAT externe (Minisat). Dans leur formulation, les variables représentent les positions exactes des objets dans le conteneur, c'est à dire que dans chaque dimension, pour chaque position possible une variable booléenne signale si l'objet apparaît à cette position ou non. Des variables additionnelles représentent les positions relatives des objets entre eux (à gauche, à droite, au dessus, au dessous). T. Soh et al. introduisent également des contraintes additionnelles pour éviter de reconsidérer des placements équivalents par symetrie. Enfin, durant la résolution, Minisat génère de nouvelles clauses à partir des conflits rencontrés. T. Soh et al. ont mis en place un mécanisme de mémorisation qui permet de réutiliser ces clauses dans les recherches suivantes (cette mémorisation est possible du fait que les instances SAT successives sont incrémentales). La modélisation de T. Soh et al. utilise $\mathcal{O}(W \times H \times n + n^2)$ variables booléennes pour un problème avec n objets et un conteneur de largeur W et de hauteur H.

Plusieurs approches de type CSP ont été proposées pour les problèmes de remplissage avec des rectangles. Par exemple, en 2004, R. Korf [8] définit un CSP dans lequel les variables représentent les positions des objets. Les contraintes assurant que les objets ne se chevauchent pas sont des disjonctions d'inégalités. Les tailles des domaines sont donc très proches de la taille du conteneur et les performances du solveur en sont directement affectées. En 2006, M. D. Moffit et al. [11] proposent d'améliorer cette approche en introduisant des variables additionelles représentant les positions relatives des objets entres eux. Remarquons que cette formalisation est proche de celle de T. Soh et al. Les relations d'ordre entre les objets sont représentées dans des graphes orientés, et le calcul des plus courts chemins dans ces graphes permet de détecter l'inconsistance du CSP.

4 Un nouveau codage SAT pour OPP

Nous proposons un nouveau codage SAT basé sur la caractérisation de S. P. Fekete et al. pour le problème de placement othogonal en d dimensions. L'idée consiste à représenter les graphes d'intervalles par des suites ordonnées de cliques, en utilisant des variables booléennes pour signaler la présence des objets dans les cliques. Les variables que nous utilisons pour le codage sont les suivantes :

$e^i_{x,y}$: **vrai** si l'arête $\{x, y\}$ est dans G_i,

$c^i_{x,a}$: **vrai** si le sommet x est dans la clique a,

$p^i_{x,y,a}$: **vrai** si les sommets x et y sont dans la clique a,

u^i_a : **vrai** si la clique a n'est pas vide,

Dans ce qui suit, O désigne l'ensemble des n sommets correspondants aux n objets du problème et d est le nombre de dimensions du problème. Nous allons maintenant établir l'ensemble des formules permettant de représenter les propriétés (P1), (P2) et (P3) définies par S. P. Fekete et al. Tout d'abord, chaque objet doit apparaitre dans au moins une clique dans chaque dimension.

1. **[Les objets sont placés]** $x \in O, 1 \leq i \leq d,$
$$c^i_{x,1} \vee \ldots \vee c^i_{x,n}$$

Ensuite, la suite des cliques doit être ordonnée comme indiqué dans la propriété 1.

2. **[Suite ordonnée de cliques]**
$x \in O, 1 \leq i \leq d, 1 \leq a < b - 1 < n,$
$$(c^i_{x,a} \wedge c^i_{x,b}) \Rightarrow c^i_{x,a+1}$$

Ainsi, si un objet x apparait dans deux cliques C_a et C_b alors x doit apparaitre dans les cliques $C_{a+1}, \ldots, C_{b-1}$. En fait, il suffit de forcer le placement de l'objet x dans la clique C_{a+1}, la contrainte se propage ensuite sur les cliques suivantes par récurrence.

Il faut également que les objets ne se chevauchent pas dans l'espace.

3. **[Pas de chevauchement]** $x, y \in O,$
$$\neg e^1_{x,y} \vee \ldots \vee \neg e^d_{x,y}$$

Il suffit d'imposer que chaque arête n'apparaisse pas dans tous les graphes.

Dans la formule qui suit, nous appellons stable minimal infaisable d'un graphe G_i tout stable pour lequel le retrait d'un de ses sommets le rend i-faisable. S^i désigne l'ensemble des stables minimaux infaisables. Pour satisfaire la propriété (P2), il suffit de forcer l'ajout d'au moins une arête entre deux sommets d'un stable minimal infaisable dans chaque dimension.

4. **[Stable faisabilité]** $1 \leq i \leq d, N \in S^i$,
$$\bigvee_{x,\, y \in N} e^i_{x,y}$$

Pour illustrer la formule ci-dessus, considérons l'exemple dans lequel trois objets x, y et z en deux dimensions ne peuvent être placés côte à côte dans la dimension i mais toute paire parmi ces trois objets peut être placée dans le conteneur. Nous avons donc que $\{x, y, z\}$ est un stable minimal infaisable. D'après la formule (4), nous aurons la clause $e^i_{x,y} \vee e^i_{x,z} \vee e^i_{y,z}$. Ainsi, nous obligeons l'ajout d'une de ces trois arêtes afin d'obtenir un stable i-faisable. Donc, la propriété (P2) sera bien satisfaite puisque chaque stable de chaque graphe sera faisable. Notons le cas particulier où deux objets x et y ne peuvent être placés côte à côte dans la dimension i. Dans ce cas, nous aurons la clause unitaire $e^i_{x,y}$. Le solveur SAT affectera immédiatement ce littéral et propagera cette affectation.

On considère autant de cliques qu'il y a d'objets. Lors de l'affectation des objets aux cliques il se peut que des cliques soient vides. Afin d'éviter des traitements redondants, les cliques vides sont forcées à apparaître à la fin de la suite.

5. **[Pas de cliques vides]** $1 \leq i \leq d, 1 \leq a \leq n$,
$$\neg u^i_a \Rightarrow \neg u^i_{a+1}$$

Ce qui signifie que si la clique a est vide alors la clique $a + 1$ l'est aussi. Signalons que cette contrainte n'est pas nécessaire.

Enfin il faut établir les liens entre les différentes variables.

6. **[Liens entre les variables]**
- $x, y \in O, 1 \leq a \leq n, 1 \leq i \leq d$,
$$p^i_{x,y,a} \Leftrightarrow (c^i_{x,a} \wedge c^i_{y,a})$$
$$(p^i_{x,y,1} \vee \ldots \vee p^i_{x,y,n}) \Leftrightarrow e^i_{x,y}$$
- $1 \leq a \leq n, 1 \leq i \leq d$,
$$u^i_a \Leftrightarrow (c^i_{1,a} \vee \ldots \vee c^i_{n,a})$$

Les contraintes qui suivent ne sont pas nécessaires pour la validité des solutions, mais en contraignant le problème elles peuvent provoquer des échecs plus rapidement.

La formule suivante aide la propagation des affectations des objets aux cliques.

7. **[Suite ordonnée de cliques (bis)]**
$x \in O, 1 \leq a \leq n, 1 \leq i \leq d$,
$$(c^i_{x,a} \wedge \neg c^i_{x,a+1}) \Rightarrow (\neg c^i_{x,a+2} \wedge \ldots \wedge \neg c^i_{x,n})$$
$$(c^i_{x,a} \wedge \neg c^i_{x,a-1}) \Rightarrow (\neg c^i_{x,a-2} \wedge \ldots \wedge \neg c^i_{x,1})$$

Si un objet x est dans la clique C_a et qu'il n'est pas dans la clique C_{a+1} (resp. C_{a-1}) alors on en déduit qu'il ne sera pas dans les cliques placées après C_{a+1} (resp. avant C_{a-1}) sinon la propriété (P1) ne serait plus satisfaite.

Pendant l'énumération il se peut que des cliques successives soient incluses l'une dans l'autre. Cette situation est sans intérêt puisqu'elle peut être reproduite sans qu'aucune clique ne soit incluse dans une autre.

8. **[Cliques maximales]** $1 \leq a < n, 1 \leq i \leq d$,
$$(u_a \wedge u^i_{a+1}) \Rightarrow ((c^i_{1,a} \wedge \neg c^i_{1,a+1}) \vee \ldots \vee (c^i_{n,a} \wedge \neg c^i_{n,a+1}))$$
$$(u_a \wedge u^i_{a+1}) \Rightarrow ((\neg c^i_{1,a} \wedge c^i_{1,a+1}) \vee \ldots \vee (\neg c^i_{n,a} \wedge c^i_{n,a+1}))$$

Si deux cliques C_a et C_{a+1} ne sont pas vides alors au moins un sommet doit être présent dans C_a et pas dans C_{a+1} et inversement, au moins un sommet doit être présent dans C_{a+1} et pas dans C_a.

La présence d'objets identiques (dans toutes leurs dimensions) dans un problème génère de nombreux traitements similaires, puisque ces objets sont interchangeables. Une façon d'éviter ces traitements redondants consiste à contraindre l'ordre d'apparition de ces objets dans les cliques dans une des dimensions. Supposons que δ soit la dimension choisie pour cet ordre. Soit $\prec$ une relation d'ordre total sur l'ensemble des objets.

9. **[Objets identiques]**
$x, y \in O, x \equiv y$ and $x \prec y, 1 \leq a < n, a < b \leq n$,
$$(c^\delta_{y,a} \wedge c^\delta_{x,b}) \Rightarrow c^\delta_{x,a}$$

Si l'objet x tel que $x \prec y$ apparaît dans une clique de rang b supérieur au rang a d'une clique contenant y, alors l'objet x doit lui aussi apparaître dans la clique de rang b.

La modélisation d'un problème avec n objets en d dimensions nécessite donc $\mathcal{O}(d \times n^3)$ variables et $\mathcal{O}(d \times (n^4 + 2^n))$ clauses. Cependant, puisque seulement les stables minimaux infaisables sont générés dans l'instance, dans la pratique, nous espérons avoir moins de 2^n clauses issues de la formule (4).

5 Résultats expérimentaux

5.1 Problème de placement orthogonal (OPP)

Le problème consiste à déterminer si un ensemble d'objets peut être placé dans un conteneur donné. Nous avons comparé notre approche avec celle de S. P. Fekete et al. sur quelques problèmes en deux dimensions [2]. Les résultats obtenus par S. P. Fekete et al. sur ces problèmes sont disponibles dans [2]. Nous avons écarté les problèmes résolus trop rapidement ainsi que les deux problèmes non résolus par chacune des méthodes. Dans ces résultats, il est indiqué que S. P. Fekete et al. utilisent un Pentium IV 3 Ghz pour résoudre ces instances et le temps limite est fixé à 15 minutes pour chaque instance. Toutes nos expérimentations ont été exécutées sur un processeur Pentium IV 3.2 GHz et 1 GO de RAM, et nous avons utilisé Minisat 2.0 pour résoudre nos instances SAT. Nous avons également fixé le temps limite à 15 minutes.

Chaque instance est caractérisée par le nombre d'objets n, sa faisabilité (F si l'instance possède une solution, N sinon) et le pourcentage d'espace libre dans le conteneur

Instances				FS	M1			M2		
Nom	Espace	Fais.	n		Temps	#var.	#claus.	Temps	#var.	#claus.
E02F17	02	F	17	7	**4.95**	5474	26167	13.9	6660	37243
E02F20	02	F	20	-	5.46	8720	55707	**1.69**	10416	73419
E02F22	02	F	22	167	**7.62**	11594	105910	21.7	13570	129266
E03N16	03	N	16	**2**	39.9	4592	20955	47.3	5644	30259
E03N17	03	N	17	**0**	4.44	5474	27401	9.32	6660	38477
E04F17	04	F	17	13	**0.64**	5474	26779	1.35	6660	37855
E04F19	04	F	19	560	3.17	7562	46257	**1.43**	9040	61525
E04F20	04	F	20	22	5.72	8780	59857	**2.22**	10416	77569
E04N18	04	N	18	**10**	161	6462	32844	87.7	7790	45904
E05F20	05	F	20	491	6.28	8780	59710	**0.96**	10416	77422
Moyenne				> 217	23.9	7291	46159	**18.8**	8727	60894

TABLE 1 – Comparaison avec S. P. Fekete et al.

si tous les objets peuvent y être placés. La table 1 montre les caractéristiques des instances, les résultats de S. P. Fekete et al. **(FS)**, et les résultats de notre approche. Nous avons testé de nombreuses modélisations, résultants de différentes combinaisons avec les formules (1) à (9). Nous indiquons ici les résultats obtenus avec deux de ces modélisations qui nous semblent les plus pertinentes : la modélisation **M1** intègre les formules (1) à (6) et (9), tandis que la modélisation **M2** contient en plus les formules facultatives (7) et (8). Les temps indiqués sont en secondes.

Nous constatons que notre méthode est plus robuste sur ces instances que FS. De plus, nous surclassons FS sur les instances satisfiables, en particulier sur l'instance E02F20 qui n'a pas été résolue dans le temps limite par FS. Sur les instances non satisfiables, FS obtient de meilleurs résultats. Nous pensons que ceci est dû au fait que S. P. Fekete et al. calculent des bornes (voir DFF dans [4]) permettant de détecter de couper très haut dans l'arbre de recherche. Concernant nos différentes modélisations, nous n'avons pas encore pu déterminer dans quels cas une modélisation est meilleure qu'une autre. Nous pouvons simplement constater que l'augmentation de la taille de l'instance induit par la modélisation M2 par rapport à M1 ne garantit en rien une efficacité accrue de la résolution (et mène même quelques fois à un moins bon résultat). L'explication ne se trouve donc pas dans la taille de l'instance mais certainement dans l'influence de la présence de contraintes additionnelles sur le choix des variables effectué par MiniSat.

5.2 Le problème du Strip Packing

Nous avons également comparé notre approche avec celle de T. Soh et al. [12] sur des instances de strip packing en deux dimensions de la librairie OR disponible à l'adresse `http://www.or.deis.unibo.it/research.html`. Le problème consiste à déterminer la hauteur minimale d'un conteneur de largeur fixée pour pouvoir placer un ensemble d'objets donné. Comme T. Soh et al., nous effectuons une recherche dichotomique pour déterminer la hauteur minimale. La borne inférieure est celle proposée par Martello et Vigo [10]. Nous calculons une borne supérieure avec un algorithme glouton qui construit des placements triviaux. En appliquant cet algorithme avec différentes hauteurs on en déduit une borne maximale de la hauteur optimale. L'algorithme consiste simplement à empiler les uns sur les autres les objets, pris dans l'ordre décroissant de leurs largeurs. Chaque fois que la pile atteint la hauteur testée de la bande, la procédure continue avec une nouvelle pile. Si la somme des largeurs des piles est inférieure à la largeur de conteneur alors ce placement est valide. La recherche repart ensuite de la hauteur minimale avec laquelle l'algorithme glouton a trouvé un placement valide.

Pour la recherche dichotomique à partir des bornes, chaque fois qu'une hauteur doit être testée, nous générons une instance SAT et la traitons avec MiniSat 2.0. T. Soh et al. résolvent ces instances sur un Xeon 2.6 GHz avec 2 GO de RAM et utilisent également MiniSat 2.0. Les résultats sont présentés dans la table 2.

Pour chaque instance, la taille du codage est reportée (nombre de variables et nombre de clauses), ainsi que la hauteur minimale trouvée dans le temps limite fixé à 3600 secondes. Les hauteurs optimales sont indiquées en gras. L'optimalité peut être prouvée de deux manières : soit nous démontrons la consistance du problème avec une hauteur égale à la borne inférieure, soit nous trouvons une hauteur h telle que le problème est consistant avec h mais inconsistant avec $h - 1$. Nous avons écarté les instances dans lesquelles le nombre d'objets est trop grand, puisque le nombre de stables infaisables devient trop important et donc le nombre de clauses également.

Nous démontrons l'optimalité pour 16 instances parmi les 22. De plus, sur ces 16 instances, 14 sont résolues en moins de 30 secondes avec une de nos deux modélisa-

Instances				Soh et	M1				M2			
Nom	n	Largeur	LB	al.	Hauteur	#var.	#claus.	Temps	Hauteur	#var.	#claus.	Temps
HT01	16	20	20	**20**	**20**	4592	22963	13.3	**20**	5644	32267	19.4
HT02	17	20	20	**20**	**20**	5474	28669	744	**20**	6660	39745	444
HT03	16	20	20	**20**	**20**	4592	24222	18.5	**20**	5644	33526	25.5
HT04	25	40	15	**15**	16	16850	271500	1206	19	19396	305392	521
HT05	25	40	15	**15**	16	16850	337395	438	16	19396	372287	536
HT06	25	40	15	**15**	16	16850	494500	146	16	19396	528392	295
CGCUT01	16	10	23	**23**	**23**	4592	26745	5.89	**23**	5644	36049	9.71
CGCUT02	23	70	63	65	66	13202	115110	1043	70	15360	188222	1802
GCUT01	10	250	1016	**1016**	**1016**	1190	4785	0.11	**1016**	1606	7237	0.04
GCUT02	23	250	1133	1196	1259	8780	105810	37.3	1196	10416	123522	1241
NGCUT01	10	10	23	**23**	**23**	1190	5132	0.23	**23**	1606	7584	0.09
NGCUT02	17	10	30	**30**	**30**	5474	29662	1.6	**30**	6660	40738	2.74
NGCUT03	21	10	28	**28**	**28**	10122	108138	273	**28**	11924	128542	580
NGCUT04	7	10	20	**20**	**20**	434	1661	0.01	**20**	640	2577	0.01
NGCUT05	14	10	36	**36**	**36**	3122	15558	6.01	**36**	3930	21906	4.44
NGCUT06	15	10	31	**31**	**31**	3810	18629	1.92	**31**	4736	26361	2.91
NGCUT07	8	20	20	**20**	**20**	632	2535	0	**20**	900	3855	0
NGCUT08	13	20	33	**33**	**33**	2522	11870	2.74	**33**	3220	17010	9.73
NGCUT09	18	20	49	**50**	50	6462	33765	391	50	7790	46825	53.3
NGCUT10	13	30	80	**80**	**80**	2522	11790	0.75	**80**	3220	16930	0.39
NGCUT11	15	30	50	**52**	**52**	3810	18507	19.7	**52**	4736	26239	25.9
NGCUT12	22	30	79	**87**	**87**	11594	173575	886	**87**	13570	196931	24.5

TABLE 2 – Résultats sur les instances de la librairie OR et comparaison avec T. Soh et al.

tions. Dans leurs résultats, T. Soh et al. n'ont pas indiqué les temps de calcul. Cependant, ils résolvent optimalement plus de problèmes que nous. Nous pensons que la capacité de leur solveur à mémoriser et à réutiliser les clauses générées par Minisat durant les recherches est un réel avantage. Dans de futurs travaux, nous espérons confirmer cette hypothèse en caractérisant les clauses pouvant être réutilisées par notre approche durant les recherches successives.

6 Conclusion et discussion

Nous avons proposé un codage SAT de la modélisation de S. P. Fekete et al. qui surclasse significativement leur méthode sur des instances satisfiables. Sur les instances non satisfiables, S. P. Fekete et al. obtiennent de meilleurs résultats. Nous pensons que ceci est dû à l'apport des DFF qui permettent d'effectuer des coupures très tôt durant la recherche. Nous envisageons donc d'intégrer le calcul des DFF directement dans la résolution effectuée par MiniSat afin de bénéficier de ces mêmes coupures.

Nous avons également expérimenté notre codage sur des instances de strip packing. Bien que résolvant moins d'instances que T. Soh et al., nous obtenons des résultats encourageants. Nous allons dans des travaux futurs tenter de caractériser les situations dans lesquelles les clauses générées par le solveur SAT peuvent être réutilisées. De plus, dans la plupart des instances, certains objets sont de tailles identiques (dans toutes les dimensions). Donc de nombreux stables minimaux infaisables représentent des situations identiques dans le sens où les objets qui les composent sont de tailles identiques dans cette dimension. Afin de limiter autant que possible le nombre de clauses correspondant aux stables minimaux infaisables, nous mettrons en place dans le solveur SAT un mécanisme de détection des stables infaisables qui devrait considérablement alléger les formules produites par notre codage.

Références

[1] Andreas Brandstädt, Van Bang Le, and Jeremy P. Spinrad. *Graph classes : a survey*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[2] F. Clautiaux, J. Carlier, and A. Moukrim. A new exact method for the two-dimensional orthogonal packing problem. *European Journal of Operational Research*, 183(3) :1196–1211, 2007.

[3] S. P. Fekete and J. Schepers. A combinatorial characterization of higher-dimensional orthogonal packing. *Mathematics of Operations Research*, 29(2) :353–368, 2004.

[4] S. P. Fekete and J. Schepers. A general framework for bounds for higher-dimensional orthogonal packing problems. *Mathematical Methods of Operations Research*, 60(2) :311–329, 2004.

[5] S. P. Fekete, J. Schepers, and J. van der Veen. An exact algorithm for higher-dimensional orthogonal packing. *Operations Research*, 55(3) :569–587, 2007.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability – A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, 1979.

[7] M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs.* Academic Press, New York, 1980.

[8] Richard E. Korf. Optimal rectangle packing : New results. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 142–149. AAAI, 2004.

[9] Norbert Korte and Rolf H. Möhring. An incremental linear-time algorithm for recognizing interval graphs. *SIAM J. Comput.*, 18(1) :68–81, 1989.

[10] S. Martello, M. Monaci, and D. Vigo. An exact approach to the strip-packing problem. *Journal on Computing*, 15(3) :310–319, 2003.

[11] Michael D. Moffitt and Martha E. Pollack. Optimal rectangle packing : A meta-csp approach. In Derek Long, Stephen F. Smith, Daniel Borrajo, and Lee McCluskey, editors, *ICAPS*, pages 93–102. AAAI, 2006.

[12] T. Soh, K. Inoue, N. Tamura, M. Banbara, and H. Nabeshima. A sat-based method for solving the two-dimensional strip packing problem. In *Proceedings of the 15th RCRA workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2008.

Une nouvelle technique de filtrage basée sur la décomposition de sous-réseaux de contraintes

Philippe Jégou **Cyril Terrioux**

LSIS - UMR CNRS 6168
Université Paul Cézanne (Aix-Marseille 3)
Avenue Escadrille Normandie-Niemen 13397 Marseille Cedex 20, France
{philippe.jegou, cyril.terrioux}@univ-cezanne.fr

Résumé

Dans ce papier, nous introduisons une nouvelle technique de filtrage pour les réseaux de contraintes. Elle est basée sur une propriété appelée *cohérence structurelle*. Il s'agit d'une cohérence paramétrable que nous noterons w-*SC*. Cette cohérence est basée sur une approche significativement différente de celles en usage. Alors que les cohérences classiques s'appuient généralement sur des propriétés locales étendues à l'ensemble du réseau, cette cohérence partielle considère à l'opposé la cohérence globale de sous-problèmes. Ces sous-problèmes sont définis par des graphes de contraintes partiels dont la largeur arborescente est bornée par une constante w, qui correspond au paramètre associé à la cohérence. Nous introduisons un algorithme de filtrage qui permet d'obtenir la w-SC cohérence. Cette cohérence est ensuite analysée pour la positionner par rapport aux cohérences classiquement utilisées dans les CSP. Cette étude montre que cette nouvelle cohérence est généralement incomparable avec celles figurant dans la littérature. Enfin, nous présentons des résultats expérimentaux préliminaires pour évaluer l'utilité de cette approche.

1 Introduction

Un CSP (Constraint Satisfaction Problem [16]), parfois également appelé réseau de contraintes, est exprimé par la donnée d'un ensemble fini de variables X, qui doivent être affectées dans leurs domaines (finis) de valeurs donnés par D, et une solution doit satisfaire un ensemble fini de contraintes C. Pour résoudre un CSP, les approches usuelles sont basées sur les algorithmes de type backtracking, dont la complexité en temps est de l'ordre de $O(e.d^n)$ où n est le nombre de variables, e est le nombre de contraintes et d est la taille maximum des domaines. Pour résoudre efficacement un CSP, ces algorithmes exploitent en général des techniques de filtrages, soit avant la recherche sous forme de pré-traitements, soit durant la recherche. La qualité de l'outil de filtrage utilisé joue alors un rôle crucial au niveau de l'efficacité de la recherche.

Les effets d'une méthode de filtrage consistent généralement en la suppression de valeurs dans les domaines. Ces valeurs peuvent être supprimées car nous avons la garantie qu'elles ne peuvent figurer dans aucune solution (elles sont dites incohérentes). Ainsi, les filtrages sont-ils basés sur la notion de *cohérence*.

Dans la mesure où la suppression de toutes les valeurs incohérentes est généralement irréaliste d'un point de vue pratique (c'est un problème NP-difficile), les filtrages sont basés sur des propriétés de cohérences moins fortes, appelées *cohérences partielles*. Ainsi, une valeur peut satisfaire une cohérence partielle, même si elle n'apparaît dans aucune solution (elle est globalement incohérente). A l'opposé, d'autres valeurs peuvent contredire la cohérence partielle considérée, et ainsi être retirées des domaines, sans altérer la satisfiabilité d'un CSP. Les cohérences partielles qui ont été définies jusqu'à présent [2] sont généralement définies par des propriétés de *cohérence locale* dont l'exploitation est étendue à l'ensemble du réseau de contraintes considéré. Par exemple, pour satisfaire la cohérence d'arc (AC) qui est la cohérence partielle la plus utilisée, une valeur doit posséder au moins une valeur compatible (appelée *support*) dans le domaine des variables figurant dans son voisinage au niveau du réseau de contraintes. Sinon, cette valeur est retirée de son domaine (filtrée) et cette suppression peut conduire à d'autres suppressions dans les domaines des variables voisines. En utilisant un mécanisme appelé *propagation de contraintes*, la première suppression peut être

étendue à l'ensemble du réseau. Ainsi, les cohérences partielles correspondent généralement à des propriétés locales qui doivent finalement être vérifiées sur l'ensemble du réseau. D'un point de vue pratique, l'intérêt d'une cohérence partielle est lié à son pouvoir de filtrage ainsi qu'au coût de sa réalisation (en termes de complexité en temps et en espace).

Dans cette contribution, nous introduisons une nouvelle forme de cohérence partielle qui doit satisfaire principalement deux critères, en particulier sur les jeux de données (instances) les plus difficiles à résoudre :

- l'efficacité pratique,
- la puissance de filtrage.

Pour cela, cette nouvelle cohérence devra être :

- paramétrable, de sorte à contrôler la complexité du filtrage, et donc *a priori*, son efficacité pratique,
- adaptée à la structure du réseau de contraintes, en exploitant ses sous-structures,
- adaptée à la dureté des contraintes, en privilégiant l'exploitation des contraintes les plus difficiles à satisfaire.

Cette nouvelle cohérence est appelée *w-SC* car il s'agit d'une cohérence paramétrée (w est le paramètre considéré) qui est basée sur des propriétés Structurelles du réseau.

Elle est définie en considérant une relaxation du CSP (i.e. un sous-problème) qui sera constituée par un réseau de contraintes partiel dont la largeur arborescente (sa tree-width) est bornée par une constante w [15]. Nous avons choisi ce type de sous-problèmes car leur traitement peut s'appuyer sur des algorithmes dont la complexité en temps est polynomiale en w, mais aussi du fait des progrès observés ces dernières années au niveau des techniques dites de décompositions, qui peuvent désormais les résoudre de façon extrêmement efficace d'un point de vue pratique (voir par exemple [13]). De plus, alors que les cohérences partielles classiques considèrent les contraintes sans véritablement tenir compte de leur dureté, ici, nous aurons la possibilité de prendre en compte celle-ci assez naturellement en sélectionnant le sous-problème servant au filtrage de sorte à ce que les contraintes les plus dures y figurent.

Plus précisément, étant donné un sous-réseau de contraintes correspondant à un graphe partiel de largeur arborescente bornée, nous dirons qu'une valeur vérifie la w-SC cohérence si celle-ci figure au moins dans une solution de ce sous-problème.

Il s'agit donc d'une approche fondée sur une idée similaire à la notion de *cohérence inverse* [11], même si formellement, elle est de nature différente.

Ainsi, cette nouvelle cohérence nous permet de définir un nouveau type de filtrage qui s'avère finalement très différent de ceux en usage dans les CSP. Notamment, nous montrerons que la w-SC cohérence est incomparable avec les méthodes reconnues parmi les plus efficaces dans le domaine, comme AC ou SAC [7]. En particulier, les expériences montreront que le comportement de la w-SC cohérence est radicalement différent de celui d'AC ou de SAC, étant notamment plus efficace pour les jeux de données difficiles, tout en s'avérant moins efficace sur des jeux de données faciles (problèmes sous-contraints).

D'un point de vue opérationnel, nous introduisons ici un algorithme de filtrage dont la complexité en temps est $O(n^2.w.d^{w+2})$.

Ce papier est organisé comme suit. La section suivante rappelle les notions classiques de cohérences partielles et les filtrages qui leur sont associés. Dans la section 3, nous introduisons la w-SC cohérence et le filtrage qui lui est associé. La section 4 présente une analyse théorique sur les relations qui peuvent se présenter entre la w-SC cohérence et les autres propriétés de cohérence alors que la section 5 fournit une analyse expérimentale de ce filtrage. Enfin, la section 6 présente les suites qui pourront être données à ce travail.

2 Notions de base

2.1 Notations

Un *problème de satisfaction de contraintes à domaines finis* ou *réseau de contraintes fini* (X, D, C) est défini par la donnée d'un ensemble de variables $X = \{x_1, \dots x_n\}$, d'un ensemble de domaines $D = \{D(x_1), \dots D(x_n)\}$ (le domaine $D(x_i)$ contient les valeurs possibles pour la variable x_i), et d'un ensemble de contraintes C portant sur les variables. Une contrainte $c_i \in C$ est définie d'une part par sa portée $S_C(c_i)$ et d'autre part par une relation de compatibilité associée $R_C(c_i)$. La portée d'une contrainte est un sous-ensemble ordonné de variables, c'est-à-dire $S_C(c_i) = (x_{i_1}, x_{i_2}, \dots x_{i_{a_i}})$ où a_i est appelée *arité* de la contrainte c_i. La relation $R_C(c_i) \subseteq D(x_{i_1}) \times D(x_{i_2}) \cdots \times D(x_{i_{a_i}})$ définit les combinaisons de valeurs compatibles pour les variables de $S_C(c_i)$, chaque combinaison de valeurs compatibles permettant de satisfaire la contrainte. Ici, nous noterons par S_C l'ensemble des portées de contraintes, à savoir $S_C = \{S_C(c_1), S_C(c_2) \dots S_C(c_e)\}$ où $e = |C|$ est le nombre de contraintes. Une solution de (X, D, C) est une affectation de l'ensemble des variables qui satisfait l'ensemble des contraintes. Sans manque de généralité, nous supposerons que chaque variable apparaît une fois au moins dans la portée d'une contrainte. Si toutes les contraintes d'un CSP sont binaires (i.e. elles portent sur deux variables exactement), alors la structure de ce réseau binaire (appelé dans ce cas CSP binaire) peut être représentée par un graphe (X, S_C)

appelé *graphe de contraintes*.

Nous supposerons que les relations ne sont pas vides et peuvent être représentées par des tables comme dans le cadre de la Théorie des Bases de Données Relationnelles. De plus, et sans manque de généralité, nous supposerons que le réseau de contraintes est connexe et normalisé (deux contraintes différentes ont des portées différentes) et pour simplifier le propos, nous ne considérerons ici que les réseaux binaires. Une contrainte c_k telle que $S_C(c_k) = (x_i, x_j)$, c_k sera alors notée c_{ij}. En général, les CSP sont résolus en utilisant des algorithmes de type backtracking qui peuvent être réellement efficaces en pratique, notamment s'ils exploitent judicieusement des méthodes de filtrage avant et/ou pendant la recherche, afin d'éviter l'exploration de zones de l'espace de recherche inutiles. Ces filtrages sont formellement basés sur des notions dites de cohérences partielles.

2.2 Cohérences partielles

La cohérence partielle la plus ancienne et si l'on peut dire, la plus populaire, est appelée *cohérence d'arc* (AC). Etant donné un CSP $P = (X, D, C)$, une valeur $v_i \in D(x_i)$ vérifie la cohérence d'arc vis-à-vis de la contrainte $c_{ij} \in C$ si et seulement s'il existe une valeur valide $v_j \in D(x_j)$ telle que $(v_i, v_j) \in R_C(c_{ij})$. On dit alors que v_j est un support de v_i pour la contrainte c_{ij}. Une valeur $v_i \in D(x_i)$ vérifie la cohérence d'arc si v_i vérifie la cohérence d'arc vis-à-vis de chaque contrainte $c_{ij} \in C$. Un domaine $D(x_i)$ vérifie la cohérence d'arc sur c_{ij} si et seulement si $\forall v_i \in D(x_i)$, la valeur v_i vérifie la cohérence d'arc, et le CSP $P = (X, D, C)$ vérifie la cohérence d'arc si et seulement si $\forall D(x_i) \in D$, le domaine $D(x_i)$ vérifie la cohérence d'arc pour toute contrainte $c_{ij} \in C$. Un filtrage des domaines basé sur AC revient à supprimer les valeurs qui ne vérifient pas la cohérence d'arc. Quand une valeur v_i est supprimée, un mécanisme appelé *propagation de contraintes* est mis en œuvre de sorte à supprimer les valeurs dont l'unique support était la valeur supprimée v_i (aucune autre valeur de $D(x_i)$ n'est compatible avec elle), et ce processus peut être étendu à toutes les autres valeurs, sous les mêmes conditions. Pour les réseaux binaires, plusieurs algorithmes très efficaces ont été proposés comme, par exemple, AC-2001 [5] ou AC3rm [14]. La meilleure complexité en temps obtenue est $O(e.d^2)$. Ce type d'algorithmes est véritablement efficace en pratique, à tel point que le filtrage peut également être mis en œuvre au sein d'un algorithme de recherche de solutions. Néanmoins, le pouvoir de filtrage d'AC se révèle parfois limité du fait de l'aspect très local inhérent à la définition de la cohérence associée. Aussi, des propriétés de cohérence plus forte, et qui permettent donc l'obtention de filtrages plus puissants ont donc été définies. Notamment, [9] a introduit la notion de *k-cohérence* qui est définie en considérant des sous-ensembles de k variables. Pour la k-cohérence, une nouvelle contrainte d'arité égale à $k-1$ est ajoutée au réseau lors du filtrage à partir du moment où une affectation pourtant cohérente sur $k-1$ variables ne peut être étendue à une $k^{ème}$ variable. Si le réseau vérifie la i-cohérence, pour $2 \leq i \leq k$, le CSP est dit *fortement k-cohérent*. A titre de rappel, la cohérence appelée *Strong-PC* aussi dite *cohérence forte de chemin* correspond à la *forte 3-cohérence*. Plus la valeur de k est élevée, plus la puissance du filtrage est forte. Malheureusement, la complexité en temps et en espace est alors $O(n^k.d^k)$ [6] et ce type de filtrage recèle ainsi des inconvénients considérables. Du fait de cette complexité, ces filtrages se révèlent généralement inutilisables, même pour de petites valeurs de k ($k = 3$ s'avère souvent irréaliste en pratique). De plus, ce type de filtrage va rajouter de nouvelles contraintes dans le réseau, dont l'arité est de l'ordre de $k-1$, et pour lesquelles l'espace requis se révèle ainsi très rapidement prohibitif, même pour des petites valeurs de k. Afin d'éviter de tels problèmes, principalement le second induit par l'ajout de contraintes additionnelles, d'autres propriétés de cohérences partielles ont été proposées dans la littérature. Par exemple, [11] a proposé la *k-inverse cohérence*. Le filtrage associé supprime les valeurs qui ne peuvent être étendues à $k-1$ variables pour former une affectation cohérente. Pour ce filtrage, plus puissant qu'AC, le problème relatif à la complexité en espace n'est certes plus présent, mais le problème de la complexité en temps demeure dans la mesure où celle-ci est de l'ordre de $O(n^k.d^k)$. Aussi, ces auteurs ont-ils naturellement suggéré de limiter l'exploitation de cette cohérence à de petites valeurs de k. Dans [11], une autre sorte de cohérence inverse, qui est définie dans un esprit similaire, a été introduite. Elle est appelée *NIC* pour *neighborhood-inverse consistency* (cohérence de voisinage inverse). Ici, le filtrage d'un domaine est induit par la compatibilité des valeurs associées aux variables par rapport au sous-problème défini par son voisinage dans le réseau. De fait, la complexité est alors liée au degré maximum Δ d'une variable dans le graphe de contraintes, et l'algorithme qui a été proposé possède donc une complexité en $O(\Delta^2.(n + e.d).d^{\Delta+1})$.

Une autre façon pour définir des cohérences partielles est fondée sur la prise en compte de sous-problèmes induits par des affectations de la forme $x_i = v_i$, en testant alors leur cohérence partielle. Par exemple, un CSP est dit *Singleton arc-cohérent* (ce qui est noté *SAC*) [7] si pour tous les domaines et pour toutes leurs valeurs $v_i \in D(x_i)$, le sous-problème induit par l'affectation $x_i = v_i$ vérifie la cohérence d'arc sur les sous-domaines. La complexité en temps

du meilleur algorithme connu pour le filtrage associé est $O(e.n.d^3)$ [4].

Pour conclure ce rapide panorama sur les cohérences partielles et filtrages associés, nous devons rappeler que les cohérences partielles généralement considérées comme étant les plus intéressantes d'un point de vue pratique en termes de filtrages sont AC ou SAC qui semblent receler le meilleur compromis entre le coût en temps (et donc l'efficacité pratique) et la puissance de filtrage.

3 La Cohérence Structurelle

3.1 La w-SC Cohérence

La Cohérence Structurelle est basée sur la notion de graphe partiel dont la largeur arborescente (*tree-width*) est bornée par une constante w. La largeur arborescente d'un graphe se définit à partir de la notion de décomposition arborescente (*tree-decomposition*) qui fut introduite formellement dans [15]. Elle a déjà été exploitée dans le domaine des CSP notamment pour définir des classes d'instances polynomiales [12] mais aussi, et surtout, pour proposer des méthodes de résolution particulièrement efficaces pour résoudre des réseaux de contraintes, sous la réserve que ceux-ci possèdent des propriétés topologiques intéressantes [8, 13].

Notre objectif ici s'avère cependant différent de ces précédents travaux puisque nous allons exploiter ces notions essentiellement afin de définir des cohérences partielles. Pour cela, nous introduisons la notion de w-*PST* (pour *partial spanning tree-decomposition*) qui consiste en un graphe partiel dont la décomposition arborescente est de largeur w. Avant cela, nous rappelons la définition de décomposition arborescente :

Définition 1 *Une décomposition arborescente d'un graphe $G = (V, E)$ est une paire (N, T) où $T = (I, F)$ est un arbre avec un ensemble de nœuds I et d'arêtes F, et $N = \{N_i : i \in I\}$ une famille de sous-ensembles de V, telle que chaque sous-ensemble (appelé cluster) N_i est un nœud de T et vérifie :*
(i) $\cup_{i \in I} N_i = V$,
(ii) $\forall \{x, y\} \in E$, $\exists i \in I$ avec $\{x, y\} \subseteq N_i$, et
(iii) $\forall i, j, k \in I$, si k est sur le chemin entre i et 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 sur toutes ses décompositions arborescentes.

Nous définissons maintenant des graphes partiels possédant une largeur arborescente donnée.

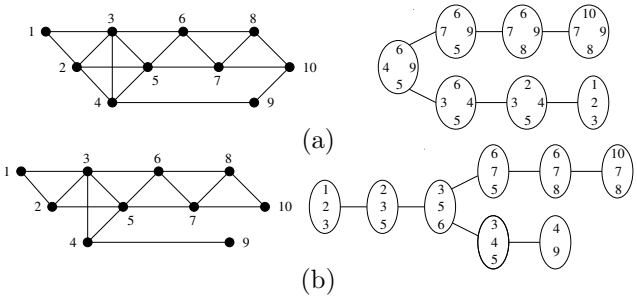


FIG. 1 – (a) Un graphe de largeur arborescente 3 et la décomposition arborescente correspondante. (b) Un 2-PST du graphe donné en (a), et une décomposition arborescente.

Définition 2 *Etant donné un graphe $G = (V, E)$, un graphe partiel de G est un graphe $G' = (V, E')$ où $E' \subseteq E$. Une décomposition arborescente partielle recouvrante de largeur w pour G, notée w-PST, est un graphe partiel de G dont la largeur arborescente est w.*

Le graphe donné dans la figure 1(a) est un 3-PST parce que sa largeur arborescente vaut 3 (une décomposition arborescente optimale est donnée dans cette figure). Dans la figure 1, nous avons le graphe partiel induit par la suppression des arêtes $\{2, 4\}$ et $\{9, 10\}$ dans le graphe donné dans la figure 1(a). Il s'agit d'un 2-PST car sa largeur arborescente vaut 2 comme permet de s'en assurer la décomposition arborescente optimale donnée dans cette figure.

Nous introduisons maintenant la notion de sous-problème d'un CSP induit par l'affectation d'une variable.

Définition 3 *Etant donné un CSP $P = (X, D, C)$, une variable $x_i \in X$ et une valeur $v_i \in D(x_i)$, le sous-problème de P induit par l'affectation $x_i = v_i$ est $P|_{x_i = v_i} = (X, D', C')$ où $D'(x_i) = \{v_i\}$ et pour tout $j \neq i, D'(x_j) = D(x_j)$ et $C' = C$, excepté pour les relations associées aux contraintes incluant x_i qui sont restreintes aux tuples où figure la valeur v_i.*

Avant de définir la cohérence structurelle basée sur un w-PST, nous devons introduire la notion de problème relaxé qui est défini par un sous-ensemble de contraintes d'un CSP donné.

Définition 4 *Etant donné un CSP $P = (X, D, C)$ et $W \subseteq S_C$, le problème relaxé de P induit par W est le CSP $P(W) = (X, D, C')$ où $W = S_{C'}$.*

Pour définir la w-SC cohérence, le problème relaxé est défini par un sous-ensemble de contraintes formant une décomposition arborescente partielle recouvrante de largeur w.

Algorithme 1: Comp-w-SC (**In** : (X, W) : Graph ; **InOut** : $P = (X, D, C)$: CSP)

```
1 for x_i ∈ X do
2     D'(x_i) ← ∅;
3 for x_i ∈ X do
4     for v_i ∈ D(x_i) do
5         if v_i ∉ D'(x_i) then
6             if Solution(P(W)|_{x_i=v_i}, Sol) then
7                 for v_j ∈ Sol do
8                     D'(x_j) ← D'(x_j) ∪ {v_j}
9             else D(x_i) ← D(x_i) − {v_i}
```

Définition 5 *Etant donné un CSP $P = (X, D, C)$ et un w-PST $G = (X, W)$ de (X, S_C) :*

- *La valeur $v_i \in D(x_i)$ est w-SC-cohérente par rapport à G si et seulement si $P(W)|_{x_i=v_i}$ possède une solution.*
- *Le domaine $D(x_i)$ est w-SC-cohérent par rapport à G si et seulement si $\forall v_i \in D(x_i)$, la valeur v_i est w-SC-cohérente par rapport à G.*
- *Le CSP $P = (X, D, C)$ est w-SC-cohérent par rapport à G si et seulement si $\forall D(x_i) \in D$, $D(x_i)$ est w-SC-cohérent par rapport à G.*

Il faut noter que cette définition de cohérence est associée à la notion d'affectation de variable à des valeurs de domaines. Elle peut aisément être généralisée à des affectations plus larges, qui porteraient sur des sous-ensembles de variables. Par manque de place, et pour simplifier cette présentation, nous limiterons la définition de w-SC-cohérence à l'affectation d'une seule variable.

3.2 Filtrage associé

Comme il est d'usage classiquement pour les cohérences dans les CSP, le filtrage associé à la w-SC-cohérence consiste tout simplement à supprimer les valeurs qui ne satisfont pas la propriété correspondante.

Définition 6 *Etant donné un CSP $P = (X, D, C)$ et un w-PST $G = (X, W)$ de (X, S_C), le CSP filtré par l'exploitation de la w-SC-cohérence est le CSP noté $w\text{-}SC(P, W) = (X, D', C')$ où :*

- *$D' = \{D'(x_1), \dots D'(x_n)\}$ où $\forall D'(x_i) \in D'$, $D'(x_i) = \{v_i \in D(x_i) : v_i$ est w-SC-cohérente par rapport à $G\}$.*
- *$S_{C'} = S_C$.*
- *$\forall c'_{ij} \in C', R_{C'}(c'_{ij}) = R_C(c_{ij}) \cap D'(x_i) \times D'(x_j)$.*

Notons qu'étant donné un CSP et un w-PST (X, W) de (X, S_C), $w\text{-}SC(P, W)$ est unique. De plus, pour s'assurer que la w-SC-cohérence définit un filtrage valide, nous devons nous assurer qu'aucune valeur filtrée n'apparaît dans une solution du CSP considéré. C'est nécessairement le cas puisque les valeurs éliminées n'apparaissent dans aucune solution du CSP relaxé considéré. Nous présentons maintenant l'algorithme appelé *Comp-w-SC* qui réalise le filtrage w-SC correspondant. Contrairement aux algorithmes de filtrage classiques, comme notamment ceux réalisant AC, l'obtention de cette cohérence partielle ne nécessite pas le recours à des mécanismes de propagation une fois qu'une suppression de valeur a été réalisée. En effet, alors que les algorithmes classiques réalisent des suppressions, puis propagent celles-ci, dans *Comp-w-SC*, une fois qu'une valeur v_i a été validée en obtenant une solution dans laquelle elle figure, elle ne pourra plus être supprimée et ainsi v_i se trouve définitivement validée. Ceci est tout simplement dû au fait que la valeur v_i apparaît dans une solution du CSP relaxé, en compagnie d'autres valeurs (que l'on peut assimiler aux supports de la valeur v_i). De plus, puisque ces valeurs figurent dans une solution, elles ne pourront plus être supprimées car ces valeurs se trouvent elles aussi validées et pour elles, il ne sera pas nécessaire de tester leur w-SC cohérence.

Dans l'algorithme *Comp-w-SC*, la fonction *Solution($P(W)|_{x_i=v_i}$, Sol)* est appelée pour réaliser le test de cohérence. Si v_i apparaît dans une solution $Sol = (v_1, v_2, \dots v_i, \dots v_n)$ de $P(W)$, la fonction renvoie *true* et *Sol* constitue l'autre résultat de cet appel. Dans le cas contraire, la valeur *false* est renvoyée. Dans *Comp-w-SC*, D' correspond à l'ensemble des domaines contenant les valeurs déjà validées et donc mémorisées pendant la phase de filtrage. Ainsi, si une valeur v_i est déjà présente dans $D'(x_i)$, c'est tout simplement parce que cette valeur figure déjà dans une solution calculée et il ne sera donc pas nécessaire de tester ultérieurement sa w-SC-cohérence. Notons enfin qu'à la fin de l'exécution de *Comp-w-SC*, pour toutes les variables x_i, nous avons $D(x_i) = D'(x_i)$.

Théorème 1 *La complexité en temps de Comp-w-SC est $O(n^2.w.d^{w+2})$ alors que sa complexité en espace est $O(n.w.d^s)$ avec s la taille du plus grand séparateur dans le w-PST.*

Preuve : La fonction *Solution($P(W)|_{x_i=v_i}$)* est appelée au plus $n.d$ fois et le coût d'un appel à cette fonction est borné par $n.w.d^{w+1}$. En effet, elle peut être implémentée en utilisant des algorithmes de résolution basés sur les décompositions arborescentes de CSP comme TC [8] notamment, ou encore, pour être plus efficace en pratique comme BTD [13].

De plus, nous savons que la complexité en espace d'une méthode de résolution par décomposition comme BTD est relative à la taille des séparateurs entre les clusters de la décomposition arborescente.

Aussi, si s est la taille du plus grand séparateur dans le w-PST, comme le nombre de séparateurs est majoré par $n-1$, la complexité en espace se trouve bornée par $O(n.w.d^s)$. □

4 Relations avec les autres cohérences

Pour évaluer la puissance de filtrage de w-SC, nous présentons ici une analyse qui est développée dans un esprit similaire à celui qui a guidé l'analyse présentée dans [7] qui fournit une comparaison entre plusieurs cohérences locales. Ici, nous considérerons AC, SAC, Strong-PC, et plus généralement, la k-cohérence forte ainsi que la k-inverse cohérence. Ce type de comparaisons est basé sur l'existence de relations formelles entre cohérences dont nous rappelons le principe. Nous dirons qu'une cohérence CO_1 est *plus forte* qu'une cohérence CO_2 (ce sera noté $CO_2 \leq CO_1$) si pour tout CSP P pour lequel CO_1 est vérifiée, alors CO_2 est également vérifiée. Ainsi, tout algorithme obtenant CO_1 supprimera au moins les valeurs qui seront supprimées en utilisant CO_2. Nous dirons qu'une cohérence CO_1 est *strictement plus forte* qu'une cohérence CO_2 (ce sera noté $CO_2 < CO_1$) si $CO_2 \leq CO_1$ et s'il existe au moins un CSP P pour lequel CO_2 est vérifiée alors que CO_1 ne l'est pas. Notons par ailleurs que ces relations sont naturellement transitives. Finalement, nous dirons que CO_1 et CO_2 sont *incomparables* si aucune relation entre elles n'est vérifiée.

Théorème 2 *1-SC $<$ AC et pour $w > 1$, w-SC et AC sont incomparables.*

Preuve : Il est clair qu'un 1-PST correspond exactement à un arbre (sous l'hypothèse de connexité). Aussi, puisque dans un réseau de contraintes constitué par un arbre, une valeur apparaît dans une solution si et seulement si elle vérifie AC, 1-SC ne peut filtrer plus de valeurs que ne le fait la cohérence d'arc, qui pour sa part va considérer l'ensemble des contraintes du problème. Par conséquent, nous avons déjà 1-SC $\leq$ AC. De plus, puisque d'autres valeurs du réseau peuvent être supprimées par AC via l'exploitation de contraintes qui n'apparaissent pas dans le 1-PST considéré, nous avons plus précisément 1-SC $<$ AC.

Maintenant, si nous considérons w-SC et AC avec $w > 1$, ces deux cohérences s'avèrent incomparables. En effet, il suffit de constater que pour AC, une valeur du domaine d'une variable est supprimée si elle ne posséde pas de support pour une contrainte qui n'apparaît pas dans un w-PST. Dans ce cas, cette valeur sera peut être conservée par un filtrage de type w-SC. Inversement, une valeur peut être supprimée par un filtrage de type w-SC mais pas par AC. En effet, si nous considérons un CSP constitué de trois variables x_1, x_2 et x_3 ayant toutes le même domaine avec deux valeurs et de trois contraintes de différence c_{12}, c_{13} et c_{23}, AC ne filtre aucune valeur alors que 2-SC appliqué sur l'ensemble du problème (qui est un 2-PST) va supprimer toutes les valeurs. □

Théorème 3 *1-SC $<$ SAC et pour $w > 1$, w-SC et SAC sont incomparables.*

Preuve : Puisque 1-SC $<$ AC et puisque AC $<$ SAC, par transitivité de $<$, nous avons 1-SC $<$ SAC.

Par ailleurs, puisque SAC considère toutes les contraintes qui apparaissent dans le réseau, nécessairement, le filtrage peut supprimer les valeurs qui n'ont pas été supprimées par 2-SC. Inversement, si on considère l'exemple (c) présenté en page 216 de [7] et qui est un 2-PST satisfaisant SAC, nous pouvons facilement voir que 2-SC supprimera une valeur. Par conséquent, 2-SC et SAC sont incomparables. Ainsi, plus généralement, w-SC et SAC sont incomparables.□

Avant d'établir d'éventuelles relations avec des cohérences plus fortes, nous établissons les relations entre différents niveaux de w-SC cohérence. Pour les relations entre w-SC avec différentes valeurs de w, nous supposerons prendre en compte des CSP qui possèdent des décompositions arborescentes partielles recouvrantes de largeurs différentes telles que w-PST $\subseteq$ $(w+1)$-PST (i.e. les arêtes du w-PST considéré sont incluses dans celles du $(w+1)$-PST). Sans cela, nous ne disposerions d'aucune garantie sur la comparison entre w-SC et $(w+1)$-SC.

Théorème 4 *Si w-PST $\subseteq$ $(w+1)$-PST, alors w-SC $<$ $(w+1)$-SC.*

Preuve : Si w-PST $\subseteq$ $(w+1)$-PST, les valeurs supprimées par w-SC cohérence en considérant le w-PST sont nécessairement supprimées en considérant le $(w+1)$-PST. De plus, puisqu'il existe des contraintes dans le $(w+1)$-PST qui ne figurent pas dans le w-PST, des valeurs supplémentaires seront détruites par la $(w+1)$-SC cohérence en considérant le $(w+1)$-PST. Par conséquent, nous avons w-SC $<$ $(w+1)$-SC. □

Sur ces bases, et en appliquant le même principe que dans le théorème 2, nous établissons la propriété suivante qui peut être vue comme sa généralisation :

Théorème 5 *k-SC $<$ $(k+1)$-cohérence forte.*

Preuve : Il est facile de voir que la largeur [10] d'un w-PST est exactement w. Par conséquent,

en appliquant les résultats présentés dans [10] et qui mettent en relation la largeur d'un réseau de contraintes avec le niveau de cohérence forte vérifiée, nécessairement, chaque valeur qui apparaît dans un domaine vérifiant la $(k+1)$-cohérence forte appartient à une solution de ce k-PST. Ainsi, elle ne sera pas supprimée par l'obtention de la k-SC cohérence. En outre, puisque d'autres valeurs de ce réseau peuvent être supprimées par la $(k+1)$-cohérence forte en exploitant des contraintes qui n'apparaissent pas dans le w-PST (avec $w = k$), il est facile de voir que la $(k+1)$-cohérence forte peut supprimer plus de valeurs que le filtrage par k-SC cohérence, et par conséquent, nous avons bien k-SC $<$ $(k+1)$-cohérence forte. □

Théorème 6 *Pour $k > 2$, la k-cohérence forte et k-SC sont incomparables.*

Preuve : Pour montrer que pour $k > 2$, k-SC et la k-cohérence forte sont incomparables, il est alors suffisant de montrer que certaines valeurs peuvent être supprimées par k-SC alors qu'elles ne sont pas filtrées par la k-cohérence forte. Considérons le CSP correspondant au problème de la k-coloration d'un graphe complet à $k+1$ sommets. Ce réseau est un k-PST mais il n'admet pas de solution. Par conséquent, l'obtention de la k-SC cohérence sur ce réseau supprimera toutes les valeurs. Par contre, si maintenant nous considérons l'application de la k-cohérence forte, aucune valeur ne sera supprimée. Par ailleurs, comme la k-cohérence forte prend en compte plus de contraintes qu'il n'en apparaît dans un k-PST, des valeurs seraient supprimées par la k-cohérence forte alors qu'elles ne peuvent l'être par k-SC. Par conséquent, pour $w > 2$, k-SC et la k-cohérence forte sont incomparables. □

Corollaire 1 *2-SC $<$ Strong-PC.*

Finalement, notons que nous pouvons tout à fait remplacer la k-cohérence forte par la k-inverse-cohérence [11] dans l'énoncé du théorème 4. Il est également facile d'établir que NIC et w-SC (pour $w > 2$) sont incomparables.

La figure 2 synthétise les relations entre cohérences usuelles. Les flèches représentent les relations de type $>$ alors que les lignes pointillées indiquent que les deux cohérences concernées sont incomparables.

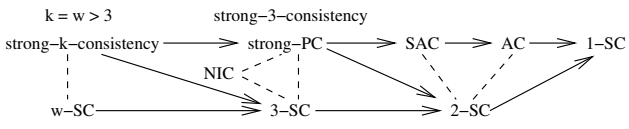


FIG. 2 – Relations entre cohérences

5 Expérimentations

5.1 Protocole expérimental

La qualité du filtrage réalisé par w-SC dépend fortement du w-PST considéré. Dans les résultats expérimentaux présentés ici, le calcul du w-PST est obtenu en utilisant une méthode heuristique qui s'appuie sur la notion de k-arbre [1]. L'exploitation des k-arbres se justifie ici car il s'agit de graphes, qui, pour une largeur arborescente donnée, possèdent le nombre maximum d'arêtes et devraient ainsi potentiellement contenir le nombre maximum de contraintes. Un graphe G est un k-arbre si G est le graphe complet à k sommets (il s'agit alors du k-arbre trivial) ou s'il existe un sommet x de degré k dont le voisinage est un graphe complet et si le graphe obtenu en supprimant de G le sommet x et toutes les arêtes qui lui sont incidentes est un k-arbre. Sur la base de cette définition inductive, on constate que les k-arbres peuvent être construits en commençant par un graphe à k sommets, et que pour chaque ajout de sommet x, celui-ci est connecté aux sommets d'une k-clique déjà présents dans le graphe précédent. Dans le cadre de nos expérimentations, et pour la méthode heuristique que nous utilisons, nous choisirons à chaque étape le sommet x qui minimise $\prod_{c_{ij}} t_{ij}$ où c_{ij} est une contrainte recouverte par la clique formée par x et les sommets de la k-clique considérée dans le graphe précédent et t_{ij} est le rapport entre le nombre de tuples interdits et le nombre total de tuples potentiels. t_{ij} est généralement appelée la dureté de la contrainte c_{ij}. La clique initiale sera par ailleurs choisie en utilisant un procédé similaire. Les choix opérés par cette heuristique sont donc motivés par la volonté d'obtenir des sous-problèmes les plus contraints possibles (cf. choix d'un maximum de contraintes de dureté maximum) de sorte à obtenir la puissance de filtrage la plus forte. Nous avons également implémenté une autre méthode qui consiste à calculer d'abord un k-arbre grâce à la méthode décrite précédemment puis à essayer d'ajouter heuristiquement autant de contraintes que possible tout en garantissant que la largeur arborescente ne dépasse pas w. Les contraintes sont considérées dans l'ordre décroissant de leur dureté. Dans les résultats fournies, nous noterons w-SC1 (respectivement w-SC2) les résultats obtenus lors de l'application de notre algorithme avec un w-PST construit selon la première méthode (respectivement la seconde). Dans les deux cas, nous exploitons un k-arbre avec $k = w$ et le sous-problème relatif au w-PST considéré est résolu grâce à la méthode de décomposition BTD [13].

Pour le filtrage AC, nous avons implémenté AC2001 [5]. Le choix de AC2001 pourrait se discuter au niveau expérimental si on tient compte du temps de cal-

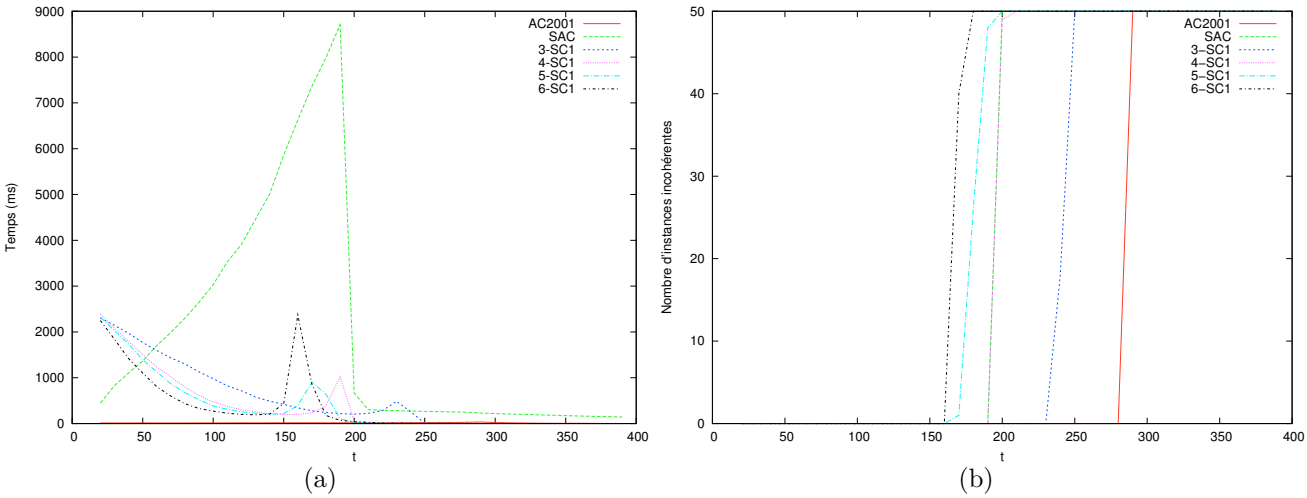


FIG. 3 – Résultats obtenus sur des instances aléatoires pour les classes (200,20,5970,t) : (a) temps d'exécution en millisecondes, (b) nombre d'instances incohérentes détectées.

cul dans la mesure où d'autres algorithmes peuvent se révéler plus rapides, mais, nous verrons qu'ici, cette question ne se pose finalement pas. Pour le filtrage SAC, nous avons utilisé une implémentation naïve de SAC. Tous les algorithmes sont implémentés en C. Notons que nous avons également comparé nos résultats avec une version plus élaborée de SAC (en l'occurrence SAC3 [3] fournie dans le solveur Java Abscon). Concernant le temps d'exécution, notre implémentation est parfois moins efficace que SAC3 mais cela ne change en rien la nature de la comparaison avec w-SC. Bien sûr, les deux versions de SAC ont exactement le même pouvoir de filtrage. Aussi, nous considérons uniquement notre version naïve dans les résultats fournis.

On pourrait, en première analyse, penser que ce type de filtrage est particulièrement adapté aux problèmes structurés (au sens d'une largeur arborescente de petite valeur). Cependant, ce n'est pas le cas, car, dans la mesure où w-SC travaille sur des graphes partiels, il ne semble pas nécessaire de se focaliser sur des graphes structurés. Les résultats expérimentaux montrent d'ailleurs que w-SC s'avère plus intéressant que SAC, par exemple, que ce soit en temps d'exécution ou en capacité de filtrage quand la densité s'accroît et que par conséquent la largeur arborescente augmente.

Ces trois algorithmes ont été comparés sur des instances aléatoires. Le générateur[1] considère 4 paramètres en entrée : n, d, e et t. Il génère des CSP de la classe (n, d, e, t) avec n variables dont la taille uniforme des domaines vaut d et qui possèdent e contraintes binaires ($0 \leq e \leq \frac{n(n-1)}{2}$) pour lesquelles t tuples sont interdits ($0 \leq t \leq d^2$). Les résultats présentés correspondent aux moyennes observées sur des jeux de 50 instances (ayant toutes des graphes de contraintes connexes) par classe. Les expérimentations ont été réalisées sur un PC sous Linux équipé d'un processeur Intel Pentium IV 3, 2 GHz et doté de 1 Go de mémoire.

[1] Voir la page http ://www.lirmm.fr/∼bessiere/generator.html

5.2 Comparaisons d'AC et SAC avec w-SC

Nous avons testé plusieurs classes d'instances obtenues en faisant varier le nombre de variables (jusqu'à 200 variables), la taille des domaines (jusqu'à 40 valeurs), la densité du graphe de contraintes et la dureté des contraintes. Toutefois, par manque de place, nous ne fournirons ici que les résultats obtenus en faisant variant t pour 200 variables, 20 valeurs par domaine et 5970 contraintes (densité de 30%) dans la figure 3 et les résultats obtenus sur quelques classes réprésentatives dans le tableau 1. Notons que nous ne fournissons pas dans la figure 3 les résultats obtenus par w-SC2 car ils sont très proches de ceux de w-SC1 (w-SC2 détecte simplement quelques instances supplémentaires comme étant incohérentes tout en demandant un temps d'exécution légèrement plus important).

Avant de comparer notre algorithme avec AC ou SAC, nous nous intéressons d'abord au choix d'une valeur convenable pour w. Dans la figure 3, si on considère le nombre d'instances détectées comme incohérentes par w-SC, nous pouvons constater que ce nombre augmente lorsque la valeur de w croît. Un tel résultat est prévisible dans la mesure où avec de plus grandes valeurs pour w, w-SC prend en compte plus de contraintes et est donc capable d'accomplir un fil-

Classes	AC			SAC			6-SC1			6-SC2		
(n,d,e,t)	tps	#inc	#suppr	tps	#inc	#suppr	tps	#inc	#suppr	tps	#inc	#suppr
(100,20,495,275)	1,8	0	9,24	198	50	104,68	70	20	486,82	441	48	79,20
(100,20,990,220)	2,4	0	0,22	11987	11	92,04	105	21	566,08	240	48	79,32
(100,20,1485,190)	3,4	0	0	4207	0	0,40	286	31	494,40	187	49	38,30
(100,40,495,1230)	4,6	0	0,92	3239	50	345,06	270	21	621,94	5709	48	106,76
(100,40,990,1030)	5,8	0	0	13229	0	0,08	515	30	809,34	4954	48	176,42
(100,40,1485,899)	8,2	0	0	11166	0	0	1622	32	902,14	1854	48	181,18
(200,10,1990,49)	2,6	0	20,96	88	50	38,28	128	22	350,62	72	48	56,36
(200,10,3980,35)	5,8	0	0,92	10503	49	261,74	248	0	34,86	637	0	249,56
(200,10,5970,30)	7,8	0	0,24	11335	0	11	423	0	54,62	708	2	241,34
(200,20,995,290)	4,6	0	57,58	224	50	65,52	190	26	670,32	7464	49	78,42
(200,20,1990,245)	6	0	3,36	3716	50	256,62	192	32	592,96	1109	50	20
(200,20,3980,195)	12,4	0	0,04	34871	0	1,82	573	25	808,46	592	49	70,48
(200,20,5970,165)	17	0	0	23307	0	0,04	2242	10	1179,88	1600	43	280,3

TAB. 1 – Temps d'exécution (en ms), nombre d'instances détectées comme incohérentes et nombre moyen de valeurs supprimées. Toutes les instances considérées n'ont aucune solution.

trage plus puissant. Ce phénomène est également vrai pour le temps qui augmente avec w. Selon nos observations, la valeur 6 pour w semble correspondre au meilleur compromis entre le temps d'exécution et la puissance de filtrage. D'une part, en exploitant des 6-PST, 6-SC1 (ou 6-SC2) prend en compte suffisamment de contraintes pour assurer un filtrage efficace. D'autre part, avec des valeurs de w plus grandes, le nombre de valeurs supprimées et donc le nombre d'instances incohérentes détectées n'augmentent guère tandis que le temps d'exécution et l'espace requis peuvent croître significativement par rapport à ceux observés pour $w = 6$. Ensuite, si nous comparons w-SC1 et w-SC2, nous pouvons observer dans le tableau 1 que généralement w-SC2 détecte plus d'instances incohérentes que w-SC1. Une fois de plus, un tel résultat est prévisible car w-SC2 exploite plus de contraintes que w-SC1. Pour la même raison, w-SC2 est souvent plus coûteux en temps que w-SC2.

Maintenant, si nous comparons le filtrage de w-SC avec celui de AC ou de SAC, nous pouvons noter dans la figure 3 que 3-SC détecte autant d'inconsistance que AC tandis que SAC se révèle tantôt meilleur, tantôt moins bon que w-SC selon la valeur de w. Néanmoins, nous pouvons observer que 6-SC détecte souvent plus d'instances incohérentes que SAC. Sur les 650 instances considérées dans le tableau 1, SAC est souvent meilleur que 6-SC1 (310 instances détectées comme incohérentes par SAC contre 270 par 6-SC1) mais moins bons que 6-SC2 (310 instances contre 530).

Concernant le temps d'exécution, selon la figure 3(a), AC est généralement plus rapide que w-SC, sauf pour les instances qui sont trivialement incohérentes. Dans ce dernier cas, w-SC parvient très souvent à détecter l'incohérence simplement en supprimant toutes les valeurs de la première variable considérée alors que AC effectue beaucoup de suppressions. Par rapport à SAC, w-SC est plus coûteux en temps sur les instances qui ne sont pas suffisamment dures (pour $t < 50$ sur la figure 3(a)) et donc trivialement cohérentes, car de nombreux appels à BTD sont requis. En revanche, quand la dureté est proche du seuil cohérent / incohérent ou au-delà, w-SC s'avère plus rapide que SAC. En effet, w-SC a besoin de supprimer moins de valeurs que SAC pour détecter l'incohérence car, par construction, il vérifie (et éventuellement supprime) chaque valeur d'une variable avant de traiter la variable suivante alors que dans AC ou SAC, les valeurs d'une variable donnée sont généralement supprimées à des moments différents (du fait du mécanisme de propagation). Ceci explique également les différences au niveau du nombre de valeurs filtrées entre 6-SC1 et 6-SC2 dans le tableau 1. Enfin, nous avons constaté que le comportement de w-SC par rapport à AC ou SAC s'améliore quand la densité du graphe de contraites augmente. Par conséquent, w-SC parvient à obtenir des temps d'exécution nettement meilleurs que SAC et à détecter plus d'instances incohérentes qu'AC et SAC dans la zone voisine du seuil cohérent / incohérent.

6 Discussion et conclusion

Nous avons proposé une nouvelle cohérence partielle paramétrée appelée w-SC-cohérence qui permet d'exploiter les propriétés internes d'un réseau de contraintes, à la fois au niveau structurel mais aussi au niveau de la dureté des contraintes. En l'état, cette nouvelle cohérence permet de définir des filtrages de domaines, à savoir la suppression de valeurs potentielles pour les variables. Cette nouvelle cohérence partielle diffère dans son approche de celles qui avaient été proposées dans la littérature jusqu'à présent. Ceci

peut se constater à la fois au niveau de sa comparaison théorique avec les cohérences partielles classiques mais également au niveau des premières expérimentations que nous avons réalisées. En effet, sur le plan théorique, le pouvoir de filtrage est en général incomparable avec les cohérences usuelles telles que la cohérence d'arc et ses généralisations mais aussi avec la cohérence SAC. Sur le plan pratique, nous avons pu également observer (non reporté dans la partie expérimentale) que ses capacités de filtrage font que ce ne sont pas les même valeurs qui sont supprimées et que les temps de calculs diffèrent au niveau des régions par rapport aux cohérences usuelles.

Il s'agit donc potentiellement d'une approche complémentaires à celles qui existaient précédemment.

Ceci conduit naturellement à proposer comme suite à ce travail, l'élaboration de propriétés de cohérences hybrides qui pourraient combiner w-SC avec d'autres cohérences telles que AC ou SAC, de sorte à fournir des filtrages à la fois plus robustes en termes de temps de calculs, mais aussi et surtout plus puissants en termes de pouvoir filtrant. Parmi les extensions potentielles à ce travail, w-SC pourrait également être étendue aux contraintes d'arité quelconque, ce qui ne semble pas particulièrement difficile d'un point de vue technique. Nous pouvons également généraliser w-SC en étendant le filtrage à des affectations partielles, par exemple en définissant le principe d'une k-w-SC cohérence qui produirait de nouvelle contraintes d'arité k. Dans les travaux qu'il faut également développer, même si 6-SC s'avère plus rapide et constitue un filtrage plus puissant que SAC notamment, du moins sur les jeux d'instances que nous avons testés, il serait utile de mieux identifier les bonnes valeurs du paramètre w. Comme cette cohérence s'appuie sur la notion de réseaux de contraintes partiels, formellement introduits ici en termes de graphes w-PST, une piste de recherche naturelle serait d'évaluer la densité requise des w-PST à considérer pour une valeur donnée de w. Une autre extension naturelle de ce travail consisterait également en l'intégration de cette nouvelle cohérence pour l'exploitation de son filtrage pendant une recherche, alors que nous avons focalisé notre travail pour le moment uniquement au niveau des pré-traitements. Ce type d'étude pourrait être entrepris tout d'abord dans le cadre des méthodes de résolution classiques basées sur le backtracking, mais aussi, et du fait de la proximité naturelle avec les méthodes fondées sur la décomposition de réseaux, sur les techniques à base justement de décomposition. Enfin, une étude approfondie devrait être menée sur le cadre général proposé dans [17] en étudiant différents types de sous-problèmes, pas seulement ceux liés aux décompositions arborescentes partielles recouvrantes, mais ce type de développement déborde du simple cadre de l'extension du travail sur la w-SC cohérence.

Références

[1] L. Beineke and R. Pippert. Properties and characterizations of k-trees. *Mathematika*, 18 :141–151, 1971.

[2] C. Bessiere. *Constraint Propagation*, chapter 3, pages 29–83. Handbook of Constraint Programming, F. Rossi, P. van Beek, T. Walsh, Elsevier, 2006.

[3] C. Bessière, S. Cardon, R. Debruyne, and C. Lecoutre. Efficient Algorithms for Singleton Arc Consistency. *Constraints*, 2010. A paraître.

[4] C. Bessière and R. Debruyne. Optimal and suboptimal singleton arc consistency algorithms. In *Proceedings of IJCAI*, pages 54–59, 2005.

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

[6] M.C Cooper. An optimal k-consistency algorithm. *Artificial Intelligence*, 41(1) :89–95, 1989.

[7] R. Debruyne and C. Bessière. Domain Filtering Consistencies. *JAIR*, 14 :205–230, 2001.

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

[9] E. Freuder. Synthesizing constraint expressions. *CACM*, 21(11) :958–966, 1978.

[10] E. Freuder. A Sufficient Condition for Backtrack-Free Search. *JACM*, 29 (1) :24–32, 1982.

[11] E. Freuder and C. D. Elfe. Neighborhood inverse consistency preprocessing. In *Proceedings of AAAI*, pages 202–208, 1996.

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

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

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

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

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

[17] G. Verfaillie, D. Martinez, and C. Bessière. A Generic Customizable Framework for Inverse Local Consistency. In *Proceedings of AAAI*, pages 169–174, 1999.

Extraction de motifs n-aires utilisant la PPC

Mehdi Khiari, Patrice Boizumault, Bruno Crémilleux
GREYC (CNRS - UMR 6072) – Université de Caen
Boulevard du Maréchal Juin
14000 Caen
{Prénom.Nom}@info.unicaen.fr

Résumé

Dans cet article, nous proposons une approche PPC permettant d'extraire des motifs n-aires (i.e. combinant plusieurs motifs locaux) en fouille de données. Dans un premier temps, l'utilisateur modélise sa requête à l'aide de contraintes portant sur plusieurs motifs locaux. Puis, un solveur de contraintes génère l'ensemble correct et complet des solutions. Notre approche permet de modéliser de manière flexible des ensembles de contraintes portant sur plusieurs motifs locaux et ainsi de découvrir des motifs plus synthétiques et ainsi plus recherchés par l'utilisateur. A notre connaissance, il s'agit de la première approche générique pour traiter ce problème. Les expérimentations menées montrent la pertinence et la faisabilité de l'approche proposée.

1 Introduction

L'Extraction de Connaissances dans les Bases de Données (ECBD) a pour objectif la découverte d'informations utiles et pertinentes répondant aux intérêts de l'utilisateur. L'extraction de motifs sous contraintes est un cadre proposant des approches et des méthodes génériques pour la découverte de motifs locaux [2]. Mais, ces méthodes ne prennent pas en considération le fait que l'intérêt d'un motif dépend souvent d'autres motifs et que les motifs les plus recherchés par l'utilisateur (cf. section 2.2) sont fréquemment noyés parmi une information volumineuse et redondante. C'est pourquoi la transformation des collections de motifs locaux en modèles globaux tels que les classifieurs ou le clustering [13] est une voie active de recherche et la découverte de motifs sous contraintes portant sur des combinaisons de motifs locaux est un problème majeur. Dans la suite, ces contraintes sont appelées *contraintes n-aires*, et les motifs concernés, *motifs n-aires*.

Peu de travaux concernant l'extraction de motifs n-aires ont été menés et les méthodes développées sont toutes ad hoc [20]. La difficulté de la tâche explique l'absence de méthodes génériques : en effet, si l'extraction de motifs locaux nécessite déjà le parcours d'un espace de recherche très conséquent, celui-ci est encore plus grand pour l'extraction de motifs n-aires (le passage de un à plusieurs motifs augmente fortement la combinatoire). Ce manque de généricité est un frein à la découverte de motifs pertinents et intéressants car chaque contrainte n-aire entraîne la conception et le développement d'une méthode ad hoc.

Dans cet article, nous proposons une approche générique pour modéliser et extraire des motifs n-aires grâce à la Programmation par Contraintes (PPC). Notre approche procède en deux étapes. Tout d'abord, l'utilisateur modélise sa requête à l'aide de contraintes portant sur plusieurs motifs locaux. Ces contraintes traduisent des propriétés ensemblistes sur les items et les motifs (inclusion, appartenance, ...) ou des propriétés numériques sur les fréquences et tailles de ces motifs. Puis, un solveur de contraintes génère l'ensemble correct et complet des solutions. Un grand avantage de cette approche est de pouvoir modéliser de manière flexible des ensembles de contraintes portant sur plusieurs motifs locaux et ainsi de découvrir des motifs plus appropriés aux besoins de l'utilisateur. Il n'est plus nécessaire de développer une méthode ad hoc chaque fois que l'on veut extraire de nouveaux motifs n-aires. A notre connaissance, il s'agit de la première approche générique pour traiter ce problème.

La fertilisation croisée entre l'extraction de motifs et la PPC est un domaine de recherche émergeant. Un travail fondateur [6] propose une formulation PPC des contraintes sur les motifs locaux, mais il ne traite pas de l'extraction des motifs n-aires. Dans des travaux

Trans.	Items					
o_1	A	B			c_1	
o_2	A	B			c_1	
o_3			C		c_1	
o_4			C		c_1	
o_5			C		c_1	
o_6	A	B	C	D		c_2
o_7			C	D		c_2
o_8			C			c_2
o_9				D		c_2

TAB. 1 – Exemple de contexte transactionnel $\boldsymbol{r}$.

antérieurs [10, 11], nous avons proposé une approche hybride, reposant sur l'utilisation jointe d'un extracteur de motifs locaux et des Constraint Satisfaction Problems (CSP), pour extraire les motifs n-aires. Dans cet article, nous montrons l'apport d'une nouvelle approche fondée uniquement sur la PPC.

L'article est organisé comme suit : la section 2 présente le contexte général et introduit quelques définitions. Notre approche est décrite dans la section 3, et nous l'illustrons en modélisant plusieurs contraintes n-aires. La section 4 présente la modélisation des contraintes n-aires à l'aide de CSP. La section 5 dresse un bref état de l'art sur l'extraction de motifs n-aires et présente notre approche hybride. Dans la section 6, nous présentons notre nouvelle approche fondée uniquement sur la PPC. La section 7 compare l'approche PPC avec l'approche hybride. Enfin, la section 8 conclut en dressant quelques perspectives sur l'utilisation de la PPC pour l'extraction de motifs.

2 Contexte et motivations

2.1 Définitions

Soit $\mathcal{I}$ un ensemble de littéraux distincts appelés *items*, un motif ensembliste[1] d'items correspond à un sous-ensemble non vide de $\mathcal{I}$. Ces motifs sont regroupés dans le langage $\mathcal{L}_\mathcal{I} = 2^\mathcal{I} \backslash \emptyset$. Un contexte transactionnel est alors défini comme un multi-ensemble de motifs de $\mathcal{L}_\mathcal{I}$. Chacun de ces motifs, appelé transaction, constitue une entrée de la base de données. Ainsi, le tableau 1 présente un contexte transactionnel $\boldsymbol{r}$ où 9 transactions étiquetées $o_1, \ldots, o_9$ sont décrites par 6 items $A, \ldots, D, c_1, c_2$.

L'extraction de motifs a pour but la découverte d'informations à partir de tous les motifs ou d'un sous ensemble de $\mathcal{L}_\mathcal{I}$. L'extraction sous contraintes cherche la collection de tous les motifs de $\mathcal{L}_\mathcal{I}$ présents dans $\boldsymbol{r}$ et satisfaisant un prédicat appelé *contrainte*. Ces motifs sont appelés *motifs locaux* ; ce sont des régularités observées dans certaines parties des données. La localité de ces motifs provient du fait que, vérifier s'ils satisfont une contrainte donnée, peut s'effectuer indépendamment des autres motifs.

Il y a de nombreuses contraintes permettant d'évaluer la pertinence et la qualité des motifs locaux. Un exemple bien connu est celui de la contrainte de fréquence qui permet de rechercher les motifs X ayant une fréquence $freq(X)$ supérieure à un seuil minimal fixé $minfr > 0$. De nombreux travaux [16] remplacent la fréquence par d'autres mesures permettant d'évaluer l'intérêt des motifs locaux recherchés. C'est le cas de la mesure d'aire : soit X un motif, $aire(X)$ est le produit de la fréquence de X par sa taille, i.e., $aire(X) = freq(X) \times long(X)$ où $long(X)$ désigne la longueur (i.e., le nombre d'items) de X.

2.2 Motivations

En pratique, l'utilisateur est très souvent intéressé par la découverte de motifs plus riches que les motifs locaux et qui révèlent des caractéristiques et propriétés de l'ensemble de données étudié. De tels motifs portant sur plusieurs motifs locaux sont appelés *motifs n-aires* et permettent l'expression de *contraintes n-aires*.

Défintion 1 (contrainte n-aire)**.** *Une contrainte c est dite n-aire si elle porte sur plusieurs motifs locaux.*

Défintion 2 (motif n-aire)**.** *Un motif X est dit n-aire si il apparaît dans (au moins) une contrainte n-aire.*

Les contraintes n-aires permettent de modéliser un large ensemble de motifs utiles à l'utilisateur tel que la découverte de règles d'exceptions [20] ou la détection de règles susceptibles de donner lieu à des conflits de classifications dans le contexte de la classification associative [23]. D'autres contraintes n-aires sont présentées à la section 3.

Exemple 1 : E. Suzuki s'est intéressé à la découverte de paires de règles incluant une règle d'exception [20]. Une règle d'exception est une règle qui exprime une situation déviant d'un comportement général modélisé par une autre règle : l'intérêt de cette définition est d'expliciter la nature d'une exception par rapport à un comportement général et admis. Formellement, les règles d'exception sont définies comme suit (I est un item, par exemple une valeur de classe, X et Y sont des motifs locaux) :

$$e(X,Y,I) \equiv \begin{cases} vrai & \text{si } \exists Y \in \mathcal{L}_\mathcal{I} \text{ tq } Y \subset X, \\ & (X \backslash Y \rightarrow I) \wedge (X \rightarrow \neg I) \\ faux & \text{sinon} \end{cases}$$

[1] Dans cet article, nous nous intéressons au cas des motifs ensemblistes

Dans une telle paire de règles, $X \backslash Y \rightarrow I$ est une règle générale et $X \rightarrow \neg I$ est une règle d'exception qui révèle ainsi une information inattendue. Cette définition demande à ce que la règle générale soit de forte fréquence et de forte confiance tandis que la règle d'exception est peu fréquente mais de très forte confiance (la confiance d'une règle $X \rightarrow Y$ est mesurée par le rapport $freq(X \cup Y)/freq(X)$). La comparaison entre la règle générale et la règle d'exception ne peut pas être modélisée par une approche reposant uniquement sur les motifs locaux. Par contre, elle se modélise aisément à l'aide des contraintes n-aires. Donnons un exemple de règle d'exception à partir du tableau 1. En prenant 2/3 comme seuil de confiance d'une règle, la règle $AC \rightarrow \neg c_1$ est une règle d'exception puisque nous avons conjointement $A \rightarrow c_1$ et $AC \rightarrow \neg c_1$. E. Suzuki a proposé une méthode fondée sur une estimation probabiliste [20] pour extraire de telles paires de règles. Mais, cette approche est totalement dédiée à ce type de motifs.

Exemple 2 : Considérons l'exemple du transcriptome et de l'analyse d'expressions de gènes : le biologiste est vivement intéressé par la recherche de groupes de synexpressions. Les motifs locaux, composés de tags (ou gènes), qui satisfont la contrainte d'aire (cf. section 2.1), sont susceptibles de donner lieu à des groupes de synexpressions [12]. D'autre part, il faut être capable de prendre en compte l'incertain qui est présent dans ce type de données [1]. Les contraintes n-aires sont une façon naturelle de concevoir des motifs tolérants aux fautes et candidats à être des groupes de synexpressions : ceux-ci sont définis par l'union de plusieurs motifs locaux satisfaisant une contrainte d'aire et ayant un fort recouvrement entre eux. Plus précisément, à partir de deux motifs locaux X et Y, on définit la contrainte n-aire suivante :

$$c(X,Y) \equiv \begin{cases} aire(X) > min_{aire} \wedge \\ aire(Y) > min_{aire} \wedge \\ aire(X \cap Y) > \alpha \times min_{aire} \end{cases}$$

où min_{aire} est le seuil minimal d'aire et α est un paramètre fourni par l'utilisateur pour fixer le recouvrement minimal entre motifs locaux.

3 Exemples de contraintes n-aires

Dans cette section nous présentons plusieurs exemples de contraintes n-aires. Certaines d'entre elles ont déjà été introduites à la section 2.2.

3.1 Règles d'exception

Soient X et Y deux motifs, et I un item tel que I et $\neg I \in \mathcal{I}$ (I et $\neg I$ peuvent représenter deux classes présentes dans le jeu de données). Soient les seuils de fréquence *minfr* et *maxfr* et les seuils de confiance δ_1 et δ_2. La contrainte n-aire relative aux règles d'exception se modélise comme suit :

- $X \backslash Y \rightarrow I$ doit être une règle fréquente de forte confiance : $freq((X \backslash Y) \sqcup I) \geq minfr \wedge freq(X \backslash Y) - freq((X \setminus Y) \sqcup I) \leq \delta_1$.
- $X \rightarrow \neg I$ doit être une règle rare de forte confiance : $freq(X \sqcup \neg I) \leq maxfr \wedge (freq(X) - freq(X \sqcup \neg I)) \leq \delta_2$.

En résumé :

$$e(X,Y,I) \equiv \begin{cases} \exists Y \subset X \text{ tq :} \\ freq((X \setminus Y) \sqcup I) \geq minfr \wedge \\ (freq(X \setminus Y) - freq((X \setminus Y) \sqcup I)) \leq \delta_1 \wedge \\ freq(X \sqcup \neg I) \leq maxfr \wedge \\ (freq(X) - freq(X \sqcup \neg I)) \leq \delta_2 \end{cases}$$

3.2 Règles inattendues

Padmanabhan et Tuzhilin ont introduit dans [18] la notion de règle *inattendue* $X \rightarrow Y$ par rapport à une croyance $U \rightarrow V$ où U et V sont des motifs. Une règle inattendue est définie dans [18] par :

1. $Y \wedge V$ n'est pas valide,
2. $X \wedge U$ est valide (XU est un motif fréquent),
3. $XU \rightarrow Y$ est valide ($XU \rightarrow Y$ est une règle fréquente et de confiance suffisante),
4. $XU \rightarrow V$ n'est pas valide (soit $XU \rightarrow V$ n'est pas une règle fréquente, soit $XU \rightarrow V$ est une règle de faible confiance).

Etant donnée une croyance $U \rightarrow V$, une règle inattendue $un(X,Y)$ est modélisée par :

$$un(X,Y) \equiv \begin{cases} freq(Y \cup V) = 0 \wedge \\ freq(X \cup U) \geq minfr_1 \wedge \\ freq(X \cup U \cup Y) \geq minfr_2 \wedge \\ freq(X \cup U \cup Y)/freq(X \cup U) \geq minconf \wedge \\ (freq(X \cup U \cup V) < maxfr \vee \\ freq(X \cup U \cup V)/freq(X \cup U) < maxconf) \end{cases}$$

3.3 Groupes de synexpressions

La recherche de groupes de synexpressions à partir de n motifs locaux se modélise à l'aide de la contrainte n-aire suivante :

$$synexpr(X_1, ..., X_n) \equiv \begin{cases} \forall\ 1 \leq i < j \leq n, \\ aire(X_i) > min_{aire} \wedge \\ aire(X_j) > min_{aire} \wedge \\ aire(X_i \cap X_j) > \alpha \times min_{aire} \end{cases}$$

où min_{aire} désigne la surface minimale (définie à la section 2.1) et α est un seuil, défini par l'utilisateur,

permettant de quantifier le recouvrement minimal souhaité. Cet exemple montre comment on peut modéliser des motifs complexes et tolérants aux fautes tels que les groupes de synexpressions.

3.4 Conflits de classification

La combinaison des motifs locaux est un point clé de la qualité d'un classifieur à base d'associations qui est généralement construit à partir de règles fréquentes et de forte confiance. Typiquement, les paires de règles ayant un important chevauchement entre leurs prémisses et concluant sur des classes distinctes sont particulièrement susceptibles de donner lieu à un conflit de classification. En effet, lorsqu'une règle d'une telle paire est déclenchée par un exemple à classer, l'autre règle concluant sur une autre valeur de classe est fortement susceptible d'être également déclenchée car les prémisses des deux règles sont relativement similaires. Ce double déclenchement conduira à un conflit de classification. En étant capable de prendre en compte plusieurs motifs locaux, les contraintes n-aires permettent de modéliser de façon naturelle de tels conflits de classification. Soient $X \rightarrow c_1$ et $Y \rightarrow c_2$ deux règles fréquentes et de forte confiance, une paire de règles susceptibles de donner lieu à un conflit de classification s'exprime de la façon suivante :

$$c(X,Y) \equiv \left\{ \begin{array}{l} freq(X) \geq minfr \wedge \\ freq(Y) \geq minfr \wedge \\ (freq(X \sqcup \{c_1\})/freq(X)) \geq minconf \wedge \\ (freq(Y \sqcup \{c_2\})/freq(Y)) \geq minconf \wedge \\ long(X \cap Y) \geq (long(X) + long(Y))/4 \end{array} \right.$$

Les quatre premières contraintes d'inégalité portent sur la fréquence et la confiance des règles de classification. La dernière contrainte décrit le chevauchement souhaité : les deux règles doivent avoir en commun au moins la moitié des items de leurs prémisses. Notons qu'il est simple pour l'utilisateur de modifier les paramètres de la contrainte n-aire et/ou ajouter de nouvelles contraintes pour modéliser des types de conflits de classification plus spécifiques.

4 Modélisation sous forme de CSP

4.1 Aperçu général

Soit $\boldsymbol{r}$ un jeu de données ayant nb transactions et $\mathcal{I}$ l'ensemble de ses items. La recherche de motifs ensemblistes peut se modéliser à l'aide de deux CSP $\mathcal{P}$ et $\mathcal{P}'$ inter-reliés :

1. CSP ensembliste $\mathcal{P}=(\mathcal{X},\mathcal{D},\mathcal{C})$ où :
   - $\mathcal{X}=\{X_1,...,X_n\}$. Chaque variable X_i représente un motif ensembliste inconnu.
   - $\mathcal{D}=\{D_{X_1},...,D_{X_n}\}$. Le domaine initial de chaque variable X_i est $[\{\} \,..\, \mathcal{I}]$.
   - $\mathcal{C}$ est une conjonction de contraintes ensemblistes formulées à l'aide d'opérateurs ensemblistes ($\cup$, $\cap$, $\setminus$, $\in$, $\notin$, ...).
2. CSP numérique $\mathcal{P}'=(\mathcal{F},\mathcal{D}',\mathcal{C}')$ où :
   - $\mathcal{F}=\{F_1,...,F_n\}$. Chaque variable F_i est la fréquence du motif X_i.
   - $\mathcal{D}'=\{D_{F_1},...,D_{F_n}\}$. Le domaine initial de chaque variable F_i est $[1 \,..\, nb]$.
   - $\mathcal{C}'$ est une conjonction de contraintes numériques telles que : $<, \leq, \neq, =, ...$

4.2 Exemple des règles d'exception

Contrainte	Formulation
$freq((X \setminus Y) \sqcup I) \geq minfr$	$F_2 \geq minfr$ $\wedge$ $I \in X_2$ $\wedge$ $X_1 \subsetneq X_3$
$freq(X \setminus Y) - freq((X \setminus Y) \sqcup I) \leq \delta_1$	$F_1 - F_2 \leq \delta_1$ $\wedge$ $X_2 = X_1 \sqcup I$
$freq(X \sqcup \neg I) \leq maxfr$	$F_4 \leq maxfr$ $\wedge$ $\neg I \in X_4$
$freq(X) - freq(X \sqcup \neg I) \leq \delta_2$	$F_3 - F_4 \leq \delta_2$ $\wedge$ $X_4 = X_3 \sqcup \neg I$

TAB. 2 – Formulation des contraintes

La table 2 décrit l'ensemble des contraintes primitives modélisant les règles d'exception.

- Les variables ensemblistes $\{X_1, X_2, X_3, X_4\}$ représentent les motifs recherchés :
  - $X_1 : X \setminus Y$, et $X_2 : (X \setminus Y) \sqcup I$ (règle générale),
  - $X_3 : X$, et $X_4 : X \sqcup \neg I$ (règle d'exception).
- Les variables entières $\{F_1, F_2, F_3, F_4\}$ représentent leurs fréquences.
- Contraintes ensemblistes : $\mathcal{C} = \{(I \in X_2), (X_2 = X_1 \sqcup I), (\neg I \in X_4), (X_4 = X_3 \sqcup \neg I), (X_1 \subsetneq X_3)\}$
- Contraintes numériques : $\mathcal{C}' = \{(F_2 \geq minfr), (F_1 - F_2 \leq \delta_1), (F_4 \leq maxfr), (F_3 - F_4 \leq \delta_2)\}$

5 Etat de l'art

5.1 Découverte de motifs en fouille de données

Alors qu'il existe de nombreux travaux traitant de la découverte de motifs locaux sous contraintes [5, 16], très peu d'approches considérant simultanément plusieurs motifs locaux ont été proposées : citons les "patterns teams" [14], les ensembles sous contraintes de motifs [7] ou encore la sélection de motifs suivant leur intérêt compte tenu d'autres motifs déjà sélectionnés [3]. Même si ces approches comparent explicitement les motifs locaux entre eux, elles sont principalement fondées sur la réduction de la redondance entre motifs ou

poursuivent des objectifs spécifiques tels que la classification. De part leur flexibilité, les contraintes n-aires permettent à l'utilisateur d'exprimer, dans un cadre unique de modélisation, des biais de recherche variés et donc des types de motifs très divers. Notons qu'il existe des cadres génériques pour la construction de modèles globaux à partir de motifs locaux [13, 9]. Mais, ces cadres ne proposent pas de méthode d'extraction : ils permettent seulement de mieux comparer les approches existantes.

5.2 Une approche hybride

Nous avons proposé [10, 11] une première approche fondée sur l'utilisation conjointe d'un extracteur de motifs locaux et de la PPC pour extraire des motifs n-aires. Dans cette section, nous donnons une vue d'ensemble de cette approche hybride avant de détailler chacune de ses trois étapes en considérant l'exemple des règles d'exception (cf. la section 4.2).

5.2.1 Aperçu général

La figure 1 présente une vue d'ensemble des trois étapes de notre approche :

1. Modéliser la contrainte n-aire sous forme de CSP, puis distinguer les contraintes locales (portant sur un seul motif) des autres (portant sur plusieurs motifs).
2. Résoudre les contraintes locales à l'aide d'un extracteur de motifs locaux (MUSIC-DFS[2]) [19] qui produit une représentation condensée par intervalles de tous les motifs satisfaisant les contraintes locales.
3. Résoudre les autres contraintes[3] à l'aide du solveur de contraintes ECL^iPS^e[4]. Le domaine de chaque variable résulte de la représentation condensée par intervalles calculée dans la seconde étape.

5.2.2 Partitionner les contraintes

L'ensemble de toutes les contraintes ($\mathcal{C} \cup \mathcal{C}'$) est divisé en deux sous ensembles :

- $\mathcal{C}_{loc}$ est l'ensemble des contraintes locales à résoudre par MUSIC-DFS. Les solutions sont fournies sous forme d'une représentation condensée par intervalles.

[2] `http://www.info.univ-tours.fr/~soulet/music-dfs/music-dfs.html`

[3] Ces contraintes portent sur plusieurs variables, et sont donc n-aires au sens PPC. Pour éviter toute confusion, le terme n-aire sera réservé aux contraintes portant sur plusieurs motifs locaux au sens de la fouille de données (cf. la définition 1).

[4] `http://www.eclipse-clp.org`

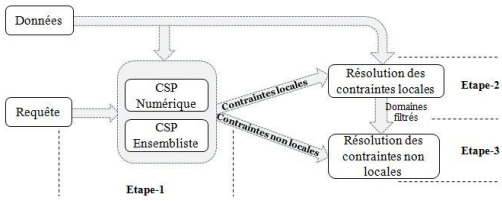


FIG. 1 – Aperçu des 3 étapes

- $\mathcal{C}_{autres}$ est l'ensemble des contraintes restantes à résoudre par ECL^iPS^e. Les domaines des variables X_i et F_i sont déduits de la représentation condensée par intervalles calculée dans l'étape précédente.

Pour les règles d'exception (cf. la table 2), cette partition s'effectue comme suit :

- $\mathcal{C}_{loc} = \{(I \in X_2), (F_2 \geq minfr), (F_4 \leq maxfr), (\neg I \in X_4)\}$
- $\mathcal{C}_{autres} = \{(F_1 - F_2 \leq \delta_1), (X_2 = X_1 \sqcup I), (F_3 - F_4 \leq \delta_2), (X_4 = X_3 \sqcup \neg I), (X_1 \subsetneq X_3)\}$

5.2.3 Résolution des contraintes locales

MUSIC-DFS est un extracteur correct et complet de motifs locaux qui permet d'extraire efficacement l'ensemble des motifs satisfaisant un large éventail de contraintes locales [19]. L'efficacité de MUSIC-DFS est due à sa stratégie de recherche en profondeur d'abord et à l'élagage de l'espace de recherche se basant sur la propriété d'anti-monotonie dans l'espace des motifs candidats. Les motifs locaux sont produits sous forme de représentation condensée composée d'intervalles disjoints où chaque intervalle synthétise un ensemble de motifs satisfaisants la contrainte [19].

Pour la méthode hybride, les contraintes locales sont résolues avant et indépendamment des autres contraintes. Ainsi, l'espace de recherche des autres contraintes sera réduit à l'espace des solutions des contraintes locales.

L'ensemble des contraintes locales $\mathcal{C}_{loc}$ est partitionné en une union disjointe de $\mathcal{C}_i$ (pour $i \in [1..n]$) où chaque $\mathcal{C}_i$ est l'ensemble des contraintes locales portant sur X_i et F_i. Chaque C_i peut être résolu de manière séparée et indépendante. Soit CR_i la représentation condensée sous forme d'intervalles issue de la résolution de $\mathcal{C}_i$ par MUSIC-DFS, $CR_i = \bigcup_p (f_p, I_p)$ où chaque I_p est un intervalle ensembliste vérifiant : $\forall x \in I_p,\ freq(x) = f_p$. Les domaines des variables X_i et F_i sont définis par :

- $D_{F_i} = \{f_p \mid f_p \in CR_i\}$,
- $D_{X_i} = \bigcup_{I_p \in CR_i} I_p$.

Exemple : Considérons $\mathcal{C}_{loc}$ l'ensemble des contraintes locales relatives aux règles d'exception, et prenons comme valeurs respectives de $(I, \neg I, minfr, \delta_1, maxfr, \delta_2)$ les valeurs suivantes : $(c_1, c_2, 2, 1, 1, 0)$. L'ensemble des contraintes locales relatives à X_2 et F_2, $\mathcal{C}_2 = \{c_1 \in X_2, F_2 \geq 2\}$, est résolu par MUSIC-DFS par la requête suivante, où les paramètres peuvent être directement déduits de $\mathcal{C}_2$:

```
---------------
music-dfs data -q "{c1} subset X2 and freq(X2)>=2;"
X2 in [A, c1]..[A, c1, B ] U [B, c1] -- F2 = 2 ;
X2 in [C, c1] -- F2 = 3
---------------
```

5.2.4 Résolution des contraintes non locales

Les domaines des variables X_i et F_i sont obtenus à partir de la représentation condensée sous forme d'intervalles de tous les motifs satisfaisant les contraintes locales. La résolution de ces contraintes par ECL^iPS^e permettra ainsi d'obtenir l'ensemble de tous les motifs satisfaisant l'intégralité des contraintes.

Exemple : Considérons $\mathcal{C}_{autres}$ l'ensemble des contraintes non locales pour les règles d'exception, les valeurs respectives de $(I, \neg I, minfr, \delta_1, maxfr, \delta_2)$ étant toujours les mêmes. La session ECL^iPS^e ci-dessous illustre comment toutes les paires de règles d'exception peuvent être obtenues par backtracking :

```
---------------
[eclipse 1]:
?- exceptionsRules(X1, X2, X3, X4).
X1=[A,B], X2=[A,B,c1], X3=[A,B,C], X4=[A,B,C,c2];
X1=[A,B], X2=[A,B,c1], X3=[A,B,D], X4=[A,B,D,c2];
.../...
---------------
```

6 Une approche PPC

Cette section présente une nouvelle approche fondée uniquement sur la PPC. Cette approche repose, elle aussi, sur la modélisation sous forme de CSP (cf Section 4). Mais, à la différence de la méthode hybride (cf. la section 5.2), cette approche dite "PPC" n'utilise pas d'extracteur de motifs locaux. La section 7 compare en profondeur ces deux approches.

La modélisation s'effectue en trois étapes :

1. Lier les transactions et les motifs n-aires,
2. Modélisation des motifs n-aires recherchés,
3. Reformulation des contraintes ensemblistes et des contraintes numériques.

Cette approche a été mise en oeuvre en Gecode[5]. Pour la première étape (cf. section 6.1), nous avons utilisé l'implantation de l'extracteur de motifs FIM_CP[6] qui est une approche PPC pour l'extraction de motifs [6]. Cette approche traite dans un cadre unifié un large ensemble de motifs locaux et de contraintes telles que la fréquence, la fermeture, la maximalité, les contraintes monotones ou anti-monotones et des variations de ces contraintes. Mais FIM_CP ne traite pas les contraintes n-aires.

[5] http://www.gecode.org

[6] http://www.cs.kuleuven.be/~dtai/CP4IM/fim_cp.php

6.1 Transactions et les motifs n-aires

Nous avons utilisé une technique similaire à celle de FIM_CP pour établir le lien entre l'ensemble des transactions et les motifs n-aires que nous recherchons.

Soit $\boldsymbol{r}$ un contexte transactionnel où $\mathcal{I}$ est l'ensemble de ses n items et $\mathcal{T}$ l'ensemble de ses m transactions. Soit $\boldsymbol{d}$ la matrice 0/1 de dimension (m, n) telle que $\forall t \in \mathcal{T}, \forall i \in \mathcal{I}$, $(d_{t,i} = 1)$ ssi $(i \in t)$. Soit M le motif (unique) recherché par FIM_CP. Deux sortes de variables de décision (de domaine $\{0, 1\}$) sont utilisées :

- $\{M_1, M_2, ..., M_n\}$ où $(M_i = 1)$ ssi $(i \in M)$,
- $\{T_1, T_2, ..., T_m\}$ où $(T_t = 1)$ ssi $(M \subseteq t)$.

Ainsi, $freq(M) = \sum_{t \in \mathcal{T}} T_t$ et $long(M) = \sum_{i \in \mathcal{I}} M_i$.

La relation entre le motif recherché M et $\mathcal{T}$ est établie via des contraintes réifiées [6] imposant que, pour chaque transaction t, $(T_t = 1)$ ssi $(M \subseteq t)$, ce qui se reformule en :

$$\forall t \in \mathcal{T}, (T_t = 1) \Leftrightarrow \sum_{i \in \mathcal{I}} M_i \times (1 - d_{t,i}) = 0 \qquad (1)$$

Le filtrage associé à chaque contrainte réifiée (cf Equation 1) procède comme suit : si l'on peut déduire que (T_t=1) (resp. T_t=0), alors la somme soit être nulle (resp. non-nulle). La propagation s'effectue de manière analogue de la partie droite vers la partie gauche de l'équivalence.

6.2 Modélisation des k motifs n-aires recherchés

Soit $\boldsymbol{r}$ un contexte transactionnel où $\mathcal{I}$ est l'ensemble de ses n items et $\mathcal{T}$ l'ensemble de ses m transactions. Soit $\boldsymbol{d}$ la matrice 0/1 de dimension (m, n) telle que $\forall t \in \mathcal{T}, \forall i \in \mathcal{I}$, $(d_{t,i} = 1)$ ssi $(i \in t)$. Soient $X_1, X_2, ..., X_k$ les k motifs n-aires recherchés.

Tout d'abord, chaque motif n-aire inconnu X_j est modélisé par n variables de décision $\{X_{1,j}, X_{2,j}, ..., X_{n,j}\}$ (de domaine $\{0, 1\}$) telles que $(X_{i,j} = 1)$ ssi l'item i appartient au motif X_j :

$$\forall i \in \mathcal{I}, (X_{i,j} = 1) \Leftrightarrow (i \in X_j) \qquad (2)$$

Puis, m variables de décision $\{T_{1,j}, T_{2,j}, ..., T_{m,j}\}$ (de domaine $\{0, 1\}$) sont associées à chaque motif n-aire inconnu X_j telles que $(T_{t,j} = 1)$ ssi $(X_j \subseteq t)$:

$$\forall t \in \mathcal{T}, (T_{t,j} = 1) \Leftrightarrow (X_j \subseteq t) \qquad (3)$$

Ainsi, $freq(X_j)=\sum_{t\in\mathcal{T}} T_{t,j}$ et $long(X_j)=\sum_{i\in\mathcal{I}} X_{i,j}$.

La relation entre le motif recherché X_j et $\mathcal{T}$ est établie via des contraintes réifiées imposant que, pour chaque transaction t, $(T_{t,j}=1)$ ssi $(X_j \subseteq t)$, ce qui se reformule en :

$$\forall j \in [1..k], \forall t \in \mathcal{T}, (T_{t,j}=1) \Leftrightarrow \sum_{i\in\mathcal{I}} X_{i,j}\times(1-d_{t,i}) = 0 \quad (4)$$

6.3 Contraintes numériques et ensemblistes

Considérons un opérateur $\texttt{op} \in \{<, \leq, >, \geq, =, \neq\}$; les contraintes numériques se reformulent comme suit :

- $freq(X_p)$ op $\alpha \rightarrow \sum_{t\in\mathcal{T}} T_{t,p}$ op α
- $long(X_p)$ op $\alpha \rightarrow \sum_{i\in\mathcal{I}} X_{i,p}$ op α

Certaines contraintes ensemblistes (telles que égalité, inclusion, appartenance,...) se reformulent directement à l'aide de contraintes linéaires :

- $X_p = X_q \rightarrow \forall i \in \mathcal{I}, X_{i,p} = X_{i,q}$
- $X_p \subseteq X_q \rightarrow \forall i \in \mathcal{I}, X_{i,p} \leq X_{i,q}$
- $i_o \in X_p \rightarrow X_{i_0,p} = 1$

Les autres contraintes ensemblistes (telles que intersection, union, différence,...) se reformulent aisément à l'aide de contraintes booléennes en utilisant la fonction de conversion $(b::\{0,1\} \rightarrow \{False, True\})$ et les opérateurs booléens usuels :

- $X_p \cap X_q = X_r \rightarrow \forall i \in \mathcal{I}, b(X_{i,r}) = b(X_{i,p}) \wedge b(X_{i,q})$
- $X_p \cup X_q = X_r \rightarrow \forall i \in \mathcal{I}, b(X_{i,r}) = b(X_{i,p}) \vee b(X_{i,q})$
- $X_p \backslash X_q = X_r \rightarrow \forall i \in \mathcal{I}, b(X_{i,r}) = b(X_{i,p}) \wedge \neg b(X_{i,q})$

Enfin, l'ensemble des contraintes, qu'elles soient réifiées, numériques ou ensemblistes, est traité par *Gecode*.

6.4 Expérimentations

Cette section a pour but de montrer la faisabilité et les apports pratiques de l'approche PPC présentée dans cet article.

6.4.1 Protocole expérimental

Différentes expérimentations ont été menées sur plusieurs jeux de données de l'UCI *repository*[7] ainsi que sur un jeu de données réelles nommé `Meningitis` et provenant de l'Hôpital Central de Grenoble. `Meningitis` recense les pathologies des enfants atteints d'une méningite virale ou bactérienne. La table 3 résume les différentes caractéristiques de ces jeux de données.

Les expérimentations ont été menées sur plusieurs contraintes n-aires : règles d'exception, règles rares et conflits de classification. La machine utilisée est un PC 2.83 GHz Intel Core 2 Duo processor (4 GB de RAM) sous Ubuntu Linux.

[7] `http://www.ics.uci.edu/~mlearn/MLRepository.html`

Jeu de données	#trans	#items	densité
`Mushroom`	8142	117	0.18
`Australian`	690	55	0.25
`Meningitis`	329	84	0.27

TAB. 3 – Description des jeux de données

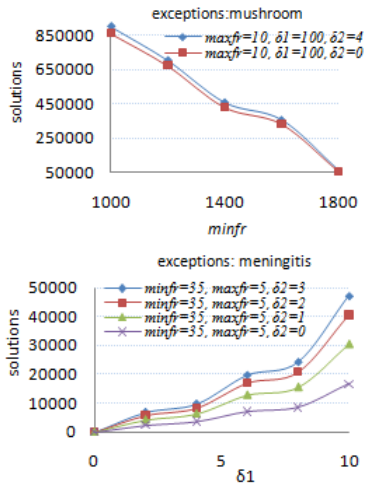


FIG. 2 – Evolution du nombre de règles d'exception

6.4.2 Correction et complétude de l'approche

Comme la résolution effectuée par le solveur de contraintes est correcte et complète, notre approche permet d'extraire l'ensemble correct et complet des motifs satisfaisant n'importe quelle contrainte n-aire. La figure 2 décrit l'évolution du nombre de paires de règles d'exception en fonction des seuils $minfr$ et δ_1 pour les jeux de données `Mushroom` et `Meningitis`. La figure 3 décrit l'évolution du nombre de conflits de classification en fonction des seuils $minfr$ et $minconf$ pour les jeux de données `Mushroom` et `Australian`. Nous avons aussi testé d'autres valeurs de ces paramètres ainsi que d'autres jeux de données. Mais, comme les résultats obtenus sont similaires, ils ne sont pas indiqués ici. Comme attendu, plus $minfr$ est petit, plus le nombre de règles d'exception est grand. Les résultats sont similaires quand δ_1 varie. Plus δ_1 est grand, plus le nombre de règles d'exception augmente (quand δ_1 augmente, la confiance décroît et il y a donc un plus grand nombre de règles générales).

6.4.3 Mise en valeur de motifs demandés par les utilisateurs

Les règles d'exception sont un cas particulier des règles rares (cf. la section 2.2). Même s'il existe quelques travaux permettant d'extraire les règles rares [21], il est impossible de distinguer les règles d'ex-

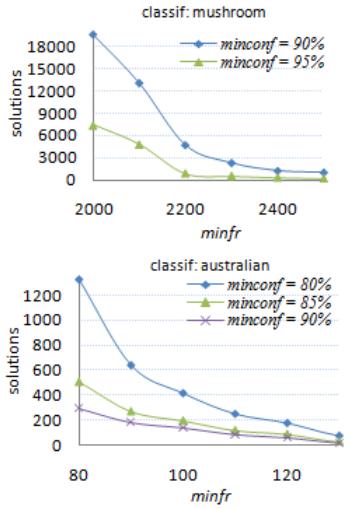


FIG. 3 – Evolution du nombre de conflits de classification

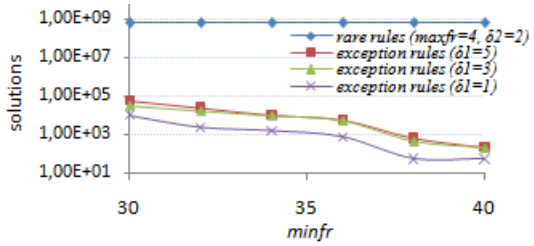


FIG. 4 – Nombre de règles d'exception vs nombre de règles rares (`Meningitis`)

ception à partir de l'ensemble des règles rares. C'est une limitation forte car la plupart des règles rares sont peu fiables, d'où l'intérêt des règles d'exception et de leur extraction. La figure 4 compare, sur le jeu de données `Meningitis`, le nombre de règles d'exception par rapport au nombre de règles rares en fonction du seuil de fréquence $minfr$ (le nombre de règles rares correspond à la ligne en haut de la figure 4). Savoir rechercher directement les règles d'exception permet de réduire de manière drastique (plusieurs ordres de magnitude) le nombre de motifs obtenus (noter que l'axe des ordonnées suit une échelle logarithmique).

Les règles inattendues sont également sources d'informations utiles. Un exemple d'une telle règle sur le jeu de données `Meningitis` est la règle dont la prémisse est composée d'un taux élevé de polynucléaires neutrophiles, de l'absence de signes neurologiques et dont la conclusion est une valeur normale pour le taux de polynucléaires immatures. Cette règle est inattendue par rapport à la croyance que des valeurs élevées de la numération des leucocytes et du taux de polynucléaires impliquent une méningite d'étiologie bactérienne. Les experts apprécient de disposer des contraintes n-aires pour la découverte de tels motifs.

6.4.4 Efficacité

Ces expérimentations permettent de quantifier les temps de calcul et le passage à l'échelle de notre approche. En pratique, les temps de calcul varient en fonction de la taille des jeux de données et de la dureté des contraintes[8]. C'est le cas des contraintes définies par des seuils de fréquence et de confiance élevés.

Pour `Meningitis` et `Australian`, l'ensemble de toutes les solutions est obtenu en quelques secondes (moins d'une seconde dans la plupart des cas). Pour `Mushroom`, les temps de calcul varient de quelques secondes pour les contraintes dures à environ 1 heure pour des seuils très faibles de fréquence et de confiance. La figure 5 décrit les temps de calcul obtenus sur `Mushroom` en fonction des seuils de fréquence et de confiance retenus. Plus une contrainte est dure, plus les temps de calcul sont petits. Les temps de calcul dépendent aussi de la taille du jeu de données traité : plus il est grand, plus le nombre de contraintes est élevé (cf. la section 7.2).

Bien évidemment, notre approche pourrait être utilisée pour extraire des motifs locaux. Nous obtenons dans ce cas les mêmes temps de calcul que [6] qui sont compétitifs vs les extracteurs de motifs locaux. Enfin, pour les règles d'exception, nous n'avons pas pu effectuer de comparaison avec la méthode ad hoc car les temps de calcul ne sont pas indiqués dans [20].

7 Comparaison des deux approches

Si l'approche hybride (cf. section 5.2) et l'approche PPC présentée dans cet article (cf. section 6) reposent sur la même modélisation, leurs mises en oeuvre diffèrent. L'approche hybride utilise un extracteur de motifs locaux afin de construire la représentation condensée sous forme d'intervalles de tous les motifs satisfaisant les contraintes locales. Cette représentation condensée permet de construire les domaines des variables du CSP ensembliste. L'approche PPC établit directement le lien entre l'ensemble des transactions et les motifs n-aires recherchés à l'aide de contraintes réifiées. Le CSP obtenu est directement résolu (sans l'aide d'un extracteur de motifs locaux).

[8] Une contrainte est dure si le ratio entre son nombre solutions et la cardinalité du produit cartésien des domaines de ses variables est faible.

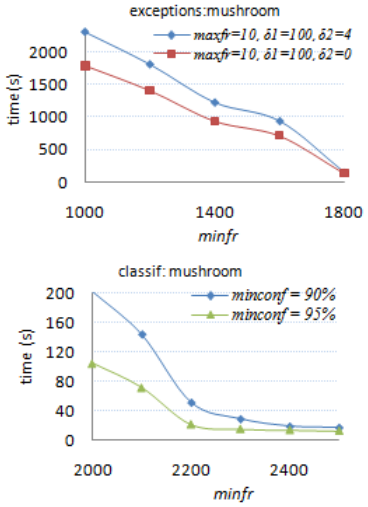


FIG. 5 – Temps d'exécution

7.1 Approche hybride : Pro/Cons

Avec l'approche hybride, la modélisation d'une contrainte n-aire peut être directement fournie au solveur de contraintes sans aucune re-formulation. Ainsi, un prototype a pu être rapidement développé en utilisant ECL^iPS^e. De plus, cette approche permet de bénéficier des avancées sur les extracteurs de motifs locaux.

Mais, les solveurs de CSP ensemblistes [8] ont du mal à gérer les unions d'intervalles. Afin d'établir la consistance aux bornes, l'union de deux intervalles ensemblistes est approximée par leur enveloppe convexe[9]. Afin de contourner ce problème, pour chaque variable X_i ayant pour représentation condensée $CR_i = \bigcup_p(f_p, I_p)$, la recherche s'effectue successivement sur chacun des I_p. Mais, si cette approche demeure correcte et complète, elle ne permet pas de profiter pleinement du filtrage car les retraits de valeurs non-viables se propagent uniquement sur les intervalles traités, et non pas sur l'intégralité des domaines.

Cette constatation pourrait paraître rédhibitoire, mais le nombre d'intervalles ensemblistes décroît rapidement lors de la prise en compte des contraintes locales. La table 4 décrit l'évolution du nombre d'intervalles ensemblistes constituant le domaine de la variable X_2 au fur et à mesure de la prise en compte des contraintes locales. (cf. l'exemple des règles d'exception à la section 5.2).

Une approche alternative serait d'utiliser des représentations condensées non exactes afin de réduire le nombre d'intervalles ensemblistes par domaine, comme

[9]L'enveloppe convexe de $[lb_1 .. ub_1]$ et $[lb_2 .. ub_2]$ est définie par $[lb_1 \cap lb_2 .. ub_1 \cup ub_2]$.

Contraintes locales	Nombre d'intervalles de D_{X_2}
-	3002
$I \in X_2$	1029
$I \in X_2 \wedge freq(X_2) >= 20$	52
$I \in X_2 \wedge freq(X_2) >= 25$	32

TAB. 4 – Nombre d'intervalles pour différentes contraintes locales (cas de D_{X_2})

par exemple une représentation condensée reposant sur les motifs fréquents maximaux [4]. Dans ce cas, le nombre d'intervalles par domaine sera beaucoup plus petit, mais en raison de l'approximation de l'union de deux intervalles par leur enveloppe convexe, il sera nécessaire de gérer les valeurs non viables qui en résultent.

7.2 Approche PPC : Pro/Cons

Tout d'abord, il est possible d'automatiser la reformulation des contraintes n-aires vers les contraintes primitives (cf. section 6.3). De plus, le propagateur des contraintes réifiées implanté en *Gecode* est très efficace. Mais, le nombre total de contraintes peut être très élevé pour des jeux de données de très grande taille. Considérons un jeu de données ayant n items, m transactions et une contrainte n-aire portant sur k motifs inconnus. Lier les transactions et les motifs n-aires nécessite $(k \times m)$ contraintes, chacune d'entre elles portant sur au plus $(n+1)$ variables (cf. l'équation 4 à la section 6.2). Ainsi, le nombre total de contraintes nécessaires à la modélisation de problèmes de très grande taille pourrait s'avérer prohibitif. En pratique, l'approche PPC permet de traiter des problèmes de grande taille (cf. section 6.4).

La différence fondamentale entre les deux approches réside dans leur façon de considérer l'ensemble des transactions. L'approche hybride utilise un extracteur de motifs locaux et le CSP résultant possède un très petit nombre de contraintes et de variables, mais ces variables ont des domaines de grande taille. À, l'inverse, l'approche PPC requiert un grand nombre de contraintes portant sur des variables de décision. Une voie médiane reste à trouver entre ces deux approches afin de pouvoir traiter des problèmes de très grande taille. L'extraction de motifs en utilisant la PPC est un domaine de recherche émergeant pour lequel il y a actuellement peu de travaux [6, 10, 11, 17].

8 Conclusions et perspectives

Nous avons proposé une approche PPC permettant de modéliser et d'extraire les motifs n-aires en fouille

de données. A notre connaissance, il s'agit de la première approche générique pour traiter ce problème. La modélisation décrite à la section 3 illustre la généricité et la flexibilité de notre approche. Les expérimentations menées (cf. section 6.4) montrent sa pertinence et sa faisabilité.

Les variables d'un CSP sont toutes quantifiées existentiellement Mais, certaines contraintes n-aires requièrent la quantification universelle pour être modélisées de manière concise et élégante, comme par exemple la contrainte *peak*. Un *peak* motif est un motif dont tous les voisins ont une valeur (selon une mesure) inférieure à un seuil. Pour cela, les QCSP [15, 22] nous semblent une piste prometteuse.

D'autre part, la découverte de connaissances dans les bases de données est un processus itératif et interactif guidé par l'utilisateur. Ce dernier ne devrait avoir qu'une vision de haut niveau du système de découverte de motifs en ayant à sa disposition un ensemble de contraintes (elles aussi de haut niveau) pour spécifier de façon déclarative les motifs désirés. Bien que nouvelle, l'approche PPC nous semble très prometteuse pour construire de tels systèmes.

Remerciements. Nous remercions le Dr P. François de l'Hôpital Central de Grenoble qui nous a fourni le jeu de données `Meningitis` et Arnaud Soulet pour les discussions fructueuses et MUSIC-DFS. Ce travail est partiellement financé par l'ANR (projet Bingo 2 ANR-07-MDCO-014).

Références

[1] J. Besson, C. Robardet, and J-F. Boulicaut. Mining a new fault-tolerant pattern type as an alternative to formal concept discovery. In *ICCS'06*, pages 144–157, Aalborg, Denmark, 2006. Springer-Verlag.

[2] F. Bonchi, F. Giannotti, C. Lucchese, S. Orlando, R. Perego, and R. Trasarti. A constraint-based querying system for exploratory pattern discovery. *Inf. Syst.*, 34(1) :3–27, 2009.

[3] B. Bringmann and A. Zimmermann. The chosen few : On identifying valuable patterns. In *12th IEEE Int. Conf. on Data Mining (ICDM-07)*, pages 63–72, 2007.

[4] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu. Mafia : A performance study of mining maximal frequent itemsets. In *FIMI*, volume 90 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2003.

[5] T. Calders, C. Rigotti, and J-F. Boulicaut. A survey on condensed representations for frequent sets. In *Constraint-Based Mining and Inductive Databases*, volume 3848 of *LNAI*, pages 64–80, 2005.

[6] L. De Raedt, T. Guns, and S. Nijssen. Constraint Programming for Itemset Mining. In *ACM SIGKDD Int. Conf. KDD'08*, Las Vegas, Nevada, USA, 2008.

[7] L. De Raedt and A. Zimmermann. Constraint-based pattern set mining. In *7th SIAM Int. Conf. on Data Mining*. SIAM, 2007.

[8] C. Gervet. Interval Propagation to Reason about Sets : Definition and Implementation of a Practical Language. *Constraints*, 1(3) :191–244, 1997.

[9] A. Giacometti, E. Khanjari Miyaneh, P. Marcel, and A. Soulet. A framework for pattern-based global models. In *IDEAL'09*, pages 433–440, 2009.

[10] M. Khiari, P. Boizumault, and B. Crémilleux. Allier CSPs et motifs locaux pour la découverte de motifs sous contraintes n-aires. In *EGC*, volume RNTI-E-19, pages 199–210. Cépaduès-Éditions, 2010.

[11] M. Khiari, P. Boizumault, and B. Crémilleux. Combining CSP and constraint-based mining for pattern discovery. In *ICCSA'10*, volume 6017 of *LNCS*, pages 432–447, 2010.

[12] J. Kléma, S. Blachon, A. Soulet, B. Crémilleux, and O. Gandrillon. Constraint-based knowledge discovery from sage data. *In Silico Biology*, 8(0014), 2008.

[13] A. Knobbe, B. Crémilleux, J. Fürnkranz, and M. Scholz. From local patterns to global models : The lego approach to data mining. In *Int. Workshop LeGo co-located with ECML/PKDD'08*, pages 1–16, Antwerp, Belgium, 2008.

[14] A. Knobbe and E. Ho. Pattern teams. In *proceedings of the 10th ECML/PKDD'06*, volume 4213 of *LNAI*, pages 577–584, Berlin, Germany, 2006.

[15] N. Mamoulis and K. Stergiou. Algorithms for quantified constraint satisfaction problems. In *proceedings of the 10th Int. Conf. CP'04*, 2004.

[16] R. T. Ng, V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *proceedings of ACM SIGMOD'98*, pages 13–24. ACM Press, 1998.

[17] S. Nijssen, T. Guns, and L. De Raedt. Correlated itemset mining in ROC space : a constraint programming approach. In *KDD'09*, pages 647–655, 2009.

[18] Balaji Padmanabhan and Alexander Tuzhilin. A belief-driven method for discovering unexpected patterns. In *KDD*, pages 94–100, 1998.

[19] A. Soulet, J. Klema, and B. Crémilleux. *Post-proceedings of the 5th Int. Workshop KDID'06*, volume 4747 of *LNCS*, pages 223–239. 2007.

[20] E. Suzuki. Undirected Discovery of Interesting Exception Rules. *Int. Journal of Pattern Recognition and Artificial Intelligence*, 16(8) :1065–1086, 2002.

[21] L. Szathmary, A. Napoli, and P. Valtchev. Towards Rare Itemset Mining. In *Proc. of the 19th IEEE ICTAI '07*, volume 1, Patras, Greece, 2007.

[22] G. Verger and C. Bessière. Guiding search in QCSP$^+$ with back-propagation. In *CP'08*, volume 5202 of *Lecture Notes in Computer Science*, pages 175–189, 2008.

[23] X. Yin and J. Han. CPAR : classification based on predictive association rules. In *proceedings of the 2003 SIAM Int. Conf. on Data Mining (SDM'03)*, 2003.

Intégration de l'optimisation par colonies de fourmis dans CP Optimizer

Madjid Khichane[1,2], Patrick Albert[1] et Christine Solnon[2]

[1] IBM
9, rue de Verdun, Gentilly 94253, France
[2] Université de Lyon
Université Lyon 1, LIRIS CNRS UMR5205, France
{madjid.khichane,palbert}@fr.ibm.com,christine.solnon@liris.cnrs.fr

Résumé

Nous présentons un algorithme générique pour la résolution de problèmes d'optimisation combinatoires. Cet algorithme est hybride et combine une approche heuristique, à savoir l'optimisation par colonies de fourmis (ACO), avec une approche complète de type "Branch &Propagate &Bound" (B&P&B), à savoir CP Optimizer (produit commercialisé par IBM Ilog). Le problème à résoudre est modélisé avec le langage de modélisation de CP Optimizer. Il est ensuite résolu par un algorithme générique qui fonctionne en deux phases séquentielles. La première phase sert à échantillonner l'espace des solutions. Pendant cette première phase, CP Optimizer est utilisé pour construire des solutions satisfaisant toutes les contraintes du problème, et le mécanisme d'apprentissage phéromonal d'ACO est utilisé pour identifier les zones prometteuses de l'espace de recherche par rapport à la fonction objectif à optimiser. Durant la deuxième phase, CP Optimizer effectue une recherche arborescente exhaustive (le "restart" [11]) guidée par les traces de phéromone accumulées lors de la première phase. Nous avons testé l'algorithme proposé sur des problèmes de sac à dos, d'affectation quadratique et d'ensemble stable maximum. Les premiers résultats sur ces trois problèmes montrent que ce nouvel algorithme améliore les performances de CP Optimizer.

1 Introduction

Les problèmes d'optimisation combinatoires (COP) sont d'une importance conséquente aussi bien dans le monde scientifique que dans le monde industriel. La plupart des COP sont *NP*-difficiles. Par conséquent, à moins que $P = NP$, ils ne peuvent pas être résolus de façon exacte en un temps polynomial. Parmi les COP qui sont *NP*-difficiles nous pouvons citer le problème du voyageur de commerce (TSP), le problème du sac-à-dos multidimensionnel (MKP) et le problème d'affectation quadratique (QAP). Pour résoudre ces COP, deux types d'approches complémentaires peuvent être utilisées, à savoir, les approches complètes et les approches heuristiques (ou métaheuristiques).

Les approches complètes explorent l'espace des combinaisons de façon exhaustive et procèdent généralement par *Branch & Bound* (B&B) : l'espace des combinaisons est structuré en un arbre qui est exploré de façon systématique, et des fonctions d'évaluation sont utilisées pour élaguer les sous-arbres dont l'évaluation de la fonction objectif est moins bonne que la meilleure solution trouvée. Cette approche permet de trouver la solution optimale en une limite de temps finie [9, 10]. Cependant, sa complexité est exponentielle dans le pire des cas. Quand le COP comporte des contraintes à satisfaire, en plus de la fonction objectif à optimiser, on peut raffiner l'approche B&B en ajoutant une phase de propagation des contraintes, ce que l'on peut résumer par *Branch & Propagate & Bound* (B&P&B) : à chaque nœud de l'arbre, les contraintes sont exploitées pour réduire l'espace de recherche en enlevant des domaines des variables les valeurs inconsistantes par rapport à une consistance locale. La programmation par contraintes (PPC) est généralement basée sur une approche B&P&B ; elle offre des langages de haut niveau pour la modélisation de COP en termes de contraintes et elle intègre toute une gamme d'algorithmes dédiés à la propagation de contraintes. Par conséquent, résoudre des COP avec la PPC ne nécessite pas beaucoup de travail de programmation. La PPC est géné-

ralement très efficace lorsque les contraintes sont suffisamment fortes pour que leur propagation réduise l'espace de recherche à une taille raisonnable. Toutefois, sur certaines instances, elle peut ne pas trouver de solutions de bonne qualité en un temps acceptable.

Les métaheuristiques contournent ce problème d'explosion combinatoire en explorant l'espace de recherche de façon incomplète : elles ignorent délibérément certaines zones qui leur semblent moins prometteuses. Un point clé réside dans leur capacité à équilibrer la diversification et l'intensification de la recherche : la diversification a pour objectif de garantir un bon échantillonnage de l'espace de recherche limitant le risque d'ignorer des zones contenant des solutions optimales ; l'intensification a pour objectif d'augmenter l'effort de recherche dans les zones prometteuses, aux alentours des bonnes solutions. Les métaheuristiques obtiennent généralement des solutions de très bonne qualité en des temps de calcul acceptables. Cependant, elles ne garantissent pas l'optimalité des solutions trouvées, car elles n'assurent pas le parcours entier de l'espace de recherche. Un autre inconvénient des métaheuristiques réside dans le fait que leur utilisation pour résoudre de nouveaux COP nécessite généralement un important travail de programmation. En particulier, la prise en compte des contraintes particulières à l'application demande de concevoir des structures de données appropriées pour évaluer rapidement les violations de contraintes avant de prendre une décision.

Beaucoup de métaheuristiques sont basées sur la recherche locale où l'espace de recherche est exploré par des perturbations élémentaires de combinaisons, e.g., le recuit simulé [5] ou la recherche Tabou [2]. Pour faciliter l'implémentation des algorithmes basés sur la recherche locale, Van Hentenryck et Michel ont conçu un langage de haut niveau basé sur les contraintes nommé Comet [3]. Comet introduit notamment la notion de variable incrémentale permettant au programmeur de concevoir de façon déclarative des structures de données qui permettent d'évaluer efficacement le voisinage de solutions.

Dans ce papier, nous proposons une approche générique pour la résolution de COP qui combine une approche complète de type B&P&B avec une métaheuristique. Nous avons implémenté et validé notre approche à l'aide de CP Optimizer, qui est un solveur générique, développé par IBM Ilog, qui utilise la technique de B&P&B. Notre objectif est de montrer qu'en combinant ce type d'approche avec une métaheuristique on améliore ses performances tout en conservant ses avantages principaux, à savoir la déclarativité et la complétude : dans notre nouvelle approche (de même qu'avec CP Optimizer), le COP à résoudre est défini de façon déclarative en termes de contraintes et il est résolu par un algorithme générique intégré au système ; cet algorithme est complet et garantit donc l'optimalité des solutions trouvées (dans la mesure où on ne borne pas les temps d'exécution).

La métaheuristique que nous utilisons pour améliorer les performances de CP Optimizer est l'optimisation par colonies de fourmis (ACO) [1]. Il s'agit d'une approche constructive qui explore l'espace de recherche par constructions itératives de nouvelles combinaisons. ACO a montré qu'elle est très efficace pour trouver rapidement de bonnes solutions, mais elle souffre des mêmes inconvénients que les autres métaheuristiques, c'est à dire, il n'y a aucune garantie quant à l'optimalité des solutions trouvées, et elle demande un important travail de programmation à chaque fois qu'elle est appliquée à un nouveau problème.

En combinant ACO avec CP Optimizer, nous prenons le meilleur des deux approches. Pour résoudre un problème dans cette nouvelle approche on procède de la même manière que lorsqu'on utilise CP Optimizer, c'est-à-dire on commence par modéliser le problème dans le langage de modélisation de CP Optimizer. Ensuite, on demande au solveur de rechercher la solution optimale. Cette recherche est décomposée en deux phases. Dans la première phase, ACO est utilisée pour échantillonner l'espace des solutions et identifier les régions de l'espace de recherche les plus prometteuses. Au cours de cette première phase, CP Optimizer est utilisé pour propager les contraintes et proposer des solutions réalisables à ACO. Dans la deuxième phase, CP Optimizer réalise une recherche basée sur le "restart" [11] et qui utilise l'approche B&P&B pour trouver la solution optimale. Au cours de cette deuxième phase, les traces de phéromones recueillies lors de la première phase sont utilisées par CP Optimizer comme heuristique d'ordre de valeurs. Cela lui permet de se concentrer en premier sur les régions les plus prometteuses de l'espace des solutions.

Rappelons que notre objectif principal n'est pas de rivaliser avec les algorithmes existants dans l'état de l'art qui sont dédiés à la résolution de problèmes spécifiques, mais de montrer qu'ACO peut améliorer le processus de recherche d'un algorithme générique utilisant la technique de B&P&B. Pour cela, nous avons choisi CP Optimizer comme référence, et nous l'avons utilisé comme une "boite noire" avec sa configuration par défaut correspondant à la stratégie Restart proposée par Refalo dans [11] : cette stratégie relance régulièrement de nouvelles recherches et utilise des *noGoods* pour apprendre de ses échecs passés ; elle utilise aussi les impacts comme heuristique de choix de variables et de valeurs.

La suite de cet article est organisée comme suit :

dans la section 2, nous rappelons quelques définitions sur les COP, la PPC et ACO. La section 3 décrit l'algorithme CPO-ACO. Dans la section 4, nous donnons quelques résultats expérimentaux sur les problèmes du sac-à-dos multidimensionnel, de l'affectation quadratique et de l'ensemble stable maximum. Nous concluons par une discussion sur certains travaux connexes et les perspectives de ce travail.

2 Contexte

2.1 Problème d'optimisation combinatoire (COP)

Un COP est défini par un quadruplet $P = (X, D, C, F)$ tel que

- $X = \{x_1, \ldots, x_n\}$ est un ensemble de n variables de décisions ;
- à chaque variable $x_i \in X$ est associé un domaine $D(x_i)$ qui est un ensemble fini d'entiers ;
- C est un ensemble de contraintes à satisfaire ;
- $F : D(x_1) \times \ldots \times D(x_n) \longrightarrow \mathbb{R}$ est la fonction objectif à optimiser.

Une affectation $\mathcal{A}$ est un ensemble de couples variable-valeur notés $< x_i, v_i >$ et correspondant à l'affectation de la valeur $v_i \in D(x_i)$ à la variable x_i. Une affectation $\mathcal{A}$ est complète si toutes les variables de X sont affectées dans $\mathcal{A}$, sinon, elle est partielle. Une affectation est inconsistante si elle viole une ou plusieurs contraintes de C, sinon elle est consistante. Une solution est une affectation consistante et complète. Une solution $\mathcal{A}$ de P est optimale si pour toute autre solution $\mathcal{A}^{'}$ de P, $F(\mathcal{A}) \leq F(\mathcal{A}^{'})$ si P est un problème de minimisation ou $F(\mathcal{A}) \geq F(\mathcal{A}^{'})$ si P est un problème de maximisation.

2.2 Résolution d'un COP par B&P&B

L'approche B&P&B résout un COP en construisant un arbre de recherche : à chaque noeud, elle choisit une variable non encore affectée x_i et elle choisit une valeur $v_j \in D(x_i)$ pour créer le point de choix $x_i = v_j \vee x_i \neq v_j$. Elle explore ensuite séparément chaque sous-arbre induit par chacune de ces deux alternatives. Cette recherche arborescente est combinée avec la propagation de contraintes et les techniques d'approximation de la fonction objectif.

La propagation de contraintes assure un certain niveau de consistance en filtrant les domaines des variables non affectées. Ce filtrage se traduit par la suppression des valeurs qui sont inconsistantes par rapport à une certaine consistance locale comme, par exemple, la consistance d'arc [7].

Les techniques d'encadrement des bornes de la fonction objectif permettent également de filtrer les domaines des variables [6]. En effet, une valeur d'une variable qui ne pourra pas figurer dans une solution qui soit meilleure que la meilleure solution trouvée jusqu'à présent sera supprimée.

Lorsque la propagation de contraintes ou l'approximation des bornes de la fonction objectif détecte un échec (un domaine d'une variable est devenu vide), cette procédure fait marche arrière pour exploiter un autre noeud de l'arbre de recherche. Cette méthode est efficace et générique. En revanche, elle ne parvient pas à résoudre certains COP pour lesquels la propagation de contraintes et/ou les techniques d'approximation ne sont pas capables de réduire suffisamment l'espace de recherche.

2.3 Heuristique d'ordre basée sur les impacts

A chaque étape d'une résolution B&P&B, il s'agit de choisir la prochaine variable à affecter ainsi qu'une valeur appartenant à son domaine. Ces choix ont une forte influence sur la taille de l'arbre de recherche et on utilise généralement des heuristiques d'ordre pour guider ces choix.

Refalo [11] a proposé des heuristiques d'ordre basées sur la notion d'impact. L'impact de l'affectation d'une valeur à une variable est défini comme étant la proportion de l'espace de recherche supprimée par la propagation de contraintes causée par cette affectation. L'impact d'une valeur est défini comme étant la moyenne de ses impacts observés et l'impact d'une variable, comme étant la moyenne de l'impact des valeurs restantes dans son domaine. Ces impacts sont utilisés pour définir des heuristiques d'ordre : à chaque nœud de l'arbre de recherche, la prochaine variable à affecter est celle ayant le plus fort impact, et la valeur affectée est celle ayant le plus petit impact.

Ces heuristiques d'ordre basées sur les impacts sont celles utilisées par défaut par CP Optimizer.

2.4 Optimisation par Colonies de Fourmis (*ACO*)

ACO est une métaheuristique constructive qui explore l'espace de recherche en construisant itérativement de nouvelles combinaisons. Cette métaheuristique a été appliquée à la résolution de nombreux COP, et il existe de nombreuses déclinaisons de ce principe très général. Nous invitons le lecteur intéressé à se référer à [1] pour plus de détails. Nous décrivons ici l'algorithme du MAX-MIN Ant System (MMAS) proposé par Stützle et Hoos dans [13], qui a montré de très bonnes performances sur de nombreux problèmes, et nous le présentons dans notre contexte de résolution d'un COP modélisé par un quadruplet (X, D, C, F).

L'idée de base est de construire des affectations complètes selon un principe glouton aléatoire, et de progressivement biaiser le modèle probabiliste utilisé pour

construire une affectation en fonction de l'expérience passée. On utilise pour cela un mécanisme simple d'apprentisage par renforcement de traces de phéromone.

Structure phéromonale. La phéromone est utilisée dans un algorithme ACO pour identifier progressivement les composants de solution les plus prometteurs, et biaiser en conséquence le modèle probabiliste utilisé pour construire des combinaisons. De façon générale, on associe à chaque composant de solution c une trace de phéromone τ_c. Cette trace de phéromone représente l'expérience passée de la colonie concernant l'utilisation de c lors de la construction d'une combinaison : plus c a participé à de bonnes combinaisons, plus τ_c a une valeur importante, et plus la probabilité de choisir c lors de la construction d'une combinaison est importante.

Le choix de la structure phéromonale (i.e., des composants sur lesquels on dépose de la phéromone) dépend du problème à résoudre. Si on considère un COP définit par un quadruplet (X, D, C, F), on peut associer une trace de phéromone $\tau(x_i, v_j)$ à chaque couple variable-valeur $< x_i \in X, v_j \in D(x_i) >$. Intuitivement, la quantité de phéromone sur $< x_i, v_j >$ représente l'intéret appris d'affecter la valeur v_j à la variable x_i. Cette structure phéromonale a démontré son efficacité à résoudre certains COP, par exemple, le QAP [13], les CSP [12] ou le problème d'ordonnancement de voitures [4].

Les traces de phéromone sont utilisées pour intensifier la recherche autour des meilleures affectations trouvées. Afin d'équilibrer l'intensification et la diversification, ces traces de phéromone sont bornées entre deux paramètres τ_{min} et τ_{max} de sorte que les différences entre deux traces soient limitées. De plus, les traces de phéromone sont initialisées à τ_{max} au début de l'exécution d'un algorithme ACO.

Construction des affectations. A chaque cycle de l'algorithme, chaque fourmi construit une affectation complète selon un principe glouton aléatoire : partant d'une affectation vide, on choisit à chaque itération une variable non affectée et une valeur à affecter à cette variable. Ce processus est répété jusqu'à ce que toutes les variables soient affectées. Le choix de la prochaine variable à affecter se fait par rapport à une heuristique donnée. Par exemple, pour le problème d'ordonnancement de voitures, les variables sont affectées par ordre croissant d'indice dans la séquence de voitures. Pour un CSP, on peut choisir la variable ayant le plus petit domaine.

Une fois qu'une fourmi a sélectionné une variable x_i, elle choisit une valeur $v_j \in D(x_i)$ en fonction de la probabilité

$$p(x_i, v_j) = \frac{[\tau(x_i, v_j)]^\alpha \cdot [\eta(x_i, v_j)]^\beta}{\sum_{v_k \in D(x_i)} [\tau(x_i, v_k)]^\alpha \cdot [\eta(x_i, v_k)]^\beta} \quad (1)$$

où $\eta(x_i, v_j)$ est le facteur heuristique associé à l'affectation de v_j à x_i. La définition de ce facteur heuristique dépend du problème considéré. Généralement, ce facteur heuristique évalue l'impact de l'affectation de v_j à x_i sur la fonction objectif. α et β sont deux paramètres qui déterminent l'influence de la phéromone et de l'heuristique dans le choix de la valeur.

Les contraintes sont généralement gérées différemment selon que le COP est faiblement ou fortement contraint. Quand le COP est faiblement contraint, il est très facile de construire une solution satisfaisant toutes les contraintes, et la difficulté réside dans le fait que l'on cherche la solution optimisant la fonction objectif. C'est le cas par exemple du QAP ou du MKP. Dans ce cas, les contraintes sont propagées après chaque affectation de sorte que les domaines ne contiennent que des valeurs consistantes, et les fourmis ne construisent que des affectations consistantes. Par exemple, pour le QAP, la contrainte imposant de ne pas implanter une même usine à plusieurs endroits est propagée en enlevant les emplacements déjà utilisés des domaines ; pour le MKP, les contraintes de capacité sont propagées en affectant à 0 les variables associées à des objets ne pouvant être sélectionnés sans violer une contrainte de capacité.

Quand le COP est fortement contraint, la construction d'une solution satisfaisant toutes les contraintes (indépendamment de la fonction objectif) peut devenir un problème difficile de sorte que les fourmis peuvent ne pas y arriver. Pour éviter cela, les contraintes peuvent être intégrées dans la fonction objectif : l'objectif est alors de trouver l'affectation satisfaisant au mieux les contraintes [12].

Mise-à-jour de la phéromone. Une fois que chaque fourmi a construit une solution, les traces de phéromone sont mises-à-jour. Dans un premier temps, toutes les traces de phéromone sont diminuées en les multipliant par $(1-\rho)$, où $\rho \in [0; 1]$ est le taux d'évaporation. Ce processus d'évaporation permet aux fourmis d'oublier progressivement les constructions anciennes pour ainsi mettre l'accent sur les constructions les plus récentes.

Dans un deuxième temps, on récompense les meilleurs affectations construites lors du dernier cycle et/ou la meilleure affectation construite depuis le début de l'exécution. Cette récompense est traduite par le dépôt d'une quantité de phéromone sur les composants de ces affectations. Lorsque les traces

de phéromone sont associées à des couples variable-valeur, la phéromone est déposée sur les couples <variable/valeur> de l'affectation à récompenser. La quantité de phéromone déposée est généralement proportionnelle à la qualité de l'affectation à récompenser. Cette quantité est souvent normalisée entre 0 et 1 et elle est généralement définie comme un ratio entre la qualité de l'affectation à récompenser et la valeur optimale (si elle est connue) ou la qualité de la meilleure affectation trouvée jusqu'alors.

3 Description de $CPO - ACO$

ACO s'est révélée très efficace pour trouver rapidement de bonnes solutions à de nombreux COP. Cependant, la conception d'un algorithme ACO pour résoudre un nouveau COP peut exiger un grand effort de programmation. En effet, si les procédures de gestion et d'exploitation de la phéromone sont très semblables d'un COP à l'autre, la résolution d'un nouveau COP nécessite d'écrire des procédures de propagation et de contrôle des contraintes dépendantes du problème considéré. Ce constat est à l'origine de notre travail : en combinant ACO avec la PPC, on peut réutiliser les nombreuses procédures disponibles pour la gestion des contraintes.

L'algorithme proposé $CPO - ACO$ fonctionne en deux phases :

- Pendant la première phase, ACO utilise CP Optimizer pour construire des solutions et le mécanisme d'apprentissage phéromonal est utilisé pour intensifier progressivement la recherche autour des meilleures solutions trouvées.
- Pendant la deuxième phase, CP Optimizer est utilisé pour rechercher la solution optimale, et les traces de phéromone recueillies au cours de la première phase sont utilisées pour guider CP Optimizer dans sa recherche.

3.1 Première phase de $CPO - ACO$

L'algorithme 1 décrit la première phase de $CPO - ACO$, les grandes lignes de cet algorithme sont décrites ci-après.

Construction d'une affectation

Lors de chaque cycle (lignes 3-14), chaque fourmi demande à CP Optimizer de construire une solution (ligne 5). Notez que pendant cette première phase, nous ne demandons pas à CP Optimizer d'optimiser la fonction objectif, mais simplement de trouver des solutions satisfaisant toutes les contraintes. Chaque nouvel appel à CP Optimizer correspond à une nouvelle recherche (restart) et CP Optimizer construit une solution selon le principe B&P&B : il propage les contraintes après chaque affectation et si un échec est détecté, il fait marche arrière pour prendre une autre décision jusqu'à ce qu'il trouve une solution. Lors de cette première phase, CP Optimizer est utilisé avec ses paramètres par défaut à l'exception de l'heuristique de choix de valeur qui lui est transmise en paramètre. Cette heuristique est basée sur ACO et définit la probabilité de choisir la valeur v_j pour une variable x_i par

$$p(v_j) = \frac{[\tau(x_i, v_j)]^\alpha \cdot [1/impact(v_j)]^\beta}{\sum_{v_k \in D(x_i)} [\tau(x_i, v_k)]^\alpha \cdot [1/impact(v_k)]^\beta}$$

où $impact(v_j)$ est l'impact observé de la valeur v_j [11]. Comme toutes les traces de phéromone sont initialisées à la même valeur (i.e., τ_{max}), au cours des premiers cycles, les impacts sont plus déterminants que les traces de phéromone dans le choix des valeurs. Toutefois, à la fin de chaque cycle les traces de phéromone sont mises à jour, et les traces de phéromones influencent de plus en plus le choix des valeurs.

Notons que CPO-ACO est plutôt destiné à résoudre des COP pour lesquels la difficulté n'est pas de construire des solutions, mais de trouver la solution qui optimise la fonction objectif. Pour ces problèmes, CP Optimizer est capable de construire très rapidement une solution (en faisant très peu de retours arrières) de sorte que l'on peut rapidement collecter un nombre significatif de solutions qui peuvent alors être utilisées par ACO pour biaiser la recherche. CPO-ACO pourrait être utilisé pour résoudre des COP plus contraints mais, dans ce cas, CP Optimizer peut avoir besoin de plus de temps pour calculer une solution satisfaisant toutes les contraintes ce qui fait que l'apprentissage phéromonal sera basé sur trop peu de solutions pour être intéressant.

Mise-à-jour de la phéromone

Une fois que chaque fourmi a construit une affectation, les traces de phéromone sont évaporées en les multipliant par $(1 - \rho)$ où $\rho \in [0; 1]$ est le taux d'évaporation des phéromones (lignes 6-7).

À la fin de chaque cycle, les meilleures solutions (par rapport à la fonction l'objectif) sont récompensées dans le but d'intensifier la recherche autour d'elles. Les meilleures solutions du cycle sont systématiquement récompensées (lignes 9-11). En revanche, la meilleure solution construite depuis le début de la recherche est récompensée seulement si elle est meilleure que les meilleures solutions du dernier cycle (lignes 12-13).

Une solution $\mathcal{A}$ est récompensée en augmentant la quantité de phéromone sur chaque couple $< x_i, v_j >$ de $\mathcal{A}$, ainsi, la probabilité d'affecter x_i à v_j lors des futures

Entrées : $P = (X, D, C, F)$ et les paramètres $\{t_{max1}, d_{min}, it_{max}, \alpha, \beta, \rho, \tau_{min}, \tau_{max}, nbAnts\}$
Sorties : une solution $\mathcal{A}_{best}$ et une matrice de phéromone $\tau : X \times D \rightarrow [\tau_{min}; \tau_{max}]$

début
  pour chaque $x_i \in X$ **et chaque** $v_j \in D(x_i)$ **faire** $\tau(x_i, v_j) \leftarrow \tau_{max}$
  répéter
    `/* Construction des solutions */`
    pour chaque $k \in \{1, \ldots, nbAnts\}$ **faire**
      Construire une solution $\mathcal{A}_k$ en utilisant CP Optimizer
    `/* Evaporation de la phéromone */`
    pour chaque $x_i \in X$ **et chaque** $v_i \in D(x_i)$ **faire**
      $\tau(x_i, v_i) \leftarrow max(\tau_{min}, (1 - \rho) \cdot \tau(x_i, v_i))$
    `/* Dépôt de phéromone */`
    Soit $\mathcal{A}_{best}$ la meilleure solution construite jusqu'à présent (y compris le dernier cycle)
    pour chaque $k \in \{1, \ldots, nbAnts\}$ **faire**
      si $\forall l \in \{1, \ldots, nbAnts\}, \mathcal{A}_k$ *est au moins aussi bon que* $\mathcal{A}_l$ **alors**
        pour chaque $< x_i, v_i > \in \mathcal{A}_k$ **faire** $\tau(x_i, v_i) \leftarrow min(\tau_{max}, \tau(x_i, v_i) + \frac{1}{1+|F(\mathcal{A}_k) - F(\mathcal{A}_{best})|})$
    si $\mathcal{A}_{best}$ *est strictement meilleur que toutes les solutions* $\{\mathcal{A}_1, ..., \mathcal{A}_{nbAnts}\}$ **alors**
      pour chaque $< x_i, v_i > \in \mathcal{A}_{best}$ **faire** $\tau(x_i, v_i) \leftarrow min(\tau_{max}, \tau(x_i, v_i) + 1)$
  jusqu'à *temps utilisé* $\geq t_{max1}$ **ou** *nombre de cycle sans amélioration de* $\mathcal{A}_{best} \geq it_{max}$ **ou** *distance moyenne de* $\{\mathcal{A}_1, ..., \mathcal{A}_{nbAnts}\} \leq d_{min}$;
  retourne $\mathcal{A}_{best}$ **et** τ
fin

Algorithme 1 – Phase 1 de $CPO - ACO$

affectations est augmentée. La quantité de phéromone ajoutée est inversement proportionnel à l'écart entre $F(\mathcal{A})$ et $F(\mathcal{A}_{best})$.

Conditions d'arrêt

La première phase s'arrête dans l'un des cas suivants : si le temps CPU t_{max1} a été atteint ; si la meilleure solution $\mathcal{A}_{best}$ n'a pas été améliorée depuis it_{max} itérations ; ou bien si la distance moyenne entre les affectations calculées pendant le dernier cycle est plus petite que d_{min}, ce qui indique que les traces de phéromone ont permis à la recherche de converger. Nous définissons la distance entre deux affectations comme étant le taux de couples variable-valeur sur lesquels les deux affectations sont différentes : la distance entre $\mathcal{A}_1$ et $\mathcal{A}_2$ est $\frac{|X| - |\mathcal{A}_1 \cap \mathcal{A}_2|}{|X|}$.

3.2 Deuxième phase de $CPO - ACO$

À la fin de la première phase, la meilleure solution construite $\mathcal{A}_{best}$ et la structure phéromonale τ sont transmises à la deuxième phase. Le coût de $\mathcal{A}_{best}$ est utilisé pour borner la fonction objectif. Ensuite, CP Optimizer est lancé pour chercher la solution optimale : dans cette deuxième phase, chaque fois que CP Optimizer trouve une meilleure solution, il borne la fonction objectif avec son coût et il fait marche arrière pour trouver de meilleures solutions ou prouver l'optimalité de la dernière solution trouvée.

Comme dans la première phase, CP Optimizer est utilisé comme une boîte noire avec ses paramètres de recherche par défaut et utilise la structure phéromonale τ et les impacts comme heuristiques d'ordre de variables. Cependant, cette fois, il ne calcule pas les probabilités de sélection des valeurs pour une variable x_i, mais il choisit la valeur qui maximise la formule

$$[\tau(x_i, v_i)]^{\alpha} \cdot [1/impact(v_i)]^{\beta}.$$

Nous avons comparé expérimentalement différentes variantes de CPO-ACO :

- Dans le but de montrer l'intérêt d'utiliser un mécanisme d'apprentissage phéromonal lors de la première phase, nous avons testé la variante où, pendant la première phase, la recherche est effectuée sans utiliser de phéromone de sorte que l'heuristique de choix de valeurs est définie par les impacts uniquement. Cette variante donne des résultats significativement moins bons.
- Pour évaluer l'intérêt de l'utilisation des traces de phéromone lors de la deuxième phase, nous

avons testé la variante où, à la fin de la première phase, nous ne retenons que $\mathcal{A}_{best}$ qui est utilisé pour borner la fonction objectif au début de la deuxième phase. La deuxième phase est alors constituée par la recherche par défaut de CP Optimizer (qui utilise uniquement les impacts comme heuristique de choix de valeurs). Cette variante obtient également des résultats beaucoup moins bons que lorsque la structure de phéromone est utilisée lors de cette deuxième phase.
– Enfin, nous avons testé la variante où, au cours de la deuxième phase, le choix de valeur pour une variable donnée est fait en utilisant la règle de transition probabiliste de ACO (voir la section 3.1) comme dans la première phase au lieu de sélectionner la valeur qui maximise la formule donnée en section 3.2. Cette variante obtient, sur la plupart des tests effectués, des résultats qui ne sont pas significativement différents de ceux qu'on donne dans la section suivante.

4 Evaluation expérimentale de CPO-ACO

4.1 Les problèmes considérés

Nous avons évalué l'algorithme CPO-ACO sur trois COP bien connus, à savoir : le problème de sac-à-dos multidimensionnel (MKP) ; le problème d'affectation quadratique (QAP) et le problème de l'ensemble stable maximum (MIS).

Le problème de sac à dos multidimensionnel (MKP) consiste à sélectionner un sous-ensemble d'objets qui satisfait un ensemble de contraintes linéaires de capacité et qui maximise la somme des profits des objets sélectionnés. Le modèle PPC que nous avons utilisé est le suivant :

– X associe une variable x_i à chaque objet i ;
– $\forall x_i \in X$, $D(x_i) = \{0, 1\}$ de sorte que $x_i = 0$ si i n'est pas sélectionné, et 1 sinon ;
– C est un ensemble de m contraintes de capacité tel que chaque contrainte $C_j \in C$ est de la forme $\sum_{i=1}^{n} c_{ij} \cdot x_i \leq r_j$ où c_{ij} est la quantité de ressource j requise par l'objet i et r_j est la quantité disponible de la ressource j ;
– la fonction objectif à maximiser est $F = \sum_{i=1}^{n} u_i \cdot x_i$ où u_i est le profit de l'objet i.

Nous avons considéré les instances académic avec 100 objets disponibles sur http ://people.brunel.ac.uk/˜mastjjb/jeb/orlib/mknapinfo.html. Nous avons considéré les 20 premières instances avec 5 contraintes de ressources (de 5-100-00 à 5-100-19), les 20 premières instances avec 10 contraintes de ressources (de 10-100-00 à 10-100-19) et les 20 premières instances avec 30 contraintes de ressources (de 30-100-00 à 30-100-19).

Le problème d'affectation quadratique (QAP) consiste à déterminer l'emplacement d'usines de façon à minimiser les distances entre les usines et les flux de produits entre les usines. Le modèle PPC que nous avons utilisé est le suivant :

– X associe une variable x_i à chaque usine i ;
– $\forall x_i \in X$, $D(x_i) = \{1, \ldots, n\}$ tel que $x_i = j$ si l'usine i est construite à l'emplacement j ;
– C contient uniquement la contrainte all-different sur toutes les variables, assurant ainsi que chaque usine est construite à un seul emplacement.
– la fonction objectif à optimiser est $F = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{x_i x_j} b_{ij}$ où $a_{x_i x_j}$ est la distance entre les emplacements des usines i et j, et b_{ij} est le flux entre les usines i et j.

Pour nos tests, nous avons utilisé les instances académiques de la QAPLIB qui sont disponibles sur www.opt.math.tu-graz.ac.at/qaplib/inst.html.

Le problème de l'ensemble stable maximum (MIS) consiste à sélectionner le plus grand sous-ensemble de sommets d'un graphe de sorte que deux sommets sélectionnés dans ce sous-ensemble ne sont pas reliés par une arête (ce problème est équivalent à la recherche d'une clique maximum dans le graphe inverse). Le modèle PPC que nous avons utilisé est le suivant :

– X associe une variable x_i à chaque sommet i ;
– $\forall x_i \in X$, $D(x_i) = \{0, 1\}$ de sorte que $x_i = 0$ si le sommet i n'est pas sélectionné et $x_i = 1$ sinon ;
– C associe une contrainte binaire c_{ij} à chaque arête (i, j) du graphe. Cette contrainte assure que les sommets i et j ne seront pas sélectionnés dans le même ensemble, i.e., $c_{ij} = (x_i + x_j < 2)$.
– la fonction objectif à maximiser est $F = \sum_{i=1}^{n} x_i$.

Les instances de MIS que nous avons utilisées pour nos tests sont toutes disponibles sur http ://www.nlsde.buaa.edu.cn/∼kexu/benchmarks/graph-benchmarks.htm.

4.2 Conditions expérimentales

Pour chaque problème, le modèle PPC a été implémenté en utilisant le langage de modélisation de CP Optimizer.

Nous comparons CPO-ACO avec CP Optimizer (noté dans la suite CPO). Dans les deux cas, nous avons utilisé la version V2.3 de CPO avec ses paramètres de recherche par défaut. Toutefois, pour CPO-ACO, les fonctions de choix de valeurs sont transmises en paramètres à CPO comme décrit dans la section

précédente. Pour CPO, les impacts sont utilisés comme heuristique de choix de variable et de valeur [11].

Pour toutes les expériences, le temps CPU total a été limité à 300 secondes sur une machine Pentium-4 2.2 Gz. Pour CPO-ACO, cette durée totale est partagée entre les deux phases : la durée de la phase 1 est au plus égale à $t_{max1} = 25\%$ du temps total. Nous avons par ailleurs fixé d_{min} à 0.05 (de sorte que la phase 1 est arrêtée dès que la distance moyenne entre les solutions du même cycle est inférieure à 5%) et it_{max} à 500 (de sorte que la phase 1 est arrêtée si $\mathcal{A}_{best}$ n'a pas été améliorée depuis 500 cycles). Le nombre de fourmis est $nbAnts = 20$; le poids du facteur phéromonal est $\alpha = 1$ et le poids du facteur heuristique est $\beta = 2$. Les traces de phéromone sont bornées entre $\tau_{min} = 0.01$ et $\tau_{max} = 1$.

Notons que nous n'avons pas utilisé le même taux d'évaporation de la phéromone pour toutes les expérimentations. En effet, pour le MKP et le MIS les variables sont binaires, ce qui fait qu'à chaque cycle une des deux valeurs possibles (0 ou 1) est récompensée, et ACO converge assez rapidement. Pour le QAP, tous les domaines sont de taille n (où n est égal au nombre de variables), donc à chaque cycle, seulement une valeur sur les n valeurs possibles est récompensée. Dans ce cas, nous avons délibérément augmenté le taux d'évaporation afin d'accélérer la convergence de ACO lors de la première phase. Par conséquent, $\rho = 0.01$ pour le MKP et le MIS et $\rho = 0.1$ pour le QAP.

Pour les deux algorithmes, CPO et CPO-ACO, nous avons effectué 30 exécutions par instance de chaque problème.

4.3 Résultats expérimentaux

Le tableau 1 donne les résultats expérimentaux obtenus par CPO et CPO-ACO sur le MKP, le QAP et le MIS. Pour chaque classe d'instances et pour chaque algorithme, ce tableau donne l'écart (en pourcentage) par rapport à la meilleure solution connue. Notons d'abord que CPO et CPO-ACO n'atteignent (presque) jamais la solution optimale connue : les meilleures solutions connues sont en effet, obtenues avec des approches dédiées. CPO et CPO-ACO sont des approches complètement génériques qui ne visent pas à concurrencer les approches dédiées. Aussi, nous avons choisi une limite raisonnable de temps CPU (300 secondes) afin de nous permettre d'effectuer un nombre suffisant de tests, pour pouvoir utiliser les tests statistiques. Avec cette limite de temps, CPO-ACO obtient des résultats compétitifs avec des approches dédiées sur le MKP (moins de 1% d'écart par rapport aux meilleures solutions connues), mais il est assez loin des meilleures solutions connues sur de nombreuses instances du QAP et MIS.

Comparons maintenant CPO avec CPO-ACO. Le tableau 1 nous montre que l'utilisation d'ACO pour guider CPO améliore le processus de recherche sur toutes les classes d'instances sauf deux. Toutefois, cette amélioration est plus importante pour le MKP que pour les deux autres problèmes. Comme les deux approches ont obtenu des résultats assez proches sur certaines classes d'instances, nous avons fait des tests statistiques (t-test avec un niveau de confiance de 95%) pour déterminer si les résultats sont significativement différents ou non. Pour chaque classe, nous avons indiqué le pourcentage d'instances pour lesquelles une approche a obtenu des résultats significativement meilleurs que l'autre (colonne $>_{t-test}$ du tableau 1). Pour le MKP, CPO-ACO est nettement meilleur que CPO sur 57 instances, alors qu'il n'est pas significativement différent pour 3 instances. Pour le QAP, CPO-ACO est nettement meilleur que CPO sur un grand nombre d'instances. Toutefois, CPO est meilleur que CPO-ACO sur une instance de la classe tai* du QAP. Pour le MIS, CPO-ACO est nettement meilleur que CPO sur 35% d'instances, mais il n'est pas significativement différent sur toutes les autres.

5 Conclusion

Nous avons proposé une approche générique pour résoudre les COP définis par un ensemble de contraintes et une fonction objectif. Cette approche générique combine une approche B&P&B avec ACO. L'idée principale de cette combinaison est de bénéficier de l'efficacité (i) de ACO pour explorer l'espace de recherche et identifier les zones prometteuses (ii) de CP Optimizer pour exploiter fortement le voisinage des meilleures solutions trouvées par ACO. Cette combinaison nous permet d'atteindre un bon équilibre entre la diversification et l'intensification de la recherche : la diversification est principalement assurée au cours de la première phase par ACO et l'intensification est assurée par CP Optimizer au cours de la deuxième phase. Nous avons montré par des expériences sur trois COP différents que cette approche hybride améliore les performances de CP Optimizer.

Il est à noter que grâce à la nature modulaire de CP Optimizer qui sépare clairement la partie modélisation du problème de la partie résolution, la combinaison de ACO et CP Optimizer a été faite de manière naturelle. Par conséquent, le programme CPO-ACO utilisé était exactement le même pour les expériences sur les différents problèmes utilisés dans cet article.

D'autres travaux ont également proposé des approches intégrant ACO avec des approches de type B&P&B. En particulier, B. Meyer a proposé, dans [8], deux algorithmes hybrides où ACO a été couplée avec

Résultats pour le MKP

			CPO				$CPO-ACO$			
Name	# I	# X	avg	(sd)	$>_{avg}$	$>_{t-test}$	avg	(sd)	$>_{avg}$	$>_{t-test}$
5.100-*	20	100	1.20	(0.30)	0%	0%	**0.46**	(0.23)	**100%**	**100%**
10.100-*	20	100	1.53	(0.31)	0%	0%	**0.83**	(0.34)	**100%**	**100%**
30.100-*	20	100	1.24	(0.06)	5%	0%	**0.86**	(0.08)	**95%**	**85%**

Résultats pour le QAP

			CPO				$CPO-ACO$			
Name	# I	# X	avg	(sd)	$>_{avg}$	$>_{t-test}$	avg	(sd)	$>_{avg}$	$>_{t-test}$
bur*	7	26	1.17	(0.43)	0%	0%	**0.88**	(0.43)	**100%**	**57 %**
chr*	11	19	12.11	(6.81)	45%	9%	**10.99**	(6.01)	**55 %**	**45 %**
had*	5	16	1.07	(0.89)	0%	0%	**0.54**	(1.14)	**100%**	**60 %**
kra*	2	30	17.46	(3.00)	0%	0%	**14.99**	(2.79)	**100%**	**100%**
lipa*	6	37	22.11	(0.82)	0%	0%	**20.87**	(0.75)	**100%**	**100%**
nug*	15	20	8.03	(1.59)	7%	0%	**5.95**	(1.44)	**93 %**	**80 %**
rou*	3	16	5.33	(1.15)	33%	0%	**3.98**	(1.00)	**67 %**	**67 %**
scr*	3	16	**4.60**	(2.4)	33%	0%	5.12	(2.60)	**67 %**	0 %
tai*	4	16	6.06	(1.35)	25%	25%	**4.84**	(1.25)	**75 %**	**50 %**

Résultats pour le MIS

			CPO				$CPO-ACO$			
Name	# I	# X	avg	(sd)	$>_{avg}$	$>_{t-test}$	avg	(sd)	$>_{avg}$	$>_{t-test}$
frb-30-15-*	5	450	9.83	(1.86)	0%	0%	**9.46**	(2.00)	**80%**	**20%**
frb-35-17-*	5	595	**11.62**	(2.05)	**60%**	0%	11.82	(2.31)	40%	0%
frb-40-19-*	5	760	13.47	(1.92)	20%	0%	**12.85**	(2.22)	**80%**	**20%**
frb-45-21-*	5	945	15.40	(2.43)	0%	0%	**14.35**	(1.82)	**100%**	**80%**
frb-50-23-*	5	1150	16.24	(2.32)	20%	0%	**15.84**	(2.00)	**80%**	**20%**
frb-53-24-*	5	1272	18.15	(2.55)	0%	0%	**16.86**	(1.84)	**100%**	**80%**
frb-56-25-*	5	1400	17.85	(2.37)	20%	0%	**16.89**	(1.08)	**80%**	**40%**
frb-59-26-*	5	1534	18.40	(2.44)	40%	0%	**18.37**	(2.16)	**60%**	**20%**

TABLE 1 – Comparaison de CPO et CPO-ACO sur le MKP, le QAP et le MIS. Chaque ligne donne successivement : le nom de la classe d'instances, le nombre d'instances dans la classe ($\#I$), le nombre moyen de variables dans ces instances ($\#X$), les résultats obtenus par CPO (resp. CPO-ACO), à savoir, le pourcentage d'écart par rapport à la meilleure solution connue (moyenne (avg) et écart type(sd)), le pourcentage d'instances pour lesquelles CPO (resp. CPO-ACO) a obtenu de meilleurs résultats en moyenne ($>_{avg}$), et le pourcentage d'instances pour lesquelles CPO (resp. CPO-ACO) est donné meilleur par le test statistique t-test.

la PPC. Il a proposé un premier couplage faible où les deux approches fonctionnent en parallèle en échangeant seulement les solutions et les bornes de la fonction objectif. Il a proposé un deuxième couplage plus fort, où la propagation de contraintes a été incorporée dans ACO pour permettre à une fourmi de revenir en arrière (backtrack) lorsqu'une affectation d'une valeur à une variable donnée échoue. Toutefois, la procédure de retour en arrière a été limitée au niveau de la dernière variable choisie. Cela signifie que, si toutes les valeurs possibles de la dernière variable choisie ont été essayées sans succès, la recherche d'une fourmi se termine par un échec (pas de solution construite). Les résultats de ce travail montrent sur le problème d'ordonnancement de tâches que le couplage plus fort est meilleur. Notons que le couplage fort proposé n'est pas une approche complète.

Nous avons également proposé dans [4] une approche hybride, notée Ant-CP, qui combine ACO avec la PPC pour résoudre les problèmes de satisfaction de contraintes (sans fonction à optimiser). Comme CPO-ACO, Ant-CP utilise le langage de modélisation de la PPC pour définir le problème, et les fourmis utilisent les procédures prédéfinies de la PPC pour propager les contraintes. Cependant, contrairement à CPO-ACO, Ant-CP effectue une recherche incomplète.

Plusieurs points méritent d'être mentionnés en tant que futures améliorations de CPO-ACO. À l'heure actuelle, CPO-ACO fonctionne plutôt bien sur les problèmes pour lesquels la recherche d'une solution réalisable est facile. Dans ce papier, nous avons appliqué CPO-ACO sur trois problèmes différents sans pour autant utiliser d'heuristiques dédiées à ces problèmes. Nous proposons d'étudier l'intérêt d'ajouter des heuristiques dépendantes des problèmes à résoudre et voir si cela lui permet de devenir compétitif avec les approches existantes dans l'état de l'art.

Le réglage des paramètres est un autre point intéressant. Pour l'instant les paramètres de CPO-ACO sont réglés en utilisant notre expérience. Cependant, on pense qu'une version adaptative qui change dynamiquement les valeurs des paramètres pendant l'exécution peut augmenter l'efficacité de l'algorithme et sa robustesse.

6 Remerciements

Les auteurs remercient Renaud Dumeur, Jérôme Rogerie, Philippe Refalo et Philippe Laborie pour les discussions fructueuses et animées sur l'optimisation combinatoire et, en particulier, les discussions sur les stratégies de recherche. Aussi, nous remercions Jérôme Rogerie et Philippe Laborie pour leur relecture de cet article.

Références

[1] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. MIT Press, 2004.

[2] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

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

[4] M. Khichane, P. Albert, and C. Solnon. Integration of *aco* in a constraint programming language. *6th International Conference on Ant Colony Optimization and Swarm Intelligence (A ANTS2008)*, (5217) :84–95, 2008.

[5] S. Kirkpatrick, C. Gellat, , and M. Vecchi. Optimization by simulated annealing. *science*, 220 :671–680, 1983.

[6] A.H. Land and A.G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28 :497–520, 1960.

[7] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems[ac1-3]. *Artificial Intelligence*, (25) :65–74, 1985.

[8] B. Meyer. *Hybrids of constructive meta-heuristics and constraint programming : A case study with ACO*. In Chr. Blum, M.J.Blesa, A. Roli, and M. Sampels, editors, Hybrid Metaheuristics-An emergent approach for optimization. Springer Verlag, New York, 2008.

[9] GL. Nemhauser and AL. Wolsey. *Integer and combinatorial optimization*. New York : Jhon Wiley and & Sons ;, 1988.

[10] CH. Papadimitriou and K. Steiglitz. *Combinatorial optimization–Algorithms and complexity*. New York :Dover ;, 1982.

[11] P. Refalo. Impact-based search strategies for constraint programming. *In CP04, 10th International Conference on Principales and Practice of Constraint Programming,*, (10) :557–571, 2004.

[12] C. Solnon. Ants can solve constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 6(4) :347–357, 2002.

[13] T. Stützle and H.H. Hoos. $\mathcal{MAX} - \mathcal{MIN}$ Ant System. *Journal of Future Generation Computer Systems*, 16 :889–914, 2000.

Génération de tests “tous-les-chemins” : quelle complexité pour quelles contraintes ?*

Nikolai Kosmatov

CEA, LIST, Laboratoire Sûreté des Logiciels, PC 94
91191 Gif-sur-Yvette France
Nikolai.Kosmatov@cea.fr

Résumé

La génération automatique de tests structurels à l'aide de la programmation par contraintes se répand de plus en plus dans l'industrie du logiciel. Parmi les critères de couverture les plus stricts, le critère *tous-les-chemins* exige la génération d'un ensemble de cas de tests tel que tout chemin d'exécution faisable du programme sous test soit exécuté par un des cas de test. Cet article étudie des aspects de calculabilité et de complexité de la génération de tests tous-les-chemins et le rapport entre la complexité et la forme des contraintes issues du programme sous test.

Nous définissons deux classes de programmes importantes pour la pratique. Nous montrons d'abord que pour une classe contenant des programmes simples avec de fortes restrictions, la génération de tests est possible en temps polynomial. Pour une classe de programmes plus large où les entrées peuvent être utilisées comme des indices de tableaux (ou décalages de pointeurs), la génération de tests tous-les-chemins s'avère NP-difficile. Quelques expérimentations montrent le temps de génération pour des exemples de programmes des deux classes.

1 Introduction

La génération automatique de tests à l'aide de la programmation par contraintes passe par la traduction du programme sous test (ou de son modèle formel) et du critère de couverture choisi en un problème de résolution de contraintes. Ensuite, un solveur de contraintes résout les contraintes et fournit *un cas de test*, c'est-à-dire des valeurs pour les variables d'entrée, qui peuvent être accompagnées d'*un oracle* décrivant le comportement attendu du programme sur ces entrées.

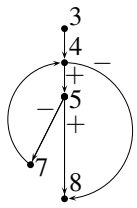
Parmi les méthodes les plus récentes, différentes techniques combinant l'exécution concrète et symbolique sont apparues durant les cinq dernières années. Elles ont permis le développement de plusieurs outils de génération de tests pour les programmes C tels que PathCrawler [2, 14, 15], DART [4], CUTE [12], EXE [3]. Ces techniques se sont montrées particulièrement pertinentes pour le *test orienté chemins*. Par exemple, le critère de couverture *tous-les-chemins* exige la génération d'un ensemble de cas de tests tel que tout chemin d'exécution activable (dit aussi *faisable*) du programme sous test soit exécuté par un cas de test. L'ensemble d'entrées possibles est supposé fini (sinon le nombre de chemins peut être infini, *cf* Section 3). Ce critère étant très fort et souvent inatteignable, des critères plus faibles restreignent la couverture aux chemins de longueur limitée, ou avec un nombre limité d'itérations de boucles, etc. Les chemins sont souvent explorés dans une recherche en profondeur d'abord [4, 12, 14, 15], parfois en largeur d'abord [16] ou avec des heuristiques mixtes [3]. Quand la couverture de chemins s'avère trop exigeante pour le programme sous test, on peut utiliser le critère *toutes-les-instructions* (toute instruction atteignable doit être exécutée par un cas de test) ou *toutes-les-branches* (toute branche atteignable doit être exécutée) [17].

Malgré l'utilisation croissante de la génération automatique de tests en industrie, l'ingénieur validation reste souvent incapable d'évaluer la calculabilité et la complexité de la génération pour un programme donné avec un critère particulier. Les résultats théoriques sous-jacents, souvent difficiles à trouver et à comprendre pour un praticien, traitent le cas le plus général et ne donnent que des réponses négatives. Cependant, les programmes à tester sont en pratique rarement si complexes que le “pire cas” théorique. Une étude détaillée de complexité de la génération de tests pour différents types de programmes en fonction de leurs

*La version originale a été publiée en anglais à *TAIC PART 2009*, Windsor, Royaume-Uni, septembre 2009.

```
1  #define D 4
2  int atU( int x[D], int y[D], int u ) {
3    int i = 1;
4    while( i < D ) {
5      if( u < x[i] )
6        break;
7      i++; }
8    return y[i-1];
9  }
```

(a)



(b)

FIG. 1 – **(a)** Fonction `atU`, et **(b)** son graphe de flot de contrôle

caractéristiques, avec les contraintes et les critères qui en résultent, peut paraître sans intérêt pour un théoricien, mais sera extrêmement utile en pratique.

La motivation de ce travail est d'initier une telle étude de calculabilité et complexité de la génération de tests pour certaines classes de programmes faisant apparaître certains types de contraintes. Dans cet article, nous considérons le cas de la génération de tests tous-les-chemins. Nous présentons d'abord la méthode de génération de tests tous-les-chemins avec une recherche en profondeur d'abord et illustrons son fonctionnement sur un exemple (Section 2).

Contributions. Notre principale contribution consiste à considérer deux classes de programmes. Dans le premier cas (Section 3), grâce à des restrictions sur la taille, le nombre et la forme des contraintes produites, nous montrons que la génération de tests tous-les-chemins en temps polynomial est possible (Théorème 2). La preuve s'appuie sur la méthode polynomiale de résolution des contraintes de différence de Pratt [10, 11]. Strictement parlant, nous prouvons que le temps de résolution de contraintes est polynomial, ce qui est justement la seule étape longue en pratique dans la génération. Nous déduisons la complexité polynomiale pour des critères de couverture plus faibles tels que toutes-les-branches et toutes-les-instructions (Corollaire 3).

Ensuite, la Section 4 considère une classe de programmes plus large pouvant contenir des variables d'entrée dans les indices de tableaux (ou décalages de pointeurs) et contraintes avec $\neq$. Nous donnons une simple preuve par une réduction originale du problème de circuit Hamiltonien que la génération de tests tous-les-chemins pour ces programmes peut être NP-difficile (Théorème 5), donc la génération en temps polynomial est impossible (sauf si P=NP). La Section 5 décrit quelques expériences de génération pour certaines classes de programmes considérées dans les Sections 3 et 4. Enfin, la Section 6 présente la conclusion et les perspectives de travail.

Pour rendre cet article facile à comprendre aussi bien par un spécialiste en théorie de complexité que par un praticien en test de logiciels, nous rappelons des notions des deux domaines, donnons une présentation simplifiée de la génération de tests tous-les-chemins en profondeur d'abord à l'aide de programmation par contraintes, et proposons en Section 4 une ébauche de preuve à l'aide d'un programme C très simple plutôt qu'une preuve formelle sur des machines de Turing. Le lecteur trouvera une introduction à la théorie de calculabilité et complexité dans [5] et plus d'information sur la génération de tests dans [7] et les références dans [7].

2 Génération de tests tous-les-chemins en profondeur d'abord

Cette section décrit la méthode de génération de tests tous-les-chemins à la PathCrawler pour les programmes C, appelée parfois *concolique*. Nous considérons les programmes C avec les types entiers, tableaux, pointeurs (où les variables d'entrée n'apparaissent pas dans les indices de tableaux ou décalages de pointeurs), conditionnels et boucles. Des méthodes similaires sont utilisées dans d'autres outils comme DART [4] et CUTE [12]. Soit f la fonction sous test en C.

L'outil PathCrawler [2, 14, 15] est développé au CEA LIST et comprend deux modules. Le premier, basé sur la bibliothèque CIL [9], traduit les instructions du code source C en contraintes et crée une *version instrumentée* du code qui sera exécutée pour imprimer le chemin d'exécution sur un cas de test. Ensuite, l'utilisateur peut modifier les paramètres de test par défaut et spécifier *une précondition* qui définit les conditions sur les entrées pour lesquelles la fonction f a été conçue et doit être testée. Le second module, *générateur de tests*, implémenté en Prolog, lit les contraintes de f et la précondition et génère des cas de tests pour le critère tous-les-chemins. Les chemins sont explorés en profondeur d'abord. Le générateur utilise une combinaison originale de l'exécution symbolique à l'aide de contraintes et de l'exécution concrète du code instrumenté. L'exécution symbolique traduit le problème de génération de test pour un chemin (partiel) en un problème de résolution de contraintes et appelle un sol-

(1) Mémoire	Contraintes
$\mathtt{x}[j] \mapsto X_j$ $\mathtt{y}[j] \mapsto Y_j$ $\mathtt{u} \mapsto U$	$\langle$precond$\rangle$
$\pi = \epsilon$	

$\rightsquigarrow$

Test 1 :	$U = 2$
$X_0 = 0$	$Y_0 = 19$
$X_1 = 5$	$Y_1 = 17$
$X_2 = 11$	$Y_2 = 5$
$X_3 = 16$	$Y_3 = 15$
$\sigma = 4^+, 5^+$	

$\rightarrow$

(2) Mémoire	Contraintes
$\mathtt{x}[j] \mapsto X_j$ $\mathtt{y}[j] \mapsto Y_j$ $\mathtt{u} \mapsto U$ $\mathtt{i} \mapsto 1$	$\langle$precond$\rangle, 1 < 4,$ $U \geq X_1$
$\pi = 4^+, 5^-_\star$	

$\rightsquigarrow$

Test 2 :	$U = 4$
$X_0 = 1$	$Y_0 = 1$
$X_1 = 4$	$Y_1 = 7$
$X_2 = 9$	$Y_2 = 10$
$X_3 = 14$	$Y_3 = 9$
$\sigma = 4^+, 5^+$	

$\rightarrow$

(3) Mémoire	Contraintes
$\mathtt{x}[j] \mapsto X_j$ $\mathtt{y}[j] \mapsto Y_j$ $\mathtt{u} \mapsto U$ $\mathtt{i} \mapsto 2$	$\langle$precond$\rangle, 1 < 4,$ $U \geq X_1, 2 < 4,$ $U \geq X_2$
$\pi = 4^+, 5^-_\star, 4^+, 5^-_\star$	

$\rightsquigarrow$

Test 3 :	$U = 19$
$X_0 = 2$	$Y_0 = 8$
$X_1 = 4$	$Y_1 = 5$
$X_2 = 16$	$Y_2 = 2$
$X_3 = 20$	$Y_3 = 4$
$\sigma = 4^+, 5^+$	

$\rightarrow$

(4) Mémoire	Contraintes
$\mathtt{x}[j] \mapsto X_j$ $\mathtt{y}[j] \mapsto Y_j$ $\mathtt{u} \mapsto U$ $\mathtt{i} \mapsto 3$	$\langle$precond$\rangle, 1 < 4,$ $U \geq X_1, 2 < 4,$ $U \geq X_2, 3 < 4,$ $U \geq X_3$
$\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star$	

$\rightsquigarrow$

Test 4 :	$U = 18$
$X_0 = 1$	$Y_0 = 0$
$X_1 = 11$	$Y_1 = 7$
$X_2 = 15$	$Y_2 = 9$
$X_3 = 18$	$Y_3 = 11$
$\sigma = 4^-$	

$\rightarrow$

(5) Mémoire	Contraintes
$\mathtt{x}[j] \mapsto X_j$ $\mathtt{y}[j] \mapsto Y_j$ $\mathtt{u} \mapsto U$ $\mathtt{i} \mapsto 4$	$\langle$precond$\rangle, 1 < 4,$ $U \geq X_1, 2 < 4,$ $U \geq X_2, 3 < 4,$ $U \geq X_3, 4 < 4$
$\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+_\star$	

$\rightsquigarrow$ $\langle$unsatisfaisable$\rangle$

$\rightarrow$

(6) Mémoire	Contraintes
$\mathtt{x}[j] \mapsto X_j$ $\mathtt{y}[j] \mapsto Y_j$ $\mathtt{u} \mapsto U$ $\mathtt{i} \mapsto 3$	$\langle$precond$\rangle, 1 < 4,$ $U \geq X_1, 2 < 4,$ $U \geq X_2, 3 \geq 4,$
$\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^-_\star$	

$\rightsquigarrow$ $\langle$unsatisfaisable$\rangle$

$\rightarrow$ ...

FIG. 2 – Génération de tests tous-les-chemins pour la fonction `atU` de la Figure 1

veur de contraintes pour générer un cas de test activant ce chemin. L'exécution concrète du code instrumenté compilé sur un cas de test fournit rapidement le chemin réellement exécuté. PathCrawler utilise COLIBRI, un solveur performant développé au CEA LIST, qu'il partage avec les outils GATeL [8] et OSMOSE [1].

Nous supposons que le programme sous test a au plus une instruction par ligne et une condition par décision. (Le premier module de PathCrawler réécrit les conditions multiples en conditions simples en introduisant des conditionnels supplémentaires.) Un chemin de programme ρ sera noté par une suite de numéros de lignes, e.g.

$$\rho = 3, 4^+, 5^-, 7, 4^+, 5^+, 6, 8$$

est un chemin dans le programme de la Figure 1. Pour les décisions (dans les conditionnels ou boucles), le numéro de ligne est suivi d'un "+" si la condition est vraie, et d'un "−" sinon. Puisque un chemin d'exécution est déterminé par ses décisions, nous pouvons abréger un chemin par une suite de décisions, e.g. $\rho = 4^+, 5^-, 4^+, 5^+$. Nous utiliserons cette notation abrégée. Le chemin vide est noté ϵ.

La marque "$\star$" après une décision indiquera que le parcours en profondeur d'abord a déjà complètement exploré sa négation, i.e. l'autre branche dans l'arbre des chemins d'exécution. Par exemple, la marque "$\star$" dans le chemin $\rho = 4^+, 5^-_\star, 4^+, 5^+$ signifie que nous avons déjà exploré tous les chemins de la forme $4^+, 5^+, \ldots$ et essayé de générer un test pour chacun d'entre eux.

Lors d'une session de génération, le générateur maintient les données suivantes :

- un tableau représentant la mémoire du programme à tout moment de l'exécution symbolique. Il peut être vu comme une fonction $Symb \mapsto Val$ qui associe une valeur Val (une constante ou une variable logique Prolog) à un nom symbolique $Symb$ (un nom de variable ou un élément de tableau).
- un chemin partiel π dans f. Si un cas de test est généré pour le chemin partiel π, alors σ notera la partie restante du chemin complet activé par ce cas de test.
- un store de contraintes contenant les contraintes collectées par l'exécution symbolique du chemin partiel courant π.

Nous pouvons maintenant décrire la méthode de génération. Elle comprend les étapes suivantes :

(Init) Pour chaque entrée, on crée une variable logique et l'associe à cette entrée. Pour les variables initialisées, on enregistre leurs valeurs initiales. On pose les contraintes de la précondition. Soit le chemin partiel courant π vide. Passe à l'étape 1.

(Etape 1) Soit σ vide. Le générateur exécute symboliquement le chemin partiel π, i.e. rajoute des contraintes et met à jour la mémoire selon les instructions dans π. Si une contrainte échoue, passe à l'étape 4. Sinon, passe à l'étape 2.

(Etape 2) Le solveur de contraintes est appelé pour générer un cas de test, c'est-à-dire des valeurs pour les variables d'entrée vérifiant les contraintes posées. S'il échoue, passe à l'étape 4. Sinon, passe à l'étape 3.

(Etape 3) La version instrumentée du programme est exécutée sur le cas de test généré à l'étape 2 pour retrouver le chemin complet. Il commence par π (par définition du problème de résolution de contraintes qui a permis de générer ce cas de test). Soit σ la partie restante du chemin complet. Passe à l'étape 4.

(Etape 4) Soit ρ la concaténation de π et σ. Si ρ ne contient aucune décision non marquée par une "$\star$", l'algorithme s'arrête. Sinon, si $x^\pm$ est la dernière décision non marquée dans ρ, soit π le sous-chemin dans ρ avant $x^\pm$, suivi de $x^\mp_\star$ (i.e. la négation de $x^\pm$ marquée comme traitée). Passe à l'étape 1.

Ainsi, l'étape 4 fait une recherche en profondeur d'abord pour choisir le chemin partiel suivant. Elle nie la dernière condition non marquée dans ρ pour trouver les différences les plus profondes d'abord, et marque une décision par une "$\star$" quand sa négation (i.e. l'autre branche partant de ce nœud dans l'arbre des chemins d'exécution) est complètement explorée. Par exemple, si

$$\rho = a^-_\star, b^+, c^+, d^-_\star, e^+_\star,$$

la dernière "$\star$" signifie que la recherche en profondeur d'abord a déjà exploré tous les chemins de la forme

$$a^-, b^+, c^+, d^-, e^-, \ldots$$

La précédente "$\star$" (dans $d^-_\star$) signifie que la recherche en profondeur d'abord a déjà traité tous les chemins

$$a^-, b^+, c^+, d^+, \ldots$$

La dernière décision non marquée dans ρ étant c^+, l'étape 4 prendra le sous-chemin avant cette décision $a^-_\star, b^+$ et rajoutera $c^-_\star$ pour obtenir le nouveau chemin partiel $\pi = a^-_\star, b^+, c^-_\star$. Ainsi, ce marquage de conditions avec une "$\star$" préserve les informations sur le parcours de des chemins partiels plus courts (ici, les chemins de la forme $a^+, \ldots$ sont complètement explorés), et rajoute ces informations pour la négation de la dernière condition (ici, les chemins $a^-, b^+, c^+, \ldots$ sont complètement explorés).

Nous déroulons cette méthode sur l'exemple de la fonction `atU` de la Figure 1. Cette fonction est la forme la plus simple d'interpolation. Elle prend trois paramètres, deux tableaux `x`, `y` (chacun avec `D` entiers) et un entier `u`. Soit $\psi_{\texttt{atU}}$ la précondition de `atU` définie comme suit :

$$\begin{array}{c} \texttt{D} \geq 1, \quad \texttt{x} \text{ contient } \texttt{D} \text{ éléments}, \\ \texttt{y} \text{ contient } \texttt{D} \text{ éléments}, \\ 0 \leq \texttt{x}[0] < \texttt{x}[1] < \cdots < \texttt{x}[\texttt{D}-1] \leq \mathrm{Max}, \\ \texttt{x}[0] \leq \texttt{u} \leq \texttt{x}[\texttt{D}-1], \\ 0 \leq \texttt{y}[0], \texttt{y}[1], \ldots, \texttt{y}[\texttt{D}-1] \leq \mathrm{Max}. \end{array} \qquad (\psi_{\texttt{atU}})$$

On suppose que les $\texttt{y}[j]$ sont les valeurs d'une certaine fonction h en les points $\texttt{x}[j]$, i.e. $h(\texttt{x}[j]) = \texttt{y}[j]$, $0 \leq j \leq \texttt{D} - 1$. La fonction `atU` retourne la valeur de h en le point le plus proche à gauche de `u` pour lequel la valeur de h est connue. Autrement dit, elle trouve le plus grand k tel que $\texttt{x}[k] \leq \texttt{u}$ et retourne $\texttt{y}[k]$. Max est une constante positive (e.g. la valeur maximale entière du système). Pour simplifier notre exemple, on suppose $\texttt{D} = 4$ et $\mathrm{Max} = 20$.

La session de génération de tests pour la fonction `atU` est présentée en Figure 2, où "$\rightsquigarrow$" note l'application des étapes 2 et 3, et "$\rightarrow$" l'application des étapes 4 et 1.

D'abord, (Init) crée des variables logiques X_j, Y_j ($0 \leq j \leq 3$) et U pour représenter les entrées, *cf* (1) de la Figure 2. Les deux premières lignes de ($\psi_{\texttt{atU}}$) étant satisfaites, (Init) ajoute dans le store de contraintes les $3\texttt{D}+3$ inégalités correspondant aux trois dernières lignes de ($\psi_{\texttt{atU}}$), qui sont notées par ⟨precond⟩ dans la Figure 2 :

$$\begin{gathered} 0 \leq X_0,\ X_0 < X_1,\ X_1 < X_2,\ X_2 < X_3,\ X_3 \leq 20, \\ X_0 \leq U,\ U \leq X_3, \\ 0 \leq Y_0,\ \ldots\ ,\ 0 \leq Y_3, \\ Y_0 \leq 20,\ \ldots\ ,\ Y_3 \leq 20. \end{gathered}$$

Le premier chemin partiel π étant toujours vide, l'étape 1 n'a rien à faire ici. Ensuite, l'étape 2 génère le premier cas de test, Test 1. L'étape 3 exécute le Test 1 sur la version instrumentée du programme et obtient $\sigma = 4^+, 5^+$ (ce qui abrège $3, 4^+, 5^+$).

Nous passons maintenant de (1) et Test 1 à (2) dans la Figure 2. L'étape 4 calcule $\rho = 4^+, 5^+$, où 5^+ est la dernière décision non marquée. Par conséquent, $\pi = 4^+, 5^-_\star$. Ensuite, l'étape 1 exécute symboliquement le chemin partiel π en contraintes, nœud par nœud, avec des entrées inconnues. L'exécution de l'affection 3 ajoute $\texttt{i} \mapsto 1$ dans la mémoire. L'exécution de la décision 4^+ ajoute la contrainte $1 < 4$ trivialement vraie, et l'exécution de la décision 5^- ajoute la contrainte $U \geq X_1$ après avoir remplacé les variables i, u et $\texttt{x}[1]$ par leurs valeurs actuelles dans la mémoire. Ensuite, l'étape 2 génère le Test 2, et l'étape 3 l'exécute et obtient $\sigma = 4^+, 5^+$.

Nous passons maintenant de (2) et Test 2 à (3) dans la Figure 2. L'étape 4 calcule $\rho = 4^+, 5^-_\star, 4^+, 5^+$ et déduit $\pi = 4^+, 5^-_\star, 4^+, 5^-_\star$. L'étape 1 exécute symboliquement π, l'étape 2 génère le Test 3, etc. Arrêtons-nous sur le passage de (4) et Test 4 à (5) dans la Figure 2. L'étape 4 calcule

$$\rho = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star, 4^-$$

et déduit $\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+, 5^-_\star, 4^+_\star$. La dernière contrainte $4 < 4$ ajoutée par exécution symbolique à l'étape 1 est fausse, donc le générateur passe directement à l'étape 4, qui pose $\pi = 4^+, 5^-_\star, 4^+, 5^-_\star, 4^-_\star$. Selon (6) de la Figure 2, la dernière contrainte $3 \geq 4$ ajoutée par l'étape 1 échoue de nouveau, et le générateur passe à l'étape 4. Les Etape après (6) ne sont pas détaillées en Figure 2. De même, le générateur essaie les chemins partiels $\pi = 4^+, 5^-_\star, 4^-_\star$ et $\pi = 4^-_\star$, qui sont infaisables, et s'arrête. Un cas de test est généré pour chacun des 4 chemins faisables.

En général, si pendant l'exécution symbolique aux étapes 1 ou 2, les contraintes sont insatisfaisables et aucun cas de test ne peut être généré, alors π est infaisable et l'algorithme continue l'exploration des autres chemins normalement. Si cela se produit à (Init) or à la première itération de l'étape 2, i.e. la précondition est insatisfaisable, alors l'algorithme s'arrête à l'étape 4 car ρ est vide.

3 Génération de tests tous-les-chemins en temps polynomial

Cette section présente des conditions suffisantes sous lesquelles une classe de problèmes de génération de tests tous-les-chemins devient polynomiale. Il est intuitivement clair que la génération de tests tous-les-chemins pour un programme peut prendre beaucoup de temps pour une (ou plusieurs) des raisons suivantes :

- (†) le programme a un très grand nombre de chemins, et se traduit donc par un très grand nombre de problèmes de résolution de contraintes,
- (††) les instructions du programme se traduisent par des contraintes complexes qui ne peuvent être résolues rapidement,
- (†††) le programme a de très longs chemins faisant apparaître des problèmes avec trop de contraintes.

Nous montons que des restrictions appropriées sur ces trois facteurs assurent la génération de tests tous-les-chemins en temps polynomial.

Puisque l'étape la plus coûteuse de la génération de tests tous-les-chemins en pratique est la résolution de contraintes à l'étape 2, nous allons nous focaliser sur le temps de cette résolution. Notre expérience avec l'outil PathCrawler montre que les autres étapes (instrumentation, traduction en contraintes, exécution symbolique etc.) sont très efficaces. Par exemple, le premier module de PathCrawler, qui instrumente et traduit en contraintes le programme sous test, met moins d'une minute pour des programmes de plusieurs centaines et milliers de lignes de code. Comme la performance de ces étapes dépend de nombreux détails d'implémentation et semble très satisfaisante en pratique, nous la laissons de côté ici. Notons que le temps de génération de tests dans les expérimentations de la Section 5 comprend toutes les étapes, depuis le code source jusqu'aux cas de test générés.

Nous définissons *un problème de génération de tests tous-les-chemins* comme

$$\Phi = (P, f, \psi)$$

où P est un programme, f est une fonction à tester dans P et ψ est la précondition de f. Une solution de Φ est un ensemble de cas de test satisfaisant le critère tous-les-chemins. La précondition peut contenir des informations nécessaires pour une initialisation correcte de la génération de tests (e.g. tailles de tableaux d'entrée, domaines de variables, etc.) et toute autre conditions sur les variables d'entrée réduisant les entrées admissibles de f. Nous notons par $L^\Phi_P > 0$ la taille du programme P.

Le nombre d'entrées possibles doit être fini (et non pas seulement borné par la mémoire disponible de l'ordinateur qui est supposée suffisamment grande). En effet, si le nombre d'entrées est illimité, le nombre de chemins peut

```
1  char LastChar(char str[]) {
2    while( *(str + 1) != 0 )
3      str = str + 1;
4    return * str;
5  }
```

FIG. 3 – Pour une chaîne de caractère non vide terminée par un 0, `LastChar` retourne son dernier caractère non nul

être illimité et la génération de tests ne terminera pas. Cela se produit pour la fonction `LastChar` de la Figure 3.

Nous supposons donc que ψ borne la taille maximale des entrées admissibles pour Φ (mesurée en octets, ou à une constante près, en entiers) par un $L_I^\Phi > 0$.

Un système de contraintes de différence [11] est défini comme un système de contraintes de la forme

$$x - y \leq c, \quad \text{ou} \quad x \leq c, \quad \text{ou} \quad x \geq c,$$

où x, y sont des variables entières et c est un entier. Une égalité $x - y = c$ peut être représentée comme $x - y \leq c$ et $y - x \leq -c$.

Théorème 1 ([10, 11]) *Un système de contraintes de différence peut être résolu en temps polynomial $g(m, n)$, où $g(X, Y)$ est un polynôme, n le nombre de variables et m le nombre de contraintes.*

Le lecteur trouvera plusieurs estimations g dans [11]. Nous sommes prêts à énoncer le résultat principal de cette section. Notez que les conditions (i), (ii), (iii) correspondent précisement aux facteurs (†), (††), (†††).

Théorème 2 *Soit $\mathcal{C}$ une classe de problèmes de génération de tests tous-les-chemins, et soient $g_1(X, Y)$, $g_2(X, Y)$ deux polynômes. On suppose que tout problème $\Phi = (P, f, \psi)$ de $\mathcal{C}$ satisfait les propriétés suivantes :*

(i) *le nombre de chemins dans Φ pour lesquels la méthode essaiera de générer un cas de test est borné par $g_1(L_P^\Phi, L_I^\Phi)$,*

(ii) *l'exécution symbolique de chaque chemin (y-compris les contraintes de la précondition) ajoute uniquement des contraintes de différence,*

(iii) *l'exécution symbolique de chaque chemin (y-compris les contraintes de la précondition) ajoute au plus $g_2(L_P^\Phi, L_I^\Phi)$ contraintes.*

Alors il existe un polynôme $g_3(X, Y)$ tel que le temps de résolution de contraintes total lors de la génération de tests tous-les-chemins pour Φ est borné par $g_3(L_P^\Phi, L_I^\Phi)$.

Ebauche de preuve. Sans perte de généralité, on suppose $g(X, Y)$ monotone en chaque argument pour $X > 0, Y > 0$. Soit $\Phi = (P, f, \psi)$ un problème de génération de tests tous-les-chemins dans $\mathcal{C}$. Selon (i), la méthode de génération de tests tous-les-chemins pour Φ résout au

```
1  #define D 4
2  int bsearch(int a[D], int key) {
3    int low = 0; int high = D-1;
4    while (low <= high) {
5      int mid = low + (high-low)/2;
6      int midVal = a[mid];
7      if (midVal < key)
8        low = mid+1;
9      else if (midVal > key)
10       high = mid-1;
11     else
12       return mid;
13   }
14   return -1;
15 }
```

FIG. 4 – Etant donné un élément `key` et un tableau trié `a` de taille `D`, `bsearch` fait une recherche binaire de `key` dans `a`

plus $g_1(L_P^\Phi, L_I^\Phi)$ problèmes de résolution de contraintes. D'après (ii), le problème de résolution de contraintes créé pour chaque chemin (donc, chaque chemin partiel) de Φ est un système de contraintes de différence. D'après (iii), ce sytème contient $m \leq g_2(L_P^\Phi, L_I^\Phi)$ contraintes. Le nombre de variables n est borné par L_I^Φ. Il suit que le temps de résolution total est borné par

$$g_1(L_P^\Phi, L_I^\Phi) g(m, n) \leq g_3(L_P^\Phi, L_I^\Phi),$$

où $g_3(X, Y) := g_1(X, Y) g(g_2(X, Y), Y)$. □

Nous permettons intentionnellement de borner le nombre de chemins par la taille du programme, ou la taille des entrées, ou les deux, car différentes estimations peuvent être utiles dans différents exemples. Notons que le nombre de chemins mentionnés dans (i) inclut les chemins partiels infaisables comme ceux vus dans la Section 2, mais n'inclut pas plusieurs chemins commençant par le même chemin partiel infaisable. En effet, la méthode en profondeur d'abord rajoute au plus une nouvelle contrainte aux contraintes d'un chemin partiel faisable, donc elle n'essaie jamais de générer un cas de test pour un chemin plus long qui contient un chemin partiel infaisable strictement plus court.

Appliquons le Théorème 2 à un exemple. Soit $\Phi_{\mathtt{D}} = (P_{\mathtt{D}}, \mathtt{atU}, \psi_{\mathtt{atU}})$ la famille de problèmes de génération de tests tous-les-chemins, où $\mathtt{D} > 0$ est un paramètre et $P_{\mathtt{D}}$ est le programme de la Figure 1 avec la fonction `atU` et la précondition $\psi_{\mathtt{atU}}$ définie dans la Section 2. Le nombre de variables d'entrée dans $\Phi_{\mathtt{D}}$ est $2\mathtt{D} + 1$, donc $L_I^{\Phi_{\mathtt{D}}} = 2\mathtt{D} + 1$. Le nombre de chemins partiels dans $\Phi_{\mathtt{D}}$ pour lesquels la méthode essaiera de générer un cas de test est égal à $2\mathtt{D} \leq L_I^{\Phi_{\mathtt{D}}}$, d'où (i). L'exécution symbolique des chemins de $\Phi_{\mathtt{D}}$ n'ajoute que des contraintes de différence, d'où (ii), avec $3\mathtt{D} + 3$ contraintes pour la précondition et

$2\mathtt{D}-1$ contraintes pour le chemin le plus long, d'où (iii) car $(3\mathtt{D}+3)+(2\mathtt{D}-1) \leq 3L_I^{\Phi_\mathtt{D}}$. Les conditions du Théorème 2 sont vérifiées, donc la génération de tests tous-les-chemins en temps polynomial est possible pour cette exemple.

De la même façon, ce théorème peut être appliqué à d'autres fonctions d'interpolation utilisées en pratique, ou à des fonctions de recherche dans un tableau comme la fonction `bsearch` de la Figure 4. Etant donné un tableau trié `a` de taille `D` et un élément `key`, la fonction `bsearch` effectue une recherche binaire (dichotomique) classique d'une occurence de `key` dans `a` et retourne son indice, ou -1 si `key` n'apparaît pas dans `a`. A première vue, l'affectation de la ligne 5 de la Figure 4 n'ajoute pas une contrainte de différence. En réalité, pendant l'exécution symbolique d'un chemin partiel, la partie droite des affectations lines $3, 5, 8, 10$ contient uniquement des constantes sans variables d'entrée, qui permettent un calcul direct de la nouvelle valeur de la variable et n'ajoutent pas de contrainte sur des variables d'entrée.

Puisque le critère tous-les-chemins est plus fort que d'autres critères de couverture comme toutes-les-branches ou toutes-les-instructions [17], le résultat suivant découle directement du Théorème 2.

Corollaire 3 *Soit $\mathcal{C}$ une classe de problèmes de génération de tests tous-les-chemins satisfaisant les conditions du Théorème 2. Alors le temps total de résolution de contraintes pour la génération de tests avec le critère toutes-les-branches (ou toutes-les-instructions) est polynomial.*

4 Génération de tests tous-les-chemins avec alias internes est NP-difficile

Dans cette section nous allons considérer une classe plus large de problèmes de génération de tests tous-les-chemins où les contraintes avec $\neq$ sont autorisées et les indices de tableaux (ou décalages de pointeurs) peuvent dépendre des variables d'entrée. La présence d'indices inconnus pendant l'exécution symbolique pour des entrées inconnues nous conduit au problème des *alias internes*, définis dans [6]. En effet, si `j` est une variable d'entrée, ou a reçu une valeur dépendant de variables d'entrée, l'expression `a[j]` est un alias non-trivial à un des aléments de `a`. L'utilisation des entrées `p[j]` comme indices du tableau `G` lines 26, 28 de la Figure 5a est un autre exemple d'alias internes. La génération de tests tous-les-chemins pour les programmes avec des alias internes a été considérée dans [6] où une extension de la méthode de la Section 2 pour ce type de programmes a été proposée.

Rappelons le problème du circuit Hamiltonien. *Un circuit Hamiltonien* dans un graphe orienté G est un circuit qui passe par chaque nœud une seule fois et revient vers le nœud de départ. Par exemple, la Figure 5b représente un graphe orienté avec 5 nœuds $\{0, 1, 2, 3, 4\}$ ayant un circuit Hamiltonien. On peut identifier un graphe orienté avec sa *matrice d'adjacence*. Son élément $G(i, j)$ est 1 si G a un arc allant de i à j, et 0 sinon. Nous préférons ici la notation mathématique $G(i, j)$, $p(i)$ à la notation C (en police TrueType) `G[i][j]`, `p[i]`.

Dans les lignes 5–11 de la Figure 5a, G est le graphe de la Figure 5b (avec $N = 5$ nœuds) défini par sa matrice d'adjacence. La première boucle dans la fonction HC vérifie que les éléments de p sont dans $\{0, 1, \ldots, N-1\}$ et sont tous différents (lines 15–23). Cela signifie que p est *une permutation de* $\{0, ..., N-1\}$. HC retourne 1 si p est une permutation des nœuds de G et

$$p(0) \to p(1) \to \cdots \to p(N-1) \to p(0)$$

est un circuit Hamiltonien dans G, et 0 sinon (lignes 25–32). La précondition ψ_{HC} est définie comme suit :

$$\begin{array}{c} p \text{ contient } N \text{ éléments,} \\ 0 \leq p(j) \leq MAXINT. \end{array} \qquad (\psi_{\mathrm{HC}})$$

Nous allons admettre la Conjecture 4, conséquence de la fameuse conjecture $P \neq NP$ que l'on croit vraie, et formuler le résultat principal de cette section.

Conjecture 4 ([5, Section 10.4.4]) *Il n'existe pas d'algorithme décidant en temps polynomial si un graphe orienté donné contient un circuit Hamiltonien.*

Théorème 5 *Il n'existe pas d'algorithme polynomial pour résoudre les problèmes de génération de tests tous-les-chemins pour les programmes avec des alias internes.*

Ebauche de preuve. Supposons le contraire. Soit A un algorithme polynomial qui, étant donné un problème de génération de tests $\Phi = (P, f, \psi)$, génère une liste

$$(t_1, \rho_1), \ldots, (t_k, \rho_k)$$

où t_i est un cas de test, ρ_i est le chemin d'exécution activé par l'exécution de f sur t_i, et $\rho_1, \ldots, \rho_k$ sont tous les chemins faisables de f. "Un algorithme polynomial" signifie qu'il existe $K, m > 0$ tels que le nombre de pas de A est toujours borné par le polynôme $K(L_P^\Phi + L_I^\Phi)^m$, où L_P^Φ est la taille de P et L_I^Φ est la taille maximale des entrées.

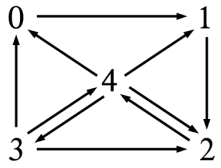
Alors nous pouvons construire un nouvel algorithme B qui, étant donné la matrice d'adjacence G d'un graphe orienté et son nombre de nœuds N,

$B1$) construit un programme P_G similaire à celui de la Figure 5a avec les N et G donnés,

$B2$) exécute A sur le problème $\Phi_G = (P_G, \mathrm{HC}, \psi_{\mathrm{HC}})$,

$B3$) dit "oui" si A a généré un cas de test pour le chemin retournant 1, et "non" sinon.

```
1   #define N 5 // number of vertices in graph G
2   typedef int graph[N][N];
3   typedef int perm[N];
4   // graph G is defined by its adjacency matrix :
5   graph G = {
6     0,1,0,0,0,
7     0,0,1,0,0,
8     0,0,0,0,1,
9     1,0,1,0,1,
10    1,1,1,1,0
11  };
12
13  int HC(perm p){
14    int i, j;
15    for( i = 0; i < N; i++ ){
16      if( p[i] < 0 )
17        return 0;
18      if( p[i] > N-1 )
19        return 0;
20      for( j = i+1; j < N; j++ )
21        if( p[i] == p[j] )
22          return 0;
23    }
24  // we checked that p is a permutation of {0,...,N-1}
25    for( i = 1; i < N; i++ )
26      if( G[ p[i-1] ][ p[i] ] != 1 )
27        return 0;
28    if( G[ p[N-1] ][ p[0] ] != 1 )
29      return 0;
30  // we checked that p defines the Hamiltonian cycle
31  // p(0) -> p(1) -> ... -> p(N-1) -> p(0) in G
32    return 1;
33  }
```



(a) **(b)**

FIG. 5 – **a)** Etant donné un graphe G avec N nœuds, statiquement défini par sa matrice d'adjacence, la fonction HC vérifie si p est une permutation de nœuds définissant le circuit Hamiltonien $p(0) \to p(1) \to \cdots \to p(N-1) \to p(0)$.
b) Le graphe G a le circuit Hamiltonien $0 \to 1 \to 2 \to 4 \to 3 \to 0$.

La taille des entrées (N, G) de B est proportionnelle à N^2 :

$$|G| \leq |(N, G)| \leq 2|G|,\ |G| \sim N^2.$$

Montrons que B est un algorithme polynomial. En effet, pour certains $K_j > 0$, $B1$ copie des chaînes de caractères de taille $\leq K_1 N^2$. $B2$ exécute l'algorithme A sur le problème Φ_G avec $(L_P^\Phi + L_I^\Phi) \leq K_2 N^2$, donc il fait $\leq K(K_2 N^2)^m$ pas. La fonction HC a $\leq K_3 N^2$ chemins faisables, et la taille de chaque chemin est $\leq K_4 N^2$. $B3$ lit la liste de cas de test générés qui peut contenir $\leq K_3 N^2$ couples (t_i, ρ_i), chacun de taille $\leq K_5 N^2$. Donc B fait $\leq K_6 N^{2m} + K_7 N^4 + K_8 N^2$ pas.

Il est clair que le chemin retournant 1 dans la fonction HC de P_G est faisable si et seulement si G a un circuit Hamiltonien. On déduit que B est un algorithme polynomial décidant si un graphe donné G a un circuit Hamiltonien. La contradiction avec la Conjecture 4 termine la preuve. □

L'intérêt du Théorème 5 réside dans sa généralité : il est vrai même pour les programmes les plus simples, comme celui de la Figure 5a, où

- le nombre de chemins est borné par un polynôme en la taille du programme, *cf* (†),
- la longueur des chemins est bornée par un polynôme en la taille du programme aussi, *cf* (†††),
- la fonction sous test f contient seulement des entiers,

	atU		bsearch	
D	tests	temps	tests	temps
4	4	0.50 s	9	0.49 s
10	10	0.52 s	21	0.56 s
50	50	1.38 s	101	2.91 s
100	100	6.06 s	201	12.10 s
500	500	5 m 39 s	1001	6 m 50 s
1000	1000	30 m 46 s	2001	32 m 47 s

FIG. 6 – Résultats de génération de tests tous-les-chemins pour les fonctions `atU` de la Figure 1 et `bsearch` de la Figure 4 pour différentes valeurs de `D`.

	HC		
N	chemins	tests	temps
4	15	38	0.66 s
5	21	140	2.10 s
6	28	747	19.75 s
7	36	5 075	4 m 33 s
8	45	40 364	26 m 58 s
9	55	362 934	4 h 2 m 24 s

FIG. 7 – Résultats de génération de tests tous-les-chemins pour la fonction HC de la Figure 5a sur des graphes complets G_N.

tableaux, conditionnels, affectations et boucles avec un nombre fixe d'itérations,

– f ne contient pas d'appels de fonction, ni d'*alias externes* (qui apparaissent quand f contient des pointeurs en entrée qui peuvent faire référence à certaines cases mémoire de deux façons différentes, *cf* [6]).

Dans cet exemple, la complexité est due à $(\dagger\dagger)$, i.e. la forme des contraintes incluant des alias internes et $\neq$, bien que (i), (iii) du Théorème 2 soient vérifiés.

Remarque. En fait, nous avons montré que le problème de génération de tests tous-les-chemins pour ces programmes est NP-difficile, c'est-à-dire au moins aussi difficile que le problème du circuit Hamiltonien ou tout autre problème NP-complet. Un spécialiste en théorie de complexité remarquera que notre ébauche de preuve peut être transformé en une preuve complète car un ordinateur peut être simulé par une machine de Turing en temps polynomial et inversement [5, Section 8.6]. Une représentation appropriée pour N et p va résoudre le problème de grandes valeurs dépassant la taille du mot sur notre ordinateur.

5 Expérimentations

Cette section présente des expérimentations avec l'outil PathCrawler qui montrent les résultats de génération de tests tous-les-chemins pour des exemples de classes de programmes considérées dans les Sections 3 et 4. Les expériences ont été faites sur un portable Intel Core 2 Duo avec 1Gb RAM.

La Figure 6 montre les résultats expérimentaux pour deux fonctions, `atU` de la Figure 1 et `bsearch` de la Figure 4, pour différentes valeurs du paramètre `D`. Pour chaque `D`, les colonnes "tests" et "temps" montrent le nombre de cas de test générés et le temps de la génération. Ici, PathCrawler génère exactement un cas de test pour chaque chemin faisable, donc le nombre de chemins faisables est égal au nombre de cas de test.

Nous voyons que le temps de génération avec l'outil PathCrawler pour ces fonctions a une croissance assez lente (clairement sous-exponentielle) avec le paramètre `D`. Donc, comme présenti dans la Section 3, la génération de tests tous-les-chemins reste traitable pour ces programmes avec des centaines et des milliers de variables d'entrée, et PathCrawer fournit une méthode de génération efficace pour ces programmes.

D'autre part, la Figure 7 montre les résultats pour la fonction HC de la Figure 5a sur le graphe orienté complet G_N avec N nœuds (i.e. ayant un arc de i à j pour tous i, j). La colonne "chemins" montre le nombre de chemins faisables. Ici, en présence d'alias internes, PathCrawler génère des cas de tests superflus (*cf* [6]).

Comme l'avait prédit la Section 4, le temps de génération croît très rapidement (comme la factorielle environ), et la génération de tests tous-les-chemins devient intraitable déjà pour $N > 10$. Pour des graphes incomplets G, le nombre de tests et le temps de génération sont différents, mais leur croissance reste sur-exponentielle.

Nous avons essayé un exemple similaire issu d'un logiciel industriel avec plusieurs centaines de lignes de code C (qui ne sera pas décrit ici en détail pour des raisons de propriété intellectuelle), où les entrées ont été utilisées aussi comme des indices d'un tableau à deux dimensions. Nous avons obtenu des résultats similaires : la génération de tests tous-les-chemins devient déjà intraitable pour des programmes avec environ 20 variables d'entrée.

6 Conclusion et perspectives

On dit souvent que la génération de tests tous-les-chemins est intraitable sans vraiment donner une caractérisation des programmes pour lesquels elle peut être traitable, ou une description des structures de langage qui peuvent la rendre intraitable. Il semble important de pouvoir répondre à ces questions.

Cet article traite le problème d'évaluation de complexité potentielle de la génération de tests tous-les-chemins pour diverses classes de programmes. En utilisant le résultat de Pratt sur la résolution de contraintes de différence [10], nous avons prouvé un théorème qui stipule des conditions suffisantes pour qu'une classe de programmes permette une génération de test tous-les-chemins en temps polynomial. Il montre pour la première fois que la génération de test tous-

les-chemins peut être traitable pour certains programmes rencontrés en pratique.

Nous avons également construit une réduction originale du problème du circuit Hamiltonien pour montrer que la génération de tests tous-les-chemins pour une classe de programmes plus large, où les indices de tableaux et les décalages de pointeurs peuvent dépendre des entrées, est intraitable (NP-difficile). Cela donne un exemple concret des phénomènes de programmation qui peuvent rendre la génération de tests tous-les-chemins infaisable en pratique. Nous avons vu cette situation sur un exemple industriel. Ces résultats sont accompagnés de quelques expérimentations avec l'outil PathCrawler.

Les perspectives de travail incluent une étude détaillée des effets de divers phénomènes de langages de programmation sur les contraintes produites et la complexité de la génération de tests.

L'existence d'algorithmes polynomiaux de résolution pour d'autres types de contraintes (tels que "range contraintes") [11, 13] laisse espérer d'autres résultats positifs pour la génération automatique de tests qui permettront de mieux comprendre ses limites d'application en pratique.

Certes, le test exhaustif tous-les-chemins risque d'être intraitable dans de nombreux cas, mais la génération de tests pour d'autres critères de couverture (toutes-les-instructions, toutes-les-branches, etc.) peut être plus facile. Nous espérons que ce type d'études aidera les ingénieurs validation à anticiper la calculabilité et la complexité de la génération de tests et à choisir un critère de couverture approprié pour un programme donné.

Remerciements. Merci à Sébastien Bardin, Bernard Botella, Mickaël Delahaye, Philippe Herrmann, Bruno Marre et Nicky Williams pour de fructueuses discussions.

Références

[1] Sébastien Bardin and Philippe Herrmann. Structural testing of executables. In *ICST'08*, pages 22–31, Lillehammer, Norway, April 2008.

[2] Bernard Botella, Mickaël Delahaye, Stéphane Hong-Tuan-Ha, Nikolai Kosmatov, Patricia Mouy, Muriel Roger, and Nicky Williams. Automating structural testing of C programs : Experience with PathCrawler. In *AST'09*, Vancouver, Canada, May 2009.

[3] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE : automatically generating inputs of death. In *CCS'06*, pages 322–335, Alexandria, VA, USA, November 2006.

[4] P. Godefroid, N. Klarlund, and K. Sen. DART : Directed automated random testing. In *PLDI'05*, pages 213–223, Chicago, IL, USA, June 2005.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.

[6] Nikolai Kosmatov. All-paths test generation for programs with internal aliases. In *ISSRE'08*, pages 147–156, Seattle, WA, USA, November 2008.

[7] Nikolai Kosmatov. *Artificial Intelligence Applications for Improved Software Engineering Development : New Prospects*, chapter XI : Constraint-Based Techniques for Software Testing. Advances in Intelligent Information Technologies Book Series. IGI Global, 2009.

[8] Bruno Marre and Agnès Arnould. Test sequences generation from Lustre descriptions : GATeL. In *ASE'00*, pages 229–237, Grenoble, France, September 2000.

[9] George C. Necula, Scott McPeak, S.P. Rahul, and Westley Weimer. CIL : Intermediate language and tools for analysis and transformation of C programs. In *CC'02*, pages 213–228, Grenoble, France, April 2002.

[10] V. Pratt. Two easy theories whose combination is hard. Technical report, MIT, Cambridge, Massachusetts, USA, September 1977.

[11] G. Ramalingam, Junehwa Song, Leo Joskowicz, and Raymond E. Miller. Solving systems of difference constraints incrementally. *Algorithmica*, 23(3) :261–275, 1999.

[12] K. Sen, D. Marinov, and G. Agha. CUTE : a concolic unit testing engine for C. In *ESEC/FSE'05*, pages 263–272, Lisbon, Portugal, September 2005.

[13] Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theor. Comput. Sci.*, 345(1) :122–138, 2005.

[14] Nicky Williams, Bruno Marre, and Patricia Mouy. On-the-fly generation of k-paths tests for C functions : towards the automation of grey-box testing. In *ASE'04*, pages 290–293, Linz, Austria, September 2004.

[15] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. PathCrawler : automatic generation of path tests by combining static and dynamic analysis. In *EDCC'05*, pages 281–292, Budapest, Hungary, April 2005.

[16] Zhongxing Xu and Jian Zhang. A test data generation tool for unit testing of C programs. In *QSIC'06*, pages 107–116, Beijing, China, October 2006.

[17] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4) :366–427, 1997.

Problèmes d'apprentissage de contraintes

Arnaud Lallouet[1], Matthieu Lopez[2], Lionel Martin[2]
[1]GREYC, Université de Caen, [2]LIFO, Université d'Orléans
[1]prenom.nom@info.unicaen.fr, [2]prenom.nom@univ-orleans.fr

Abstract

Il est reconnu que la création d'un modèle de réseaux de contraintes requiert une bonne expérience du domaine. Pour cette raison, des outils pour générer automatiquement de tels réseaux ont gagné en intérêt ces dernières années. Ce papier présente un système basé sur la programmation logique inductive capable de construire un modèle de contraintes à partir de solutions et non-solutions de problèmes proches. Le modèle est exprimé dans un langage mi-niveau. Nous montrons que les approches de PLI classique ne sont pas capables de résoudre cette tâche d'apprentissage et nous proposons une nouvelle approche basée sur le raffinement d'une solution appelée graine. Nous présentons des résultats expérimentaux sur des jeux de données allant des puzzles aux problèmes d'emploi du temps.

1 Introduction

La programmation par contraintes (CP) est un formalisme pour modéliser et résoudre une large gamme de problèmes de décision, des puzzles arithmétiques aux emplois du temps en passant par les problèmes de programmation de tâches industrielles. Cependant, il est reconnu par la communauté [16] que le modelage en CP requiert une connaissance approfondie pour être produit avec succès. Des problèmes majeurs pour les utilisateurs débutants sont leur connaissance très limitée sur le choix des variables, comment trouver les contraintes et comment améliorer leur modèle pour le rendre efficace. Dans ce processus, trouver les bonnes contraintes est crucial et il existe beaucoup de travaux sur la compréhension[18] et l'automatisation[4] de la tâche de modelage.

A notre connaissance, seulement le système CONACQ dans [9] et les papiers ultérieurs [4, 5] cherche à apprendre un réseau de contraintes avec un algorithme basé sur l'espace des versions. Cependant, une limitation principale de l'approche est que l'utilisateur doit fournir l'ensemble exact des variables ainsi que des solutions et des non-solutions pour son problème. Qu'un utilisateur veuille construire un modèle alors qu'il a déjà des solutions est discutable. En contraste, et d'un point de vue cognitif, il est plus intéressant pour un utilisateur d'obtenir un modèle pour un nouveau problème en ayant des solutions et non-solutions de problèmes liés. Pour illustrer, nous considérons qu'un utilisateur veut modéliser un problème d'emploi du temps scolaire en ayant seulement des solutions et non-solutions, produites à la main, pour des instances historiques des précédentes années. Malgré la généralisation à un cadre apprentissage actif[5], l'utilisateur doit fournir à CONACQ un jugement sur des solutions potentielles du problème actuel.

Certains langages de modelage comme OPL [12], Essence' [10] ou MiniZinc[15] fournissent un cadre pour modéliser des problèmes de contraintes avec un niveau d'abstraction moyen. L'utilisateur fournit des règles et des paramètres qui sont combinés dans un processus de réécriture pour générer un Problème de Satisfaction de Contraintes (CSP) adapté au véritable problème à résoudre. Apprendre une telle spécification à partir de problèmes déjà résolus (données historiques) fournirait un modèle qui pourrait être réutilisé dans un nouveau contexte avec différents paramètres. Par exemple, après la génération du problème d'emploi du temps scolaire, le modèle pourra être construit à partir des données courantes comme le nombre de cours, d'enseignants, les nouvelles salles de cours ...

Dans ce papier, nous présentons une manière d'acquérir une telle spécification utilisant la Programmation Logique Inductive (ILP). Les exemples et contre-exemples pour le concept à apprendre sont définis comme des interprétations dans un langage logique que nous appelons *langage de description* et le CSP en sortie est exprimé par des contraintes dans un *langage de contraintes*. La spécification est exprimée par des règles du premier ordre qui associent à un ensemble de prédicats dans le langage de description (corps d'une

règle), un ensemble de contraintes dans le langage de contraintes (tête d'une règle). Nous n'utilisons pas directement un langage de modelage comme Essence ou Zinc pour rester proche d'un système de règles mais les règles que nous apprenons ont un niveau d'abstraction équivalent aux langages de modelage intermédiaire comme Essence', Minizinc ou OPL. En particulier, ils permettent l'utilisation d'arythmétique et de paramètres et ils peuvent être réécrits pour générer un problème contraint de taille différente. Chercher de telles règles est un réel problème pour les techniques d'ILP puisque nos règles présentent des difficultés reconnus dans la communauté ILP telles la recherche aveugle et la traversée de "plateau". Pour dépasser ces difficultés, nous proposons une nouvelle technique basée sur une recherche bidirectionnelle consistant à progressivement réduire les bornes de l'espace de recherche. Pour cela, nous nous basons sur la structure de la saturation d'un exemple pour biaiser l'espace de recherche. La saturation est un objet bien connu en ILP permettant de connaître les différentes caractéristiques d'un exemple en fonction d'une base de connaissance.

Nous organisons ce papier de la manière suivante. Nous introduisons le langage de règles et décrivons comment un modèle peut se réécrire en CSP. Nous présentons ensuite le problème d'apprentissage sous la forme d'un problème d'ILP classique et fournissons les clés pour comprendre et reproduire notre algorithme. Nous présentons divers résultats sur des problèmes classiques de contraintes comme l'emploi du temps, l'ordonnancement d'ateliers et le classique n-reines.

2 Apprentissage de problèmes contraints

2.1 Le langage cible

Soient V un ensemble de variables et $D = (D_X)_{X \in V}$ leurs domaines. Une *contrainte* est une relation c sur un sous-ensemble de variables. Nous notons $var(c)$ les variables sur lesquelles c est définie et par $sol(c) \subseteq D^{var(c)}$ l'ensemble des tuples définissant c. Un CSP est un triplé (V, D, C), où V et D sont définis comme ci-dessus et C est un ensemble de contraintes. Un langage de modelage fournit une manière de spécifier un CSP avec une certaine abstraction. De nombreux langages donnent la possibilité d'utiliser des arguments pour paramétrer leurs modèles [10, 14]. Les paramètres sont alors fournis dans un fichier séparé. Nous présentons dans ce papier un langage de modelage en logique du premier ordre qui retient la notion de paramètres et pose des contraintes pour toutes les substitutions de variables en accord avec les paramètres. Nous ne considérons pas des objets comme les fonctions, souvent utilisées par langage de modelage destiné à un utilisateur humain. Nous appelons notre langage $\mathcal{ML}$.

Une spécification de problème contraint (CPS) dans le langage $\mathcal{ML}$ consiste en un ensemble de règles décrivant quand une contrainte doit être posée dans une instance d'un CSP. Soit T un ensemble de types. Soit $V = (V_t)_{t \in T}$ et $(Const_t)_{t \in T}$ respectivement un ensemble de variables typées et de constantes. Un terme est soit une variable soit une constante. Les prédicats ont aussi des types et sont divisés en deux ensembles disjoints P_D et P_C correspondant respectivement au corps et à la tête de la règle. Les prédicats du corps forment le *langage de description*. Ils sont utilisés pour décrire les exemples et contre-exemple et pour introduire les variables de la règle. Ils ont également une ou plusieurs déclarations de mode : pour chaque argument, on précise s'il s'agit d'une entrée ou d'une sortie du prédicat. Les entrées sont notées $+$ et les sorties $-$. Par exemple, le prédicat *sum(X,Y,Z)* avec la sémantique $X + Y = Z$ pourrait avoir le mode *sum(+,+,-)*. Les prédicats de la tête forment le *langage de contraintes* et sont utilisés pour définir des contraintes qui doivent être satisfaites si le corps est vrai. Ces prédicats sont les précurseurs des contraintes et seront transformés en contraintes pendant la phase de réécriture. Un atome est une expression de la forme $p(t_1, \ldots, t_k)$, où p est un prédicat d'arité k et $t_1, \ldots, t_k$ des termes. Un littéral est un atome ou la négation d'un atome.

La syntaxe de nos règles est la suivante :

$$
\begin{array}{rcl}
\mathit{rule} & ::= & \forall\ \mathit{variables} :\ \mathit{body}\ \rightarrow \mathit{head} \\
\mathit{variables} & ::= & \mathit{vs} \in \texttt{TYPE} \mid \mathit{variables}, \mathit{variables} \\
\mathit{vs} & ::= & \texttt{VARIABLE} \mid \mathit{vs}, \mathit{vs} \\
\mathit{body} & ::= & \texttt{BODY\_ATOM} \mid \mathit{body}\ \wedge\ \mathit{body} \\
\mathit{head} & ::= & \texttt{HEAD\_ATOM} \mid \neg\texttt{HEAD\_ATOM} \\
 & & \mid \mathit{head}\ \vee\ \mathit{head}
\end{array}
$$

La figure 2 présente quelques exemples de problèmes spécifiés avec ce langage. Nous avons volontairement enlevé les quantifications universelles de variables pour des raisons de place.

Le premier exemple correspond au problème de coloration de graphe où deux sommets adjacents ne peuvent avoir la même couleur. Le second est un problème d'emploi du temps simplifié où $timetable(L, T, R, S)$ représente un cours L enseigné par un professeur T dans une salle R pendant le créneau S. Deux cours ne peuvent avoir lieu dans la même salle s'ils ont lieu au même moment (première règle) et un enseignant ne peut enseigner deux cours différents au même moment. Le dernier problème est un problème d'organisation d'atelier. Une tâche J de type T doit être faite entre les heures B et E avec la machine M. Les machines peuvent faire tous les type de

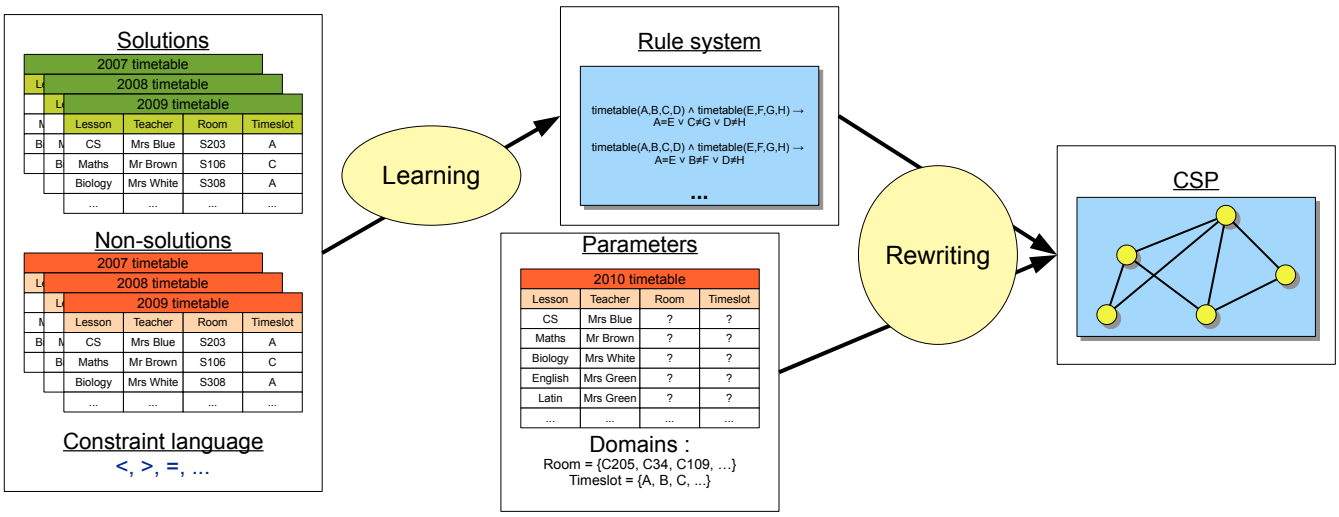


FIGURE 1 – Cadre pour l'apprentissage de problèmes contraints

Graph coloring problem

$$col(X, A) \wedge col(Y, B) \rightarrow A \neq B \ \vee \ \neg adj(X, Y)$$

Simplified school timetable

$$timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2) \rightarrow L_1 = L_2 \ \vee \ R_1 \neq R_2 \ \vee \ S_1 \neq S_2$$
$$\bigwedge$$
$$timetable(L_1, T_1, R_1, S_1) \wedge timetable(L_2, T_2, R_2, S_2) \rightarrow T_1 \neq T_2 \ \vee \ L_1 = L_2 \ \vee \ S_1 \neq S_2$$

Simplified jobshop

$$schedule(J, T, B, E, M) \rightarrow B < E$$
$$\bigwedge$$
$$schedule(J_1, T_1, B_1, E_1, M_1) \wedge schedule(J_2, T_2, B_2, E_2, M_2) \rightarrow J_1 = J_2 \ \vee \ M_1 \neq M_2 \ \vee \ B_1 > E_2 \ \vee \ E_1 < B_2$$
$$\bigwedge$$
$$schedule(J_1, T_1, B_1, E_1, M_1) \wedge schedule(J_2, T_2, B_2, E_2, M_2) \rightarrow J_1 = J_2 \ \vee \ E_1 < B_2 \ \vee \ prev(T_1, T_2)$$

FIGURE 2 – Exemples de CPS

travaux mais certains types doivent être faits avant d'autres (*prev* décrit l'ordre des types de tâches).

En contraste avec les langages de modelage classiques, la présence de disjonction rend difficile la compréhension pour un utilisateur humain. Cependant, la majorité de ces disjonctions sera supprimée lors de la compilation du CSP.

2.2 Instanciation d'un CPS

Prenons comme exemple le problème d'emploi du temps. Nous supposons que suite à la phase d'apprentissage l'utilisateur a obtenu un CPS dans le langage décrit précédemment. Pour obtenir un modèle de CSP, il doit fournir une table partiellement remplie représentant le prédicat *timetable* (voir fig. 1) et les domaines liés à son problème actuel qui sont, dans l'exemple, les salles et les créneaux. Ces tables partielles, qu'on appellera par la suite *extension partielle*, permettent de fixer les paramètres du CPS.

L'objectif de l'utilisateur est alors d'obtenir un CSP pouvant compléter cette extension partielle. Dans la figure 1, il s'agit de déterminer les salles et les créneaux où auront lieux les cours sachant que les professeurs ont déjà été choisis.

La traduction d'un CPS en CSP se décompose en deux phases. La première va compléter les extensions partielles avec des variables de CSP en leur fixant le domaine correspondant. La seconde va, pour chaque règle du CPS, chercher les substitutions possibles du corps et produire les contraintes correspondantes grâce à la tête. L'algorithme est présenté dans la figure 2.2 et est expliqué dans la suite. Il prend en paramètre un modèle CS , correspondant au CPS décrivant son problème, et les données actuelles de son problèmes : un ensemble d'extension partielle et les domaines pour chaque type.

Une extension partielle d'un prédicat p est un couple (p, E), où E est un ensemble de tuples $\langle x_1, x_2, \ldots, x_k\rangle$ définissant ce prédicat. x_i est soit une constante soit ? signifiant que l'utilisateur ne connaît pas la valeur pouvant être associée au reste du tuple. Dans la suite, on notera *ext* l'ensemble des extensions partielles fournies par l'utilisateur. La première phase (ligne 3) consiste à compléter *ext* en remplaçant les ? par des variables de CSP avec le bon domaine (donné par le prédicat).

La seconde phase consiste à produire à partir des règles du CPS les contraintes propre à l'instance de l'utilisateur. Pour cela, nous générons toutes les substitutions du corps G possible tel qu'il soit satisfait (ligne 8). Étant donné une substitution σ, le corps est satisfait si chaque atome $p(t_1, \ldots, t_k)$ de G est satisfait avec σ. L'atome $p(t_1, \ldots, t_k)$ est satisfait si $\sigma(p(t_1, \ldots, t_k))$ a un support dans *ext* ou dans le cas où p est défini intentionnellement, si $\sigma(p(t_1, \ldots, t_k))$ est valide en respect de la définition. Un problème se pose dans le cas d'un p intentionnel. Que se passe-t-il quand parmi les entrées de p, il y a une variable de CSP ? Dans ce cas, nous générons des variables de CSP auxiliaires en sorties. Par exemple, considérons l'atome suivant $sum(X, Y, Z)$ et la substitution $\{X/2, Y/v_1, Z/?\}$ avec pour domaine de la variable v_1 : $[2..6]$. Dans ce cas, la substitution est complétée avec Z/v_2 avec comme domaine pour v_2 : $[-\infty.. + \infty]$. Une fois ces substitutions calculées, l'algorithme va substituer la tête de la règle pour produire les contraintes (ligne 21). Les contraintes produites sont alors des disjonctions de contraintes. Cependant, si on prend l'exemple des emplois du temps, les variables d'un certain nombre de contraintes seront déjà fixées. Par exemple, prenons la substitution $\{L_1/Latin,\ T_1/Mrs\ Green,\ R_1/v_1,\ S_1/v_2,\ L_2/English,\ T_2/Mrs\ Green,\ R_2/v_3,\ S_2/v_4\}$ de la seconde règle. La contrainte produite par cette substitution est alors $Mrs\ Green \neq Mrs\ Green \vee Latin = English \vee v_2 \neq v_4$. Elle peut être immédiatement simplifiée en $v_2 \neq v_4$. Ce n'est cependant pas le cas à chaque fois ; si cette substitution est appliquée à la première règle, il restera une disjonction.

Algorithm : TRANSLATE$(CS, ext, domains)$

```
1.  //Complete the partial extension
2.  // with CSP variables with domains
3.  ext ← COMPLETE(ext, domains)
4.  //Initialize the variables set
5.  //with these in ext
6.  vars ← GETVAR(ext)
7.  constraints ← ∅
8.  for each G → C ∈ CS
9.  // Generate all substitution of the body
10.    subst ← GENERATEALLSUBST(G, ext)
11.    for each σ ∈ subst
12. //If there are atoms with no matching
13. // in ext,it adds aux. variables
14. // and the constraint
15.       for each atoms p(t_1, ..., t_k) ∈ G
16.       such that (p, _) ∉ ext
17.          vars.add(GETVAR(σ(p(t_1, ..., t_k))))
18.          constraints.add(σ(p(t_1, ..., t_k)))
19. //It adds the constraint
20. //corresponding to the head
21.       constraint.add(σ(C))
22. //Finally, it returns the CSP
23. return CSP(vars, domains, constraints)
```

FIGURE 3 – Traduction d'un CPS en CSP

2.2.1 Tâche d'apprentissage

Dans cette partie, nous décrivons la tâche d'apprentissage consistant à induire à partir de solutions et non-solutions d'un problème, un CPS. Le choix de notre langage était motivé par l'utilisation des techniques de programmation logique inductive (ILP). Nous présentons donc cette tâche comme un problème d'apprentissage relationnel classique.

Le but est donc d'apprendre la définition d'un CPS, noté CS qui discriminera correctement les solutions E^+ des non-solutions E^-. On dira que CS couvre un exemple e en respect avec une base de connaissance $\mathcal{B}$, noté $(e \cup \mathcal{B}) \models CS$, si pour toute substitution σ des variables de CS vers les constants de e, $\sigma(CS)$ est vrai. Un exemple est rejeté s'il n'est pas couvert. Une définition est satisfaisante si elle couvre tous les exemples de E^+ et rejette ceux de E^-. Pour résumer, nous pouvons formaliser notre problème d'apprentissage de la manière suivante : étant donnés deux ensembles E^+ et E^-, et une base de connaissance $\mathcal{B}$, trouver une définition pour CS telle que :

- $\forall e^+ \in E^+ : (e^+ \cup \mathcal{B}) \models CS$
- $\forall e^- \in E^- : (e^- \cup \mathcal{B}) \not\models CS$

Le cadre traditionnel d'apprentissage en ILP consiste à apprendre une définition composée de clauses. Un exemple est alors couvert s'il existe une substitution couvrant au moins une règle. Cette définition correspond à des formes normales disjonctives (DNF). Or, un CPS est en forme normale conjonctive. En passant au concept négatif, c'est à dire en cherchant la négation de CS rejetant les exemples de E^+ et couvrant ceux de E^- (autrement dit, en inversant les ensembles positifs et négatifs), on se ramène à l'apprentissage d'une définition en DNF et ainsi, pouvoir utilisé les techniques de l'état de l'art. Par manque de place, nous ne détaillerons pas cette transformation dans ce papier. Le lecteur intéressé pourra consulter [13]. Ainsi, dans la suite de ce papier, nous chercherons à apprendre des définitions pour la négation du problème cible. Le CPS pourra être obtenu en prenant la négation de la définition apprise.

Nous nous intéressons dans ce papier à la famille d'algorithme *separate-and-conquer*[11]. Ces algorithmes cherchent à apprendre un ensemble de règles jusqu'à ce que tous les exemples soient discriminés. Dans la suite, nous ne nous intéressons plus qu'à l'apprentissage d'une règle de la définition. Les caractéristiques principales d'un système de recherche d'une règle sont données par la définition d'un espace de recherche, un opérateur de raffinement et une heuristique capable de choisir, à chaque étape, le meilleur candidat parmi les raffinements possibles. Dans ce contexte, l'opérateur de raffinement joue un rôle essentiel et les approches sont usuellement séparées en deux catégories : d'une part, les recherches *top-down* qui commencent avec une clause générale et qui construisent des spécialisations et de l'autre, les recherches *bottom-up* qui commencent avec une clause spécifique et puis la généralisent. Dans la plupart de ces cas, l'opérateur de raffinement permet d'organiser l'espace de recherche comme un treillis, commençant soit par une clause générale nommée *top* ($\top$) soit par une spécifique nommée *bottom* ($\bot$).

3 Saturation Minimale

Nous avons observé que la majorité des systèmes d'ILP classiques échouait sur l'apprentissage de nos CPS ou finissait par trouver une solution après une recherche exhaustive. Ces échecs peuvent être expliqués par les difficultés contenues dans les problèmes considérés qui sont bien connus dans la communauté ILP. De récents travaux sur la transition de phase en ILP[17] et sur la recherche aveugle ou la traversée de "plateau"[1] montrent de réelles difficultés sur certaines classes de problèmes. C'est notamment le cas pour l'apprentissage de CPS. Dans la suite de notre article, nous proposons une recherche bidirectionnelle où l'idée principale est de réduire progressivement l'espace de recherche. À chaque étape, notre espace de recherche est défini par un couple d'hypothèses $(H_{\top_i}, H_{\bot_i})$ limitant l'espace. Notre opérateur de raffinement produit de nouvelles limites $(H_{\top_{i+1}}, H_{\bot_{i+1}})$ où $H_{\top_{i+1}}$ est plus spécifique que $H_{\top_i}$ et $H_{\bot_{i+1}}$ est plus générale que $H_{\bot_i}$. La recherche s'arrête quand $H_{\top_i} = H_{\bot_i}$ qui correspond à la règle apprise.

Notre algorithme est basé sur la structure de la saturation, une opération usuelle en ILP. La saturation est un ensemble de littéraux clos apportant un maximum d'information sur un exemple. Elle est organisée en plusieurs couches ordonnées. Un littéraux appartient à une couche k si les termes nécessaires à son introduction ont été introduits par des littéraux des couches précédentes. Nous commençons par formaliser cet objet avant de présenter notre algorithme de recherche de règle.

3.1 Saturation

Avant de définir la saturation, nous introduisons quelques notations : étant donné un littéral l, nous notons $input(l)$ et $output(l)$ les ensembles des termes associés respectivement aux entrées et aux sorties du littéral l. Nous utilisons la même notation pour une formule f en notant $input(f) = \bigcup_{l \in f} input(l)$

et $output(f) = \bigcup_{l \in f} output(l)$. De plus, l'ensemble des termes apparaissant dans un littéral l est noté $terms(l)$. $vars(f)$ est une opération qui consiste à substituer toutes les constantes d'une formules f par des variables telles que les constantes sont remplacées par des variables distinctes. Comme indiqué précédemment, la saturation d'un exemple est un ensemble de littéraux qui peut être structuré en niveaux : nous notons $layer(l)$ le niveau dans lequel apparaît le littéral k[1]. L'ensemble des littéraux appartenant au niveau k est noté $litsOfLayer(k)$ et est défini par : $litsOfLayer(k) = \{l \mid layer(l) = k\}$. Enfin, étant donnés un ensemble d'exemples positifs et un ensemble d'exemples négatifs, nous notons $p(f)$ et $n(f)$ le nombre d'exemples respectivement **p**ositifs et **n**égatifs couverts par la formule f.

Saturation d'un exemple Dans cette partie, nous présentons une formalisation de l'opérateur de saturation, consistant à collecter tous les littéraux clos qui décrivent un exemple ou qui sont liés à sa description. Cette construction sera illustrée sur l'exemple suivant, inspiré du classique exemple des trains de Michalski :

Exemple

Dans cet exemple, chaque train est composé d'un certain nombre de wagons, chaque wagon est décrit par son nombre de roues et sa taille. De plus chaque wagon transporte un certain nombre d'objets de différentes formes géométriques. La situation suivante décrit deux trains correspondant respectivement à un exemple positif et un exemple négatif du même concept cible :

$t_1^+ : \{has\_car(t_1^+, c_1), has\_car(t_1^+, c_2), has\_car(t_1^+, c_3),$
$wheels(c_1, 2), wheels(c_2, 3), wheels(c_3, 5),$
$long(c_1), long(c_2), long(c_3), load(c_1, circle, 3),$
$load(c_2, circle, 6), load(c_3, triangle, 10)\}$

$t_2^- : \{has\_car(t_2^-, c_1), has\_car(t_2^-, c_2), has\_car(t_2^-, c_3),$
$wheels(c_1, 1), wheels(c_2, 4), wheels(c_3, 3),$
$long(c_1), long(c_2), short(c_3), load(c_1, circle, 4),$
$load(c_1, rectangle, 2), load(c_2, rectangle, 5)$
$load(c_2, circle, 6), load(c_3, circle, 2)\}$

Nous considérons ici que les prédicats utilisés sont associés aux modes suivants : $has\_car(+,-)$, $shape(+,-)$, $wheels(+,-)$ et $load(+,-,-)$. Si on suppose que la définition cible caractérise les trains ayant au moins deux wagons transportant des objets de même forme et que l'un des deux a à la fois plus de roues et plus d'objets de la forme commune, alors une définition possible sous forme de clause est :

$goodtrain(T) : -has\_car(T, C_1), has\_car(T, C_2),$
$C_1 \neq C_2, wheels(C_1, W_1), wheels(C_2, W_2),$
$W_1 < W_2, load(C_1, O, L_1), load(C_2, O, L_2),$
$L_1 < L_2$

La saturation d'un exemple est obtenue en ajoutant autant de littéraux clos que possible à partir de la base de connaissance, en imposant que tous les littéraux soient connectés à l'exemple. La saturation est construite à partir des informations sur les domaines et sur les modes des prédicats. Il s'agit d'un ensemble de littéraux qui est structuré en niveaux : un littéral l appartient au niveau k si ses termes correspondant à des positions d'entrée ($input(l)$) apparaissent dans des littéraux de niveaux inférieurs et que au moins l'un de ces littéraux appartient au niveau $k-1$. Cette opération peut produire un ensemble infini, par conséquent elle est paramétrée par une profondeur maximale, noté i, correspondant au niveau maximum considéré.

Pour définir la saturation, nous caractérisons les différents niveaux qui la composent : le niveau k de saturation, noté $sat_k(S_l)$, est construit récursivement à partir de l'ensemble S_l de tous les littéraux des niveaux inférieurs à k :

$$sat_k(S_l) = \{l \mid input(l) \subseteq termsS_l \\ \wedge\ input(l) \cap output(litsOfLayer(k-1)) \neq \emptyset\}$$

Nous pouvons maintenant définir l'ensemble de tous les littéraux de niveau supérieur ou égal à k (mais bornés par i) à partir de l'ensemble S_l des littéraux introduits dans les niveaux précédents. Nous notons $sat(S_l, k, i)$ cet ensemble, il est défini par :

$$sat(S_l, k, i) = sat_k(S_l) \cup sat(S_l, k+1, i) \quad k \leq i$$
$$sat(S_l, k, i) = \emptyset \quad k > i$$

Finalement, la saturation de l'exemple e pour le concept cible p, avec une profondeur maximum i peut être écrite :

$$sat(p(e), i) = sat(\{p(e)\}, 1, i)$$

Exemple (cont.)

Nous considérons que nous disposons d'une connaissance du domaine décrivant les prédicats $\neq$ et $<$. Le prédicat $\neq$ permet de comparer d'une part les wagons et d'autre part les formes géométriques, ce prédicat a pour mode $\neq (+,+)$. Le prédicat $<$ permet de comparer les nombres de roues ou les nombres d'objets, il a pour mode $<(+,+)$. La saturation de l'exemple t_1^+ avec une profondeur maximale $i = 3$ est organisée en niveaux de la manière suivante :

1. La notion de niveau est relative à un exemple et sa saturation. Puisque la construction d'une clause est réalisée à l'aide d'un exemple graine, dans la suite, pour chaque notion qui est liée à un exemple ou sa saturation, il s'agira implicitement de l'exemple graine ou de sa saturation afin de ne pas alourdir les notations.

Niveau	*Littéraux*
0	$goodtrain(t_1^+)$
1	$has\_car(t_1^+, c_1), has\_car(t_1^+, c_2),$ $has\_car(t_1^+, c_3)$
2	$wheels(c_1, 2), wheels(c_2, 3), wheels(c_3, 4),$ $long(c_1), long(c_2), long(c_3),$ $load(c_1, circle, 3), load(c_2, circle, 5),$ $load(c_3, triangle, 10),$ $c_1 \neq c_2, c_1 \neq c_3, c_2 \neq c_3$
3	$2 < 3, 2 < 4, triangle \neq circle,$ $3 < 5, 3 < 10, 2 < 5, 2 < 10, 5 < 10$

3.2 Apprentissage de règles par saturation minimale

Dans cette section, nous présentons notre méthode d'apprentissage. À chaque étape de la recherche, la clause en construction est représentée par un ensemble de littéraux. L'espace de recherche représentant l'ensemble des clauses qui peuvent être apprises est organisé en un treillis borné par deux clauses notées $\top$ et $\bot$, et muni d'une relation d'ordre partiel permettant de comparer la généralité entre 2 clauses. L'ordre partiel considéré ici est l'inclusion, une clause s_1 est plus générale qu'une clause s_2 si s_1 est un sous-ensemble de s_2. La recherche est guidée par un exemple positif appelé "graine" et noté s dans la suite ; en notant p le prédicat cible, nous définissons $\top = \{p(s)\}$ et $\bot = sat(\top, i)$ où i est le niveau maximum de saturation. Cet espace de recherche est usuellement utilisé dans les algorithmes de l'état de l'art. En supposant que cette clause $\bot$ rejette tous les exemples négatifs (elle couvre au moins un exemple positif, la graine), l'espace de recherche contient ainsi au moins une règle discriminante, rejetant les exemples négatifs. Notre objectif est de découvrir une règle qui soit meilleure que $\bot$, dans le sens où elle couvre un maximum d'exemples positifs, tout en rejetant les exemples négatifs.

Notre algorithme d'apprentissage est basé sur une recherche bi-directionnelle qui raffine progressivement les hypothèses $H_\top$ et $H_\bot$ en conservant à chaque étape la relation $H_\bot = sat(H_\top, i)$.

Cette section est organisée en deux parties : nous présentons d'abord notre opérateur de raffinement puis l'algorithme d'apprentissage, basé sur une itération d'étapes de raffinement.

Opérateur de raffinement Cet opérateur recherche, dans une couche k, un ensemble de littéraux à ajouter à la clause en construction, $H_\top$, telle que la clause saturée correspondant $H_\bot$ discrimine les exemples positifs des négatifs. Pour déterminer cet ensemble de littéraux, nous recherchons un ensemble minimal par une recherche en largeur.

Afin de formaliser l'opérateur de raffinement, nous commençons par définir l'ensemble des couples candidats $(H'_\top, H'_\bot)$ obtenus par raffinements à partir de l'hypothèse en cours de construction $(H_\top, H_\bot)$. Cet ensemble de candidats est noté $candidates(H_\top, H_\bot, k, i)$, où k représente la couche à l'intérieur de laquelle le raffinement est recherché et i correspond à la profondeur maximale :

$$candidates(H_\top, H_\bot, k, i) = \{(H'_\top, H'_\bot) | H'_\top = H_\top \cup S_k \wedge S_k \subseteq litsOfLayer(k) \wedge H'_\bot = sat(H'_\top, k, i) \wedge n(H'_\bot) = 0\}$$

Cet ensemble est constitué de couples $(H'_\top, H'_\bot)$ où $H'_\top$ est plus spécifique que $H_\top$ et $H'_\bot$ est plus général que $H_\bot$ et pour lesquels $H'_\bot$ rejette tous les exemples négatifs.

Pour choisir un raffinement parmi l'ensemble des candidats, nous sélectionnons celui pour lequel la taille de S_k est minimum et s'il existe plusieurs raffinements possibles, nous choisissons celui qui couvre le plus grand nombre d'exemples positifs. Le nombre de candidats est donc exponentiel avec la taille de $litsOfLayer(k)$. Afin de ne pas les générer tous, nous recherchons le plus petit par une recherche en largeur, l'algorithme de raffinement peut se résumer ainsi ;

Algorithm : REFINE$(H_\top, H_\bot, k, i)$

```
1.  //To search the smallest
2.  //subset of litsOfLayer(k)
3.  for j from 0 to |litsOfLayer(k)|
4.    Scandidates = ∅
5.  //to enumerate subset
6.  // of layer k of size j
7.    for each subset S_k of litsOfLayer(k)
8.     such that |S| = j
9.       H'_⊤ = H_⊤ ∪ S_k
10.        H'_⊥ = sat(H'_⊤, k, i)
11.       if n(H'_⊥) = 0 then
12.         Scandidates.add((H'_⊤, H'_⊥))
13. //if there exist at less one candidate,
14. //the algorithm returns one covering
15. //the maximum of positive examples
16.   if Scandidates ≠ ∅ then
17.     return argmax_{(H'_⊤,H'_⊥)∈Scandidates} p(H'_⊥)
18. //if none candidate is found,
19. //it returns the same pair
20. return (H_⊤, H_⊥)
```

Exemple (cont.)

Pour illustrer notre opérateur de raffinement, supposons que l'hypothèse en cours de construction soit donnée par le couple $(H_\top, H_\bot) = (vars(\{goodtrain(t_1^+)\}), vars(sat(\{goodtrain(t_1^+)\}, 3))$

et que le raffinement soit produit par $\textsc{Refine}(H_\top, H_\bot, 1, 3)$.

L'algorithme commence par rechercher un ensemble de taille 1 dans le premier niveau. Tous les raffinements conduisent à un même couple hypothèse (a renommage de variables près) :

Niveau	*Littéraux*
0	$\mathbf{goodtrain(Vt_1^+)}$
1	$\mathbf{has\_car(Vt_1^+, Vc_1)}$
2	$wheels(Vc_1, VW2), long(Vc_1),$ $load(Vc_1, Vcircle, VL3),$
3	

où $H'_\top$ *apparaît en gras et* $H'_\bot$ *contient tous les littéraux. Ce couple n'est pas acceptable puisque* $H'_\bot$ *couvre l'exemple négatif* t_2^-.

Si nous considérons maintenant les sous-ensembles de taille 2, nous obtenons les couples suivants (à renommage de variables près) :

Niveau	*Littéraux*
0	$\mathbf{goodtrain(Vt_1^+)}$
1	$\mathbf{has\_car(Vt_1^+, Vc_1)}, \mathbf{has\_car(Vt_1^+, Vc_2)}$
2	$wheels(Vc_1, VW2), wheels(Vc_2, VW3),$ $load(Vc_1, Vcircle, VL3),$ $load(Vc_2, Vcircle, VL5), long(Vc_1),$ $long(Vc_2), Vc_1 \neq Vc_2$
3	$VW2 < VW3, VL3 < VL5$

Niveau	*Littéraux*
0	$\mathbf{goodtrain(Vt_1^+)}$
1	$\mathbf{has\_car(Vt_1^+, Vc_1)}, \mathbf{has\_car(Vt_1^+, Vc_3)}$
2	$wheels(Vc_1, VW2), wheels(Vc_3, VW4),$ $long(Vc_1), long(Vc_3),$ $load(Vc_1, Vcircle, VL3),$ $load(Vc_3, Vtriangle, VL10), Vc_1 \neq Vc_3$
3	$VW2 < VW4, Vtriangle \neq Vcircle,$ $VL3 < VL10$

Seule la première définition rejette tous les exemples négatifs, par conséquent c'est le couple correspondant qui sera retourné par l'algorithme de raffinement.

Apprentissage d'une règle Une règle intéressante est une règle qui rejette tous les exemples négatifs et qui couvre un maximum d'exemples positifs. Dans notre recherche bi-directionnelle, nous construisons par raffinements successifs des couples d'hypothèses $(H_\top, H_\bot)$, qui convergent vers un couple vérifiant $H_\top = H_\bot$. Afin de l'obtenir, le raffinement est opéré niveau par niveau, en commençant par le niveau $k = 1$ et en finissant par $k = i$ où i est le niveau maximum. Nous initialisons cette recherche par un couple correspondant à un exemple graine et à sa saturation, où toutes les constantes sont remplacées par des variables. L'algorithme suivant décrit la recherche d'une règle définissant un concept cible p, où l'espace de recherche est construit à partir d'un exemple graine s.

Algorithm : $\textsc{LearnRule}(p, s, i)$

```
1. H⊤ = {p(s)}
2. H⊥ = vars(sat(p(s), i))
3. for each layer k from 1 to i
4.     (H⊤, H⊥) = Refine(H⊤, H⊥, k, i)
5. return H⊤
```

4 Expériences

Pour illustrer l'efficacité de notre approche, nous avons produit des jeux de données sur des problèmes classiques. Pour cela, nous avons généré, pour les problèmes donnés en exemple à la section 2.1, un ensemble de solutions et de non-solutions. Pour produire un exemple positif, nous avons choisi des tailles aléatoires pour les différents ensembles (ex. pour la coloration de graphe, les nombres de sommets et de couleurs), puis résolu le CSP correspondant avec une heuristique aléatoire. Pour un exemple négatif, nous avons procédé d'une manière équivalente à la différence que les contraintes sont relâchées et que l'on s'assure qu'il y est au moins une contrainte non-satisfaite.

Pour évaluer notre méthode, nous nous sommes comparés aux seuls systèmes d'ILP classiques ayant réussi à trouver une définition. Seulement Propal (la version décrite dans [2], plus rapide que celle de [3]) et Aleph[19] ont réussi. Aleph est un système offrant de nombreuses possibilités de configurations. La première que nous avons considérée, que nous appellerons Aleph1, correspond à un parcours en largeur du treillis limité à l'exploration de 200 000 nœuds de recherche visités et à une liste ouverte infinie. La seconde, appelée Aleph2, diffère seulement sur la stratégie de recherche utilisée où nous avons considéré la recherche heuristique, où nous avons testé chaque heuristique implémenté dans Aleph avec des résultats similaires. Nous avons implémenté un prototype pour notre algorithme et la figure 4 présente les résultats obtenus avec ces systèmes. Pour calculer la précision du CPS appris (troisième colonne), nous avons généré de nouveaux exemples et calculé le rapport d'exemples correctement discriminés (couverts pour les exemples positifs, rejetés pour les négatifs) sur le nombre total d'exemples. Plus le concept cible est compliqué, plus Propal et Aleph2 trouvent des CPS incorrectes. Pour les n-reines, nous avons stoppé Propal après 10 heures de calcul. Cela illustre la difficulté que rencontre les approches top-down quand la recherche est aveugle. Aleph1 réussit sur tous les jeux de donnés. Cependant, il nécessite un important temps de calcul pour les jeux de données compliqués. Notre méthode réussi

sur tous les jeux de données : elle trouve des CPS précis en peu de temps. Même si le sens est identique, les règles apprises ne sont cependant pas toujours exactement celles attendus. Par exemple, pour les n-reines, la contrainte apprise pour la diagonale est :

$$position(Q1, X1, Y1) \wedge\ position(Q2, X2, Y2)$$
$$\wedge\ ecart(Y1, Y2, V1) \wedge\ ecart(Y2, Y2, V2)$$
$$\rightarrow\ Q1 = Q2 \vee\ V2 \neq V1$$

Notre prototype et nos jeux de données peuvent obtenus en contactant par e-mail les auteurs.

5 Conclusion

Nos travaux cherchent à éviter les limitations rencontrées avec CONACQ [4]. À notre connaissance, CONACQ et notre système sont les seules études concernant l'acquisition de CSP. Dans [7], Bessière et al. propose une méthode pour automatiquement générer des points de vue à partir d'exemples. Cependant, aucune méthode n'est donnée pour générer les contraintes sur ces variables. Alors que l'acquisition n'est pas très étudiée, plusieurs travaux existent sur la découverte de contraintes impliquées ou rendondantes [8, 6]. Dans ce cas, la tâche d'apprentissage consiste à apprendre seulement à partir d'exemple positifs puisque la discrimination des solutions et non-solutions est déjà assurée par le modèle. Ces approches sont complémentaires à la notre et les CPS, et les CSP générés à partir d'eux, gagneraient à être reformulés pour être plus efficaces.

Dans ce papier, nous avons présenté un cadre pour obtenir automatiquement un modèle abstrait de CSP. Notre approche, basée sur la programmation logique inductive, nécessite des exemples que l'utilisateur considère comme des solutions et non-solutions de problèmes liés. Ensuite, il obtient une spécification (un CPS) qui peut être traduit par la suite en CSP en ajoutant les données de son véritable problème. Notre apport principal concerne la tâche d'apprentissage. Même si nos CPS sont écrits en logique du premier ordre, les systèmes classiques échouent lors de l'apprentissage. Les problèmes de recherche aveugle et de transition de phase rendent notre classe de problèmes difficiles à résoudre. Nous avons donc développé un nouvel algorithme de recherche bidirectionnel basé sur le raffinement progressif des bornes de l'espace de recherche. Les résultats sont très encourageants et ouvrent la perspective d'enrichir le langage pour permettre d'exprimer de nouveaux problèmes.

Références

[1] Érick Alphonse and Aomar Osmani. On the connection between the phase transition of the covering test and the learning success rate in ilp. *Machine Learning*, 70(2-3) :135–150, 2008.

[2] Érick Alphonse and Céline Rouveirol. Lazy propositionalisation for relational learning. In Werner Horn, editor, *ECAI*, pages 256–260. IOS Press, 2000.

[3] Érick Alphonse and Céline Rouveirol. Extension of the top-down data-driven strategy to ilp. In Stephen Muggleton, Ramón P. Otero, and Alireza Tamaddoni-Nezhad, editors, *ILP*, volume 4455 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2006.

[4] Christian Bessière, Remi Coletta, Frédéric Koriche, and Barry O'Sullivan. Acquiring constraint networks using a sat-based version space algorithm. In *AAAI*. AAAI Press, 2006.

[5] Christian Bessière, Remi Coletta, Barry O'Sullivan, and Mathias Paulin. Query-driven constraint acquisition. In Veloso [20], pages 50–55.

[6] Christian Bessière, Remi Coletta, and Thierry Petit. Learning implied global constraints. In *IJCAI*, pages 44–49, 2007.

[7] Christian Bessiere, Joil Quinqueton, and Gilles Raymond. Mining historical data to build constraint viewpoints. In *Proceedings CP'06 Workshop on Modelling and Reformulation*, pages 1–16, 2006.

[8] John Charnley, Simon Colton, and Ian Miguel. Automatic generation of implied constraints. In *ECAI*, pages 73–77, 2006.

[9] Remi Coletta, Christian Bessière, Barry O'Sullivan, Eugene C. Freuder, Sarah O'Connell, and Joël Quinqueton. Semi-automatic modeling by constraint acquisition. In Francesca Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 812–816. Springer, 2003.

[10] Alan M. Frisch, Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The design of essence : A constraint language for specifying combinatorial problems. In Veloso [20], pages 80–87.

[11] Johannes Furnkranz. Separate-and-conquer rule learning. *Artificial Intelligence Review*, 13 :3–54, 1999.

[12] P. Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

benchmark	Propal			Our algorithm		
	# learned rules	time (s)	acc.	# learned rules	time (s)	acc.
Graph coloring	1	0	100%	1	0.17	100%
School timetable	3	11	98,33%	2	0.69	100%
Job-shop	6	103	87,78%	5	7.37	100%
N-queens	-	-	-	3	29.11	100%
	Aleph1			Aleph2		
Graph coloring	1	0.24	100%	1	0.14	100%
School timetable	1	1.24	100%	1	0.31	100%
Job-shop	3	1051.03	100%	6	1130.88	96%
N-queens	3	489.49	100%	3	4583.84	61.67%

FIGURE 4 – Expérience sur des problèmes classiques

[13] Wim Van Laer. *From Propositional to First Order Logic in Machine Learning and Data Mining*. PhD thesis, Katholieke Universiteit Leuven, June 2002.

[14] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, Maria Garcia de la Banda, and Mark Wallace. The design of the zinc modelling language. *Constraints*, 13(3) :229–267, 2008.

[15] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc : Towards a standard cp modelling language. In *CP*, pages 529–543, 2007.

[16] Jean-Francois Puget. Constraint programming next challenge : Simplicity of use. In Mark Wallace, editor, *International Conference on Constraint Programming*, volume 3258 of *LNCS*, pages 5–8, Toronto, CA, 2004. Springer. Invited paper.

[17] Alessandro Serra, Attilio Giordana, and Lorenza Saitta. Learning on the phase transition edge. In *IJCAI*, pages 921–926, 2001.

[18] Barbara M. Smith. Modelling. In T. Walsh F. Rossi, P. van Beek, editor, *Handbook of Constraint Programming*, chapter 11, pages 377–406. Elsevier, 2006.

[19] Ashwin Srinivasan. *A learning engine for proposing hypotheses (Aleph)*.

[20] Manuela M. Veloso, editor. *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2007.

Stratégies Dynamiques pour la Génération de Contre-exemples*

Le Vinh Nguyen[1] **Hélène Collavizza**[1] **Michel Rueher**[1]
Samuel Devulder[2] **Thierry Gueguen**[2]

[1] Université de Nice-Sophia Antipolis/CNRS, 2000, route des Lucioles - Les Algorithmes
BP 121 - 06903 Sophia Antipolis Cedex - France
[2] Geensys, 120 Rue René Descartes, 29280 Plouzané - France
{lvnguyen, helen, rueher}@polytech.unice.fr
{samuel.devulder, thierry.gueguen}@geensys.com

Résumé

La vérification de propriétés de sûreté est incontournable dans le processus de validation des logiciels critiques. Lorsque les outils de vérification formelle échouent à démontrer certaines propriétés, la recherche de contre-exemples des propriétés violées est un point crucial, notamment pour les programmes complexes pour lesquels les jeux de test sont difficiles à générer. Nous proposons dans cet article différentes stratégies dynamiques de génération de jeux de test qui violent une post-condition de programmes écrits en C ou en Java. Notre approche est basée sur la génération dynamique d'un système de contraintes lors de l'exploration du graphe de flot de contrôle du programme. Ces stratégies ont été évaluées sur des exemples académiques et sur des applications réelles. Les expérimentations sur un contrôleur industriel qui gère les feux de clignotants d'une voiture et sur une implémentation d'un système de détection de collisions du trafic aérien ont montré que notre système est bien plus performant qu'un outil de model checking de premier plan comme CBMC ou qu'un système de génération de jeux de test à base de contraintes comme Euclide.

Abstract

Checking safety properties is mandatory in the validation process of critical software. When formal verification tools fail to prove some properties, testing is necessary. Generation of counterexamples violating some properties is therefore an important issue, especially for tricky programs the test cases of which are very difficult to compute. We propose in this paper different constraint based dynamic strategies for generating structural test cases that violate a post-condition of C or JAVA programs. These strategies have been evaluated on standard benchmarks and on real applications. Experiments on a real industrial Flasher Manager controller and on the public available implementation of the Traffic Collision Avoidance System (TCAS) show that our system outperforms state of the art model checking tools like CBMC or constraint based test generation systems like Euclide.

1 Introduction

La vérification de programmes fait appel à des techniques de preuves formelles, des techniques de tests fonctionnels et structurels, des revues et analyses manuelles de code. Lorsque les outils de vérification formelle échouent à prouver les post-conditions d'un programme, ce travail est souvent fait à la main. De plus, la vérification formelle se base sur un modèle abstrait qui ne capture pas toujours toutes les fonctionnalités de l'implémentation, et qui est souvent de taille limitée. Aussi, le test est toujours nécessaire, et la génération automatique de contre-exemples d'une propriété reste un point crucial dans le processus de vérification des programmes. En particulier, pour les applications temps réel, la génération de cas de test pour des périodes de temps réalistes reste un problème difficile.

Dans cet article, nous proposons de nouvelles stratégies dynamiques pour générer des cas de test structurel qui violent une post-condition de programmes écrits en C ou en Java[1]. Notre approche repose sur les observa-

*Ce travail a été partiellement soutenu par le projet CAVERN (ANR-07-SESUR-003), ainsi que par le projet TESTEC (ANR-07 TLOG 022).

[1]Nous traitons actuellement un sous-ensemble de ces lan-

tions suivantes :

- lorsque le programme est sous une forme DSA[2], un chemin d'exécution peut être construit de manière dynamique. En d'autres termes, il n'est pas nécessaire d'explorer le GFC (Graphe de Flot de Contrôle) du programme de manière séquentielle (descendante ou ascendante), mais les blocs compatibles sur un chemin peuvent être collectés de manière non déterministe.
- une partie significative du programme peut n'avoir aucun impact sur une propriété donnée[3]. En particulier, les parties du programme qui ne contiennent pas de variables de la post-condition, ni de variables liées aux variables de la post-condition, peuvent être ignorées quand on cherche un contre-exemple.

Dans l'approche préconisée, nous utilisons d'abord une technique classique, connue sous le nom de *slicing*, pour supprimer de façon statique les blocs du programme où n'apparaissent pas les variables pertinentes pour la propriété considérée. Nous explorons ensuite ce GFC réduit en suivant une variable particulière de la post-condition. Toutefois, il ne s'agit pas d'un parcours arrière chronologique : nous construisons les chemins de manière dynamique pour détecter au plus tôt des chemins inconsistants, c'est à dire sans examiner l'ensemble des noeuds desdits chemins. Par contre, en cas d'erreur, tous les noeuds correspondant au chemin du contre exemple doivent être examinés.

Concrètement la stratégie *DPVS* (Dynamic Postcondition-Variables driven Strategies) travaille avec un ensemble de contraintes S et une file de variables. Nous avons essayé différentes gestions de la file des variables : FIFO, LIFO et TAS (trié par rapport au niveau des variables dans le GFC). Les meilleurs résultats ont été obtenus avec les stratégies LIFO et TAS. Nous avons aussi essayé différentes combinaisons de solveurs de contraintes sur les domaines finis (CP) et de solveurs de programmation linéaire (LP). Aucune de ces combinaisons n'est systématiquement meilleure qu'une autre mais les résultats sont assez prévisibles : lorsque les domaines des variables sont petits, CP se comporte mieux alors que LP est généralement meilleur si le programme a beaucoup d'expressions linéaires. Nous avons évalué *DPVS* sur des benchmarks standards ainsi que sur deux applications réelles :

- l'implémentation d'un système de gestion des collisions de trafic aérien (TCAS) ;
- une application réelle industrielle, un contrôleur qui gère certaines fonctions liées aux feux clignotants d'un véhicule, appelée *Clignotant* dans la suite.

Sur ces applications réelles, *DPVS* surpasse des outils de model checking de premier plan comme CBMC ainsi qu'un système de génération de test basé sur les contraintes comme Euclide.

1.1 Etat de l'art

Les outils récents de génération de tests structurels orientés chemins (par exemple, PathCrawler[20], Dart[13], CUTE[4]) sont basés sur la sélection de chemin, l'exécution symbolique et l'exécution concolitique. Ils sont très efficaces pour générer des jeux de test qui garantissent une certaine couverture, mais ils n'ont pas été conçus pour trouver les données de test qui violent certaines propriétés ; ils pourraient néanmoins le faire en effectuant une recherche exhaustive, mais ceci serait très coûteux.

Les model-checkers bornés [11], comme CBMC[5], peuvent trouver des contre-exemples de propriétés de programmes C[12, 17]. Ces outils transforment le programme et la post-condition en une formule conditionnelle et utilisent les solveurs SAT ou SMT pour prouver que cette formule est satisfaite ou pour trouver un contre-exemple.

La Programmation Logique par Contraintes (CLP) permet d'implémenter des techniques d'exécution symbolique [4], et a été utilisée pour la génération de tests de programmes (e.g., [16, 18, 21, 1]). Gotlieb et al ont montré comment transformer des programmes impératifs en programmes CLP : InKa [16] a été un pionnier dans l'utilisation de CLP pour générer des données de test pour les programmes en C. Denmat et al ont développé TAUPO, un successeur de InKa qui utilise des relaxations linéaires dynamiques [10]. Cela accroît la capacité de résolution du solveur en présence de contraintes non-linéaires, mais les auteurs ne publient des résultats expérimentaux que sur quelques programmes académiques en C.

Euclide [14] est également un successeur de InKa. Il a trois fonctions principales : la génération de données de test structurel, la génération de contre-exemples et la preuve partielle des programmes C critiques. Euclide construit les contraintes de manière incrémentale et combine des techniques standards de programmation par contraintes et des techniques spécifiques pour

gages incluant les entiers et les tableaux.

[2]La forme DSA (Dynamic Single Assignment)[2] est une représentation intermédiaire inspirée de la forme SSA bien connue en compilation. La forme DSA est une transformation qui préserve la sémantique du programme tout en assurant que chaque variable sur un chemin d'exécution ne soit définie qu'une seule fois.

[3]Nous supposons ici que la post-condition est une conjonction de propriétés.

[4]cf. `http://osl.cs.uiuc.edu/~ksen/cute/`

[5]cf. `http://www.cprover.org/cbmc`

traiter les nombres à virgule flottante et les contraintes linéaires.

CPBPV [6, 7, 8] est un outil basé sur les contraintes dont le but est de vérifier la conformité d'un programme avec ses spécifications. L'idée clef de CPBPV est d'utiliser un ensemble de contraintes pour représenter le programme et ses spécifications, et d'explorer de manière non-déterministe les chemins d'exécution de longueur bornée sur cet ensemble de contraintes. CPBPV fournit un contre-exemple lorsque le programme n'est pas conforme à sa spécification.

Les stratégies de recherche d'Euclide et de CPBPV ne sont pas bien adaptées à la recherche d'un contre-exemple. En effet, CPBPV est fondé sur une exploration descendante des chemins car il a été conçu pour la vérification partielle des programmes. Euclide - qui a été conçu pour la génération de données de test - explore dynamiquement des alternatives faisables mais utilise aussi une exploration descendante. Ces stratégies peuvent donc devenir très coûteuses lorsque le but est seulement de trouver un contre-exemple sur un programme complexe. Par contre, *DPVS* est une stratégie ascendante dynamique qui a été conçue pour trouver des contre-exemples.

1.2 Plan du papier

La section 2 illustre notre approche sur un petit exemple et introduit les nouveaux algorithmes de recherche que nous avons définis. La section 3 décrit les benchmarks et les applications que nous avons utilisées pour valider notre approche. La section 4 présente les résultats expérimentaux et les perspectives.

2 DPVS, la nouvelle stratégie de recherche à base de contraintes

Dans cette section, nous décrivons en termes très généraux les principes de notre approche et le processus de recherche sur un exemple simple. Ensuite, nous détaillons l'algorithme.

2.1 Un petit exemple

La stratégie *DPVS* construit incrémentalement un système de contraintes S et gère une file de variables Q. Q est initialisée avec les variables de la propriété de la post-condition pour laquelle nous cherchons un contre-exemple, alors que S est initialisé avec la négation de cette propriété. Tant que Q n'est pas vide, *DPVS* extrait la première variable v et cherche un bloc du programme où la variable v est définie. Toutes les nouvelles variables (sauf les variables d'entrée du programme) de cette définition sont mises dans Q. La définition de la variable v, ainsi que les conditions nécessaires pour parvenir à la définition de v sont ajoutées à S. Si S est inconsistant, *DPVS* fait un retour arrière et cherche une autre définition de v, sinon les conditions duales aux conditions ajoutées à S sont coupées pour éviter de perdre du temps à explorer des chemins incompatibles. Lorsque Q est vide, le solveur de contraintes est utilisé pour chercher un ensemble de valeurs des variables d'entrée qui violent la propriété, c'est-à-dire un contre-exemple.

Nous illustrons maintenant ce processus sur un exemple très simple, le programme foo présenté dans la figure 1. Ce programme comporte deux post-conditions : $p_1 : c >= d + e$ et $p_2 : f > -e * b$.

Le GFC du programme foo est présenté dans la figure 2. Notons que le programme a été transformé en une forme DSA[6]. Supposons que nous voulons prouver la propriété p_1. Avant de chercher un contre-exemple, nous calculons la forme compactée du GFC, où les nœuds qui ne sont pas liés à la propriété à prouver sont supprimés (cf. figure 3).

Les figures 4 et 5 présentent les chemins explorés par *DPVS*. Le processus de recherche sélectionne d'abord le nœud (4) où la variable c_0 est définie. Pour atteindre le nœud (4), l'état dans le nœud (0) doit être vrai. Cette condition est donc ajoutée à l'ensemble de contraintes S et le branchement alternatif est coupé. A cette étape, S contient les contraintes suivantes : $(c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = a_0 \wedge a_0 \geq 0)$ qui peut être simplifié en $(a_0 < 0 \wedge a_0 \geq 0)$. Cet ensemble de contraintes est inconsistant et *DPVS* sélectionne donc le nœud (8) où la variable c_0 est également définie. Pour atteindre le nœud (8), la condition dans le nœud (0) doit être fausse. La négation de cette condition est donc ajoutée à S et l'autre branche est coupée. S contient donc les contraintes suivantes : $(c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = b_0 \wedge a_0 < 0 \wedge d_0 = 1 \wedge e_0 = -a_0)$ qui peut être simplifié à $(a_0 < 0 \wedge b_0 < 0)$. Cet ensemble de contraintes est consistant et le solveur calcule une solution, par exemple, $(a_0 = -1, b_0 = -1)$. Ces valeurs des variables d'entrée sont un jeu de test qui viole la propriété p_1 du programme foo.

Ce petit exemple montre comment fonctionne *DPVS* : cette nouvelle stratégie recueille le maximum d'informations sur les variables qui influencent la post-condition afin de détecter les inconsistances le plus tôt possible ; ceci est particulièrement efficace quand un petit sous-ensemble du système de contraintes est inconsistant.

[6] Nous ne considérons ici que des programmes bornés où la taille des tableaux, et le nombre d'itérations des boucles sont limités, nous simplifions des $\phi-$fonctions lors de construction de la forme SSA.

```
void foo(int a, int b)
1.  int c, d, e, f;
2.  if(a >= 0) {
4.      if(a < 10) {
5.          f = b − 1;
6.      }
7.      else {
8.          f = b − a;
9.      }
10.     c = a;
11.     if(b >= 0) {
12.         d = a; e = b;
13.     }
14.     else {
15.         d = a; e = −b;
16.     }
17. }
18. else {
19.     c = b; d = 1; e = −a;
20.     if(a > b) {
21.         f = b + e + a;
22.     }
23.     else {
24.         f = e ∗ a − b;
25.     }
26. }
27. c = c + d + e;
28. assert(c >= d + e); // property p1
29. assert(f >= −b ∗ e); // property p2
```

FIG. 1 – Programme *foo*

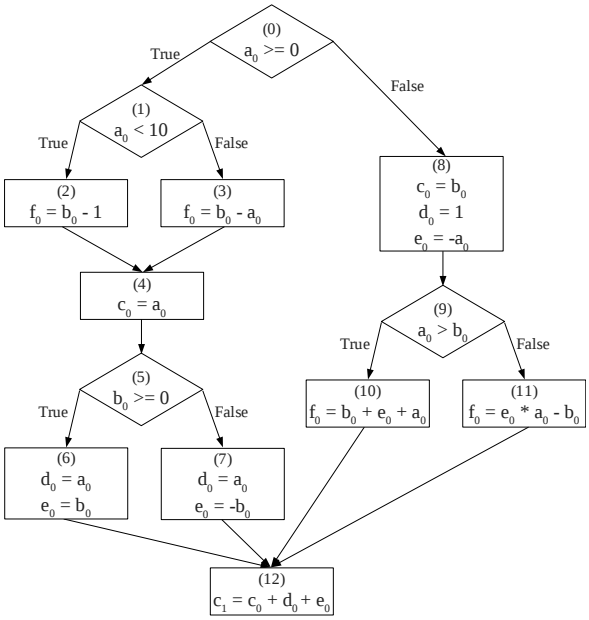


FIG. 2 – GFC du programme *foo* en forme DSA

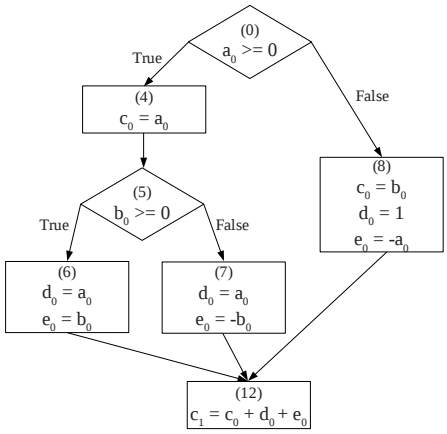


FIG. 3 – GFC compacté pour la propriété p_1

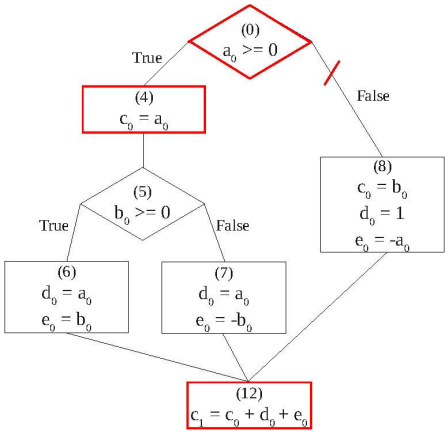


FIG. 4 – Processus de recherche pour p_1, étape 1

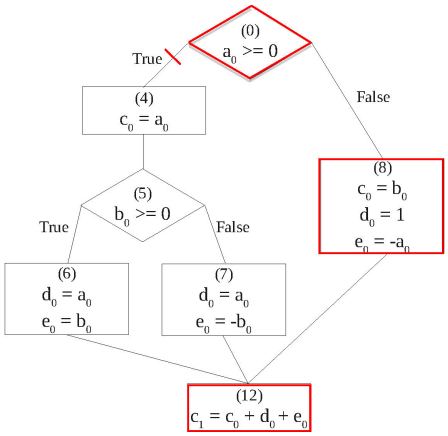


FIG. 5 – Processus de recherche pour p_1, étape 2

2.2 Algorithme

Nous détaillons ici l'algorithme *DPVS* (cf. figure 2.2). Nous calculons au préalable les informations suivantes :

- $du[x]$: l'ensemble des blocs où la variable x est définie ;
- $anc_c[u]$: l'ensemble des prédécesseurs de u qui sont des nœuds conditionnels ;
- $dr[u, v]$: un booléen qui est vrai (resp. faux) lorsque la condition du prédécesseur v du nœud u doit être vrai (resp. fausse) pour atteindre u.

DPVS utilise également les structures de données suivantes :

- M : l'ensemble des variables marquées (une variable est marquée si elle a déjà été mise dans la file) ; M est initialisé avec $\emptyset$;
- S : l'ensemble des contraintes qui est initialisé avec $const(pred \wedge \neg(prop))$ où *const* est une fonction qui transforme une expression sous forme DSA en un ensemble de contraintes ;
- Q : l'ensemble des variables temporaires, initialisé par $V(prop)$; l'ordre de traitement des éléments de Q est spécifié par le paramètre O_q

DPVS sélectionne une variable dans Q et essaie de trouver un contre-exemple avec sa première définition, si elle échoue, il essaie de manière itérative avec les autres définitions de la variable sélectionnée.

DPVS définit la couleur du nœud conditionnel u à rouge (resp. bleu) lorsque l'état de u est défini à true (false) dans le chemin courant. En d'autres termes, lorsque la couleur est rouge (resp. bleu) le branchement à droite (resp. gauche) de u est coupé. $color[u]$ est initialisé à vide pour tous les nœuds.

DPVS retourne *sol* qui est soit une instance des variables d'entrée de P satisfaisant le système de contraintes S ou $\emptyset$ si S n'a pas de solution. Les solutions sont calculées par la fonction *solve*, un solveur sur les domaines finis. La fonction *solve* est une procédure de décision complète sur les domaines finis. La fonction $isfeasible$ (ligne 27) effectue uniquement un test de la consistance partielle mais elle est beaucoup plus rapide que la fonction *solve*, ce qui justifie notre choix puisque ce test est effectué à chaque fois que la définition d'une variable est ajoutée à S.

Il est facile de montrer que *sol*, la solution calculée par *DPVS* est un contre-exemple. En effet, ces valeurs des données d'entrées satisfont les contraintes générées à partir de :

- *pred*, la pre-condition ;
- $\neg prop$, la négation d'une partie de la post-condition ;
- la définition de toutes les variables dans $V(prop)$ et la définition de toutes les variables (sauf les variables d'entrée), introduites par ces définitions ;
- toutes les conditions nécessaires pour atteindre les définitions ci-dessus.

Ainsi, il existe au moins un chemin exécutable qui prend des valeurs d'entrée *sol* et calcule une sortie qui viole la propriété *prop*. Lorsque aucune solution ne peut être trouvée, tous les chemins sont contradictoires avec la négation de la post-condition, et nous pouvons donc affirmer qu'il n'existe pas de valeurs d'entrée qui violent la propriété *prop*.

Algorithm 1 : *DPVS*

Function $DPVS(M, S, Q, O_q)$
returns *counterexample*

```
1:  if Q = ∅ then
2:    return solve(S)
3:  else
4:    x ← POP(Q, O_q)
5:    for all u ∈ du[x] do
6:      Cut ← FALSE ; SAVE vector Color
7:      S1 ← S∧ const(def[x, u])
            % def[x, u] denotes the definition of x in
        block u
8:      V_new ← V(def[x, u]) \ M
9:      PUSH(Q, V_new, O_q) ; add(V_new, M)
10:     for all v ∈ anc_c[u] do
11:       if color[v] = blank then {%no branch is cut
          off}
12:         V_new ← V(condition[v]) \ M
13:         PUSH(Q, V_new, O_q) ; add(V_new, M)
14:         if dr[u, v] then {% Condition must be true}
15:           S1 ← S1∧ cons(condition[v]))
16:           color[v] ← red % Cut the right branch
17:         else {% Condition must be false}
18:           S1 ← S1 ∧ ¬ cons(condition[v])
19:           color[v] ← blue % Cut the left branch
20:         end if
21:       else
22:         if (color[v] = red ∧ dr[u, v])
                    ∨ (color[v] = blue ∧ ¬(dr[u, v]))
            then {%no branch is reachable}
23:           Cut ← TRUE
24:         end if
25:       end if
26:     end for
27:     if ¬Cut ∧ isfeasible(S1) then
28:       result ← DPVS(M, S1, Q, O_q)
29:       if result ≠ ∅ then
30:         return result
31:       end if
32:     end if
33:     RESTORE vector Color
34:   end for
35:   return ∅
36: end if
```

3 Expérimentations et Applications

Dans cette section, nous décrivons les exemples et applications utilisés pour évaluer *DPVS*. Il s'agit d'exemples académiques, d'une implémentation du système de gestion des collisions de trafic aérien (TCAS), et enfin d'une application industrielle réelle, un contrôleur qui gère certaines fonctions liées aux feux clignotants d'un véhicule.

3.1 Exemples académiques

3.1.1 Programme tritype

Le programme tritype est un benchmark standard pour la génération de cas de test et pour la vérification de programmes car il contient de nombreux chemins infaisables : seulement 10 chemins correspondent à des entrées réelles du programme, en raison de la complexité des instructions conditionnelles. Ce programme prend en entrée trois entiers positifs (les côtés d'un triangle) et retourne la valeur 1, (resp 2, 3) si les entrées correspondent à un triangle scalène (resp. isocèle, équilatéral) et la valeur 4 s'il ne s'agit pas d'un triangle.

Nous considérons aussi deux variations du programme *Tritype* :

1. *Triperimeter* qui retourne le périmètre du triangle lorsque les entrées correspondent à un triangle, sinon il retourne -1.
2. *Tritimes* qui retourne le produit des entrées.

Ces deux variations sont plus difficiles que le programme *Tritype* lui-même : *Triperimeter* retourne une expression linéaire sur les entrées et *Tritimes* retourne une expression **non** linéaire alors que*Tritype* retourne une constante.

Pour ces trois programmes, nous avons également examiné certaines versions où une erreur a été introduite.

3.1.2 Recherche binaire

Le deuxième exemple illustre le cas des programmes contenant des tableaux et des boucles. Il s'agit d'un programme de recherche binaire qui détermine si une valeur v est présente dans un tableau trié t.

3.2 Système anti-collisions de trafic aérien

Cette application concerne un composant logiciel connu du système anti-collisions de trafic aérien (TCAS). Il s'agit d'un système critique embarqué destiné à éviter les collisions en vol entre aéronefs [5]. Nous considérons ici le programme et la spécification qui ont été écrits par Arnaud Gotlieb [14, 15] à partir d'une version préliminaire donnée dans [19]. Nous résumons ici leurs caractéristiques principales.

Ce programme contient 173 millions de lignes de code C, y compris les conditions imbriquées, les opérateurs logiques, des définitions de type, les macros et les appels de fonction. La fonction principale à vérifier prend 14 variables globales comme entrées. 10 propriétés ont été identifiées et formalisées dans la littérature (cf. [14] pour une présentation complète des ces propriétés).

3.3 Clignotant

Ce dernier benchmark est un composant logiciel industriel temps-réel[7]. Il illustre la façon dont nous traitons les propriétés sur plusieurs unités de temps. La complexité du code C est comparable à la complexité du benchmark TCAS, mais nous devons vérifier une propriété pour plusieurs exécutions du code.

3.3.1 Description du module

Le *Clignotant* est un contrôleur qui gère certaines fonctions liées aux feux clignotants d'un véhicule. La figure 6 fournit un modèle Simulink simplifié de ce contrôleur. Les entrées et sorties du modèle sont :

- CBSW_HAZARD_L : Activation du clignotant gauche
- CBSW_HAZARD_R : Activation du clignotant droit
- WARNING : Activation des warning
- RF_KEY_LOCK : Fermeture du véhicule par la clé
- RF_KEY_UNLOCK : Ouverture du véhicule par la clé
- FLASHER_ACTIVE : Fonction clignotant activée
- CMD_FLASHER_L : Commande du clignotant gauche
- CMD_FLASHER_R : Commande du clignotant droit

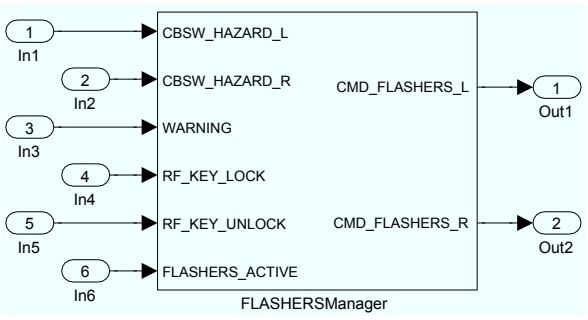


FIG. 6 – Modèle Simulink simplifié du *Clignotant*

[7] Cet exemple provient d'un constructeur automobile et a été fourni par Geensys (cf. `http://www.geensys.com/?Home&l=en`)

Certaines fonctions du *Clignotant* sont prioritaires sur d'autres. Nous avons cherché à vérifier la propriété p_1 : ***Les clignotants ne restent jamais en position allumée, même en cas de coupure***

3.3.2 Programme à vérifier

Afin de vérifier la propriété p_1, le modèle Simulink du *Clignotant* est d'abord traduit[8] en une fonction C, nommée fonction $f1$. La fonction $f1$ possède 6 entrées et 2 sorties. Elle comprend 300 lignes de code, 81 variables booléennes, et 28 variables entières. La fonction $f1$ se compose principalement d'instructions conditionnelles imbriquées et d'instructions d'affectation avec des expressions linéaires ou constantes (cf. partie de code, figure 7).

```
and1_a = (Switch5 && !Unit_Delay3_a_DSTATE);
if(and1_a - Unit_Delay_c_DSTATE != 0)
    rtb_Switch_b = 0;
else{
    add_a = 1 + Unit_Delay1_b_DSTATE;
    rtb_Switch_b = add_a;
}
superior_a = (rtb_Switch_b >= 3);
```

FIG. 7 – Un partie de code de la fonction $f1$

Notre objectif est de vérifier s'il existe une séquence de données d'entrée qui viole la propriété p_1 du *Clignotant*. Cette propriété concerne le comportement du *Clignotant* pour une période de temps infinie. En pratique, on ne peut vérifier qu'une version limitée de la propriété p_1 : nous considérons que la propriété est violée si les feux restent allumés pour une séquence de N unités de temps consécutives. Nous introduisons donc une boucle (bornée par N) qui compte le nombre de fois où la sortie du *Clignotant* a été consécutivement vraie. Si ce compteur est égal à N à la sortie de la boucle, alors la propriété est violée. La partie du programme C correspondant à la version bornée est donnée dans la figure 8.

Le programme tritype et ses spécifications en JML[9], le programme Bsearch, le programme TCAS avec la spécification pour la propriété à vérifier, ainsi que le modèle Simulink détaillé du *Clignotant* sont disponibles à l'adresse `http://users.polytech.unice.fr/~rueher/Publis/jfpc10_fr_annexe.pdf`.

[8]Cette traduction est faite avec un outil propriétaire de Geensys.

[9]cf. `http://www.cs.ucf.edu/leavens/JML/`

```
count = 0;
// number of time where the output has been
// consecutively true int count = 0; consider N
// periods of time
for(int i = 0; i < N; i++) {
// call to f1 function to compute the outputs
// according to non deterministic input values
f1();
if (Model_Outputs4)
   // the output has been consecutively true one
   // more time
   count ++;
else
   // the output is not consecutively true
   count = 0;
}
// if count is less than N, then the property is
// verified
assert(count < N)};
```

FIG. 8 – Le programme C à vérifier

4 Expérimentations et discussion

Dans cette section, nous présentons les expérimentations que nous avons faites pour valider notre approche et nous discutons des travaux futurs.

4.1 Outils

Nous avons comparé les performances de *DPVS* avec CBMC, Euclide et CPBPV*.

Comme indiqué précédemment, CBMC[10] est l'un des meilleurs model checkers bornés actuellement disponibles. Nous avons utilisé la version 3.3.2 qui appelle le solveur *minisat*2.

CPBPV* est une version optimisée de CPBPV [7, 8] qui est implémenté dans `Comet`[11]. Il utilise un ensemble de contraintes pour représenter le programme et ses spécifications, et il explore des chemins exécutables de longueur bornée sur ces ensembles de contraintes. Toutefois, contrairement à CPBPV, il travaille sur un GFC compacté. Une propagation de bornes est également effectuée dans l'étape initiale.

Euclide [14] est un framework de test à base de contraintes conçu pour la génération de données structurales de test, pour la génération de contre-exemple et de preuves partielles des programmes C critiques. Nous n'avons pas pu évaluer Euclide sur l'application *Clignotant*. En effet, en raison d'un bug, Euclide ne pouvait pas traiter ce programme C[12]. Les perfor-

[10]cf. `http://www.cprover.org/cbmc`

[11]`Comet`, une marque commerciale de Dynadec (cf. `http://dynadec.com`) est une plate-forme d'optimisation hybrides.

[12]Nous avons contacté les auteurs, mais ils ne pouvent pas corriger ce bug pour ce moment.

mances d'Euclide sur l'application TCAS sont publiées par les auteurs dans [14]. Les expérimentations des auteurs ont été réalisées sur un processeur Intel Core Duo 2,4GHz avec 2Go de RAM, un ordinateur très semblable à celui que nous avons utilisé.

DPVS est implémenté en `Comet`. Nous donnons ici les expérimentations de deux versions : LIFO et TAS. Nous avons essayé différentes combinaisons de solveurs domaines finis (CP) et solveurs de programmation linéaire (LP). Nous présentons ici les résultats pour deux combinaisons :

- CP-CP : le solveur CP est utilisé pour vérifier la consistance partielle à chaque nœud et pour la recherche d'une solution ;
- LP-CP : le solveur LP est utilisé pour vérifier la consistance partielle d'une relaxation linéaire du système de contraintes à chaque nœud, et le solveur CP est utilisé pour chercher une solution.

Les expérimentations avec $CBMC$,$CPBPV^*$ ont été effectuées sur un processeur Intel Core Duo T8300 2,4 GHz avec 2Go de RAM pour tous les tests sauf l'application *Clignotant* où un Quad Core Intel Xeon X5460 3,16 GHz avec 16Go de mémoire a été utilisé. Tous les temps sont donnés en secondes. OoM signifie "out of memory" tandis que TO signifie "timeout" dans les différents tableaux. TO a été fixé à 3 minutes pour tous les tests.

4.2 Expérimentations sur des exemples académiques

Le tableau 1 fournit les résultats expérimentaux pour le programme *Tritype* et de ses variations. Tous les solveurs ont trouvé un contre-exemple pour les programmes erronés en très peu de temps. Pour les programmes corrects, les preuves sont plus difficiles. En effet, tous les chemins doivent être explorés, contrairement aux versions erronées où la résolution s'arrête dès que la première erreur est trouvée. Toutefois, les systèmes qui utilisent un solveur linéaire ont pu vérifier la correction partielle de Tritype et Triperimeter. La version correcte du programme non-linéaire *Tritimes* est la plus difficile à traiter. Seules les versions de $DPVS$ et $CPBPV^*$ qui combinent LP et CP ont pu la traiter. Cela peut s'expliquer par les raisons suivantes :

1. Les décisions sont faciles à tester avec un solveur linéaire ;
2. Les décisions sont ajoutées incrémentalement dans l'ensemble de contraintes ;
3. Le solveur CP qui est appelé à la fin des chemins peut bénéficier des décisions qui ont été ajoutées à l'ensemble de contraintes grâce à un mécanisme de substitution des sous expressions.

Les points (1) et (2) garantissent que des chemins infaisables sont rapidement coupés. Pour expliquer le point (3) considérons la propriété $p : r = i*j*k$ à vérifier. Supposons que la décision $i = j$ a été prise sur un chemin, et donc la valeur retournée par le programme est $i * k * i$. Le système de contraintes contiendra les trois contraintes $r = i*j*k$, $i = j$, $\neg(r = i*k*i)$[13], et le solveur CP peut facilement détecter l'inconsistance.

Le tableau 2 donne les résultats expérimentaux sur une version correcte du programme de *recherche binaire*. CBMC et $DPVS$ ne peuvent pas traiter ce benchmark. CBMC perd beaucoup de temps dans le processus de dépliage. Les stratégies utilisées par $DPVS$ ne sont pas bien adaptées non plus pour ce programme très spécifique. La version LP-CP de CPBPV* est par contre très efficace. Elle utilise une approche descendante et ajoute incrémentalement les décisions prises sur un chemin. Cette stratégie est particulièrement bien adaptée pour ce programme de *recherche binaire* qui a une pre-condition très contraignante.

Tab. 1 – Tritype (Typ), Tritimes (Tm), Triperimeter (per) , entiers de 16 bits

Prog	CBMC 3.3.2	DPVS LIFO CP-CP	CPBPV* CP-CP	DPVS LIFO LP-CP	CPBPV* LP-CP
Typ-OK	0.161	TO	TO	0.781	0.304
Typ-KO	0.012	0.087	0.127	0.030	0.009
Per-OK	0.717	TO	TO	0.054	0.054
Per-KO	0.015	0.002	0.027	0.014	0.018
Tm-OK	TO	TO	TO	1.048	0.865
Tm-KO	0.050	0.070	0.002	0.029	0.009

Tab. 2 – Cherche binaire (entiers de 16 bits)

Length	CBMC 3.3.2 (MiniSAT2)	DPVS LIFO LP-CP	CPBPV* LP-CP
4	5.732	0.529	0.107
8	110.081	35.074	0.298
16	TO	TO	1.149
32	TO	TO	5.357
64	TO	TO	27.714
128	TO	TO	153.646

4.3 Expérimentations sur TCAS

Les résultats pour l'application TCAS sont présentés dans le tableau 3. CBMC a de bons résultats sur ce problème, mais les performances de la combinaison CP-CP de $DPVS$ sont meilleures. Ceci s'explique par le fait que TCAS est essentiellement un problème SAT : le programme contient de nombreux booléens

[13] Nous ne détaillons pas ici les renommages DSA.

et les entiers ne peuvent prendre que quelques valeurs. C'est aussi pourquoi la combinaison CP-CP de CPBPV* est meilleure que la combinaison de LP-CP CPBPV* sur ce problème.

Les différentes stratégies de *DPVS* donnent des résultats semblables sur ce problème. Les expérimentations sur *TCAS* ont été réalisées pour des entiers de 16 bits car nous voulions comparer nos résultats à ceux publiés dans [15]. Euclide est beaucoup plus lent que les autres solveurs, mais il est implémenté en Prolog.

TAB. 3 – TCAS (entiers de 16 bits)

Property	CBMC 3.3.2	Euclide (SICtus Prolog)	DPVS LIFO CP-CP	CPBPV* CP-CP	CPBPV* LP-CP
P1A-OK	0.101	0.7	0.021	0.173	8.960
P1B-OK	0.063	0.7	0.019	0.211	13.737
P2A-OK	0.097	0.6	0.020	0.175	9.168
P2B-KO	0.064	0.7	0.011	0.014	0.165
P3A-KO	0.061	5.4	0.008	0.036	0.615
P3B-OK	0.066	1.2	0.011	0.100	0.4431
P4A-KO	0.075	6.8	0.008	0.045	1.446
P4B-KO	0.060	2.7	0.008	0.014	0.254
P5A-OK	0.068	0.6	0.044	0.197	20.627
P5B-KO	0.065	1.0	0.015	0.014	0.195

4.4 Expérimentations sur l'application Clignotant

TAB. 4 – *Clignotant*

N	CBMC 3.3.2	DPVS HEAP CP-CP	DPVS LIFO CP-CP	DPVS HEAP LP-CP	DPVS LIFO LP-CP
5	0.134	0.026	0.032	0.565	0.953
10	0.447	0.055	0.068	2.484	4.134
20	1.766	0.118	0.149	9.081	15.320
30	4.231	0.194	0.248	20.847	35.066
50	12.92	0.345	0.458	57.797	96.693
75	32.747	0.602	0.842	137.104	TO
100	58.279	2.750	3.394	TO	TO
150	138.192	1.552	2.365	TO	TO
200	OoM	OoM	8.082	TO	TO

CBMC et *DPVS* ont trouvé des contre-exemples qui violent la propriété p_1. Ils ont également généré des données d'entrée pour lesquelles le clignotant reste allumé de manière continue pour une séquence de N unités de temps.

Par exemple, voici une séquence d'entrées qui viole la propriété p_1 pour une séquence de 5 unités de temps :

```
[(0,1,0,0,0,1),(0,1,0,0,0,1),(0,1,0,0,0,1),
(0,0,1,0,0,1),(0,0,0,0,0,1)]
```

où $(0,1,0,0,0,1)$ sont les valeurs des entrées de in_1 à in_6 pour la première période, $(0,1,0,0,0,1)$ les valeurs des entrées pour la deuxième période et ainsi de suite.

Le tableau 4 montre que la combinaison CP-CP de *DPVS* est bien meilleure que les combinaisons LP - CP sur l'application *Clignotant*. On notera que par manque de mémoire, CBMC n'est pas capable de générer des contre exemples pour plus de 200 unités de temps alors que *DPVS*, avec la stratégie TAS et la combinaison CP-CP, peut générer des contre-exemples pour 400 unités de temps. CPBPV* n'est pas capable de générer des contre-exemples pour plus de 10 unités de temps.

L'utilisation d'un solveur de contraintes sur les domaines finis pour calculer la consistance partielle est beaucoup efficace sur ce programme qu'un solveur de programmation linéaire. Ceci s'explique pour deux raisons :

- la relaxation linéaire nécessaire par le solveur LP introduit de nombreux points de choix alors que ceci n'est pas le cas pour le solveur CP. En effet, considérons une condition telle que $x == y$ dans un `if`. La négation de cette condition correspond à la contrainte $x! = y$ qui nécessite la génération de deux systèmes de contraintes linéaires : l'un avec la contrainte $x < y$, l'autre avec la contrainte $x > y$.
- les domaines des variables entières sont petits pour cette application, et l'étape de propagation des bornes que nous effectuons avant l'exploration du graphe permet de réduire fortement la taille de ces domaines. Les tests de consistance avec le solveur CP sont donc très efficaces.

4.5 Discussion et travaux futurs

Les premières expérimentations avec *DPVS* sont très encourageantes. *DPVS* se comporte bien sur des exemples académiques et obtient de bien meilleurs résultats que CBMC sur deux applications réelles. Il faut souligner ici que la génération de cas de test pour des séquences de périodes de temps assez grandes est un problème crucial dans les applications temps-réel.

Ces résultats sont très prometteurs car *DPVS* est encore un prototype académique tandis que CBMC est un des tout meilleurs systèmes de model checking actuellement disponibles. *DPVS* est également bien meilleur que Euclide sur l'application TCAS. Naturellement, d'autres expérimentations sur des applications réelles sont nécessaires pour affiner et pour valider l'approche proposée. Il nous faut en particulier arriver à mieux expliciter les caractéristiques des programmes pour lesquels cette stratégie est effectivement bien adaptée.

Les travaux futurs concernent l'extension de notre prototype. Actuellement, il y a beaucoup de restrictions sur les programmes C et Java que le prototype peut traiter. En particulier, nous ne traitons que

des programmes qui ne provoquent pas un déroutement, e.g., nous ne traitons pas les programmes qui contiennent des divisions par zéro ou des erreurs qui provoquent la levée d'une exception. Nous acceptons des tableaux de taille bornée, mais le type des données d'entrée est limité aux booléens et aux entiers. Nous travaillons aussi actuellement sur l'interface entre un solveur CP et notre propre solveur sur les nombre à virgule flottante [4].

Références

[1] Elvira Albert, Miguel Gómez-Zamalloa, and Germán Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *LOPSTR 2008*, LNCS (Springer verlag), vol. 5438 : 4–23. Springer, 2008.

[2] Mike Barnett and K. Rustan M. Leino. Weakest-Precondition of Unstructured Programs In *Information Processing Letters*, vol. 93(6) :281–288.

[3] Thomas Bochot, Pierre Virelizier, Hélène Waeselynck, and Virginie Wiels. Model checking flight control systems : The Airbus experience. In *ICSE 2009*, pages 18–27. IEEE, 2009.

[4] Bernard Botella, Arnaud Gotlieb, and Claude Michel. Symbolic execution of floating-point computations. *Softw. Test., Verif. Reliab.*, 16(2) :97–121, 2006.

[5] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzè. Using symbolic execution for verifying safety-critical systems. In *ESEC / SIGSOFT FSE*, pages 142–151, 2001.

[6] Hélène Collavizza and Michel Rueher. Exploration of the capabilities of constraint programming for software verification. In *TACAS*, LNCS (Springer verlag),vol. 3920 :182–196, 2006.

[7] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV : A Constraint-Programming Framework for Bounded Program Verification. In *CP 2008*, LNCS (Springer verlag), vol. 5202 :327–341, 2008.

[8] Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. CPBPV : A Constraint-Programming Framework for Bounded Program Verification. *Constraints Journal*, Springer Verlag, vol. 15(2) :238-264, 2010.

[9] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4) :451–490, 1991.

[10] Tristan Denmat, Arnaud Gotlieb, and Mireille Ducassé. Improving constraint-based testing with dynamic linear relaxations. In *Proc. of ISSRE*, pages 181–190. IEEE Computer Society, 2006.

[11] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7) :1165–1178, 2008.

[12] Gordon Fraser, Franz Wotawa, and Paul Ammann. Testing with model checkers : a survey. *Softw. Test., Verif. Reliab.*, 19(3) :215–261, 2009.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart : directed automated random testing. In *PLDI*, pages 213–223. ACM Press, 2005.

[14] Arnaud Gotlieb. Euclide : A Constraint-Based Testing Framework for Critical C Programs. In *ICST'09*, pages 151–160. IEEE Computer Society, 2009.

[15] Arnaud Gotlieb. TCAS software verification using constraint programming. *The Knowledge Engineering Review, (Accepted for publication)*, 2010.

[16] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *ISSTA*, pages 53–62, 1998.

[17] Paula Herber, Florian Friedemann, and Sabine Glesner. Combining model checking and testing in a continuous hw/sw co-verification process. In *TAP 2009*, LNCS (Springer verlag), vol. 5668 : 121–136, 2009.

[18] Daniel Jackson and Mandana Vazir. Finding bugs with a constraint solver. In *ISSTA*, pages 14–25. ACM Press, 2000.

[19] Carolos Livadas, John Lygeros, and Nancy A. Lynch. High-Level Modeling and Analysis of TCAS. In *IEEE Real-Time Systems Symposium*, pages 115–125, 1999.

[20] N.Williams, B.Marre, P.Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In *Dependable Computing -EDCC'05*, pages 281–292, 2005.

[21] Nguyen Tran Sy and Yves Deville. Automatic test data generation for programs with integer and float variables. In *ASE*, pages 13–21. IEEE Computer Society, 2001.

Résolution exacte de MinSAT *

Chu Min Li [1] **Felip Manyà** [2] **Zhe Quan** [1] **Zhu Zhu** [1]

[1] MIS, Université de Picardie Jules Verne, 5 Rue du Moulin Neuf, 80000 Amiens, France
[2] IIIA-CSIC, Campus UAB, 08193 Bellaterra, Spain

{chu-min.li,zhu.zhu}@u-picardie.fr felip@iiia.csic.es quanzhe@gmail.com

Résumé

MinSAT est le problème consistant à trouver une affectation qui minimise le nombre de clauses satisfaites dans une formule CNF. Nous présentons une approche originale pour résoudre MinSAT de façon exacte, qui repose sur le codage MaxSAT et les solveurs MaxSAT. Notre étude empirique fournit la preuve que, en l'absence de spécifiques solveurs exacts pour MinSAT, l'approche générique proposée dans ce papier est compétitive.

Abstract

MinSAT is the problem of finding a truth assignment that minimizes the number of satisfied clauses in a CNF formula. We present an original approach for exact MinSAT solving, which relies on solving MinSAT using MaxSAT encodings and MaxSAT solvers. Our empirical investigation provides evidence that, in the absence of genuine exact solvers for MinSAT, the generic approach proposed in this paper is competitive.

1 Introduction

MaxSAT est le problème consistant à trouver une affectation qui maximise le nombre de clauses satisfaites dans une formule CNF, et MinSAT est le problème consistant à trouver une affectation qui minimise le nombre de clauses satisfaites. En dépit que MaxSAT est plus populaire et a attiré l'intérêt de la communauté SAT ces dernières années [8], nous pensons qu'il vaut la peine d'étudier MinSAT et de mettre au point des techniques de résolution exacte rapide pour MinSAT.

MinSAT a des intérêts théoriques et pratiques. Du point de vue théorique, nous soulignons les travaux existants sur les algorithmes d'approximation pour MinSAT (voir e.g. [2, 6, 11] et les références y citées). Du point de vue pratique, nous aimerions mettre l'accent sur les réductions existantes à problème MinSAT dans les domaines tels que la bio-informatique [5] et l'apprentissage supervisé [4].

Malheureusement, au meilleur de notre connaissance, il n'existe pas de solveur exact pour MinSAT similaire aux solveurs modernes séparation et évaluation développés pour MaxSAT. Compte tenu de l'énorme quantité d'efforts consacrés jusqu'à présent à la conception et au développement de solveurs compétitifs exacts pour MaxSAT, il semble difficile d'avoir de spécifiques solveurs exacts efficaces pour MinSAT dans un proche avenir. Donc, nous pensons qu'il vaut la peine d'explorer la résolution exacte de MinSAT à l'aide d'une méthode générique de résolution de problème.

Notre approche générique pour la résolution exacte de MinSAT s'appuie sur la définition des codages efficaces et originaux de MinSAT en MaxSAT, résoudre les problèmes codés par un solveur MaxSAT moderne, puis en déduire une solution optimale de MinSAT à partir d'une solution optimale de MaxSAT. Pour être plus précis, nous définissons trois codages. Le premier est une réduction directe de MinSAT en MaxSAT partiel. Les deux autres codages sont plus complexes, car ils commencent par une réduction de MinSAT en MaxClique, puis par une seconde réduction de MaxClique en MaxSAT partiel. La différence entre les deux codages est que le premier utilise un codage classique de MaxClique en MaxSAT partiel, tandis que le deuxième utilise un nouveau codage de Maxclique en MaxSAT qui repose sur une partition du graphe concerné en cliques maximales. L'étude empirique que nous avons réalisée montre que cette approche générique est compétitive.

Notre objectif est de profiter de l'évolution récente de MaxSAT, qui ont produit une technologie qui devient très compétitive dans la résolution de problème d'optimisation. D'autre part, il existe une large gamme de problèmes de minimisation qui ont un codage plus naturel et plus com-

*Ce travail est partiellement financé par la Generalitat de Catalunya avec la subvention 2009-SGR-1434, et les projets de recherche *Ministerio de Ciencia e Innovación* CONSOLIDER CSD2007-0022, INGENIO 2010, Acción Integrada HA2008-0017, TIN2007-68005-C04-04 et TIN2009-14704-C03-01.

pact en MinSAT qu'en MaxSAT. Résoudre directement et efficacement MinSAT peut faciliter le travail de l'utilisateur. Notre approche permet ainsi de contribuer à diffuser les résultats de la recherche et la technologie de la communauté SAT à d'autres communautés travaillant sur les problèmes d'optimisation.

Ce papier est structuré comme suit. Dans la section 2, nous donnons quelques définitions. Dans la section 3, nous définissons le codage direct de MinSAT en MaxSAT partiel. Dans la section 4, nous définissons les codages qui sont fondés en premier lieu sur la réduction de MinSAT à MaxClique, puis MaxClique vers MaxSAT partiel. Dans la section 5, nous présentons l'étude empirique, afin d'évaluer notre approche de la résolution exacte de MinSAT. Dans la section 6, nous donnons les conclusions et suggérons quelques directions pour les futures recherches.

2 Preliminaires

Considérons un ensemble de variables booléennes $\{x_1,x_2,...,x_n\}$, un littéral est une variable x_i ou sa négation $\bar{x}_i$, une clause est un *ou* logique de quelques littéraux. Une formule CNF (Conjunctive Normal Form) ϕ est un ensemble de clauses. MaxSAT est le problème consistant à trouver une affectation de valeurs de vérité qui maximise le nombre de clauses satisfaites dans une formule CNF, et MinSAT est le problème consistant à trouver une affectation qui minimise le nombre de clauses satisfaites. Min-2-SAT (Min-3-SAT) est le problème MinSAT restreint à des clauses avec au plus 2 (3) littéraux.

Il est connu que, pour toute instance de MinSAT, il y a une affectation qui satisfait zéro clauses si chaque variable apparaît uniquement avec la polarité positive ou seulement avec la polarité négative. De même, si toutes les clauses sont unitaires, la solution à MinSAT est facilement obtenue en affectant une variable à vrai si elle est apparaît dans moins de clauses que sa négation, et la variable à faux sinon. Cependant, en général, MinSAT (même Min-2-SAT) est NP-difficile [6].

Nous considérons également l'extension de MaxSAT connu sous le nom de MaxSAT partiel, car cette extension permet de mieux représenter et résoudre certains problèmes NP-difficiles. Un problème MaxSAT partiel est une formule CNF dont certaines clauses sont *relaxables* ou *souples* et les restes sont *non-relaxables* ou *dures*. Les clauses dures sont représentées entre crochets. Alors que MaxSAT consiste à trouver une affectation de valeurs de vérité qui maximise le nombre de clauses satisfaites, résoudre une instance MaxSAT partiel revient à trouver une affectation qui satisfait toutes les clauses dures et le maximum clauses souples. Étant donné un graphe non orienté $G(V, E)$, V est l'ensemble de sommets et E est l'ensemble d'arêtes, le problème de couverture minimale de sommets consiste à trouver un sous-ensemble minimum V' de V, tel que pour chaque arête $\{u, v\}$ dans E, au moins un des sommets u et v est dans V'. Une clique dans un graphe non orienté $G = (V, E)$ est un sous-ensemble C de V tel que, pour tous les deux sommets de C, il existe une arête les reliant. Cela revient à dire que le sous-graphe induit par C est complet. Une clique maximale est une clique qui ne peut être étendue en ajoutant un sommet supplémentaire, et une clique maximum est une clique de la plus grande cardinalité possible. Le problème de clique maximum (MaxClique) pour un graphe non orienté G consiste à trouver une clique maximum dans G. Une partition en cliques d'un graphe non orienté $G = (V, E)$ est une partition de V en sous-ensembles disjoints $V_1, \ldots, V_k$ tels que, pour $1 \leq i \leq k$, le graphe induit par V_i est une clique.

3 Codage 1 : Coder directement MinSAT en MaxSAT partiel

Définition 1 *Étant donnée une instance MinSAT I composée de l'ensemble des clauses $\mathcal{C}_I = \{C_1, \ldots, C_m\}$ et l'ensemble de variables X_I, le codage direct de I en MaxSAT est défini comme suit :*

- *L'ensemble des variables propositionnelles est $X_I \cup \{c_1, \ldots, c_m\}$, où $\{c_1, \ldots, c_m\}$ est un ensemble de variables auxiliaires.*
- *Pour toute clause $C_i \in \mathcal{C}_I$, la clause dure $c_i \leftrightarrow C_i$ est ajoutée.*
- *Pour toute variable auxiliaire c_i, la clause souple unitaire $\neg c_i$ est ajoutée.*

L'ensemble des clauses dures implique que c_i est vraie si et seulement si C_i est satisfaite, ou de manière équivalente, c_i est faux si et seulement si C_i est falsifié. Puisque notre objectif est de maximiser le nombre de clauses falsifiées, nous ajoutons une unité clause de $\neg c_i$ pour chaque $C_i \in \mathcal{C}_I$. Par conséquent, si les variables auxiliaires affectées à vrai dans une solution optimale du codage MaxSAT sont $\{c_{i_1}, \ldots, c_{i_k}\}$, alors $\{C_{i_1}, \ldots, C_{i_k}\}$ est un ensemble minimum de clauses satisfaites dans l'instance I de MinSAT. Nous pouvons construire une affectation optimale de MinSAT en utilisant l'affectation des variables de X_I à partir d'une affectation optimale de MaxSAT partiel.

Exemple 1. Soit I l'instance MinSAT $\{C_1, C_2, C_3, C_4, C_5\}$, $C_1 = a \vee b$, $C_2 = a \vee \neg b$, $C_3 = \neg a \vee b$, $C_4 = \neg a \vee \neg b$, et $C_5 = a \vee c$. Le codage

FIGURE 1 – Un graphe auxiliaire (à gauche) et son complément (à droite)

direct MaxSAT pour I est :

$$
\begin{array}{l}
[c_1 \leftrightarrow a \vee b] \\
[c_2 \leftrightarrow a \vee \neg b] \\
[c_3 \leftrightarrow \neg a \vee b] \\
[c_4 \leftrightarrow \neg a \vee \neg b] \\
[c_5 \leftrightarrow a \vee c] \\
\neg c_1 \\
\neg c_2 \\
\neg c_3 \\
\neg c_4 \\
\neg c_5
\end{array}
$$

Puisque $c_2 = c_3 = c_4 = vrai$ et $c_1 = c_5 = a = b = c = faux$ est la solution optimale pour MaxSAT partiel, alors $\{C_2, C_3, C_4\}$ est un ensemble minimum de clauses satisfaites dans I, et $a = b = c = faux$ est une solution optimale de MinSAT.

4 Coder MinSAT en MaxSAT par l'intermédiare de MaxClique

Soit I l'instance MinSAT composée de l'ensemble de clauses $\mathcal{C}_I$ et l'ensemble de variables X_I. Le graphe auxiliaire $G_I(V_I, E_I)$ correspondant à I est construit comme suit [11] : chaque sommet de V_I correspond à une clause dans $\mathcal{C}_I$. Pour chaque deux sommets v_i et v_j de V_I, l'arête $\{v_i, v_j\}$ est dans E_I si et seulement si les clauses correspondantes c_i et c_j sont telles qu'il existe une variable $x \in X_I$ apparaîssant sous forme positive dans une clause et sous forme négative dans l'autre clause. Le complément d'un graphe auxiliaire G_I est le graphe $\overline{G_I}$ avec les même sommets tel que deux sommets de $\overline{G_I}$ sont adjacents si et seulement s'ils ne sont pas adjacents dans G_I.

L'intuition dans le graphe auxiliaire est la suivante : chaque arête dans le graphe auxiliaire entre les sommets correspondent aux clauses c_i et c_j partageant au moins une variable sous forme positive dans une clause et négative dans l'autre. Aucune affectation de variables ne peut contredire c_i et c_j en même temps ; Ainsi une affectation qui viole c_i doit nécessairement satisfaire c_j, et réciproquement.

Exemple 2. Soit I l'instance de MinSAT $\{C_1, C_2, C_3, C_4, C_5\}$, $C_1 = a \vee b$, $C_2 = a \vee \neg b$, $C_3 = \neg a \vee b$, $C_4 = \neg a \vee \neg b$, et $C_5 = a \vee c$. La figure 1 montre le graphe auxiliaire G_I correspondant à I (à gauche) et son complément $\overline{G_I}$ (à droite).

4.1 Codage 2 : Codage basé sur MaxClique

Marathe et Ravi [11] ont prouvé que le nombre de clauses d'une instance I de MinSAT satisfaites par une affectation optimale est égale à la cardinalité d'une couverture minimum de sommets dans le graphe auxiliaire G_I. D'autre part, l'ensemble des sommets n'appartenant pas à une clique maximum dans le graphe complémentaire $\overline{G_I}$ est une couverture minimum de sommets de G_I. Cela découle du fait que, pour n'importe quel graphe $G(V, E)$, $V' \subseteq V$ est une couverture de sommets si et seulement si $V - V'$ est une clique dans le complément de G. Par conséquent, le nombre de clauses d'une instance I MinSAT satisfaites par une solution optimale est égal au nombre total de sommets moins la cardinalité d'une clique maximum de $\overline{G_I}$.

En outre, une affectation optimale pour l'instance MinSAT I peut être déduite d'une couverture minimum de sommets pour G_I comme suit [11] : les variables apparaîssant dans les clauses correspondant aux sommets qui n'appartiennent pas à la couverture minimum de sommets doivent être affectées de façon à ce que ces clauses soient violées, ce qui est possible parce que, par construction de G_I, ces clauses ne contiennent à la fois x et $\bar{x}$ pour tout $x \in X_I$; d'autres variables sont affectées à une valeur arbitraire.

Par conséquent, nous pouvons déduire une affectation optimale pour l'instance I de MinSAT d'une clique maximum de $\overline{G_I}$: les variables apparaîssant dans les clauses correspondant à sommets appartenant à la clique maximum doivent être affectées en sorte que ces clauses soient violées, d'autres variables sont affectées arbitrairement. Par conséquent, afin de trouver une affectation optimale pour l'instance MinSAT I, nous pouvons chercher une clique maximum dans $\overline{G_I}$. Le problème de MinSAT, par l'intermédiare de MaxClique, est naturellement codé en un problème MaxSAT partiel comme suit.

Définition 2. *Étant donnée une instance I de MinSAT composée de l'ensemble de clause $\mathcal{C}_I = \{C_1, \ldots, C_m\}$, le codage MaxSAT de I basé sur MaxClique est défini comme suit :*

– *L'ensemble de variables propositionnelles est* $\{c_1, \ldots, c_m\}$.
– *Pour toutes les deux clauses* C_i, C_j *de* $\mathcal{C}_I$ *tel que* C_i *contient une occurrence de* l *et* C_j *contient une occurrence de* $\neg l$*, la clause dure* $\neg c_i \vee \neg c_j$ *est ajoutée.*
– *Pour chaque variable propositionnelle* c_i*, une clause unitaire souple* c_i *est ajoutée.*

Observons que chaque variable propositionnelle correspond à un sommet de $\overline{G_I}$, et qu'il existe une clause dure pour tous les deux sommets non adjacents dans $\overline{G_I}$, signifiant que les deux sommets ne peuvent pas être dans la même clique, et une clause unitaire souple pour chaque sommet de $\overline{G_I}$. L'ensemble des variables propositionnelles affectées à vrai dans une solution optimale de l'instance MaxSAT forme une clique maximum de $\overline{G_I}$.

Exemple 3. Soit I l'instance MinSAT $\{C_1, C_2, C_3, C_4, C_5\}$, $C_1 = a \vee b$, $C_2 = a \vee \neg b$, $C_3 = \neg a \vee b$, $C_4 = \neg a \vee \neg b$, et $C_5 = a \vee c$. Le codage MaxSAT pour I basé sur MaxClique est

$$
\begin{array}{l}
[\neg c_1 \vee \neg c_2] \\
[\neg c_1 \vee \neg c_3] \\
[\neg c_1 \vee \neg c_4] \\
[\neg c_2 \vee \neg c_3] \\
[\neg c_2 \vee \neg c_4] \\
[\neg c_3 \vee \neg c_4] \\
[\neg c_3 \vee \neg c_5] \\
[\neg c_4 \vee \neg c_5] \\
c_1 \\
c_2 \\
c_3 \\
c_4 \\
c_5
\end{array}
$$

Notons que contrairement au codage direct (codage 1), le sens prévu de c_i est que c_i est vraie si est seulement si C_i est violée. La partie dure implique que chaque paire de clauses C_i et C_j contenant des littéraux complémentaires ne peuvent pas être violées simultanément. Comme notre objectif est de maximiser le nombre de clauses falsifiées, nous ajoutons une clause souple unitaire c_i pour chaque clause $C_i \in \mathcal{C}_I$. Dans une solution optimale de I, $c_1 = c_5 = vrai$ et $c_2 = c_3 = c_4 = faux$, les sommets correspondant aux variables affectées à vrai appartiennent à une clique maximum de $\overline{G_I}$ et les autres sommets appartiennent à une couverture minimum de sommets en G_I. Puisque une couverture minimum de sommets de G_I est de taille 3, le nombre minimum de clauses satisfaites dans I est 3. Nous pouvons construire une affectation optimale pour MinSAT en falsifiant C_1 et C_5, qui sont les clauses correspondant aux sommets appartenant à une clique maximum de $\overline{G_I}$. Par conséquent, si a, b, et c sont affectées à faux, on obtient une affectation optimale pour l'instance I de MinSAT.

4.2 Codage 3 : Codage amélioré basé sur MaxClique

Le codage de MinSAT en MaxSAT basé sur MaxClique peut être amélioré en réduisant le nombre de clauses souples en considérant le fait suivant : si le graphe auxiliaire d'une instance de MinSAT contient une clique $C = \{c_{i_1}, \ldots, c_{i_k}\}$, alors la partie dure contient les clauses $\neg c_{i_j} \vee \neg c_{i_h}$ pour tout j, h tel que $1 \leq j < h \leq k$. Remarquons que ces clauses dures codent la au-plus-une condition : au plus un littéral dans $\{c_{i_1}, \ldots, c_{i_k}\}$ peut être satisfait. Par conséquent, nous pouvons remplacer les k clauses souples $c_{i_1}, \ldots, c_{i_k}$ par la clause souple $c_{i_1} \vee \cdots \vee c_{i_k}$, car le nombre de clauses satisfaites est le même dans les deux cas dans une solution optimale.

En général, étant donné un graphe auxiliaire $G_I(V_I, E_I)$ d'une instance MinSAT I, une partition en cliques $V_1, \ldots, V_k$ de G_I, le codage amélioré de MinSAT en MaxSAT basé sur MaxClique a, pour chaque clique V_i dans la partition, une clause souple formée par la disjonction des sommets de V_i.

Définition 3. *Étant donnée une instance* I *de MinSAT composée de l'ensemble de clause* $\mathcal{C}_I = \{C_1, \ldots, C_m\}$*, le codage MaxSAT amélioré de* I *basé sur MaxClique est défini comme suit :*

– *L'ensemble de variables propositionnelles est* $\{c_1, \ldots, c_m\}$.
– *Pour toutes les deux clauses* C_i, C_j *de* $\mathcal{C}_I$ *tel que* C_i *contient une occurrence de* l *et* C_j *contient une occurrence de* $\neg l$*, la clause dure* $\neg c_i \vee \neg c_j$ *est ajoutée.*
– *Dans une partition du graphe auxiliaire* $G_I(V_I, E_I)$ *en cliques, une clause souple est ajoutée pour chaque clique qui est un* ou *logique des variables correspondantes aux sommets dans la clique.*

Exemple 4. Soit I l'instance MinSAT $\{C_1, C_2, C_3, C_4, C_5\}$, $C_1 = a \vee b$, $C_2 = a \vee \neg b$, $C_3 = \neg a \vee b$, $C_4 = \neg a \vee \neg b$, et $C_5 = a \vee c$. Le codage amélioré de I en MaxSAT basé sur MaxClique est :

$$
\begin{array}{l}
[\neg c_1 \vee \neg c_2] \\
[\neg c_1 \vee \neg c_3] \\
[\neg c_1 \vee \neg c_4] \\
[\neg c_2 \vee \neg c_3] \\
[\neg c_2 \vee \neg c_4] \\
[\neg c_3 \vee \neg c_4] \\
[\neg c_3 \vee \neg c_5] \\
[\neg c_4 \vee \neg c_5] \\
c_1 \vee c_2 \vee c_3 \vee c_4 \\
c_5
\end{array}
$$

Nous remarquons que $\{\{c_1, c_2, c_3, c_4\}, \{c_5\}\}$ est une partition de clique du graphe auxiliaire de I (cf. Figure 1), et que les six premières clauses dures signifient qu'au plus un des littéraux c_1, c_2, c_3, c_4 est satisfait par une affectation optimale.

Puisque le problème de trouver une partition minimale de cliques d'un graphe est NP-difficile, nous proposons d'appliquer une méthode heuristique pour partitionner le graphe en cliques pour obtenir le codage amélioré. L'heuristique utilisée dans notre étude empirique est l'algorithme 1.

Étant donné un graphe G et une liste P de cliques disjointes de G, P est initialement vide, nous dénotons par #c(v) le nombre de cliques dans P dans lesquelles le sommet v peut être inséré pour les agrandir. Algorithm 1 partitionne G en cliques en sélectionnant le sommet v avec le minimum #c(v), dans le cas où il y a plusieurs sommets avec le minimum #c(v), celui qui a le moindre degré (c'est à dire avec le minimum sommets adjacents) est choisi. Puis l'algorithme insère v dans la première clique de P où v peut être inséré si #c(v)>0, et crée une nouvelle clique dans P pour y insérer v sinon. Soit C une clique dans P, et v_i et v_j sont deux sommets non adjacents qui peuvent être respectivement insérés dans C pour obtenir une plus grande clique, observons que l'insertion de v_i dans C empêche v_j d'être inséré dans C, et réciproquement. L'intuition dans la sélection de v dans l'algorithme 1 est que le sommet le plus contraint (c'est à dire avec le moins de possibilités dans P) est inséré en premier, en évitant de créer une nouvelle clique pour ce sommet après l'insertion d'autres sommets.

Nous avons comparé l'algorithme 1 avec les heuristiques naturelles suivantes :

`H-Min` : même que l'algorithme 1 sauf que chaque fois que le sommet avec le plus petit degré est sélectionné pour être inséré dans une clique ;

`H-Max` : même que l'algorithme 1 sauf que chaque fois que le sommet avec le plus grand degré est sélectionné pour être inséré dans une clique ;

`H-Maximal-Clique-Min` : une clique maximale est calculée à la fois. Tant qu'il y a des sommets qui peuvent être insérés dans une clique pour l'agrandir, un de ces sommets avec le plus petit degré est inséré dans la clique. Après que la clique devient maximale et retiré du graphe, d'autres cliques maximales sont calculées de la même manière dans le graphe restant jusqu'à ce que le graphe devienne vide ;

`H-Maximal-Clique-Max` : même que H-Maximal-clique-Min, sauf que chaque fois que le sommet avec le plus grand degré est inséré dans la clique en construction.

L'Heuristique H-Min est utilisée dans [14] pour partitionner un graph en ensembles indépendents, et les heuristiques H-Maximal-Clique-Min et H-Maximal-Clique-Max sont utilisées dans [3, 12] pour partitionner aussi un graphe en ensemble indépendents. Ces partitions sont utilisées pour donner une borne supérieure de la caridinalité d'une clique maximum dans le graphe. Leur complexité en temps est meilleure que l'algorithme 1, puisque le calcul de #c n'est pas très simple dans l'algorithme 1.

Algorithm 1: cliquePartition(G)

Input: Un graphe G=(V, E)
Output: Une partition en cliques de G

```
1  begin
2      P ← ∅;
3      while G n'est pas vide do
4          Pour chaque sommet v dans G, calculer le
           nombre de cliques #c dans P dans lesquelles v
           est adjacent à tous les sommets;
5          v ← le sommet de G avec le minimum #c et
           puis avec le minimum degré ;
6          supprimer v de G;
7          if il y a une clique C dans P dans laquelle v
           est adjacent à tous les sommets then
8              ajouter v à C ;
9          else
10             créer une nouvelle clique C;
11             ajouter v à C;
12             P ← P ∪ {C} ;
13     retourner P ;
14 end
```

Cependant, l'algorithme 1 est quand même rapide et est significativement meilleur que toutes les autres heuristiques, car il donne la plus petite partition selon nos résultats expérimentaux (non reportés ici). Nous aimerions également faire remarquer que nous obtenons une meilleure borne inférieure initiale de MaxSAT si nous réduisons le nombre de clauses souples. Ainsi l'algorithme 1 permet le minimum nombre de clauses souples dans le codage amélioré et la meilleure borne inférieure initial de MaxSAT parmi toutes heuristiques testées.

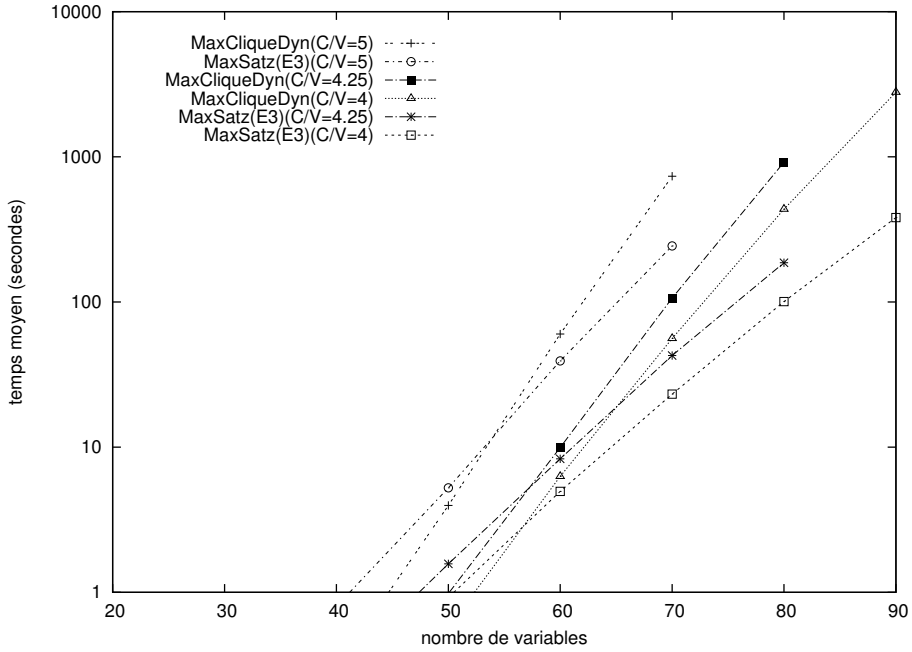
5 Résultats expérimentaux

Nous avons effectué une comparaison empirique des trois codages de MinSAT en MaxSAT sur plusieurs solveurs MaxSAT, et comparé nos résultats avec les résultats obtenus par la résolution de MinSAT par les deux meilleurs solveurs exacts de MaxClique à nos connaissances.

Les solveurs utilisés dans notre étude empirique sont les suivants :

– MaxSatz [9] : nous avons utilisé la version pour MaxSAT pondéré partiel qui a participé à l'évaluation

FIGURE 2 – *Min-3-SAT, échelle logarithmique*



MaxSAT 2009 avec un petit modification de l'heuristique utilisée pour sélectionner les variables dans la détection des littéraux d'échec : alors que la version de l'évaluation MaxSAT 2009 applique la détection des littéraux d'échec à toutes les variables apparaîssant au moins deux fois négativement et deux fois positivement dans les clauses binaire, dans la version utilisée dans ce papier, elle est appliqué à toutes les variables ayant au moins une occurrence positive et au moins deux occurrences négatives dans les clauses binaires, et à toutes les variables ayant au moins une occurrence négative et au moins deux occurrences positives dans les clauses binaires. Ceci parce que les variables dans une instance MaxSAT codant une instance MaxClique a seulement une occurrence positive, même si elles peuvent avoir plusieures occurrences négatives dans les clauses dures.

– WBO [10] : nous avons utilisé la version de ce solveur de MaxSAT pondéré partiel qui a participé à l'évaluation MaxSAT 2009.
– PM2 [1] : nous avons utilisé la version de ce solveur de MaxSAT partiel qui a participé à l'évaluation MaxSAT 2009.
– SAT4J-Maxsat : [1] nous avons utilisé la version de ce solveur de MaxSAT pondéré partiel qui a participé à l'évaluation MaxSAT 2009.
– MaxCliqueDyn [7] : nous avons obtenu le code source de ce solveur MaxClique de D. Janezic, qui est l'un de ses auteurs, en Janvier 2010.
– Cliquer [13] : nous avons utilisé la dernière version publique de ce solveur, qui a été publié en 2008. [2]

Les MaxClique solveurs MaxCliqueDyn et Cliquer résolvent une instance I de MinSAT en cherchant une clique maximum dans le graphe auxiliaire complémentaire $\overline{G_I}$. Les MaxSAT solveurs MaxSatz, WBO, PM2, et SAT4J-Maxsat résolvent I en utilisant trois codages de MinSAT en MaxSAT partiel qui sont définis dans les sections précédentes.

Comme benchmarks, nous avons généré des instances aléatoires de Min-2-SAT et Min-3-SAT dont chaque clause contient exactement 2 et 3 littéraux respectivement. Le nombre de variables dans les instances Min-2-SAT et Min-3-SAT variait de 20 à 100. Les ratios #clause sur #variable pour Min-2-SAT sont 1 et 3, et les ratios #clause sur #variable pour Min-3-SAT sont 4, 4,25 et 5. Des expérimentations ont été effectuées sur un MacPro avec un processeur Intel Xeon à 2,8 GHz et 4 Gb de mémoire avec Mac OS X 10.5. Le temps d'exécution est limité à 3 heures par instance.

Le tableau 1 (resp. le tableau 2) contient les résultats expérimentaux obtenus pour les instances de Min-2-SAT (resp. Min-3-SAT) avec les solveurs de MaxSAT et de MaxClique. Le tableau 1 et le tableau 2 montrent, pour

1. http ://www.sat4j.org/

2. http ://users.tkk.fi/pat/cliquer.html

TABLE 1 – Nombre d'instances résolues en moins de 3 heures et temps d'exécution en secondes de MaxSatz, MaxCliqueDyn (Dyn dans le tableau), Cliquer, PM2, WBO et SAT4J-Maxsat sur Min-2-SAT aléatoire. C/V est le ratio #clause sur #variable.

instance		*MaxSatz*			*Dyn*	*Cliquer*	*PM2*			*WBO*			*sat4j-maxsat*		
#var	C/V	*E*1	*E*2	*E*3			*E*1	*E*2	*E*3	*E*1	*E*2	*E*3	*E*1	*E*2	*E*3
20	1.0	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.07	0.04
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*
30	1.0	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.18	0.14	0.05
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*
40	1.0	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.48	0.27	0.05
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*
50	1.0	0.01	0.01	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.92	0.87	0.06
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*
60	1.0	0.01	0.08	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	2.71	2.73	0.06
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*
70	1.0	0.02	0.21	0.00	0.02	0.00	0.01	0.00	0.00	0.00	0.00	0.00	17.40	27.02	0.05
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*
80	1.0	0.03	1.18	0.00	0.02	0.00	0.01	0.00	0.00	0.02	0.00	0.00	505.9	190.6	0.08
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*
90	1.0	0.58	9.38	0.00	0.02	0.00	0.01	0.00	0.00	0.02	0.01	0.00	2710	1995	0.08
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(24)*	*(45)*	*(50)*
100	1.0	0.26	27.34	0.00	0.02	0.00	0.01	0.01	0.00	0.02	0.01	0.00	2596	6358	0.11
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(14)*	*(12)*	*(50)*
20	3.0	0.01	0.03	0.00	0.02	0.00	0.02	0.02	0.00	0.01	0.01	0.00	0.81	1.20	0.07
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*
30	3.0	0.03	1.39	0.01	0.02	0.00	0.14	0.14	0.00	0.14	0.22	0.00	112.5	1530	0.17
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(32)*	*(50)*
40	3.0	0.12	52.95	0.01	0.03	0.01	0.19	0.51	0.00	0.07	0.52	0.00	9904	-	0.20
		(50)	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(4)*	*(0)*	*(50)*
50	3.0	0.89	2053	0.01	0.04	1.23	5.25	7.89	0.00	42.35	51.12	0.01	-	-	0.43
		(50)	*(48)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(0)*	*(0)*	*(50)*
60	3.0	7.42	-	0.02	0.07	15.88	49.35	79.77	0.01	207.6	93.17	0.20	-	-	2.41
		(50)	*(0)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(0)*	*(0)*	*(50)*
70	3.0	43.25	-	0.03	0.13	68.01	144.3	135.8	0.01	225.1	143.9	0.08	-	-	2.42
		(50)	*(0)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(0)*	*(0)*	*(50)*
80	3.0	201.2	-	0.05	0.25	302.4	199.9	231.8	0.02	416.8	432.9	0.70	-	-	3.35
		(50)	*(0)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(0)*	*(0)*	*(50)*
90	3.0	1355	-	0.08	0.91	895.5	353.1	330.3	0.05	558.8	295.1	1.37	-	-	5.38
		(49)	*(0)*	*(50)*	*(50)*	*(42)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(0)*	*(0)*	*(50)*
100	3.0	3258	-	0.11	1.51	1185	256.7	455.3	0.09	548.5	665.9	29.27	-	-	96.91
		(46)	*(0)*	*(50)*	*(50)*	*(44)*	*(10)*	*(50)*	*(50)*	*(50)*	*(50)*	*(50)*	*(0)*	*(0)*	*(50)*

TABLE 2 – Nombre d'instances résolues en moins de 3 heures et temps d'exécution en secondes de MaxSatz, MaxCliqueDyn (Dyn dans le tableau) et Cliquer sur Min-3-SAT aléatoire. C/V est le ratio #clause sur #variable.

instance		*MaxSatz*			*Dyn*	*Cliquer*
#var	C/V	*E*1	*E*2	*E*3		
20	4.00	0.02*(50)*	0.10*(50)*	0.01*(50)*	0.02*(50)*	0.00*(50)*
30	4.00	0.24*(50)*	7.73*(50)*	0.04*(50)*	0.03*(50)*	0.04*(50)*
40	4.00	3.28*(50)*	507.8*(50)*	0.19*(50)*	0.09*(50)*	4.40*(50)*
50	4.00	49.30*(50)*	- *(0)*	0.92*(50)*	0.57*(50)*	454.4*(50)*
60	4.00	742.4*(50)*	- *(0)*	4.96*(50)*	6.29*(50)*	3886*(11)*
70	4.00	5735*(34)*	- *(0)*	23.21*(50)*	56.01*(50)*	- *(0)*
80	4.00	- *(0)*	- *(0)*	100.7*(50)*	436.0*(50)*	- *(0)*
90	4.00	- *(0)*	- *(0)*	381.5*(50)*	2788*(50)*	- *(0)*
100	4.00	- *(0)*	- *(0)*	1658*(33)*	5610*(9)*	- *(0)*
20	4.25	0.02*(50)*	0.14*(50)*	0.01*(50)*	0.02*(50)*	0.00*(50)*
30	4.25	0.30*(50)*	12.05*(50)*	0.05*(50)*	0.03*(50)*	0.07*(50)*
40	4.25	4.67*(50)*	992.5*(50)*	0.28*(50)*	0.12*(50)*	15.94*(50)*
50	4.25	75.6*(50)*	- *(0)*	1.57*(50)*	0.98*(50)*	945.0*(49)*
60	4.25	1153*(50)*	- *(0)*	8.31*(50)*	9.94*(50)*	5385*(10)*
70	4.25	5989*(5)*	- *(0)*	42.77*(50)*	106.4*(50)*	- *(0)*
80	4.25	- *(0)*	- *(0)*	186.3*(50)*	917.4*(50)*	- *(0)*
90	4.25	- *(0)*	- *(0)*	760.4*(50)*	4453*(41)*	- *(0)*
100	4.25	- *(0)*	- *(0)*	2819*(26)*	- *(0)*	- *(0)*
20	5.00	0.03*(50)*	0.39*(50)*	0.02*(50)*	0.02*(50)*	0.00*(50)*
30	5.00	0.63*(50)*	55.91*(50)*	0.14*(50)*	0.04*(50)*	0.26*(50)*
40	5.00	10.87*(50)*	5693*(48)*	0.80*(50)*	0.30*(50)*	63.98*(50)*
50	5.00	226.8*(50)*	- *(0)*	5.24*(50)*	3.97*(50)*	3035*(35)*
60	5.00	3803*(48)*	- *(0)*	39.3*(50)*	60.14*(50)*	- *(0)*
70	5.00	- *(0)*	- *(0)*	243.4*(50)*	735.2*(50)*	- *(0)*
80	5.00	- *(0)*	- *(0)*	1512*(50)*	6355*(41)*	- *(0)*
90	5.00	- *(0)*	- *(0)*	5167*(39)*	- *(0)*	- *(0)*
100	5.00	- *(0)*	- *(0)*	- *(0)*	- *(0)*	- *(0)*

FIGURE 3 – *Min-2-SAT*

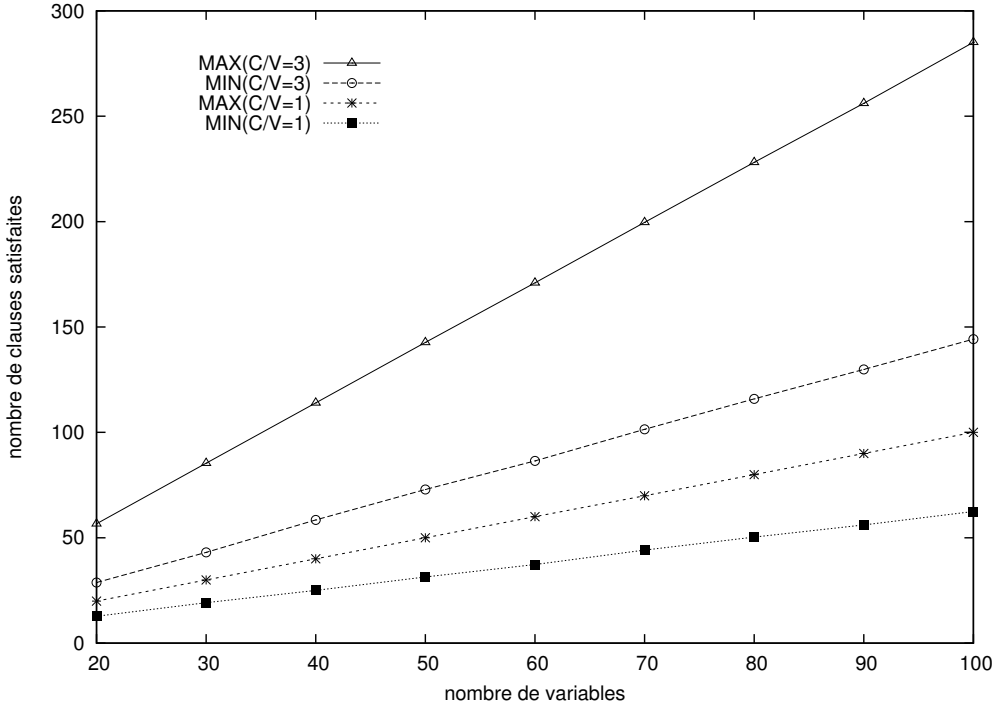
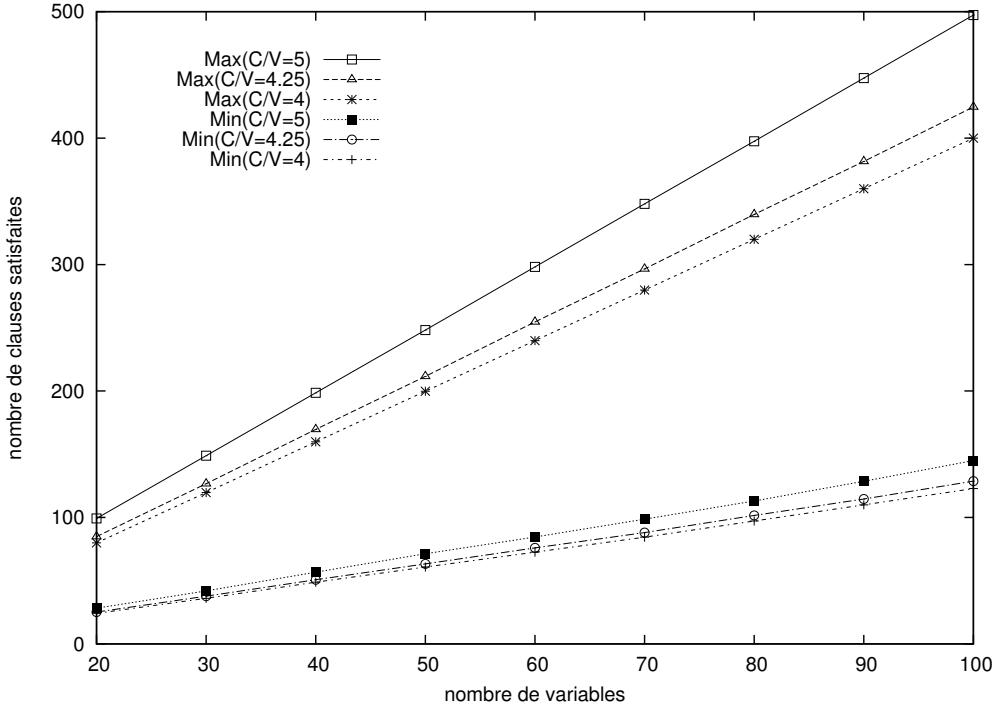


FIGURE 4 – *Min-3-SAT*



chaque solveur, le nombre d'instances résolues en moins de 3 heures (entre parenthèses) sur un ensemble de 50 instances à chaque point, et le temps moyen nécessaire pour résoudre ces instances résolues. Les solveurs MaxSAT qui ne sont pas inclus dans le tableau 2 ont été loin d'être compétitifs sur les instance Min-3-SAT. Les trois codages sont désignés par $E1$ (codage 1), $E2$ (codage 2), et $E3$ (codage 3).

Les résultats expérimentaux montrent que le codage direct de MinSAT en MaxSAT partiel (Codage 1) est meilleur que le codage basé sur MaxClique (Codage 2) pour tous les solveurs MaxSAT. Toutefois, lorsque le codage basé sur MaxClique est amélioré en utilisant une bonne partition en cliques de G_I, le codage basé sur MaxClique (Codage 3) est le plus performant pour tous les solveurs de MaxSAT. Plus important encore, le codage 3 fait MaxSatz significativement meilleur pour trouver une clique maximum dans $\overline{G_I}$ que les solveurs spécifiques de MaxClique comme MaxCliqueDyn et Cliquer. La figure 2 montre que, pour les trois ratios #clause sur #variable de Min-3-SAT, la différence de performance entre MaxSatz (utilisant le codage 3) et MaxCliqueDyn croît avec le problème de la taille. Ce phénomène surprenant mérite une étude future.

Les figures 3 et 4 montrent les nombres maximum et minimum de clauses satisfaites (pour 2-SAT et 3-SAT). Dans ces figures, Min représente le nombre minimum de clauses satisfaites et Max représente le nombre maximum de clauses satisfaites. Nous pouvons voir que si le ratio #clause sur #variable (C/V) augmente, alors l'écart entre Min et Max ne cesse d'augmenter.

6 Conclusions

Nous avons présenté une approche originale générique pour résoudre le problème MinSAT de façon exacte, qui utilise les codages MaxSAT et des solveurs MaxSAT, et avons montré qu'en particulier, l'approche basée sur le réduction de MaxSAT à MaxClique, puis réduire MaxClique à MaxSAT en utilisant le codage amélioré (le codage 3) est meilleure que d'utiliser directement les algorithmes exacts de MaxClique pour résoudre MinSAT, au moins pour les instances testées. En l'absence d'un solveur efficace de séparation et évaluation spécifique pour MinSAT, nous pensons que notre Codage 3 fournit une alternative qui est appropriée et compétitive pour résoudre des problèmes réels de MinSAT, et permet de promouvoir la recherche sur MinSAT.

Ce papier, malgré son caractère préliminaire, contient d'importantes contributions et suggère futures directions de recherche telles que l'élaboration de systèmes d'inférence, la conception des méthodes de calcul de bornes inférieures et supérieures spécifiques pour MinSAT, l'extension de notre approche au formalisme des clauses dures et clauses souples pondérées, et les applications des technologies MinSAT pour résoudre des problèmes industriels.

Remerciements : le problème MinSAT a, à l'origine, été demandé par Laurent Simon au premier auteur.

Références

[1] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. Solving (weighted) partial MaxSAT through satisfiability testing. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT-2009, Swansea, UK*, pages 427–440. Springer LNCS 5584, 2009.

[2] Adi Avidor and Uri Zwick. Approximating min 2-sat and min 3-sat. *Theory of Computing Systems*, 38(3) :329–345, 2005.

[3] T. Fahle. Simple and fast : Improving a branch-and-bound algorithm for maximum clique. In *Proceedings of ESA-2002*, pages 485–498, 2002.

[4] Giovanni Felici and Klaus Truemper. A minsat approach for learning in logic domains. *INFORMS Journal on Computing*, 14(1) :20–36, 2002.

[5] Avraham Goldstein, Petr Kolman, and Jie Zheng. Minimum common string partition problem : Hardness and approximations. *Electr. J. Comb.*, 12, 2005.

[6] Rajeev Kohli, Ramesh Krishnamurti, and Prakash Mirchandani. The minimum satisfiability problem. *SIAM J. Discrete Mathematics*, 7(2) :275–283, 1994.

[7] J. Konc and D. Janezic. An improved branch and bound algorithm for the maximum clique problem. *Communications in Mathematical and in Computer Chemistry*, 58 :569–590, 2007.

[8] Chu Min Li and F. Manyà. Max-SAT, hard and soft constraints. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 613–631. IOS Press, 2009.

[9] Chu Min Li, Felip Manyà, and Jordi Planes. New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30 :321–359, 2007.

[10] Vasco M. Manquinho, João P. Marques Silva, and Jordi Planes. Algorithms for weighted boolean optimization. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, SAT-2009, Swansea, UK*, pages 495–508. Springer LNCS 5584, 2009.

[11] M. V. Marathe and S. S. Ravi. On approximation algorithms for the minimum satisfiability problem. *Information Processing Letters*, 58 :23–29, 1996.

[12] P. R. J. Ostergard. A fast algorithm for the maximum clique problem. In *Discrete Applied Mathematics 120 (2002)*, pages 197–207, 2002.

[13] P. R. J. Ostergard. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120 :197–207, 2002.

[14] E. Tomita and T. Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *Proc. Discrete Mathematics and Theoretical Computer Science, LNCS 2731*, pages 278–289, 2003.

Un Algorithme de séparation et évaluation Efficace Basé sur MaxSAT pour le Problème Maxclique

Chu-Min LI **Zhe Quan**

Université de Picardie Jules Verne,
Amiens, France
chu-min.li@u-picardie.fr quanzhe@gmail.com

Résumé

Fréquemment, Les algorithmes de séparation et évaluation pour le problème Maxclique utilisent une borne supérieure basée sur la partition d'un graphe en ensembles indépendants, pour obtenir la borne supérieure d'une clique maximum du graphe, qui ne peut pas être très fine pour les graphes imparfaits. Dans ce papier, nous proposons un nouveau codage de Maxclique en MaxSAT et l'utilisation de l'approche MaxSAT pour améliorer la borne supérieure basée sur la partition P. De cette manière, la puissance d'algorithmes de partitionnement d'un graphe spécifiques pour Maxclique et la performance de la méthode MaxSAT dans la logique propositionnelle sont naturellement associées pour résoudre Maxclique. Les résultats expérimentaux montrent que l'approche est très efficace sur les graphes aléatoires difficiles et sur les benchmarks Maxclique de DIMACS, et permet de fermer un problème DIMACS ouvert.

Mots-Clés : séparation et évaluation, Maxclique, MaxSAT

Abstract

State-of-the-art branch-and-bound algorithms for the maximum clique problem (Maxclique) frequently use an upper bound based on a partition P of a graph into independent sets for a maximum clique of the graph, which cannot be very tight for imperfect graphs. In this paper we propose a new encoding from Maxclique into MaxSAT and use MaxSAT technology to improve the upper bound based on the partition P. In this way, the strength of specific algorithms for Maxclique in partitioning a graph and the strength of MaxSAT technology in propositional reasoning are naturally combined to solve Maxclique. Experimental results show that the approach is very effective on hard random graphs and on DIMACS Maxclique benchmarks, and allows to close an open DIMACS problem.

Keywords : Branch-and-Bound, Maxclique, MaxSAT

1 Introduction

Considérons un graphe non orienté G=(V, E), où V est un ensemble de n sommets $\{v_1, v_2, ..., v_n\}$ et E est un ensemble de m arêtes. L'arête (v_i, v_j) avec $i \neq j$ connecte les deux sommets v_i et v_j. Une clique de G est un sous-ensemble C de V tel que tous les deux sommets de C sont connectés par une arête. Le problème Maxclique consiste à trouver une clique de G de cardinalité maximum dénotée par $\omega(G)$.

Un ensemble indépendant de G est un sous-ensemble I de V de sorte que aucun sommet n'est connecté à un autre dans I. Étant donné G, le GCP (Graph Coloring Problem) demande de trouver le nombre minimum de couleurs (i.e., le nombre chromatique $\chi(G)$) nécessaires pour colorer les sommets de G tel que deux sommets connectés ne partagent pas la même couleur. La résolution de GCP est équivalent à partitionner G en un nombre minimum d'ensembles indépendants, parce que tous les sommets colorés par la même couleur dans G constituent un ensemble indépendant. Nous avons $\chi(G) \geq \omega(G)$, puisque chaque sommet dans une clique doit être affecté à une couleur différente. Un graphe G est parfait si $\chi(G')$=$\omega(G')$ pour tout sous-graphe induit G' de G. Maxclique est un très important problème combinatoire NP-difficile, car il apparaît dans beaucoup d'applications du monde réel. De nombreux efforts ont été consacrés à le résoudre. Voir [11] où Pardolos et Xu ont donné un survol des premiers travaux intensifs sur Maxclique. Dans la littérature, nous dis-

tinguons deux types d'algorithmes pour Maxclique : des méthodes d'approximation incluant les algorithmes de recherche locale stochastique, e.g. [12], et les algorithmes exacts incluant les algorithmes par séparation et évaluation, e.g. [14, 5, 10, 1, 2, 13]. Les algorithmes d'approximation sont capables de résoudre des problèmes Maxclique de grande taille, mais ne garantissent pas l'optimalité de leurs solutions. Les algorithmes exacts garantissent l'optimalité des solutions qu'ils trouvent. Dans cet papier, nous nous concentrons sur les algorithmes de séparation et évaluation pour Maxclique.

Les algorithmes de séparation et évaluation pour Maxclique utilisent fréquemment une solution heuristique pour GCP comme une borne supérieure. Il y a deux inconvénients dans cette approche : (i) le nombre d'ensembles indépendants dans une partition calculée de G n'est généralement pas minimum, puisque GCP est lui-même NP-difficile ; (ii) même si la partition est minimale, il peut y avoir une grande différence entre $\chi(G)$ et $\omega(G)$ lorsque G n'est pas parfait, de sorte que $\chi(G)$ n'est pas une borne supérieure de bonne qualité de $\omega(G)$. Ces deux inconvénients pourraient probablement expliquer la stagnation de la recherche sur les algorithmes exacts pour Maxclique. Ainsi, des bornes supérieures de meilleure qualité sont nécessaires dans les algorithmes de séparation et évaluation pour Maxclique.

Récemment des progrès considérables ont été faits dans la résolution de MaxSAT (voir [6] pour une présentation de ces progrès). Etant donné un ensemble de variables booléennes $\{x_1,x_2,...,x_n\}$, un littéral l est une variable x_i ou sa négation $\bar{x}_i$, une clause est un *ou* logique des littéraux. Une formule CNF ϕ est un ensemble de clauses. Le problème MaxSAT demande de trouver une affectation des valeurs de vérité (0 ou 1) aux variables booléennes pour maximiser le nombre de clauses satisfaites dans ϕ. Maxclique peut être codé en MaxSAT puis résolu en utilisant un solveur MaxSAT. Malheureusement, les solveurs MaxSAT ne sont pas compétitifs pour résoudre Maxclique, parce que leur raisonnement n'est pas guidé par les propriétés structurelles du graphe.

Dans cet papier, nous montrons que la technologie développée pour résoudre MaxSAT peut être utilisée pour améliorer les bornes supérieures dans l'algorithme de séparation et évaluation pour Maxclique. Nous présentons tout d'abord quelques préliminaires et un algorithme de séparation et évaluation de base pour Maxclique appelé MaxCLQ, avant de présenter quelques bornes supérieures précédentes pour Maxclique. Ensuite, nous proposons un nouveau codage de Maxclique en MaxSAT, ce qui nous permet d'utiliser la technologie MaxSAT pour améliorer les précédente bornes supérieures pour Maxclique. Nous étudions ensuite le comportement de notre approche intégrée dans MaxCLQ, et comparons MaxCLQ avec les meilleurs algorithmes exacts à nos connaissances sur les graphes aléatoires et sur les benchmarks Maxclique de DIMACS qui est largement utilisés [1] pour évaluer les algorithmes de Maxclique dans la littérature. Les résultats expérimentaux montrent que MaxCLQ est très efficace grâce à la technologie MaxSAT intégrée puisqu'il est, à notre connaissance, capable de résoudre un problème ouvert DIMACS pour la première fois.

2 Preliminaires

Une clique C d'un graphe G=(V, E) est maximum si aucune clique de cardinalité plus grande n'existe dans G. G peut avoir plusieurs cliques maximums. Soit V' un sous-ensemble de V, le sous-graphe G' de G induit par V' est défini comme G'=(V', E'), où $E' = \{(v_i, v_j) \in E \mid v_i, v_j \in V'\}$. Etant donné un sommet v de G, l'ensemble des sommets voisins de v est noté $\Gamma(v)$=$\{v' \mid (v, v') \in E\}$. La cardinalité $|\Gamma(v)|$ de $\Gamma(v)$ est appelée le degré de v. G_v désigne le sous-graphe induit par $\Gamma(v)$ et $G\backslash v$ est le graphe induit par $V\backslash\{v\}$. $G\backslash v$ est obtenue en enlevant v et toutes les arêtes connectant v à G. La densité d'un graphe de n sommets et m arêtes est égale à $2m/(n(n-1))$.

Soit un problème MaxSAT ϕ, une affectation de valeurs de vérité aux variables boolennes satisfait un littéral positif x si x=1, satisfait un littéral négatif $\bar{x}$ si x=0, et satisfait une clause si au moins un littéral dans la clause est satisfait. Nous distinguons spécialement les clauses unitaires qui ne contiennent qu'un seul littéral, et les clauses vides qui ne contiennent pas de littéral et ne peuvent pas être satisfaites. Un problème MaxSAT ϕ est dit partiel, si certaines clauses de ϕ sont dures, c'est-à-dire qu'elles doivent être satisfaites dans toutes les solutions, et les autres clauses sont souples et peuvent être violées dans une solution. Un problème MaxSAT partiel demande de trouver une affectation satisfaisant toutes les clauses dures en maximisant le nombre de clauses souples satisfaites.

La façon habituelle de coder un problème Maxclique en un problème MaxSAT est d'introduire une variable booléenne x pour chaque sommet v de G, avec le sens que x=1 si et seulement si v appartient à la clique maximum cherchée. Ensuite, une clause binaire dure $\bar{x}_i \vee \bar{x}_j$ est ajoutée pour chaque paire de sommets v_i et v_j qui ne sont pas connectés, signifiant que v_i et v_j ne peuvent pas être dans la même clique. Il est clair que toute affectation satisfaisant toutes les clauses dures donne une clique. Enfin, une clause unitaire souple x est ajoutée pour chaque sommet v. Maximiser le nombre de clauses souples satisfaites, tout en satisfaisant toutes les clauses dures donne une clique maximum.

Par exemple, l'instance Maxclique dans la figure 1 peut être codée en une instance MaxSAT composée des clauses souples : $\{x_1, x_2, x_3, x_4, x_5, x_6\}$, et des clauses dures :

1. disponibles à http://cs.hbg.psu.edu/ txn131/clique.html

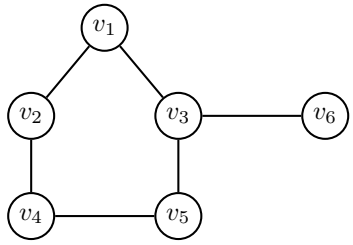


FIGURE 1 – Un simple graphe imparfait ($\chi(G)$=3 et $\omega(G)$=2)

$\{\bar{x}_1 \vee \bar{x}_4,\ \bar{x}_1 \vee \bar{x}_5,\ \bar{x}_2 \vee \bar{x}_3,\ \bar{x}_2 \vee \bar{x}_5,\ \bar{x}_3 \vee \bar{x}_4,\ \bar{x}_1 \vee \bar{x}_6,\ \bar{x}_2 \vee \bar{x}_6,\ \bar{x}_4 \vee \bar{x}_6,\ \bar{x}_5 \vee \bar{x}_6\}$.

3 Un Algorithme de séparation et évaluation de base pour Maxclique

L'Algorithme 1 montre le pseudo-code d'un algorithme de séparation et évaluation pour Maxclique, inspiré par l'algorithme de séparation et évaluation MaxSatz pour MaxSAT [7]. Nous illustrons le principe de cet algorithme en utilisant le graphe G de la figure 1. Initialement C=$\emptyset$ et LB=0, et MaxCLQ (G, $\emptyset$, 0) trouve une clique maximum ($\{v_3, v_6\}$) de G comme suit.

Algorithm 1: MaxCLQ(G, C, LB), un algorithme de séparation et évaluation de base pour Maxclique

Input: Un graphe G=(V, E), une clique C, et une borne inférieure LB représentant la cardinalité de la plus grande clique trouvée jusqu'à présent
Output: $C \cup C'$, où C' est une clique maximum de G, si $|C \cup C'| > LB$, $\emptyset$ sinon

```
1  begin
2      si |V|=0 alors retourner C ;
3      UB ← surestimation(G)+|C| ;
4      si LB ≥ UB alors retourner ∅ ;
5      choisir un sommet v de degré minimum de G ;
6      C1 ← MaxCLQ(Gv, C ∪{v}, LB) ;
7      si |C1| > LB alors LB ← |C1| ;
8      C2 ← MaxCLQ(G\v, C, LB) ;
9      si |C1| ≥ |C2| alors retourner C1 ; sinon
       retourner C2 ;
10 end
```

Dans la ligne 3, la function surestimation(G) donne une borne supérieure clairement supérieure à 0 pour une clique maximum de G, puisque G n'est pas vide. Ensuite MaxCLQ choisit v_6 comme sommet de branchement (ligne 5), car il est de degré minimum. Ainsi toutes les cliques de G sont implicitement divisées en deux ensembles : l'ensemble des cliques contenant v_6 et l'ensemble des cliques ne contenant pas de v_6. Le premier appel récursif (ligne 6) MaxCLQ(G_{v_6}, $\{v_6\}$, 0) retourne une clique ($\{v_3, v_6\}$) contenant v_6, où G_{v_6}=($\{v_3\}$, $\emptyset$).

Puis, la ligne 7 change la borne inférieure LB à 2. Le deuxième appel récursif MaxCLQ($G\backslash v_6$, $\emptyset$, 2) essaie de trouver une clique maximum ne contenant pas de v_6 et plus grande que 2 dans $G\backslash v_6$ (ligne 8), où $G\backslash v_6$ est le cycle composé de $\{v_1,v_2,v_3,v_4,v_5\}$. Si la fonction surestimation($G\backslash v_6$) utilise une méthode de coloration et retourne 3 comme borne supérieure (rappelons que $\chi(G\backslash v_6)$=3) pour une clique maximum de $G\backslash v_6$, MaxCLQ($G\backslash v_6$, $\emptyset$, 2) doit faire deux autres appels récursifs en choisissant un sommet de branchement, e.g. v_1 : MaxCLQ($(G\backslash v_6)_{v_1}$, $\{v_1\}$, 2) et MaxCLQ($(G\backslash v_6)\backslash v_1$, $\emptyset$, 2). L'exécution de ces deux appels retourne tous les deux $\emptyset$. Ainsi, MaxCLQ($G\backslash v_6$, $\emptyset$, 2) retourne $\emptyset$ à la ligne 9. Cependant, si la fonction surestimation($G\backslash v_6$) utilise l'approche proposée dans ce papier, il retournera 2 comme borne supérieure, de telle sorte que UB=LB et MaxCLQ($G\backslash v_6$, $\emptyset$, 2) retourne directement $\emptyset$ (line 4) sans d'autres appels récursifs, puisque UB=LB signifie qu'une clique de cardinalité plus grande ne peut pas être trouvée.

Enfin, MaxCLQ(G, $\emptyset$, 0) retourne C_1=$\{v_3, v_6\}$ à la ligne 9.

4 Survol des bornes supérieures précédentes pour Maxclique

Les algorithmes de séparation et évaluation pour Maxclique utilisent fréquemment des solutions heuristiques de GCP qui peuvent être obtenues en un temps raisonnable comme leurs bornes supérieures. Cette approche se base sur la propriété suivante :

Proposition 1 *Soit $\omega(G)$ la cardinalité d'une clique maximum du graphe G. Si G peut être partitionné en k ensembles indépendants, alors $\omega(G) \leq k$.*

Comme un prétraitement, l'algorithme Cliquer [10] partitionne G, en déterminant un ensemble indépendant à la fois. Tant qu'il y a des sommets qui peuvent être ajoutés dans l'ensemble indépendant en construction, l'un de ces sommets avec le plus grand degré est ajouté. Puis Cliquer construit une clique maximum incrémentalement en considérant les sommets dans l'ordre inverse de celui dans lequel les sommets sont ajoutés dans les ensembles indépendants, la borne supérieure des cliques maximums contenant un sommet donné est rapidement déterminée de cette façon.

Falhe [2] améliore l'algorithme de Carraghan et Pardalos [1], en utilisant l'heuristique constructive DSATUR pour colorer les sommets un par un, tout en partitionnant en parallèle le graphe en ensembles indépendants. Comme les sommets sont colorés ou insérés dans un ensemble indépendant dans l'ordre décroissant ou croissant selon leur degré, quatre solutions heuristiques de GCP sont obtenues, et la meilleure est utilisée comme la borne supérieure.

Regin [13] utilise une borne supérieure basée sur un algorithme de matching. Un matching, qui correspond à un ensemble d'ensembles indépendants de taille 2 ici, est calculé en parcourant les sommets d'un graphe, et en considérant qu'une arête existe si deux sommets ne sont pas connectés.

MCQ [15] colorie les sommets dans un ordre prédéterminé. Supposons que les ensembles indépendants actuels sont S_1, S_2, ..., S_k (dans cet ordre, k est égal à 0 au début du processus de coloration), MCQ insère le premier sommet v dans l'ordre prédéterminé dans le premier S_i tel que v est non-connecté à tous les sommets déjà dans S_i. Si une telle S_i n'existe pas, une nouvel ensemble indépendant S_{k+1}) est ouvert et v est inséré dedans. Après que tous les sommets sont insérés dans les ensembles indépendants, ils sont réordonnés en fonction de leur ensemble indépendant, les sommets de S_i précédant les sommets de S_j si $i < j$. Ce procédé de coloration est exécuté pour G_v après chaque branchement sur le sommet v. L'ordre prédéterminé de sommets G_v est hérité de G. MCR [14] améliore MCQ avec un meilleur ordre de sommets du graphe d'entrée initial, mais utilise le même processus de coloration pour calculer la borne supérieure.

MaxCliqueDyn [5] est également obtenu en améliorant MCQ. Alors que MCQ (ainsi que MCR) ne calcule que le degré des sommets à la racine de l'arbre de recherche pour le graphe d'entrée initial, MaxCliqueDyn recalcule dynamiquement le degré des sommets pour certains nœuds situés à proximité de la racine de l'arbre de recherche choisis en utilisant un paramètre, et réordonne les sommets dans l'ordre décroissant de leur degré avant la coloration de ces sommets. Le calcul dynamique des degrés près de la racine de l'arbre de recherche rend MaxCliqueDyn plus rapide que MCQ pour les graphes aléatoires lorsque leur densité est comprise entre 0,7 à 0,95, mais plus lent que MCQ lorsque la densité des graphes est inférieure à 0,7.

5 Nouveaux codages de Maxclique en MaxSAT

Definition 1 *Soit G un graphe partitionné en ensembles indépendants, le codage de Maxclique en MaxSAT basé sur les ensembles indépendants est défini comme suit : (1) chaque sommet v de G est représenté par une variable booléenne x, (2) une clause dure $\bar{x}_i \vee \bar{x}_j$ est ajoutée pour chaque paire de sommets non-connectés (v_i, v_j), et (3) une clause souple est ajoutée pour chaque ensemble indépendant qui est un ou logique des variables représentant les sommets dans l'ensemble indépendant.*

Il est facile de voir que le codage habituel Maxclique en MaxSAT présenté dans la section préliminaire est un cas particulier de la définition 1, dans lequel chaque ensemble indépendant ne contient qu'un seul sommet.

Par exemple, le graphe de la figure 1 peut être partitionné en trois ensembles indépendants $\{v_1, v_4, v_6\}$, $\{v_2, v_3\}$, $\{v_5\}$. Par conséquent, le codage de Maxclique en MaxSAT basé sur les ensembles indépendants contient trois clauses souples : $\{x_1 \vee x_4 \vee x_6, x_2 \vee x_3, x_5\}$, et les mêmes clauses dures comme dans le codage habituel : $\{\bar{x}_1 \vee \bar{x}_4, \bar{x}_1 \vee \bar{x}_5, \bar{x}_2 \vee \bar{x}_3, \bar{x}_2 \vee \bar{x}_5, \bar{x}_3 \vee \bar{x}_4, \bar{x}_1 \vee \bar{x}_6, \bar{x}_2 \vee \bar{x}_6, \bar{x}_4 \vee \bar{x}_6, \bar{x}_5 \vee \bar{x}_6\}$.

On observe que les clauses dures exigent que au plus un littéral puisse être satisfait dans une clause souple. Etant donné une affectation satisfaisant toutes les clauses dures, une clause souple satisfaite a exactement un littéral satisfait du fait des clauses dures. L'ensemble indépendant correspondant a un sommet dans la clique donnée par l'affectation. Donc, nous avons :

Proposition 2 *Soit ϕ une instance MaxSAT partielle codant une instance Maxclique en basant sur une partition du graphe G en ensembles indépendants, l'ensemble des variables booléennes de ϕ affectées à vrai dans une solution optimale de ϕ donne une clique maximum de G.*

Différentes partitions de G en ensembles indépendants donnent différentes instances MaxSAT. Le tableau 1 compare trois partitions des graphes aléatoires de 200 sommets et de certains graphes DIMACS. Le codage Max est basé sur une partition calculée en utilisant l'algorithme de coloration MCQ pour insérer des sommets un par un dans un ensemble indépendant dans l'ordre décroissant de leur degré (i.e., les sommets avec le degré maximum sont insérés en premier). Min est similaire à Max sauf que les sommets sont insérés dans l'ordre croissant de leur degré (i.e., les sommets avec le degré minimum sont insérés en premier). Le codage Habituel correspond à celui présenté dans la section préliminaire, dans lequel chaque clause souple est unitaire. Nous reportons, pour chaque codage, le nombre d'ensembles indépendants dans le graphe, et le temps nécessaire pour trouver une clique maximum par deux solveurs MaxSAT de l'état de l'art : MaxSatz [7] et Minimaxsat [4]. A chaque densité des graphes aléatoires, 50 graphes sont générés et codés en MaxSAT. Le runtime moyen et le nombre moyen d'ensembles indépendants k sont reportés, k étant aussi le nombre de clauses souples dans le codage. Pour les deux solveurs MaxSAT, le codage Max est le meilleur, le nombre de clauses souples étant le plus petit. Nous allons utiliser ce codage pour appliquer la technologie MaxSAT dans la section suivante.

Heras et Larrosa [3] proposent un prétraitement pour améliorer le codage habituel en utilisant deux règles d'inférence basées sur la résolution MaxSAT. Ce prétraitement permet d'améliorer la borne supérieure initiale basée sur le nombre de clauses souples initiales dans le codage habituel, mais ne semble pas améliorer le temps de Mini-MaxSAT pour résoudre les instances DIMACS. Toutefois, le prétraitement permet d'accélérer Minimaxsat pour certaines instances spécifiques avec solution optimale cachée

TABLE 1 – Durée [sec.] de Maxsatz (version 2009, MSZ dans le tableau) et Minimaxsat (Mini dans le tableau) sur un MacPro 2,8 GHz avec 4 Go de mémoire pour trois codages différents de Maxclique en MaxSAT, k est le nombre de clauses souples dans un codage

$Graph$		$Habituel$			Min			Max		
name	density	k	MSZ	Mini	k	MSZ	Mini	k	MSZ	Mini
200	0.40	200	2.11	2.25	30.6	1.31	0.80	27.3	1.10	**0.76**
200	0.60	200	36.8	18.3	45.4	11.2	7.68	40.8	**7.22**	7.27
200	0.80	200	9691	576.3	67.4	265.7	278.5	61.1	**117.8**	269.2
brock200_1	0.74	200	1344	156.7	60	103.6	100.4	56	**52.9**	67.5
brock200_2	0.50	200	6.72	4.36	36	2.58	2.09	33	2.28	**1.79**
brock200_3	0.61	200	45.3	13.7	45	9.44	7.95	43	7.48	**6.88**
keller4	0.65	171	40.5	34.0	34	6.16	8.81	32	**2.89**	7.39
p_hat300-1	0.24	300	3.09	2.67	33	2.51	1.12	22	1.81	**0.96**
p_hat300-2	0.49	300	105.3	7.54	69	69.7	3.64	46	3.06	**2.72**
p_hat300-3	0.74	300	>3h	452.1	95	2326	163.5	72	185.3	**129.2**

et certaines protein alignment instances.

Malgré que les deux nouveaux codages permettent un gain significatif pour MaxSatz et MiniMaxSAT par rapport au codage habituel, les solveurs MaxSAT ne sont pas encore en mesure de battre les solveurs Maxclique spécifiques sur les instances aléatoires et les instances DIMACS, comme nous allons voir dans la section 7. Nous pensons que la raison principale de cette inefficacité des solveurs MaxSAT est que ces solveurs n'exploitent pas la structure des graphes, contrairement aux algorithmes spécifiques. En revanche, nous allons montrer que l'association de l'exploitation de structure des graphes et la puissance de raisonnement propositionnel de la technologie MaxSAT permet grandement d'ameliorer les algorithmes spécifiques pour Maxclique.

6 Utilisation de la technologie MaxSAT pour améliorer la borne supérieure de Maxclique

Etant donnée une instance MaxSAT partiel, un solveur MaxSAT de séparation et évaluation doit trouver une affectation qui minimise le nombre de clauses souples insatisfaites et satisfait toutes les clauses dures. Dans ce but, le solveur doit sous-estimer, à chaque noeud de l'arbre de recherche, le nombre de clauses souples qui seront violées par une affectation complète étendant l'affectation partielle en cours. Une approche proposée dans [9] et qui s'est révélée très puissante dans MaxSatz et Minimaxsat consiste à détecter des sous-ensembles inconsistants de clauses souples disjoints. Un sous-ensemble de clauses souples est inconsistant si le sous-ensemble, en conjonction avec l'ensemble des clauses dures, permet de déduire une contradiction (une clause vide). Le nombre de sous-ensembles inconsistants de clauses souples disjoints est une minoration du nombre de clauses souples violées par une affectation complète étendant l'affectation partielle en cours et satisfaisant toutes les clauses dures, car il y a au moins une clause souple violée dans chaque sous-ensemble inconsistant de clauses souples.

Cette approche, appelée UP, détecte les sous-ensembles inconsistants de clauses disjoints avec la propagation des clauses unitaires comme suit :

Etant donnée une formule CNF ϕ, UP stocke toutes les clauses unitaires de ϕ dans une file Q, et applique la propagation des clauses unitaires dans une copie φ de ϕ, en successivement prenant une clause unitaire l de Q pour la satisfaire (i.e., l=1). La satisfaction de l signifie que toutes les clauses contenant l sont satisfaites et retirées de φ, et que $\bar{l}$ est enlevé des autres clauses, ce qui peut produire des nouvelles clauses unitaires et des clauses vides dans φ (une clause devient vide si elle ne contenait que $\bar{l}$). Les nouvelle clauses unitaires sont stockées à la fin de Q. La propagation des clauses unitaires continue jusqu'à ce que'il n'existe plus de clause unitaire dans Q ou jusqu'à ce qu'une clause vide soit produite. Dans ce dernier cas, les clauses utilisées dans la propagation pour produire la clause vide constituent un sous-ensemble inconsistant de clauses. Une fois qu'un sous-ensemble inconsistant des clauses est identifié, ces clauses sont retirées de ϕ. Ainsi UP continue de détecter d'autres sous-ensembles inconsistants de clauses tant qu'il reste des clauses unitaires dans ϕ. UP est amélioré dans [8] par la détection des littéraux d'échec. Un littéral l est d'échec si, quand il est satisfait et $\bar{l}$ retiré de toutes les clauses le contenant, la propagation des clauses unitaires produit une clause vide. Si les deux littéraux l et $\bar{l}$ sont d'échec, l'union des clauses utilisées pour produire les deux clauses vides constitue un sous-ensemble inconsistant.

Dans le cas de MaxSAT partiel, après l'obtention d'un sous-ensemble inconsistant des clauses en utilisant les approches ci-dessus, les clauses dures dans le sous-ensemble ne sont pas retirées de ϕ et peuvent être re-utilisées dans la détection d'autres contradictions, mais les clauses souples dans le sous-ensemble sont retirées de ϕ. En d'autres termes, les clauses dures peuvent appartenir à plusieurs sous-ensembles inconsistants de clauses, parce qu'elles doivent être satisfaites de toute façon. Par contre, une clause souple ne peut pas appartenir à deux sous-ensembles inconsistants de clauses différents.

Nous adaptons la détection des littéraux d'échec pour détecter des sous-ensembles inconsistants de clauses souples dans une instance MaxSAT partielle codant une instance MaxClique. Nous ne détectons pas si un littéral négatif est d'échec ou non, car une variable peut avoir plusieurs occurrences négatives, mais une seule occurrence positive (dans une clause souple). Au lieu de cela, nous détectons si chaque littéral dans une clause souple est d'échec. En fait, soit une clause souple $c = x_1 \vee x_2 \vee ... \vee x_t$, si tous les x_i $(1 \leq i \leq t)$ sont littéral d'échec, l'union de toutes les clauses souples utilisées pour produire une clause vide pour chaque x_i, avec c, constitue un sous-ensemble inconsistant.

Par exemple, dans l'instance MaxSAT codant le graphe de la Figure 1 basée sur les ensembles indépendants présentée dans la dernière section, il y a trois clauses souples $\{x_1 \vee x_4 \vee x_6,\ x_2 \vee x_3,\ x_5\}$ correspondant aux trois ensembles indépendants dans une partition optimale du graphe. Les clauses dures sont : $\{\bar{x}_1 \vee \bar{x}_4, \bar{x}_1 \vee \bar{x}_5, \bar{x}_2 \vee \bar{x}_3, \bar{x}_2 \vee \bar{x}_5, \bar{x}_3 \vee \bar{x}_4, \bar{x}_1 \vee \bar{x}_6, \bar{x}_2 \vee \bar{x}_6, \bar{x}_4 \vee \bar{x}_6, \bar{x}_5 \vee \bar{x}_6\}$. La borne supérieure initiale d'une clique maximum est 3, qui est la meilleure borne supérieure qui peut être obtenue en utilisant un procédé de coloration. Nous montrons que x_5 est un littéral d'échec, permettant d'améliorer la borne supérieure. Posons x_5=1 pour satisfaire la troisième clause souple, $\bar{x}_5$ est retiré des clauses dures $\bar{x}_1 \vee \bar{x}_5$, $\bar{x}_2 \vee \bar{x}_5$, et $\bar{x}_5 \vee \bar{x}_6$, les trois nouvelles clauses unitaires impliquent que x_1=0, x_2=0 et x_6=0, puis x_1 et x_6 soient retirés de la première clause souple, et x_2 de la deuxième, les deux clauses devenant unitaires. Donc, x_4 et x_3 doivent être affectés à 1, rendant la clause dure $\bar{x}_3 \vee \bar{x}_4$ vide. Donc, x_5 est un littéral d'échec et les trois clauses souples utilisées dans la propagation pour produire la clause vide constituent un sous-ensemble inconsistant, parce que ce sous-ensemble, en conjonction avec les clauses dures, permet de déduire une contradiction. Puisqu'au plus deux clauses souples peuvent être satisfaites par une affectation satisfaisant toutes les clauses dures, nous améliorons la borne supérieure pour une clique maximum de 3 à 2.

En général, nous avons :

Proposition 3 *Soit $\omega(G)$ la cardinalité d'une clique maximum d'un graphe G. Si G peut être partitionné en k ensembles indépendants, et il existe s sous-ensembles inconsistants de clauses souples disjoints dans l'instance MaxSAT codant G en basant sur la partition de G en k ensembles indépendants, alors $\omega(G) \leq k - s$.*

En se basant sur la proposition 3, la fonction surestimation(G) présentée dans l'algorithme 2 partitionne d'abord G en ensembles indépendants de la même manière que MCQ, sauf que le degré de chaque sommet de G est exact. Par conséquent, la qualité de la partition est sans doute meilleure (i.e., la partition contient vraisemblablement moins d'ensembles indépendants), puisque les sommets plus contraints (i.e. avec plus de voisins connectés) sont insérés en premier dans les ensembles indépendants. Ensuite la fonction code le graphe en une instance MaxSAT basée sur la partition, ce qui correspond au codage Max présenté dans la dernière section, et utilise la technologie MaxSAT efficace pour améliorer la borne supérieure donnée par la partition.

Avec l'utilisation de la fonction surestimation(G), la différence essentielle entre MaxCLQ et un solveur MaxSAT pour Maxclique se présente comme suit. Alors qu'un solveur MaxSAT utilise un codage MaxSAT fixe et ne repartitionne jamais de sous-graphe pendant la recherche, MaxCLQ re-partitionne de façon dynamique le sous-graphe à chaque noeud de l'arbre de recherche et code ce sous-graphe en MaxSAT pour améliorer la borne supérieure donnée par la partition, de sorte que la force des algorithmes spécifiques pour Maxclique dans le partitionnement de graphe et la puissance de la technologie MaxSAT dans le raisonnement propositionnel sont naturellement combinées dans MaxCLQ pour résoudre Maxclique.

Algorithm 2: surestimation(G), une surestimation de la cardinalité d'une clique maximum de G

Input: Un graphe G=(V, E)
Output: Borne supérieure pour une clique maximum de G

```
1  begin
2      P ← ∅;
3      G' ← G;
4      while G' est non vide do
5          v ← le sommet de dégrée maximum de G';
6          retirer v de G';
7          if il exist un ensemble indépendant S dans P
           où v est non-connecté à tous les sommets then
8              insérer v dans S ;
9          else
10             créer un nouvel ensemble indépendant S;
11             insérer v dans S;
12             P ← P ∪ {S} ;
13     Coder G en une instance MaxSAT φ en basant sur
       P;
14     s ← 0;
15     while φ contient une clause souple qui n'est pas
       testée do
16         c ← la clause souple de φ de taille minimum
           qui n'est pas testée;
17         marquer c comme "testée";
18         if chaque littéral dans c est d'échec then
19             retirer c et toutes les clauses souples
               faisant les littéraux de c d'échec de φ;
20             s ← s+1 ;
21     retourner |P| − s ;
22 end
```

7 Evaluation comparative de MaxCLQ

Nous comparons la performance de MaxCLQ avec celle de MaxCLQ$^-$ qui est identique à MaxCLQ sauf qu'il n'utilise pas la technologie MaxSAT pour améliorer la borne supérieure dans la fonction surestimation(G) (i.e. la fonction retourne toujours |P| dans MaxCLQ$^-$). Nous comparons aussi MaxCLQ avec les meilleurs algorithmes exacts pour Maxclique à notre connaissance : Cliquer, Algorithme Regin, MCR, MaxCliqueDyn (MCQdyn en bref). Le calcul de

la borne supérieure dans MaxCLQ$^-$ est le même que MCR et MCQDyn, à l'exception que MCR calcule seulement le degré de chaque sommet une fois dans la racine de l'arbre de recherche, et MCQdyn recalcule en plus le degré de chaque sommet dans certains noeuds proches de la racine choisis en utilisant un parametre, alors que MaxCLQ$^-$ recalcule le degré de chaque sommet dans tous les noeuds de l'arbre de recherche. En utilisant des informations exactes de degré de sommets, la fonction surestimation(G) donne à priori une borne supérieure de meilleure qualité dans MaxCLQ$^-$ que dans MCR et MCQdyn, mais le recalcul de degré de chaque sommet dans chaque noeud de l'arbre de recherche est consommateur du temps, surtout quand l'arbre de recherche est grand.

Les exécutions de MaxCLQ, MaxCLQ$^-$, Cliquer et MCQdyn sont réalisées sur un MacPro Xeon 2,8 GHz avec processeur Intel et 4 Go de mémoire (produit au début de 2008) avec MAC OS X 10.5. Nous utilisons la dernière version de Cliquer publiée en 2008 [2] et nous l'exécutons avec ses paramètres par défaut. MCQdyn a été obtenu de l'un de ses auteurs (D. Janezic) en janvier 2010 et exécuté avec le meilleur paramètre 0,025. Les temps d'exécutions de MCR et Regin sont normalisés à partir de ceux reportés comme suit.

Les temps d'exécution du programme benchmark dfmax pour les graphes DIMACS r100.5, r200.5, r300.5, r400.5 et r500.5 sur le MacPro sont respectivement de 0,002, 0,033, 0,270, 1,639, 6,281. Le temps d'exécution de dfmax reporté pour l'ordinateur (Pentium4 2,20 GHz CPU avec Linux) exécutant MCR est respectivement 0,00213, 0,0635, 0,562, 3,48, 13,3 pour les mêmes graphes. Ainsi, nous divisons les temps d'exécution de MCR par 2,12 (= (13.3/6.281 +3,48/1,639)/2, la moyenne des deux plus grands ratios). Les temps d'exécution de dfmax pour r100.5-r500.5 de l'ordinateur Regin (Pentium4 2Ghz mobile avec 512 Mo de mémoire) ne sont pas reportés, mais l'ordinateur Regin est environ 10% plus lent que l'ordinateur qui exécute MCR, donc nous divisons les temps d'exécution reportés de Regin par 2,33. Cette normalisation est basée sur la méthode établie dans le Second Challenge d'Implémentation DIMACS pour Cliques, Coloration, et Satisfiabilité, elle est également utilisée dans [14] pour comparer MCR avec d'autres algorithmes. Regin [13] normalise les temps d'exécution des algorithmes en comparant les différents ordinateurs. Parmi les nombreux algorithms comparés avec MCR dans [14] et avec Regin dans [13], nous citons l'algorithme Fahle [2] qui améliore l'algorithme proposé par Carraghan et Pardalos [1] et implémenté dans dfmax. MCR et Regin sont significativement plus rapide que l'algorithme de Fahle.

Le tableau 2 compare les temps d'exécution réels de MaxCLQ, MaxCLQ$^-$, Cliquer et MCQdyn avec les temps d'exécution normalisés de MCR pour les graphes aléatoires jusqu'à 500 sommets. MaxCLQ, MaxCLQ$^-$, Cliquer et MCQdyn résolvent 50 graphes à chaque point et le temps d'exécution moyen est reporté. Les graphes de faible densité (par exemple, des graphes de 150 sommets et de densité 0,7) qui sont résolus en moins de 1 seconde par l'ensemble des cinq algorithmes sont exclus pour économiser l'espace. MaxCLQ est sensiblement meilleur que MaxCLQ$^-$ et l'accélération augmente avec la densité lorsque le nombre de sommets est fixe, et avec le nombre de sommets lorsque la densité est fixe. MCR est plus rapide que MaxCLQ pour les graphes de relativement faible densité (densité $< 0{,}7$), car il n'a pas à recalculer le degré des sommets à chaque noeud de l'arbre de recherche sauf pour celui de la racine. Toutefois, de façon substantielle MaxCLQ est plus rapide que MCR et que d'autres algorithmes pour les graphes de densité $\geq 0{,}7$, et l'accélération croit aussi avec le nombre de sommets et la densité du graphe.

TABLE 2 – Temps d'exécution [s] pour les graphes aléatoires. Pour Cliquer, MCQdyn et MaxCLQ$^-$, "-"signifie que l'instance ne peut être résolue en moins de 3 heures ; pour les MCR, "-" signifie que le temp d'exécution n'est pas disponible. Les temps d'exécution de Regin pour les graphes aléatoires ne sont pas disponibles. s est l'amélioration moyenne de la borne supérieure par MaxSAT dans un noeud d'arbre, et Rate est le taux de réussite de la technologie MaxSAT dans MaxCLQ pour couper les sous-arbres (expliqué plus loin), en moyenne de 50 graphes à chaque point.

n	density	Cliquer	MCR	MCQdyn	MaxCLQ$^-$	MaxCLQ	s	Rate
150	0.80	3.49	0.36	0.32	0.64	**0.16**	2.85	0.85
150	0.90	433.3	3.59	1.74	2.69	**0.25**	4.15	0.87
150	0.95	1513	2.01	0.74	1.17	**0.05**	2.68	0.54
200	0.70	2.06	**0.44**	0.47	1.06	**0.44**	2.35	0.82
200	0.80	125.0	8.32	5.12	10.12	**2.27**	3.19	0.87
200	0.90	-	462.4	90.73	138.3	**9.98**	4.87	0.91
200	0.95	-	-	81.4	121.9	**2.40**	6.10	0.90
300	0.60	3.27	**0.87**	1.02	2.32	1.49	2.18	0.74
300	0.70	112.1	14.57	12.12	28.25	**10.29**	2.81	0.82
300	0.80	-	844.3	423.8	805.9	**158.3**	3.39	0.88
300	0.90	-	-	-	-	**6695**	5.52	0.92
500	0.50	7.08	**2.13**	2.87	6.87	6.25	2.17	0.60
500	0.60	176.7	**37.71**	38.84	107.9	54.27	2.70	0.74
500	0.70	-	1797	1496	3527	**1147**	2.95	0.83

Le tableau 3 compare les temps d'exécution réels de MaxCLQ, MaxCLQ$^-$, Cliquer et MCQdyn avec les temps d'exécution normalisés de MCR et Regin sur les benchmarks Maxclique DIMACS. Afin d'économiser de l'espace, nous excluons les instances très faciles qui sont résolues par les six algorithmes en moins de 2 secondes et les cinq instances ouvertes (MANN_a81, hamming10-4, johnson32-2-4, keller6, et p_hat1500-3) qu'aucun algorithme exact à nos connaissances n'est capable de résoudre. Sauf 8 instances faciles que MaxCLQ résout aussi rapidement, MaxCLQ est nettement plus rapide que tous les autres algorithmes, surtout pour les graphes durs et denses. En particulier, MaxCLQ est capable de fermer l'instance ouverte p_hat1000-3.

Dans un arbre de recherche de MaxCLQ, soit p le nombre de noeuds dans lesquels la partition donne déjà

2. disponible à http ://users.tkk.fi/pat/cliquer.html

TABLE 3 – Temps d'exécution [sec] pour DIMACS benchmarks. "d" correspond à la densité. Pour Cliquer, MCQdyn et MaxCLQ$^-$, "-" signifie que l'instance ne peut pas être résolue en moins de 24 heures, sauf dans le cas de keller5, p_hat1500-2 et p_hat1000-3 qui ne peut pas être résolu en moins de 5 jours ; pour Regin et MCR, "-" signifie que le temps d'exécution n'est pas disponible. s est l'amélioration moyenne de la borne supérieure par MaxSAT dans un noeud d'arbre, et Rate est le taux de réussite de la technologie MaxSAT dans MaxCLQ pour couper les sous-arbres (expliqué plus loin).

name	n	d	ω	Cliquer	Regin	MCR	MCQdyn	MaxCLQ$^-$	MaxCLQ	s	Rate
brock200_1	200	0.74	21	6.37	4.60	1.13	0.96	2.28	**0.67**	2.66	0.86
brock400_1	400	0.75	27	22182	4867	1137	703.5	1447	**370.84**	2.92	0.86
brock400_2	400	0.75	29	5617	3395	465.10	309.0	664.9	**178.70**	2.86	0.86
brock400_3	400	0.75	31	1667	1922	766.51	565.0	971.3	**290.06**	2.81	0.85
brock400_4	400	0.75	33	247.7	2597	409.43	320.4	605.6	**167.30**	3.15	0.85
brock800_1	800	0.65	23	-	-	10712	8821	22821	**8815**	2.92	0.80
brock800_2	800	0.65	24	-	-	9679	8125	21001	**7690**	2.77	0.81
brock800_3	800	0.65	25	26014	-	6546	5565	13559	**5285**	2.73	0.80
brock800_4	800	0.65	26	6108	-	4561	4240	9625	**3880**	2.71	0.80
MANN_a27	378	0.99	126	-	7.93	1.98	3.10	6.86	**0.66**	3.21	0.75
MANN_a45	1035	0.996	345	-	-	2931	2006	8965	**255.67**	6.30	0.91
hamming10-2	1024	0.99	512	0.19	0.45	**0.16**	2.26	69.54	7.92	0.34	0.07
keller5	776	0.75	27	-	-	-	31038	78505	**9687**	3.60	0.87
p_hat300-3	300	0.74	36	496.6	17.47	7.45	4.91	9.79	**2.07**	3.07	0.85
p_hat500-2	500	0.50	36	134.6	14.03	2.12	1.53	3.92	**0.90**	3.26	0.83
p_hat500-3	500	0.75	50	-	5470	1256	349.4	634.4	**55.95**	4.56	0.91
p_hat700-1	700	0.25	11	0.09	2.58	**0.07**	0.14	0.71	0.80	1.88	0.33
p_hat700-2	700	0.50	44	15417	109.8	30.19	12.6	25.80	**4.87**	3.79	0.84
p_hat700-3	700	0.75	62	-	-	-	6187	12178	**1033**	4.64	0.91
p_hat1000-1	1000	0.24	10	1.11	11.93	**0.35**	0.58	2.84	2.53	1.21	0.77
p_hat1000-2	1000	0.49	46	-	7230	1656	412.9	1038	**146.54**	4.21	0.89
p_hat1000-3	1000	0.74	68	-	-	-	-	-	**200760**	5.45	0.93
p_hat1500-1	1500	0.25	12	8.01	206.4	**2.97**	4.31	22.37	15.85	2.19	0.82
p_hat1500-2	1500	0.51	65	-	-	-	61461	105909	**8848**	4.95	0.92
san1000	1000	0.50	15	**0.08**	44.12	3.35	0.74	1.56	1.46	1.67	0.61
san200_0.9_2	200	0.90	60	13.36	1.124	2.92	0.79	1.14	**0.10**	3.39	0.58
san200_0.9_3	200	0.90	44	503.4	78.41	**0.11**	3.43	5.80	0.22	4.86	0.82
san400_0.7_1	400	0.70	40	-	9.99	1.09	0.52	0.67	**0.21**	3.95	0.78
san400_0.7_2	400	0.70	30	3081	28.98	0.21	0.20	**0.09**	**0.09**	1.10	0.19
san400_0.7_3	400	0.70	22	4.47	117.3	2.12	2.04	2.67	**0.75**	2.66	0.83
san400_0.9_1	400	0.90	100	-	729.6	2.5	30.95	56.99	**1.97**	6.58	0.86
sanr200_0.7	200	0.70	18	1.88	1.845	**0.37**	0.39	0.86	0.38	2.40	0.80
sanr200_0.9	200	0.90	42	46593	64.41	204.72	51.57	80.51	**5.72**	5.07	0.91
sanr400_0.5	400	0.50	13	0.97	7.35	**0.52**	0.74	2.13	1.73	2.36	0.57
sanr400_0.7	400	0.70	21	2852	1347	237.26	177.8	429.25	**141.48**	2.52	0.85

une borne supérieure plus petite ou égale à la borne inférieure pour couper les sous-arbres enracinés dans ces noeuds (i.e. $|P|+|C| \leq LB$), que q désigne le nombre de noeuds dans lesquels la technologie MaxSAT réduit avec succès la borne supérieure à la borne inférieure (i.e., $|P|+|C| > LB$, mais $|P|-s+|C| \leq LB$) pour couper les sous-arbres enracinés dans ces noeuds, et que r désigne le nombre de noeuds dans lequel la technologie MaxSAT n'a pas réussi à réduire la borne supérieure à la borne inférieure (i.e., $|P|-s+|C|>LB$). Notons que l'arbre de recherche a p+q feuilles. Dans le tableau 2 et le tableau 3, en plus de temps d'exécution, nous rapportons pour MaxCLQ l'amélioration moyenne s de la borne supérieure dans les (q+r) noeuds grâce à la technologie MaxSAT (nous excluons les p feuilles où la technologie MaxSAT n'a pas eu besoin d'etre appliquée), et le taux de réussite q/(q+r) de la technologie MaxSAT pour couper les sous-arbres. Le taux de réussite est très élevé, surtout pour les grands graphes et dense (jusqu'à 93%), expliquant la bonne performance de MaxCLQ.

8 Conclusion

Nous avons proposé un nouveau codage de Maxclique en MaxSAT fondé sur une partition d'un graphe en ensembles indépendants, chaque ensemble indépendant étant codé en une clause souple. Le nombre de clauses souples est nettement plus petit dans le nouveau codage que dans le codage habituel de Maxclique en MaxSAT. Le nouveau codage nous permet d'intégrer naturellement la technologie MaxSAT dans un algorithme de séparation et évalua-

tion pour améliorer la borne supérieure basée sur la partition. Les résultats expérimentaux montrent que de nombreux sous-arbres sont coupés de cette manière et l'algorithme résultant MaxCLQ est très efficace, surtout pour les instances difficiles, et il est capable de fermer une instance DIMACS ouverte.

Nous pensons que l'utilisation de la technologie MaxSAT pour améliorer la borne supérieure de Maxclique est une direction de recherche très prometteuse dans l'avenir. Nous envisageons d'intégrer d'autres technologies MaxSAT effectives dans MaxCLQ pour améliorer encore sa borne supérieure et résoudre les instances Maxclique difficiles telles que celles de l'BHOSLIB benchmark [3].

Références

[1] R. Carraghan, P. M. Pardalos, An exact algorithm for the maximum clique problem. *Operations Research Letters* 9(6) : 375-382 (1990)

[2] T. Fahle, Simple and fast : Improving a branch-and-bound algorithm for maximum clique. In *Proceedings of ESA-2002*, pp. 485-498, 2002.

[3] F. Heras, J. Larrosa, A Max-SAT Inference-Based Preprocessing for Max-Clique, in proceedings of SAT'2008, LNCS 4996, pp. 139-152, 2008.

[4] F. Heras, J. Larrosa, and A. Oliveras, MiniMaxSAT : An efficient weighted Max-SAT solver. *Journal of Artificial Intelligence Research*, 31 :1-32, 2008.

[5] J. Konc, D. Janezic, An improved branch and bound algorithm for the maximum clique problem, *Communications in Mathematical and in Computer Chemistry* 58 (2007) pp. 569-590.

[6] C. M. Li and F. Manyà, Max-sat, hard and soft constraints. In A. Biere, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*. Pages 613-631, IOS Press, 2009.

[7] C. M. Li, F. Manyà, and J. Planes, New inference rules for Max-SAT. *Journal of Artificial Intelligence Research*, 30 :321-359, 2007.

[8] C. M. Li, F. Manyà, and Jordi. Planes, Detecting disjoint inconsistent subformulas for computing lower bounds for maxsat, In *Proceedings of AAAI'06*, pages 86-91. AAAI Press, 2006.

[9] C. M. Li, F. Manyà and J. Planes, Exploiting unit propagation to compute lower bounds in branch and bound MaxSAT solvers, In *proceedings of CP'05*, LNCS 3709 Springer, 2005, pp 403-414.

[10] P. R. J. Ostergard, A fast algorithm for the maximum clique problem, *Discrete Applied Mathematics* 120 (2002), 197-207.

[11] P. M. Pardalos, J. Xue, The maximum clique problem. *Journal of Global Optimization* 4 : 301-328, 1994

[12] W. Pullan, H. H. Hoos, Dynamic Local Search for the Maximum Clique Problem. *Journal of Artificial Intelligence Research*, Vol. 25, pp. 159-185, 2006.

3. http ://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm

[13] J.-C. Regin, Solving the maximum clique problem with constraint programming. In *Proceedings of CPAIOR'03*, Springer, LNCS 2883, pp. 634-648, 2003.

[14] E. Tomita, T. Kameda, An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *J. Glob Optim* (2007) 37 :95-111.

[15] E. Tomita, T. Seki, An efficient branch-and-bound algorithm for finding a maximum clique. In *Proc. Discrete Mathematics and Theoretical Computer Science*. LNCS 2731, pp. 278-289 (2003).

Réparation locale de réfutation de formules SAT

Nicolas Prcovic
LSIS - Aix-Marseille Universités
nicolas.prcovic@lsis.org

Résumé

Nous traitons du problème de la démonstration qu'une formule booléenne est insatisfiable. Nous présentons un schéma de réparation locale de preuve d'insatisfiabilité par résolution. L'idée principale est de considérer les preuves linéaires dont le nombre de résolutions autorisées est bornée par une constante comme étant équivalentes à des problèmes de satisfaction de contraintes (CSP) dont chaque solution est une réfutation de la formule booléenne. Dès lors, toute technique de réparation locale applicable aux CSP peut être réutilisée a priori, une fois définis le voisinage d'une "réfutation à réparer" et sa proximité par rapport à une véritable réfutation. Nous proposons des critères pour évaluer les réfutations à réparer ainsi que des opérateurs de voisinage qui réaffectent une variable du CSP (changement de clause) ou permutent des valeurs de variables (déplacement de clause dans la preuve).

Abstract

We address the problem of proving that a boolean formula is unsatisfiable. We present a local repair scheme which proves unsatisfiability by resolution. The main idea is to consider the linear proofs which number of allowed resolutions is bounded by a constant as being equivalent to constraint satisfaction problems (CSP), which solutions are refutations of the boolean formula. Therefore, any local repair technique applicable to CSP can be reused, once defined the neighborhood of a "refutation to be repaired" and its proximity to a true refutation. We propose criteria to evaluate the proofs to be repaired and define neighborhood operators that re-assign a variable of CSP (change of a clause) or swap the values of variables (move of a clause inside the proof).

1 Introduction

Parmi les 10 défis posés à la communauté par Selman et al [8] en 1997, le cinquième, consistant à trouver un algorithme de recherche locale efficace pour démontrer l'insatisfiabilité d'une formule booléenne (ie, sa réfutation), est le seul à n'avoir pas donné lieu à des résultats satisfaisants, malgré quelques tentatives intéressantes [6, 1]. Dans cet article, nous partons du résultat de [3], où il est démontré que le problème de trouver la plus petite réfutation par résolution d'une formule booléenne est NP-difficile *quand la longueur de la preuve est une fonction polynomiale* du nombre de variables. Nous proposons une nouvelle approche consistant à transformer la question sous la forme d'un problème de satisfaction de contraintes (CSP) dont chaque solution est une réfutation de la formule booléenne qu'il représente. L'avantage de cette transformation est que l'on peut immédiatement appliquer n'importe quelle méthode de recherche de solution d'un CSP à notre problème, notamment une recherche locale.

Informellement, il s'agit de construire une preuve *linéaire* par résolution dont la longueur est bornée par une constante k. Une preuve est linéaire quand toute nouvelle résolvante est inférée à partir de la résolvante précédemment inférée et d'une autre clause [5]. Dans ce contexte, construire une réfutation, c'est choisir k clauses successives de telle manière que la dernière résolvante soit la clause vide. Il s'agit donc bien de reformuler un problème de co-NP en un problème NP-complet. Ceci est possible (et n'implique pas que co-NP = NP) car nous restreignons la question initiale à celle d'une *preuve par résolution qui soit polynomialement bornée* par le nombre de variables de la formule. Cette restriction n'est pas gênante dans la mesure où, d'une part, les réfutations sont en général basées sur la résolution, et d'autre part, il faut que la taille de la preuve soit présuposée polynomialement bornée pour qu'un algorithme de recherche de cette preuve puisse s'exécuter en un temps raisonnable.

Dans un premier temps, nous ferons des rappels sur les formules booléennes et les méthodes pour déterminer leur insatisfiabilité. Puis, nous formaliserons comment, à partir d'une formule booléenne, nous pouvons

définir le CSP dont chaque solution est une réfutation linéaire de cette formule. Ensuite, nous définirons ce qu'est le voisinage d'une dérivation et comment évaluer sa proximité à une réfutation. Enfin, nous comparerons notre approche à l'existant.

2 Notions préliminaires

Une formule booléenne, sous sa forme dite CNF, est un ensemble de clauses. Une clause est un ensemble de littéraux. Un littéral est soit une variable booléenne, soit sa négation. On définit Var(l) comme étant la variable du littéral l. Les variables peuvent être affectées à la valeur vrai ou faux. Une clause représente une disjonction de l'ensemble de ses littéraux. Une formule SAT représente une conjonction de ses clauses. Un modèle d'une formule ϕ est une affectation de variables qui est tel que ϕ est vraie. Une formule n'ayant pas de modèle est dite insatisfiable. La clause vide, notée $\square$, est toujours fausse et si une formule la contient alors elle est insatisfiable. Si une clause contient un littéral l et son opposé $\neg l$, elle est toujours vraie et on l'appelle *tautologie* sur Var(l).

Les systèmes formels de preuve propositionnelle permettent de dériver d'autres formules à partir d'une formule ϕ, grâce à des règles d'inférence. En particulier, le système de Robinson [7] permet de dériver des clauses induites à partir des clauses d'une formule ϕ grâce à la règle de *résolution* :

$$C_1, C_2 \vdash C_1 \backslash \{l\} \cup C_2 \backslash \{\bar{l}\}$$

On notera $C_1 \otimes C_2$ la clause inférée par résolution de C_1 avec C_2, qu'on appellera *résolvante* sur Var(l) de C_1 et C_2. L'intérêt d'un système basé sur la résolution est qu'il permet d'établir des réfutations de formules (ie, des preuves de leur insatisfiabilité) dans la mesure où une formule est insatisfiable si et seulement si il existe une suite de résolutions qui dérive la clause vide.

Une dérivation peut se représenter grâce à un arbre binaire dont les feuilles sont des clauses de ϕ, les nœuds internes sont des résolvantes et la racine est la clause dérivée (cf figure 1).

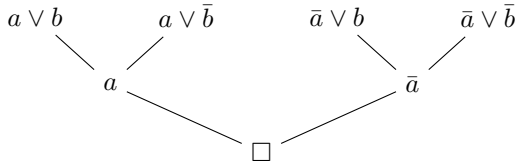


FIG. 1: Un arbre de resolution

Parmi les techniques de dérivation, certaines restreignent les possibilités d'appliquer la résolution tout en maintenant leur complétude. En particulier, les techniques de dérivation *linéaire* [5] obligent toute nouvelle résolvante à être inférée à partir de la précédente. Formellement, une dérivation linéaire d'une formule ϕ est une séquence de clauses (R_1, ..., R_n) telle que $R_1 \in \phi$ et chacun des autres R_i est la résolvante entre R_{i-1} (son parent proche) et une clause de ϕ (son parent d'entrée) ou une résolvante R_j ($j < i$) précédemment inférée (son ancêtre éloigné). R_1 est la clause de base. R_n est la clause dérivée. Si $R_n = \square$, nous avons une réfutation linéaire (cf figure 2). La SL-résolution[4] est un exemple d'algorithme permettant de rechercher systématiquement la réfutation linéaire d'une formule. Le principe de cet algorithme est d'effectuer une recherche arborescente en profondeur d'abord qui à chaque pas choisit une nouvelle clause à ajouter à la fin de la dérivation courante. La nouvelle clause doit permettre de produire une nouvelle résolvante non tautologique. L'algorithme s'arrête quand la clause vide est produite ou qu'il a essayé suffisamment de possibilités de construire une réfutation linéaire pour établir que la formule est non réfutable (et donc satisfiable).

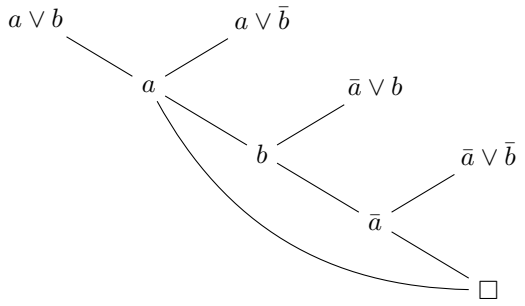


FIG. 2: Une résolution linéaire

Un problème de satisfaction de contraintes (CSP) est un triplet (X, D, C) où X=$\{x_1, ..., x_k\}$ est un ensemble de variables, D=$\{D_1, ..., D_k\}$ est l'ensemble des domaines dans lesquels ces variables peuvent prendre respectivement leur valeur, et C est un ensemble de contraintes qui expriment la compatibilité des affectations des variables. Une solution du CSP est une affectation de toutes les variables à une valeur de leur domaine telle que toutes les contraintes de C sont respectées.

3 Réfutation d'une formule SAT comme solution d'un CSP

Nous allons reformuler notre problème de réfutation d'une formule SAT sous la forme d'un CSP, uniquement dans le but de montrer concrètement comment

il peut se ramener à un problème NP-complet. Nous partons du fait que pour construire une dérivation linéaire de longueur k d'une formule $\phi = \{C_1, ..., C_m\}$, il faut choisir k clauses qui vont permettre d'effectuer la dérivation $\{C_{m+1}, ..., C_{m+k}\}$. Si on appelle x_i le choix du numéro de la i^e clause permettant d'inférer C_{m+i}, on a $x_i \in D_i = \{1, ..., m+i-1\}$. L'affectation de tous les x_i représente une réfutation linéaire si la contrainte $C_{x_1} \otimes C_{x_2} \otimes ... C_{x_k} = \square$ est respectée, avec $C_{m+1} = C_{x_1}$ et $\forall i > m+1$, $C_i = C_{i-1} \otimes C_{x_{i-m}}$. Ceci constitue un CSP dont la seule contrainte est globale et se vérifie en temps polynomial.

Notre objectif étant de définir une méthode de réparation de réfutation, nous pouvons en théorie imaginer commencer par affecter les variables x_i de façon à obtenir une dérivation d'une clause qui ne soit pas vide puis à modifier les affectations pour obtenir pas à pas une réfutation. Cependant, la difficulté est que nous ne pouvons pas choisir indépendamment les valeurs des variables car il faut qu'à chaque étape de la dérivation on puisse appliquer une résolution, qui nécessite qu'un littéral apparaisse positivement dans une clause et négativement dans l'autre. Or, la liberté de changer n'importe quelle affectation est un critère important pour concevoir un algorithme de recherche locale. C'est pourquoi nous choisissons de généraliser la règle d'inférence de telle manière qu'elle puisse s'appliquer à *tout* couple de clause. Pour ceci, nous intégrons une règle d'inférence qu'on retrouve classiquement dans les sytèmes logiques formels, la *règle d'affaiblissement* :

$$C_1,\, C_2 \vdash C_1 \cup C_2$$

Cette règle n'est pas intégrée dans les méthodes de réfutation basée sur la résolution car $C_1 \cup C_2$ est subsumée par C_1 et par C_2 et ne doit donc pas être utilisée à la place de C_1 ou C_2 pour dériver la clause vide. En particulier, si on peut effectuer une résolution entre C_1 et C_2 sur l, effectuer un affaiblissement de C_1 et C_2 produit une clause égale à la résolvante de C_1 et C_2 augmentée de $\{l,\, \bar{l}\}$. Nous introduisons cependant les affaiblissements car ils sont logiquement corrects et que nous pourrons les supprimer ultérieurement par réparation. Ceci nous conduit donc à remplacer la résolution par une règle d'inférence qui la généralise en prenant en compte la possibilité d'affaiblissement :

$$C_1,\, C_2 \vdash C_1 \backslash \{l\} \cup C_2 \backslash \{\bar{l}\} \text{ si } l \in C_1 \text{ et } \bar{l} \in C_2 \text{ et } C_1 \cup C_2 \text{ sinon}$$

En définissant l'opérateur binaire $\odot$ associatif à gauche qui est tel que $C = C_1 \odot C_2$ ssi $C_1,\, C_2 \vdash C$, la contrainte globale devient $C_{x_1} \odot C_{x_2} \odot ... C_{x_k} = \square$. Nous obtenons ainsi la propriété très utile que toute affectation de l'ensemble des variables x_i permet d'obtenir la dérivation (logiquement correcte) d'une clause.

Remarquons que, contrairement aux méthodes habituelles de réfutations basées sur la résolution, nous ne rejetons pas les tautologies inférées. Dans le cadre habituel, une tautologie $\{l, \bar{l}\} \cup C$ peut être écartée car une résolution sur l avec une clause $\{l\} \cup C'$ donne la résolvante $\{l\} \cup C \cup C'$ qui est subsumée par $\{l\} \cup C'$. Nous conservons les tautologies car elles sont logiquement correctes et que nous pourrons les supprimer ultérieurement par réparation.

A partir de maintenant, nous appelerons *clause inférée* le résultat d'une *inférence* qui consiste soit en une résolution, qui produit une résolvante, soit un affaiblissement, qui produit une *clause affaiblie*.

Réduction de la combinatoire

Nous avons un moyen de réduire la combinatoire du problème. Tel qu'il est défini jusqu'à présent, il s'agit de choisir k fois parmi au moins m et au plus $m+k$ valeurs. Or, k étant un polynôme $P(m)$, la combinatoire est en $\Theta(P(m)^{P(m)})$. Nous avons un moyen de réduire fortement cette complexité grâce à une restriction supplémentaire utilisée notamment dans la SL-résolution : les résolutions effectuées avec des ancêtres éloignés sont limitées à celles qui produisent une clause subsumant le parent proche (ie, tous les littéraux de l'ancêtre éloigné sont dans le parent proche sauf un littéral dont l'opposé est présent dans le parent proche, littéral qui permet la résolution). De plus, quand une résolution avec un ancêtre éloigné est possible, il faut l'effectuer immédiatement. Cette restriction conserve la complétude de la résolution linéaire et supprime totalement les choix relatifs aux résolutions avec ancêtres éloignés. La preuve de la complètude est donnée dans [4]. Nous pouvons donc modifier notre CSP ainsi : les variables x_i représentent les choix des *clauses d'entrée* effectués au cours de la dérivation. Entre deux choix, on applique itérativement toutes les résolutions subsumantes possibles avec ancêtre éloigné. Dans [4], il est donné une méthode (les chaînes de littéraux) pour le faire très efficacement. Remarquons qu'en conséquence, un CSP à k variables représente une dérivation de longueur supérieure ou égale à k : k inférences avec clauses d'entrée intercalées avec un certain nombre de résolutions avec un ancêtre éloigné. La combinatoire se réduit maintenant à $m^{P(m)}$.

Il est bien clair que la contrainte qui caractérise maintenant notre CSP est complexe et très particulière. Il n'y a donc pas véritablement d'intérêt pratique à reformuler notre problème sous la forme d'un CSP puis à le donner tel quel à un solveur de CSP. Cependant, par la suite, nous conserverons l'appellation CSP quand il est pratique pour la compréhension de notre méthode de considérer notre problème sous cette forme.

4 Schéma de réparation locale de réfutation

Maintenant que nous avons défini notre CSP, nous pouvons imaginer ré-utiliser directement n'importe quel algorithme de recherche locale connu dédié à la recherche rapide de solutions de CSP. Ce qui justifie l'utilisation d'une méthode de réparation de réfutation linéaire (que nous abrégerons dorénavant par RRL) est le présupposé qu'une modification locale a un impact global limité. Or, si nous considérons les arbres de résolution, nous constatons qu'ils induisent un ordre partiel sur les résolutions à effectuer. En l'occurrence, dans une réfutation linéaire qui se termine par la résolution entre l et $\bar{l}$, toutes les résolutions dérivant l peuvent être effectuées avant ou après celles dérivant $\bar{l}$. La dérivation de l n'influe pas sur celle de $\bar{l}$ et la réparation de l'une n'impacte l'autre que partiellement. Une modification locale de dérivation peut donc ne modifier que partiellement la clause dérivée.

Voici le schéma très général qu'ont les algorithmes de réparation si on les applique à notre contexte. Une fois générée une dérivation initiale D, on cherche itérativement un voisin D' de D qui améliore la qualité de D et on remplace D par D'. On s'arrête si la clause vide est dans D ou après un certain nombre d'itérations. Si D n'a pas de voisin qui l'améliore, on utilise une technique d'échappement de ce miminum local. Nous avons donc en premier lieu à définir ce qui fait la qualité d'une dérivation (ie, sa proximité à une réfutation) et ce qu'est le voisinage d'une dérivation.

4.1 Evaluation de la qualité d'une dérivation

Voici trois classes de critères permettant d'évaluer la proximité d'une dérivation par rapport à une réfutation :

- les critères liés à la taille des clauses produites :
  - La taille de la plus petite clause produite doit être minimisée. Nous devons considérer toutes les clauses inférées et non la dernière car il est possible que la clause vide apparaisse en cours de dérivation. Si deux dérivations ont leur clause la plus courte de taille égale, on favorisera la dérivation qui contient le plus de clauses de cette taille.
  - La largeur de la dérivation doit être minimisée. La largeur d'une dérivation est égale à la taille de sa plus grande clause. Dans [2], il est démontré que les réfutations courtes ont nécessairement une largeur faible. Si deux dérivations ont même largeur, on favorisera la dérivation qui contient le moins de clauses de cette largeur.
  - la taille moyenne des clauses doit être la plus faible possible. Ce critère prend en compte la taille des clauses qui ne sont pas extrêmes.
- les critères liés à la nature des clauses inférées :
  - le nombre de tautologies doit être minimisé car toute inférence entre une tautologie et une clause C produit une clause subsumée par C.
  - le nombre de clauses affaiblies doit être minimisé car elles sont subsumées par leur clause parente.
  - le nombre de clauses produites subsumées par des clauses d'entrée ou des clauses précédemment inférées doit être minimisées. Ce critère généralise le précédent car une clause peut être subsumée sans être le résultat d'un affaiblissement. Cependant le surcoût de vérification de subsumption doit être mis en balance avec les gains d'efficacité espérés grâce à une meilleure évaluation de la dérivation, ceci afin de déterminer s'il faut l'inclure en pratique.
- les critères liés à l'occurrence des littéraux :
  - Le nombre de littéraux "purs" (relativement aux clauses utilisées dans la dérivation) doit être minimisé. En effet, si un littéral apparaît dans une dérivation mais jamais son opposé, il sera toujours présent dans toutes les clauses inférées après son introduction.
  - De façon plus générale, parmi l'ensemble des clauses qui sont introduites *après* une clause inférée, il faut qu'il apparaisse au moins une fois l'opposé de chacun des littéraux de la clause inférée.

Parmi ces critères, les plus importants sont celui de la taille de la plus petite clause et celui du nombre de littéraux purs. En effet, une réfutation contient nécessairement la clause vide (de taille 0) et un nombre nul de littéraux purs. Par contre, elle peut éventuellement contenir des clauses longues, des affaiblissements et des tautologies. Notre fonction d'évaluation ordonne donc les dérivations en fonction des critères suivants, dans cet ordre (en cas d'égalité de deux dérivations sur un critère, on examine le suivant) : minimisation de la plus petite clause, maximisation du nombre de plus petites clauses, minimisation du nombre de littéraux purs, minimisation de la largeur de la dérivation, maximisation du nombre de clauses de cette largeur, minimisation du nombre de tautologies, minimisation du nombre d'affaiblissements. Cet ordre est partiellement arbitraire et d'autres combinaisons mériteraient d'être étudiées.

4.2 Voisinage d'une dérivation

Habituellement, on définit le voisin d'une affectation des variables comme étant une affectation qui diffère

d'une seule valeur. L'avantage est qu'il existe une séquence de voisins qui permet de relier toute affectation à toute autre, ce qui garantit à l'algorithme de recherche locale de pouvoir explorer tout l'espace de recherche. Cependant, dans notre cas, nous pouvons aussi considérer en plus un autre type de modification qui consiste à déplacer une clause (ou une suite de clauses) à un autre endroit de la dérivation, ce qui consiste à permuter les valeurs de variables du CSP. La mise en oeuvre de cette seconde possibilité se justifie dans la mesure où déplacer une clause peut moins perturber la dérivation que la remplacer par une autre clause.

Dit autrement, une dérivation résulte donc d'un *arrangement* de clauses prises parmi les clauses d'entrée du problème et nous pouvons nous intéresser d'abord à trouver une *combinaison* de clauses d'entrées puis à les ordonner.

Si k est la taille de la dérivation et m est le nombre de clauses de la formule, le nombre de façons de changer une clause de la dérivation est $k.m$ et le nombre de façons de déplacer une clause dans la dérivation est de l'ordre de k^2 (chacune des k clauses peut s'insérer dans un des $k-1$ autres emplacements).

4.2.1 Remplacement de clause

Nous nous intéressons dans un premier temps aux conditions de présence d'une clause dans la dérivation sans pour l'instant nous soucier de sa place. La condition la plus évidente est que, pour chacun de ses littéraux, il doit exister une autre clause de la dérivation qui contienne son littéral opposé (si on veut obtenir une réfutation). Une heuristique de choix de clause à retirer consiste donc à trouver une clause dont un littéral est "pur" en évitant autant que possible que la disparition des autres littéraux fasse que d'autres littéraux de la combinaison de clauses courante ne deviennent purs. De façon complémentaire, une heuristique d'introduction d'une nouvelle clause doit si possible introduire des littéraux opposés aux littéraux actuellement purs de la combinaison de clauses courante et éviter d'introduire des littéraux purs.

Il y a très certainement d'autres heuristiques de remplacement de clauses à découvrir.

4.2.2 Déplacement de clause

Voici les effets que peut avoir le déplacement d'une clause :

- Supprimer un affaiblissement : reculer une clause affaiblissante C à un endroit de la dérivation postérieur à l'introduction d'une clause dont un littéral est l'opposé d'un de ceux de C.
- Supprimer une tautologie : si l_2 et $\bar{l_2}$ sont dans une résolvante sur l_1, faire remonter la clause d'entrée avant l'introduction du littéral sur l_2 et après l'introduction du littéral sur l_1.
- Diminuer la taille d'une suite de clauses inférées : si aucun des nouveaux littéraux introduits par une clause d'entrée à l'étape i n'est effacé (par résolution) avant l'étape j ($j > i$), déplacer la clause d'entrée à l'étape $j-1$ permet de supprimer la présence des nouveaux littéraux entre les étapes i et j.

Il faut faire cependant attention que la disparition d'un affaiblissement ou d'une tautologie ou la diminution de la taille d'une suite de clauses peut provoquer l'apparition d'un autre affaiblissement ou d'une autre tautologie. Voici des cas de modifications qui apportent un gain clair grâce à un déplacement d'une clause. On appelle C_i la clause d'entrée introduite dans la dérivation D à l'étape i et R_{i+1} la clause inférée à partir de C_i et R_i.

- Cas où C_i est affaiblissante (cf figure 3). Soit C_j la première clause d'entrée introduite après C_i où un des littéraux $\bar{l_1}$ de C_j est opposé à un littéral l_1 de C_i. C_j produit une résolvante car $l_1 \in R_j$. Si R_{j+1} est une tautologie sur l_1 (car la résolution a été faite sur une autre variable) alors déplacer C_i à l'étape j (donc juste après C_j) fait disparaître la tautologie car l_1 n'est plus dans R_j. Il est nécessaire que R_{j+1} soit une tautologie sinon le déplacement de C_i derrière C_j fait que R_{j+1} devient un affaiblissement et nous n'aurions alors fait que déplacer l'endroit où se trouve l'affaiblissement.

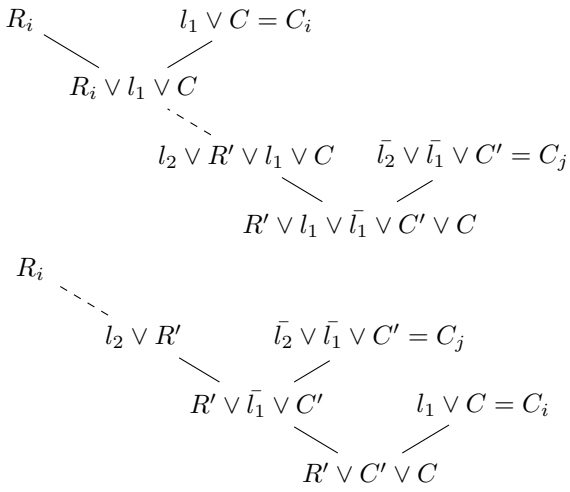


FIG. 3: Déplacement provoquant la suppression d'un affaiblissement et d'une tautologie : C_i passe juste après C_j.

– Cas où C_i permet une résolution sur $l_1, l_1 \in C_i$ (cf figure 4). Soit C_j la première clause d'entrée introduite après C_i où un des littéraux $\bar{l_2}$ de C_j est opposé à un littéral l_2 de C_i. Si R_{j+1} ne contient pas le littéral l_1, alors placer C_i à l'étape $j-1$ (juste avant C_j) permet de supprimer $C_i \backslash \{l_1\}$ de tous les R_h, $i < h < j$, c'est-à-dire de diminuer la taille de ces clauses inférées. Si R_{j+1} contient le littéral l_1, alors il faut remonter avant l'étape j jusqu'à ce que la clause inférée ne contienne plus l_1. En effet, si elle contenait l_1 alors l'absence de C_i avant elle ferait qu'elle deviendrait une tautologie sur l_1.

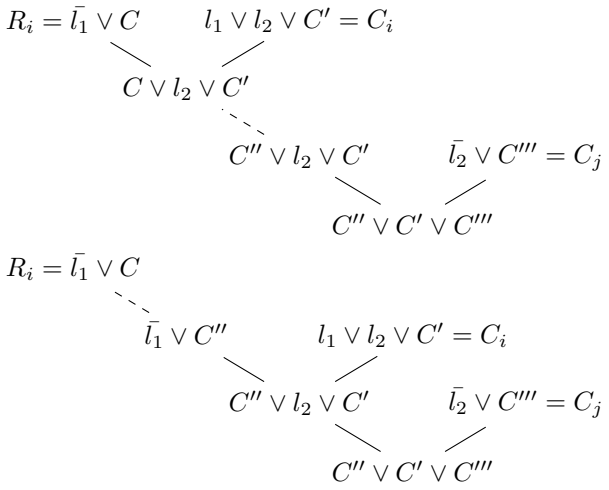


FIG. 4: Déplacement provoquant le raccourcissement d'une séquence de clauses inférées : C_i passe juste avant C_j.

Ce sont les deux cas de déplacement à effet purement positif que nous avons découvert pour l'instant mais nous pensons qu'il en reste à découvrir. Notamment, il faudrait aussi examiner les possibilités de déplacer des blocs de clauses consécutives.

5 Algorithme de réparation de réfutation linéaire

Nous avons d'abord défini l'algorithme le plus simple possible (cf figure 5). Nous partons d'une dérivation initiale D (ligne 1) et nous cherchons une meilleure dérivation `max-iterations` fois ou tant que nous n'avons pas dérivé la clause vide. Nous remplaçons une clause par une autre (ligne 4 et 5) et nous cherchons un meilleur ordre possible de clauses présentes dans la dérivation. Si cette dérivation est mieux évaluée, elle remplace la précédente (ligne 7). Une fois sorti de la boucle, nous retournons la dérivation produite.

Nous allons détailler les étapes de la méthode en précisant les options que nous avons écartées et celles que nous avons finalement retenues pour l'instant.

```
RRL(F, max-iterations)
(1) D = derivation-initiale(F)
(2) tantque clause vide not in D
    et nb-iterations < max-iterations
(3)     e = eval(D)
(4)     i = numero-clause-a-retirer(D)
(5)     c = clause-a-ajouter(F)
(6)     D' = reordonne(D \ {c_i} U {c})
(7)     si eval(D') > e alors D = D'
(8)     incrementer nb-iterations
(9) retourne D
```

FIG. 5: Algorithme de réparation de réfutation linéaire

La taille de la dérivation

Après quelques expérimentations, nous avons fixé la taille k de la dérivation à la valeur $2.m$. Cela signifie que nous nous attendons à ce que chaque clause puisse apparaître en moyenne deux fois par réfutation. Si nous voulons faire un algorithme qui ne dépend pas de k, il faudrait simplement le relancer en augmentant la valeur de k à chaque fois. Comme k doit rester un polynôme de m, il faudrait ajouter un multiple ou un diviseur de m à k à chaque relance.

Dérivation initiale

Nous pouvons affecter aléatoirement les clauses d'entrée de la dérivation initiale. Mais en pratique, nous avons constaté qu'une telle dérivation contenait essentiellement des affaiblissements qui étaient aussi souvent des tautologies. Nous avons donc aussi essayé de générer une dérivation initiale de la meilleure qualité possible grâce à un algorithme peu coûteux. Nous avons choisi d'utiliser un algorithme glouton qui construit itérativement une dérivation, à partir d'une dérivation vide, en rajoutant à chaque fois une clause d'entrée à la fin de la dérivation précédente, comme le ferait la SL-résolution mais sans backtracker. Concrètement, à l'itération i, nous inférons une nouvelle clause R_{i+1} à partir de la dernière clause inférée R_i et d'une clause d'entrée C_i que nous choisissons ainsi : il faut que R_{i+1} soit une résolvante non tautologique la plus courte possible et qu'elle ne soit pas subsumée par une résolvante précédente. Si toute résolvante non subsumée est tautologique, on accepte la plus courte. Si aucune résolution n'est possible, on choisit l'affaiblissement le plus court. En pratique, nous avons

constaté que les dérivations produites ainsi ne contenaient presque que des résolutions non tautologiques.

Remplacement des clauses

Pour l'instant, pour des raisons de simplicité, la clause à remplacer est choisie au hasard ainsi que la clause la remplaçant. Ceci garantit une bonne diversification dans le choix des clauses mais ne permet pas de trouver rapidement les bonnes clauses[1].

Déplacement des clauses

Le réordonnancement des clauses se fait en testant toutes les façons $k.(k-1)$ de déplacer une clause. Dès que le déplacement d'une clause produit une meilleure évaluation de la dérivation, le déplacement est effectué et on recommence. Lorsqu'il n'y a plus moyen de déplacer une clause en améliorant l'évaluation de la dérivation, on a terminé le réordonnancement. Cette manière de procéder permet d'améliorer la qualité des dérivations mais elle s'avère très coûteuse par rapport à ce qu'elle apporte. Nous avons donc restreint les choix de clauses à déplacer à celles responsables d'affaiblissement et de tautologie. Même avec cette restriction, le procédé reste coûteux. Faute de temps, nous n'avons pas encore pu expérimenter un réordonnancement qui n'effectuerait que les deux types de modifications présentées en section 4.2.2. Nous verrons s'ils permettent une amélioration de la dérivation proche de celle que nous obtenons déjà tout en réduisant fortement le temps de réordonnancement. Mais, par ailleurs, il faudra aussi envisager la possibilité de déplacer un bloc consécutif de clauses plutôt qu'une seule à la fois.

La RRL comme pré-processus

Même si l'objectif de la RRL est d'obtenir une réfutation, elle peut s'arrêter avant d'avoir pu dériver la clause vide. Dans ce cas, son utilisation peut quand même être utile si elle a produit des clauses courtes, unaires ou binaires. En effectuant une propagation unitaire, nous réduisons la taille de la formule initiale. En ajoutant toutes les clauses binaires à la formule initiale, nous favorisons le filtrage des clauses pour la méthode de résolution qui prendrait le relais.

C'est pourquoi, dans la dernière version de notre algorithme, nous mémorisons toutes les clauses binaires et unaires produites qui n'étaient pas dans la formule. Lorsque la RRL est terminée, si elle a échoué à trouver une réfutation mais qu'elle a produit de nouvelles clauses courtes, nous simplifions la formule en effectuant une propagation unitaire et en supprimant les clauses à littéral pur ou subsumée par une clause courte, puis nous relançons la RRL. De cette façon, nous sommes parvenus à réfuter plus de formules qu'auparavant.

[1]Nous n'avons pas encore eu le temps de tester une heuristique de choix de clause basée sur les principes donnés en section 4.2.1.

6 Travaux connexes et comparaisons expérimentales

Nous ne connaissons que deux autres tentatives de faire de la recherche locale pour démontrer l'insatisfiabilité : GUNSAT [1] et RANGER [6]. Ces deux algorithmes, bien qu'ayant des raffinements différents, sont basés sur le même principe itératif : parmi un ensemble de clauses d'entrées et de résolvantes précédemment produites, choisir au mieux deux clauses qui produiront une nouvelle résolvante (la clause la plus courte dans RANGER, un critère plus complexe dans GUNSAT). Aucun de ces deux algorithmes n'exhibe une réfutation et se contente d'essayer d'obtenir la clause vide. En imaginant qu'on associe à chaque résolvante deux pointeurs sur les clauses qui l'ont produite, on pourrait considérer que ces méthodes maintiennent un ensemble de sous-arbres (ie, de morceaux de preuves) dont certains sont suceptibles d'appartenir à la réfutation de la formule. Nous voyons bien alors ce qui différencie ces méthodes de la nôtre. Alors que nous restons focalisés sur une seule dérivation que nous nous autorisons à modifier à n'importe quel endroit, GUNSAT et RANGER essaient de fusionner des sous-arbres de preuves parmi un certain nombre de candidats. Nous avons donc des approches très différentes dont il n'est pas aisé de déterminer a priori celle qui pourrait être la plus efficace.

En fait, notre méthode doit surtout être comparée à la SL-résolution dans la mesure où la RRL est à la SL-résolution ce que sont les méthodes locales de recherche de modèles à DPLL : le sacrifice de la systématicité due à la liberté de modification d'une solution imparfaite en échange de la possiblité d'aller plus directement vers une solution. La SL-résolution ne revenant que sur le dernier choix de clause (qu'elle a ajouté à la fin de la dérivation), un mauvais choix de clause en début de dérivation peut mener à explorer inutilement un grand espace de recherche.

Premières expérimentations

Nous avons d'abord voulu vérifié l'efficacité relative de LRR par rapport à sa version systématique, la SL-résolution, et un autre algorithme classique basé sur la résolution, DP60. Nous avons pu constater que la SL-résolution ne parvenait à résoudre que de petits problèmes. Par exemple, aucun quand il y a 16 va-

riables. Par rapport à DP60, LRR devient meilleure lorsque le nombre de variables augmente.

#var	SL-res	DP60	LRR
10	3.9	0.18	0.57
16	-	0.68	1.08
20	-	2.26	1.67
24	-	13.6	10.0

FIG. 6: Comparaison entre la SL-resolution, DP60 et LRR sur 100 problèmes aléatoires insatisfiables au pic de difficulté (#clauses/#vars = 4.25).

Nous avons aussi voulu comparer LRR avec GUNSAT. GUNSAT est très souvent bien meilleure (typiquement 10 fois plus rapide) que LRR sauf sur quelques instances dans lesquelles elle reste coincée extrêmement longtemps (il faut l'arrêter avant qu'elle ait terminé) et qui augmente quelque peu artificiellement la moyenne des temps.

7 Conclusion et perspectives

Nous avons introduit une nouvelle méthode de preuve d'insatisfiabilité d'une formule SAT par une recherche locale. Elle imite parfaitement les méthodes locales destinées à trouver des solutions dans la mesure où nous avons fait en sorte de reformuler efficacement une restriction raisonnable de la question de l'insatisfaibilité d'une formule SAT en une question sur la consistance d'un CSP. Comme nous sommes face à un CSP très particulier mais qui permet de résoudre un problème très général, il est utile de raffiner et d'adapter autant que possible les méthodes de réparation locales traditionnelles. Ainsi, nous avons proposé une définition du voisinage d'une dérivation qui diffère de celle traditionnellement utilisée (consistant à modifier la valeur d'une seule variable) dans la mesure où nous incluons la possibilité de permuter les valeurs de variables (ie, les places des clauses dans la dérivation). De même, nous avons proposé des critères d'évaluation de proximité d'une affectation à une solution qui prenaient en compte le contexte (ie, la réfutation). Comme notre approche diffère grandement des techniques habituelles, nous savons qu'il faudra du temps pour étudier et perfectionner ses différents aspects afin de la rendre compétitive face aux méthodes traditionnelles de type DPLL qui bénéficient de plusieurs dizaines d'années d'efforts d'amélioration. Nous espérons néanmoins que le potentiel que nous lui voyons attirera l'attention d'autres membres de notre communauté, qui souhaiteront explorer les nombreux moyens de la rendre plus efficace, comme nous-même allons continuer à le faire.

Remerciements

Ce travail a été soutenu par un programme blanc de l'ANR (projet UNLOC).

Merci à Richard Ostrowski de m'avoir suggéré l'utilisation de la règle d'affaiblissement.

Références

[1] Gilles Audemard and Laurent Simon. Gunsat : A greedy local search algorithm for unsatisfiability. In *IJCAI*, pages 2256–2261, 2007.

[2] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow - resolution made simple. *J. ACM*, 48(2) :149–169, 2001.

[3] Kazuo Iwama. Complexity of finding short resolution proofs. In *MFCS*, pages 309–318, 1997.

[4] Robert A. Kowalski and Donald Kuehner. Linear resolution with selection function. *Artif. Intell.*, 2(3/4) :227–260, 1971.

[5] Donald W. Loveland. A linear format for resolution. In *Symposium on Automatic Demonstration*, pages 143–163, 1970.

[6] Steven David Prestwich and Inês Lynce. Local search for unsatisfiability. In *SAT*, pages 283–296, 2006.

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

[8] Bart Selman, Henry A. Kautz, and David A. McAllester. Ten challenges in propositional reasoning and search. In *IJCAI (1)*, pages 50–54, 1997.

Détection des cas de débordement flottant avec une recherche locale

Mohamed Sayah, Yahia Lebbah
Laboratoire I3S/CNRS, Université de Nice Sophia Antipolis, France
Laboratoire LITIO, Université d'Oran Es sénia, Algérie
{sayahmh, ylebbah}@gmail.com

Résumé

Dans ce papier, nous proposons une approche de génération automatique des cas de test de programmes de calcul numérique, qui donnent lieu à des débordements flottants. Afin de résoudre les contraintes sur les flottants modélisant cette problématique, nous proposons un algorithme de recherche locale mis en œuvre avec la bibliothèque multi-précision MPFR. Notre démarche de résolution procède en deux étapes : exploitation d'une recherche locale numérique pour avoir une solution approchée sur le domaine réel, puis un algorithme spécifique de recherche locale démarrant à partir de la solution de la première recherche locale, pour déterminer la solution exacte qui mène à un débordement flottant. Nous exploitons la bibliothèque multi-précision MPFR, tout au long de notre résolution en deux étapes, afin de sécuriser le solveur contre les débordements lors de la recherche numérique de la solution flottante.

1 Introduction

Les algorithmes numériques sont conçus et prouvés sur les nombres réels $\mathbb{R}$, alors que leur mise en œuvre est faite sur les nombres flottants $\mathbb{F}$ qui ne respectent pas les propriétés mathématiques des nombres réels. En pratique, l'ordinateur calcule avec l'arithmétique à virgule flottante une approximation de l'arithmétique réelle. Cette approximation se traduit par des erreurs d'arrondi ou parfois des erreurs soudaines de débordement au niveau des calculs.

Considérons l'algorithme `square` ci-dessous qui calcule la racine carrée r de deux nombres réels x et y. Dans le cas où x est égal à y, r prend la valeur de x.

1: **if** $(x = y)$ **then**
2: $\quad r \leftarrow x$;
3: **else**
4: $\quad r \leftarrow sqrt(x * y)$;
5: **end if**
6: **return** r ;

Le programme `square` avec le cas de test $x = 1.5e + 180$ et $y = 5.03e + 129$ provoque l'exception $Overflow$; le cas de test $x = 1e{-}200$ et $y = 5.03e{-}129$ provoque l'exception $Underflow$. C'est ce que nous traitons dans cet article, les cas de test relatifs au débordement flottant dans les programmes informatiques. Nous notons que parmi les méthodes de résolution, seules les méthodes globales ont été étudiées dans [5, 1]. Cependant, et vu que l'espace des flottants a une taille de nature fortement exponentielle (Par exemple, le nombre de flottants entre 0 et 1 dépend de la taille de ces flottants, il existe approximativement 2147483648 flottants.), ces méthodes se trouvent confrontées à la combinatoire qui rend impossible l'exploration de l'espace $\mathbb{F}$ des flottants. Ainsi, le recours aux techniques locales de résolution s'avère incontournable.

Notre approche adopte l'hypothèse suivante : "*les solutions d'un système de contraintes donné sur les flottants sont proches des solutions du même système sur les réels.*". D'abord, les systèmes de contraintes sur les flottants provenant d'un calcul numérique sont en pratique sur les réels. Le recours aux variables à virgule flottante est dû à l'indisponibilité des nombres réels sur machine. Puis, la solution sur les flottants est une approximation (qui peut être large dans certains cas) de la solution sur les réels.

Notre hypothèse nous motive à faire appel initialement à une première recherche locale sur les réels. Cette première recherche locale va nous fournir une première solution approchée. Ici, comme les algorithmes de recherche locale dans le domaine continu ne peuvent pas trouver des cas (solutions) de débor-

dement sans faire déborder le solveur lui même, nous recourons à une mise en œuvre avec la bibliothèque MPFR [3].

Justement, pour remédier à l'inexactitude sur les flottants de cette première solution, nous faisons appel à une deuxième recherche locale qui démarre à partir de la solution approchée fournie par la première recherche locale.

2 Description de l'approche

Dans ce contexte de génération automatique des cas de test qui donnent lieu à des débordements flottants, l'approche procède par les étapes suivantes :

1. Le programme de calcul numérique est transformé en un système de contraintes sur les flottants en exploitant la forme SSA [4].
2. Le critère exprimant le débordement est écrit sous forme de contraintes tout en exploitant les valeurs extrémales des flottants (i.e. $max(\mathbb{F})$ et $min(\mathbb{F})$). A la fin de cette étape, on obtient le problème global de satisfaction de contraintes $\mathcal{CSP}_{\mathbb{F}}$ dont les solutions sont nécessairement les cas de débordement du programme initial.
3. Une première solution approchée du $\mathcal{CSP}_{\mathbb{F}}$ est produite avec une recherche locale classique, à savoir la méthode du gradient en supposant que les domaines sont réels.
4. Une deuxième résolution à base d'une recherche locale dédiée pour les nombres flottants, permet de déterminer une solution exacte du système de contraintes et donc du débordement.

   Le principe de cette recherche locale est classique. Elle consiste à démarrer d'une instanciation initiale I_0, fournie par la première recherche locale sur les réels, et d'essayer ensuite de l'améliorer, en cherchant une meilleure instanciation, relativement à une fonction coût à minimiser, dans le voisinage de celle ci. Initialement l'instanciation courante I_c est égale à I_0. L'algorithme s'arrête une fois qu'il a atteint une instanciation qui est une solution exacte du système de contraintes sur les flottants. Comme toute recherche locale, elle s'arrête aussi si elle dépasse un nombre maximum d'itérations donné par l'utilisateur.

   L'espace de voisinage d'une instanciation I_c est l'ensemble de toutes les instanciations distantes d'un seul flottant de I_c. La fonction de coût, à minimiser, d'une instanciation est la somme des violations des différentes contraintes du problème. Les contraintes traitées sont les contraintes d'égalité de la forme $(e_1 = e_2)$, et les contraintes d'inégalité large $(e_1 \leq e_2)$; avec e_1 et e_2 des expressions données. La fonction *evalC* d'évaluation du coût de violation d'une contrainte est donnée comme suit :

$$\begin{array}{lcl} evalC(e_1 = e_2, I_c) & = & |v_1 - v_2|, \\ evalC(e_1 \leq e_2, I_c) & = & \text{si } v_1 \leq v_2 \text{ alors } 0 \\ & & \text{sinon } (v_1 - v_2). \end{array}$$

3 Discussion et conclusion

Dans ce papier, nous avons introduit une approche de génération automatique des cas de test relatifs au débordement pour les programmes de calcul numérique. Cette approche procède en deux étapes : une première étape de résolution sur les nombres réels produisant une solution approchée sur les flottants ; puis une deuxième étape avec une recherche locale visant à trouver une solution exacte sur les flottants.

Les expérimentations de cette démarche ont montré la nécessité d'une haute précision dans la première recherche locale pour que la deuxième recherche locale puisse explorer un espace réduit sur les flottants. D'où la première perspective, celle d'améliorer algorithmiquement la deuxième recherche locale pour qu'elle nécessite moins de précision dans la première recherche locale. La deuxième perspective est de comparer les résultats de notre approche avec les démarches de vérification des programmes de calcul numérique, et tout particulièrement l'outil Astrée [2].

Références

[1] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Softw. Test., Verif. Reliab.*, 16(2) :97–121, 2006.

[2] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Varieties of static analyzers : A comparison with Astrée, invited paper. In *Proc. First IEEE & IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE '07*, pages 3–17, Shanghai, China, 6–8 June 2007.

[3] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann. MPFR : A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw*, 33(2) :13, 2007.

[4] A. Gotlieb. *Automatic Test Data Generation using Constraint Logic Programming.* PhD thesis, Université de Nice — Sophia Antipolis, France, 2000.

[5] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating point numbers. *Lecture Notes in Computer Science (LNCS)*, pages 2239 :524–538, 2001.

Filtrage basé sur des contraintes *tous différents* pour l'isomorphisme de sous-graphe

Christine Solnon
LIRIS, CNRS UMR 5205, Université de Lyon, Université Lyon 1
christine.solnon@liris.cnrs.fr

Résumé

Le problème de l'isomorphisme de sous-graphe consiste à rechercher une copie d'un graphe motif dans un graphe cible. Ce problème peut être résolu par une exploration exhaustive combinée avec des techniques de filtrage visant à élaguer l'espace de recherche. Cet article introduit un nouvel algorithme de filtrage basé sur des contraintes *tous différents* conditionnelles. Nous montrons que ce filtrage est plus fort que les autres filtrages, dans le sens où il coupe plus de branches, et qu'il est également plus efficace, dans le sens où il permet de résoudre de nombreuses instances plus rapidement.

1 Introduction

Les graphes sont utilisés dans de nombreuses applications pour représenter des objets structurés tels que, par exemple, des molécules, des images ou des réseaux biologiques. Dans beaucoup de ces applications, on cherche une copie d'un graphe motif dans un graphe cible. Ce problème, connu sous le nom d'isomorphisme de sous-graphe, est NP-complet dans le cas général [5].

Le problème d'isomorphisme de sous-graphe peut être résolu par une exploration systématique combinée avec des techniques de filtrage visant à réduire l'espace de recherche. Différents niveaux de filtrage peuvent être envisagés ; certains sont plus forts que d'autres (ils réduisent plus fortement l'espace de recherche), mais ont également une complexité en temps supérieure.

Dans cet article, nous décrivons et comparons différents algorithmes de filtrage pour le problème de l'isomorphisme sous-graphe, et nous introduisons un nouvel algorithme de filtrage dont nous montrons qu'il est plus fort. Nous évaluons expérimentalement ce nouvel algorithme de filtrage sur un benchmark de près de 2000 instances, et nous montrons qu'il est beaucoup plus efficace pour une large majorité d'instances.

2 Définitions et notations

Un graphe $G = (N, E)$ se compose d'un ensemble de nœuds N et un ensemble d'arêtes $E \subseteq N \times N$. L'ensemble des voisins d'un nœud u est noté $adj(u)$ et est défini par $adj(u) = \{u' \mid (u, u') \in E\}$. Dans cet article, on considère implicitement des graphes non orientés, de telle sorte que $(u, u') \in E \Leftrightarrow (u', u) \in E$. L'extension de notre travail à des graphes orientés est directe et peut être trouvée dans [15].

Le problème de l'isomorphisme de sous-graphe entre un graphe motif $G_p = (N_p, E_p)$ et un graphe cible $G_t = (N_t, E_t)$ consiste à décider si G_p est isomorphe à un sous-graphe de G_t. Plus précisément, il s'agit de trouver une fonction injective $f : N_p \to N_t$, qui associe un nœud cible différent à chaque nœud motif, et qui préserve les arêtes du motif, à savoir, $\forall (u, u') \in E_p, (f(u), f(u')) \in E_t$. La fonction f est appelée *fonction de sous-isomorphisme*.

Notons que le sous-graphe n'est pas nécessairement induit de sorte que deux nœuds motifs qui ne sont pas reliés par une arête peuvent être appariés à deux nœuds cibles qui sont reliés par une arête.

Dans la suite, nous supposons sans perte de généralité que $N_t \cap N_p = \emptyset$. Nous notons u ou u' (resp. v ou v') les nœuds de G_p (resp. G_t), et $\#S$ la cardinalité d'un ensemble S. Nous définissons $n_p = \#N_p$, $n_t = \#N_t$, $e_p = \#E_p$, $e_t = \#E_t$ et d_p et d_t les degrés maximaux des graphes G_p et G_t.

3 Filtrages pour l'isomorphisme de sous-graphes

Dans cette section, nous montrons d'abord comment modéliser un problème d'isomorphisme de sous-graphe en un problème de satisfaction de contraintes (CSP). Ensuite, nous décrivons différents algorithmes de fil-

trage pour l'isomorphisme de sous-graphe dans les sections 3.2 à 3.5, et nous les comparons à la section 3.6.

3.1 Modélisation CSP

Un problème d'isomorphisme de sous-graphe peut être modélisé comme un CSP en associant une variable (notée x_u) à chaque nœud motif $u \in N_p$. Le domaine d'une variable x_u (noté D_u) contient l'ensemble des nœuds cibles qui peuvent être appariés à u. Intuitivement, l'affectation d'une variable x_u à une valeur v correspond à l'appariement du nœud motif u au nœud cible v. Le domaine D_u est généralement réduit à l'ensemble des nœuds cibles dont le degré est supérieur ou égal au degré de u car un nœud u ne peut être apparié à un nœud v que si $\#adj(u) \leq \#adj(v)$.

Les contraintes garantissent que l'affectation des variables à des valeurs correspond à une fonction de sous-isomorphisme. Il y a deux types de contraintes : (1) les *contraintes d'arête* garantissent que les arêtes motifs sont préservées, *i.e.*, $\forall (u,u') \in E_p, (x_u, x_{u'}) \in E_t$; et (2) les *contraintes de différence* garantissent que l'affectation correspond à une fonction injective, *i.e.*, $\forall (u,u') \in N_p^2, u \neq u' \Rightarrow x_u \neq x_{u'}$.

Un tel CSP peut être résolu en construisant un arbre de recherche. La taille de cet arbre peut être réduite en utilisant des techniques de filtrage qui propagent les contraintes pour supprimer des valeurs des domaines. Nous décrivons dans les sections 3.2 à 3.5 différentes techniques de filtrage qui peuvent être utilisées pour résoudre les problèmes d'isomorphisme de sous-graphe. Certains de ces filtrages (*FC(Diff)*, *GAC(AllDiff)*, *FC(Edges)* et *AC(Edges)*) sont génériques et peuvent être utilisés pour résoudre n'importe quel CSP tandis que d'autres (*LV2002* et *ILF (k)*) sont dédiés au problème de l'isomorphisme de sous-graphe.

3.2 Propagation des contraintes de différence

Une propagation simple par vérification en avant des contraintes de différence (notée *FC(Diff)*) peut être faite à chaque fois qu'un nœud motif u est apparié à un nœud cible v en supprimant v des domaines de tous les nœuds non appariés. Cela peut être fait en $\mathcal{O}(n_p)$.

FC(Diff) propage chaque contrainte binaire de différence séparément. Un filtrage plus fort peut être obtenu en propageant l'ensemble des contraintes de différence de façon globale, assurant ainsi que chaque nœud motif peut être apparié à un nœud cible différent. Il s'agit de la cohérence d'arc généralisée (notée *GAC(AllDiff)*). Régin a montré dans [13] comment utiliser l'algorithme de couplage maximal dans un graphe bipartie proposé par Hopcroft et Karp pour assurer *GAC(AllDiff)* en $\mathcal{O}(n_p^2 \cdot n_t^2)$.

Exemple 1 *Considérons 4 variables x_1, x_2, x_3 et x_4 telles que $D_1 = \{a\}$, $D_2 = D_3 = \{a,b,c\}$ et $D_4 = \{a,b,c,d\}$.* FC(Diff) *enlève a des domaines de x_2, x_3 et x_4.* GAC(AllDiff) *enlève en plus b et c de D_4 car si x_4 est affectée à b ou c, alors x_2 ne peut plus être affectée à une valeur différente de celles de x_3 et x_4.*

3.3 Propagation des contraintes d'arête

Une propagation simple par vérification en avant des contraintes d'arête (notée *FC(Edges)*) peut être effectuée à chaque fois qu'un nœud motif u est apparié à un nœud cible v en enlevant du domaine de chaque nœud adjacent à u tout nœud cible n'étant pas adjacent à v. Cela peut être fait en $\mathcal{O}(d_p \cdot n_t)$.

Il est possible d'aller plus loin et de vérifier, pour chaque arête motif (u,u') et chaque nœud $v \in D_u$, qu'il existe au moins un nœud $v' \in D_{u'}$ qui soit adjacent à v. Le nœud cible v' est dit *support* de l'appariement (u,v) pour l'arête (u,u'). Si un appariement (u,v) n'a pas de support pour une arête, alors v peut être enlevé de D_u. Une telle propagation des contraintes d'arête est itérée jusqu'à ce que plus aucune valeur ne puisse être enlevée, assurant ainsi la *cohérence d'arc* des contraintes d'arête (notée *AC(Edges)*), *i.e.*,
$\forall (u,u') \in E_p, \forall v \in D_u, \exists v' \in D_{u'}, (v,v') \in E_t$.

La cohérence d'arc a été très largement étudiée et différents algorithmes de filtrage ont été proposés, ayant différentes complexité en temps et en espace. Par exemple, AC4 [12] a une complexité en temps et en espace de $\mathcal{O}(c \cdot k^2)$, où c est le nombre de contraintes et k la taille du plus grand domaine. Ainsi, la complexité de *AC(Edges)* est $\mathcal{O}(e_p \cdot n_t^2)$ quand on considère AC4.

Exemple 2 *Considérons l'instance de la figure 1 (qui n'a pas de solution). Supposons que le nœud* 3 *ait été apparié au nœud E et que E ait été enlevé des domaines des autres nœuds motifs par* FC(Diff) *ou*

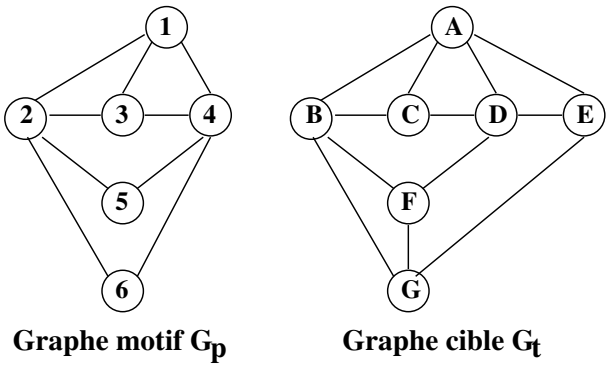


FIGURE 1 – Exemple d'instance.

GAC(AllDiff). FC(Edges) *enlève B, C et F des domaines des nœuds* 1, 2 *et* 4 *car B, C et F ne sont pas adjacents à E tandis que* 1, 2 *et* 4 *sont adjacents à* 3. AC(Edges) *enlève en plus G de D_1 car l'appariement $(1, G)$ n'a pas de support pour l'arête $(1, 4)$ (aucun des nœuds adjacents à G n'appartient à D_4).* AC(Edges) *enlève également G de D_2 et D_4.*

3.4 Propagation d'un ensemble de contraintes d'arête

FC(Edges) et *AC(Edges)* propagent chaque contrainte d'arête séparément. Un filtrage plus fort est obtenu en propageant les contraintes d'arête de façon plus globale, i.e., en propageant le fait que plusieurs nœuds doivent être adjacents à un nœud donné. En effet, un nœud motif u ne peut être apparié à un nœud cible v que si le nombre de nœuds adjacents à u est inférieur ou égal au nombre de nœuds cibles qui sont à la fois adjacents à v et appartiennent au domaine d'un nœud adjacent à u. Ainsi, Larrosa et Valiente ont proposé dans [8] un algorithme de filtrage (noté *LV2002*) qui propage cette contrainte. Plus précisément, ils définissent l'ensemble $\mathcal{F}(u, v) = \cup_{u' \in adj(u)}(D_{u'} \cap adj(v))$, qui est un sur-ensemble de l'ensemble des nœuds pouvant être appariés à des nœuds adjacents à u dans l'hypothèse où u serait apparié à v. Par conséquent, v peut être enlevé de D_u dès lors que $\#\mathcal{F}(u, v) < \#adj(u)$. Il est également possible d'enlever v de D_u dès lors qu'il existe un nœud motif $u' \in adj(u)$ tel que $D_{u'} \cap adj(v) = \emptyset$, assurant ainsi la cohérence d'arc des contraintes d'arête. L'algorithme *LV2002* a une complexité en temps de $\mathcal{O}(n_p^2 \cdot n_t^2)$.

Exemple 3 *Considérons de nouveau l'instance de la figure 1 et supposons que le nœud* 3 *ait été apparié au nœud E et que E ait été enlevé des domaines des autres nœuds motifs. Comme* AC(Edges), LV2002 *enlève les nœuds B, C, F et G de D_1, D_2 et D_4. Il enlève également les valeurs A et D de D_1. En effet,*

$$\begin{aligned} \mathcal{F}(1, A) &= (D_2 \cup D_3 \cup D_4) \cap adj(A) = \{D, E\} \\ \mathcal{F}(1, D) &= (D_2 \cup D_3 \cup D_4) \cap adj(D) = \{A, E\} \end{aligned}$$

Comme $\#\mathcal{F}(1, A) < \#adj(1)$ et $\#\mathcal{F}(1, D) < \#adj(1)$, à la fois A et D sont enlevés de D_1 qui devient vide de sorte qu'une incohérence est détectée.

3.5 Iterated Labelling Filtering (ILF(k))

Zampelli *et al* ont proposé dans [17] un algorithme de filtrage (appelé *ILF(k)*) qui exploite la structure des graphes de façon globale afin de calculer des étiquettes qui sont associées aux nœuds et qui sont utilisées pour filtrer les domaines. Plus précisément, une relation de compatibilité est définie entre les étiquettes et est utilisée pour enlever du domaine d'un nœud motif tout nœud cible dont l'étiquette n'est pas compatible.

ILF(k) est une procédure itérative qui part d'un étiquetage initial. Cet étiquetage initial peut être défini par les degrés des nœuds. Dans ce cas, la relation de compatibilité est l'ordre $\leq$ classique. Cet étiquetage initial peut être étendu pour filtrer plus de valeurs. Etant donné un étiquetage l et une relation de compatibilité $\preceq$ entre étiquettes de l, l'étiquetage étendu l' étiquette un nœud u par le multi-ensemble des étiquettes des nœuds adjacents à u. La relation de compatibilité $\preceq'$ entre ces nouvelles étiquettes est telle que $l'(u) \preceq' l'(v)$ si pour chaque occurrence x d'une étiquette de $l'(u)$ il existe une occurrence différente y d'une étiquette de $l'(v)$ telle que $x \preceq y$. Un point clé réside dans le calcul de cette nouvelle relation de compatibilité $\preceq'$, qui est effectué en $\mathcal{O}(n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$ grâce à l'algorithme de couplage maximal de Hopcroft et Karp (voir [17] pour plus de détails).

Une telle extension d'étiquetage peut être itérée. Un paramètre k est introduit afin de borner le nombre d'extensions d'étiquetage. Notons que ce processus itératif d'extension peut être arrêté avant d'atteindre cette borne k si un domaine est devenu vide ou si un point fixe est atteint (tel que plus aucune valeur ne pourra être enlevée). La complexité en temps de *ILF(k)* est $\mathcal{O}(min(k, n_p \cdot n_t) \cdot n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$.

Exemple 4 *Considérons de nouveau l'instance de la figure 1. L'étiquetage initial basé sur les degrés est*

- $l(5) = l(6) = 2$
- $l(1) = l(3) = l(C) = l(E) = l(F) = l(G) = 3$
- $l(2) = l(4) = l(A) = l(B) = l(D) = 4$

et l'ordre sur ces étiquettes est tel que 2 *est compatible avec* 2, 3 *et* 4 *;* 3 *est compatible avec* 3 *et* 4 *; et* 4 *est compatible avec* 4. *Ainsi, les nœuds C, E, F et G peuvent être enlevés de D_2 et D_4.*

L'extension de cet étiquetage initial est telle que

- $l'(1) = l'(3) = l'(E) = l'(F) = \{\{3, 4, 4\}\}$
- $l'(2) = l'(4) = \{\{2, 2, 3, 3\}\}$
- $l'(5) = l'(6) = \{\{4, 4\}\}$
- $l'(A) = \{\{3, 3, 4, 4\}\}$
- $l'(B) = l'(D) = \{\{3, 3, 3, 4\}\}$
- $l'(C) = \{\{4, 4, 4\}\}$
- $l'(G) = \{\{3, 3, 4\}\}$

et l'ordre sur ces nouvelles étiquettes est tel que $\{\{3, 4, 4\}\}$ est compatible avec $\{\{3, 3, 4, 4\}\}$ et $\{\{3, 4, 4\}\}$; $\{\{2, 2, 3, 3\}\}$ est compatible avec $\{\{3, 3, 4, 4\}\}$ et $\{\{3, 3, 3, 4\}\}$; et $\{\{4, 4\}\}$ est compatible avec $\{\{3, 3, 4, 4\}\}$, $\{\{4, 4, 4\}\}$ et $\{\{3, 4, 4\}\}$. Comme $l'(1)$ n'est pas compatible avec $l'(B)$, B est enlevé de D_1. De même, B, D et G sont enlevés de D_1, D_3, D_5 et D_6. Ce nouvel étiquetage l' peut de nouveau être étendu,

enlevant ainsi de nouvelles valeurs et prouvant finalement l'incohérence de cette instance.

3.6 Discussion

La plupart des algorithmes qui ont été proposés pour résoudre le problème d'isomorphisme de sous-graphe peuvent être décrits par rapport aux algorithmes de filtrage décrits dans les sections 3.2 à 3.5. En particulier, [11] combine *FC(Diff)* et *FC(Edges)* ; [16] combine *FC(Diff)* et *AC(Edges)* ; [14] combine *GAC(AllDiff)* et *AC(Edges)* ; [8] combine *GAC(AllDiff)* et *LV2002* ; et [17] combine *GAC(AllDiff)*, *AC(Edges)* et *ILF(k)*.

Ces différents filtrages assurent différentes cohérences. Certaines sont plus fortes que d'autres. En particulier, *GAC(AllDiff)* est plus fort que *FC(Diff)* tandis que *LV2002* est plus fort que *AC(Edges)* qui est plus fort que *FC(Edges)*. Cependant, *GAC(AllDiff)* et *FC(Diff)* ne sont pas comparables avec *FC(Edges)*, *AC(Edges)*, *LV2002* et *ILF(k)* car ils ne propagent pas les mêmes contraintes.

Les relations entre *ILF(k)* et les autres filtrages propageant des contraintes d'arête (*i.e.*, *LV2002*, *AC(Edges)* et *FC(Edges)*) dépendent des domaines initiaux : si le domaine initial de chaque variable contient tous les nœuds cibles, alors *ILF(k)* est plus fort que *LV2002*, dès lors que le nombre d'extensions k est supérieur ou égal à 2. Cependant, si des domaines ont été réduits (ce qui est souvent le cas quand le filtrage est effectué à un nœud ne se trouvant pas à la racine de l'arbre de recherche), alors *ILF(k)* n'est pas comparable avec *LV2002* et *AC(Edges)*.

En effet, *ILF(k)* n'exploite pas les domaines pour filtrer les valeurs : les étiquettes calculées ne dépendent pas des domaines des variables mais uniquement de la structure des graphes. Afin de propager plus de réductions de domaines, une possibilité consiste à démarrer le processus itératif d'extension à partir d'un étiquetage initial intégrant complètement les réductions de domaines dans la relation de compatibilité de sorte que si un nœud cible v n'appartient pas au domaine d'un nœud motif u alors l'étiquette de v n'est pas compatible avec l'étiquette de u. Avec un tel étiquetage initial (appelé l_{dom} dans [17]), *ILF(k)* est plus fort que *LV2002* dès lors que $k \geq 1$. Cependant, si ce filtrage est plus fort, il est également beaucoup plus coûteux car la complexité en pratique de *ILF(k)* dépend du nombre d'étiquettes différentes. En effet, la complexité théorique d'une itération de *ILF(k)* correspond au pire des cas où tous les nœuds ont des étiquettes différentes. Si les nombres d'étiquettes différentes de nœuds motifs et cibles sont respectivement l_p et l_t, alors la complexité d'une itération de *ILF(k)* est $\mathcal{O}(e_t + l_p \cdot l_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$.

4 Filtrage LAD

Le nouveau filtrage proposé dans cet article exploite le fait que, pour chaque fonction d'isomorphisme f et chaque nœud motif $u \in N_p$, nous avons :

1. $\forall u' \in adj(u), f(u') \in adj(f(u))$
2. $\forall (u', u'') \in adj(u)^2, u' \neq u'' \Rightarrow f(u') \neq f(u'')$

Ces deux propriétés sont des conséquences directes du fait que toute fonction de sous-isomorphisme (1) préserve les arêtes et (2) est une injection. Par rapport au CSP associé à un problème d'isomorphisme de sous-graphe, ces deux propriétés peuvent être exprimées par la contrainte conditionnelle suivante, appelée *contrainte de voisinage* :

$$x_u = v \Rightarrow \quad \forall u' \in adj(u), x_{u'} \in adj(v) \\ \wedge allDiff(\{x_{u'} | u' \in adj(u)\})$$

Cette contrainte de voisinage peut être propagée en recherchant un couplage maximal dans un graphe bipartie, comme proposé par Régin pour la contrainte globale *tous différents* [13]. Rappelons qu'un couplage d'un graphe $G = (N, E)$ est un sous-ensemble d'arêtes $m \subseteq E$ tel qu'il n'y ait pas deux arêtes de m partageant une même extrémité. Un couplage $m \subseteq E$ couvre un ensemble de nœuds N_i si chaque nœud de N_i est une extrémité d'une arête de m.

Pour chaque couple de nœuds (u, v) tel que $v \in D_u$, nous définissons le graphe bipartie qui associe un nœud à chaque nœud adjacent à u ou v et une arête à chaque couple (u', v') tel que $v' \in D_{u'}$.

Définition 1 *Etant donné $(u, v) \in N_p \times N_t$ tel que $v \in D_u$, le graphe bipartie $G_{(u,v)} = (N_{(u,v)}, E_{(u,v)})$ est tel que $N_{(u,v)} = adj(u) \cup adj(v)$ et $E_{(u,v)} = \{(u', v') \in adj(u) \times adj(v) \mid v' \in D_{u'}\}$*

S'il n'existe pas de couplage de $G_{(u,v)}$ couvrant $adj(u)$, alors les nœuds adjacents à u ne peuvent pas être appariés à des nœuds différents qui soient adjacents à v et, par conséquent, v peut être enlevé de D_u. Ce filtrage doit être itéré : quand v est enlevé de D_u, l'arête (u, v) est enlevée des autres graphes biparties de sorte que certains graphes biparties peuvent ne plus avoir de couplage couvrant. Un point clé pour une implémentation efficace de ce filtrage réside dans le fait que l'arête (u, v) n'appartient qu'aux graphes biparties $G_{(u',v')}$ tels que $u' \in adj(u)$ et $v' \in adj(v) \cap D(u')$.

Exemple 5 *Considérons l'instance de la figure 2 et définissons les domaines initiaux par rapport aux degrés des nœuds, i.e.,*

$$\begin{aligned} D_1 = D_3 = D_5 = D_6 &= \{A, B, C, D, E, F, G\} \\ D_2 = D_4 &= \{A, B, D\} \end{aligned}$$

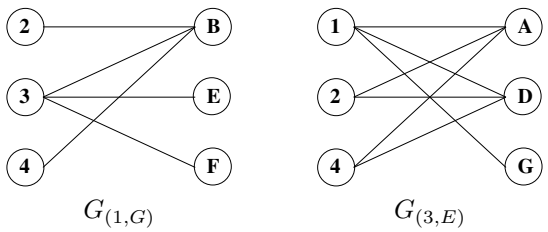


FIGURE 2 – Graphes biparties.

Le graphe bipartie $G_{(1,G)}$ est dessiné dans la partie gauche de la figure 2. Il n'existe pas de couplage de ce graphe couvrant $adj(1)$ car à la fois 2 *et* 4 *ne peuvent être couplé qu'à* B*. Par conséquent,* G *peut être enlevé de* D_1*. Notons que, sur cet exemple, le filtrage* LV2002 *ne peut enlever* G *de* D_1 *étant donné que* $\mathcal{F}(1,G) = (D_2 \cup D_3 \cup D_4) \cap adj(G) = \{B, E, F\}$ *de sorte que* $\#\mathcal{F}(1,G) \geq \#adj(1)$*. Notons également qu'une simple contrainte* allDiff *sur l'ensemble de variables* $\{x_2, x_3, x_4\}$ *ne permet pas d'enlever* G *à* D_1 *: il faut combiner cette contrainte* allDiff *avec le fait que, si* 1 *est apparié à* G*, alors* 2, 3 *et* 4 *doivent être appariés à des nœuds adjacents à* G*.*

*Le graphe bipartie $G_{(3,E)}$ est dessiné dans la partie droite de la figure 2. Il existe un couplage de ce graphe qui couvre $adj(3)$ (*e.g.*, $m = \{(1,G),(2,A),(4,D)\}$) de sorte que* E *n'est pas enlevé de* D_3*. Cependant, dès lors que* G *a été enlevé de* D_1*, l'arête* $(1,G)$ *est enlevée de* $G_{(3,E)}$ *et il n'y a plus de couplage couvrant* $adj(3)$ *(car à la fois* 1, 2 *et* 3 *ne peuvent être couplés qu'à* A *et* D*). Ainsi,* E *est aussi enlevé de* D_3*.*

L'algorithme 1 décrit la procédure de filtrage correspondante, appelée LAD (Local All Different). Cette procédure prend en entrée un ensemble S de couples de nœuds motif/cible à filtrer. A la racine de l'arbre de recherche, cet ensemble devrait contenir tous les couples possibles, *i.e.*, $S = \{(u,v) \mid u \in N_p, v \in D_u\}$. Ensuite, à chaque nouveau point de choix, S devrait être initialisé avec l'ensemble des couples (u,v) tels que $v \in D_u$ et un nœud adjacent à v a été enlevé du domaine d'un nœud adjacent à u depuis le dernier appel à LAD.

Pour chaque couple (u,v) appartenant à S, LAD vérifie qu'il existe un couplage de $G_{(u,v)}$ couvrant $adj(u)$. Si ce n'est pas le cas, v est enlevé de D_u et tous les couples (u',v') tels que u' est adjacent à u et v' est adjacent à v et appartient à $D_{u'}$ sont ajoutés à S.

Un point clé réside dans l'implémentation de la procédure vérifiant qu'il existe un couplage couvrant de $G_{(u,v)}$. Nous utilisons pour cela l'algorithme de Hopcroft et Karp [7] dont la complexité en temps est $\mathcal{O}(d_p \cdot d_t \cdot \sqrt{d_t})$. Cette complexité peut être améliorée en exploitant le fait que l'algorithme de Hopcroft et Karp est incrémental : partant d'un couplage vide, il calcule itérativement de nouveaux couplages contenant de plus en plus d'arêtes jusqu'à ce que le couplage soit maximal. Chaque itération consiste en un parcours en largeur et se fait en $\mathcal{O}(d_p \cdot d_t)$ tandis que le nombre d'itérations est en $\mathcal{O}(\sqrt{d_t})$. Cependant, si l'algorithme débute d'un couplage contenant déja k arêtes et si le couplage maximal a l arêtes alors le nombre d'itérations est également borné par $l - k$.

Algorithm 1: Filtrage LAD

Entrées: Un ensemble S de couples de nœuds motif/cible à filtrer
Sorties: Echec (si une incohérence est détectée) ou Succès. En cas de succès, les domaines sont filtrés de sorte que $\forall u \in N_p, \forall v \in D_u$, il existe un couplage de $G_{(u,v)}$ qui couvre $adj(u)$.

```
1 tant que S ≠ ∅ faire
2     Enlever un couple (u, v) de S
3     si il n'existe pas de couplage de G_(u,v) couvrant adj(u) alors
4         Enlever v de D_u
5         si D_u = ∅ alors retourner Echec
6         S ← S ∪ {(u', v') | u' ∈ adj(u), v' ∈ adj(v) ∩ D_u'}
7 retourner Succès
```

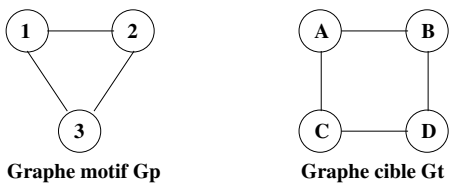
Nous utilisons cette propriété pour améliorer la complexité de LAD. Plus précisément, pour chaque nœud motif $u \in N_p$ et chaque nœud cible $v \in D_u$, nous mémorisons le dernier couplage de $G_{(u,v)}$ calculé. La complexité en espace pour mémoriser tous les couplages couvrants est $\mathcal{O}(n_p \cdot n_t \cdot d_p)$ car il y a $n_p \cdot n_t$ graphes biparties et le couplage couvrant de $G_{(u,v)}$ comporte $\#adj(u)$ arêtes. Il serait trop coûteux, à la fois en temps et en mémoire, de recopier tous ces couplages à chaque point de choix et de restituer ces couplages après chaque retour en arrière. Ainsi, nous ne recopions pas les couplages à chaque point de choix, mais nous les mettons simplement à jour à chaque fois que nous devons vérifier qu'il existe un couplage couvrant de $G_{(u,v)}$: nous mettons tout d'abord à jour le dernier couplage calculé en enlevant les couples (u',v') tels que v' n'appartient plus à $D_{u'}$; si aucun arête n'a été enlevée, alors le couplage reste valide ; sinon Hopcroft Karp est utilisé pour compléter le couplage ; si un nouveau couplage couvrant est trouvé, alors il est mémorisé.

Théorème 1 *La complexité en temps de* LAD *est* $\mathcal{O}(n_p \cdot n_t \cdot d_p^2 \cdot d_t^2)$.

Preuve.
- La complexité du calcul d'un premier couplage couvrant pour chaque graphe bipartie est $\mathcal{O}(n_p \cdot n_t \cdot d_p \cdot d_t \cdot \sqrt{d_t})$; cette étape est effectuée une seule fois, à la racine de l'arbre de recherche.
- Chaque fois qu'une valeur v est enlevée d'un domaine D_u, les couplages de tous les graphes $G_{(u',v')}$ tels que $u' \in adj(u)$ et $v' \in D_{u'} \cap adj(v)$ sont mis-à-jour. Il y a $d_p \cdot d_t$ graphes biparties dans le pire des cas et chaque mise-à-jour est faite incrémentalement en $\mathcal{O}(d_p \cdot d_t)$.
- Dans le pire des cas, seulement une valeur est enlevée lors de la mise à jour des couplages couvrants de tous les voisins et $n_p \cdot n_t$ valeurs peuvent être supprimées.

Nous montrons dans [15] que le filtrage *LAD* établit la cohérence d'arc généralisée des contraintes de voisinage, notée *GAC(voisinages)*. Nous comparons également dans [15] cette cohérence partielle avec les autres cohérences partielles introduites en 3. En particulier, nous montrons que *GAC(voisinages)* est plus forte que *LV2002* et qu'elle est équivalente à *ILF(k)* quand l'étiquetage initial est l_{dom} et quand $k = \infty$. Cependant, le filtrage *LAD*, qui établit *GAC(voisinages)*, a une complexité en temps inférieure. En effet, *ILF(k)* recalcule tous les couplages, pour tous les couples de nœuds motif/cible, à chaque itération, tandis que *LAD* ne met à jour que les couplages qui ont été effectivement impactés par une réduction de domaine.

Enfin, nous montrons dans [15] que *GAC(voisinage)* est moins forte que la cohérence d'arc singleton (SAC) [1] des contraintes d'arêtes combinées à une contrainte globale *tous différents* (notée *SAC(Edges+AllDiff)*). En effet, *SAC(Edges+AllDiff)* assure que, $\forall u \in N_p, \forall v \in D_u$, si D_u est réduit au singleton $\{v\}$, alors *AC(Edges)* combinée avec *GAC(AllDiff)* ne détecte pas d'incohérence. Dans ce cas, il existe un couplage de $G_{(u,v)}$ qui couvre $adj(u)$ de sorte que *GAC(voisinage)* est également assurée. *SAC(Edges+AllDiff)* est en fait strictement plus forte que *GAC(Neighborhood)*. Considérons par exemple l'instance suivante :



Supposons que les domaines initiaux soient $D_1 = D_2 = D_3 = \{A, B, C, D\}$. *GAC(Neighborhood)* ne réduit aucun domaine car chaque graphe bipartie $G_{(u,v)}$ admet un couplage couvrant $adj(u)$. Cependant, *SAC(Edges+AllDiff)* détecte l'incohérence : si D_1 est réduit à $\{A\}$, alors *AC(Edges)* réduit D_2 et D_3 à l'ensemble des nœuds adjacents à A (i.e., à $\{C, B\}$) et l'arête $(3, 2)$ n'a plus de support (car G_t n'a pas d'arête entre C et B) de sorte que *AC(Edges)* détecte une incohérence.

Cependant, la complexité en temps optimale pour assurer la cohérence d'arc singleton d'un CSP binaire est $\mathcal{O}(n \cdot d^3 \cdot e)$ où e est le nombre de contraintes, n le nombre de variables et d la taille des domaines [1]. Si l'on considère le CSP binaire comportant uniquement les contraintes d'arêtes, nous avons $n = n_p$, $d = n_t$ et $e = e_p$ de sorte que *SAC(Edges)* est établie en $\mathcal{O}(n_p \cdot n_t^3 \cdot e_p)$. Dans le pire des cas (si les graphes sont complets), *SAC(Edges)* a la même complexité que *LAD*, à savoir $\mathcal{O}(n_p^3 \cdot n_t^3)$. Cependant, pour des graphes moins denses, *LAD* a une complexité en temps inférieure à *SAC(Edges)*.

5 Evaluation expérimentale

5.1 Jeu d'essai

Nous considérons 1993 instances provenant de 3 bases de données différentes.

Base de données "scale-free" (classes sf-d-D-n et si-d-D-n) Cette base de données a été utilisée dans [17] pour évaluer *ILF(k)*. Les graphes de ces instances sont des réseaux scale-free qui ont été générés aléatoirement en suivant une loi de distribution exponentielle pour les degrés $P(d = k) = k^{-2,5}$ (voir [17] pour plus de détails). Il y a 5 classes. Chacune des 4 premières, notées *sf-d-D-n*, contient 20 instances satisfiables telles que le graphe cible a n nœuds dont les degrés sont bornés entre d et D, et le graphe motif est généré en sélectionnant 90% des nœuds et arêtes du graphe cible. La cinquième classe, notée *si-d-D-n*, contient 20 instances non satisfiables qui ont été générées comme les instances des autres classes, sauf que 10% de nouvelles arêtes ont été ajoutées au graphe motif afin d'obtenir des instances non satisfiables.

Base de données "GraphBase" (classe LV) Cette base de données a été utilisée dans [8] pour évaluer *LV2002*. Elle contient 113 graphes non orientés de natures variées. Nous avons considéré les 50 premiers graphes, ayant de 10 à 128 nœuds. A partir de ces 50 graphes, nous avons généré 793 instances en considérant tous les couples de graphes (G_p, G_t) qui ne sont pas trivialement résolus, *i.e.*, tels que $e_p > 0$, $n_p \leq n_t$ et $d_p \leq d_t$.

Base de données "Vflib" (classes bvg-n, bvgm-n, m4D-n, m4Dr-n et r-d-n) Cette base de données a

été utilisée dans [2] pour évaluer Vflib. Elle contient 63 classes d'instances et chaque classe contient des instances telles que le graphe cible a de 20 à 1000 nœuds. Pour chaque classe, nous n'avons considéré que 4 tailles et, pour chaque taille, nous n'avons considéré que les 10 premières instances. Nous avons regroupé les classes comme suit (voir [4] pour plus de détails sur les classes d'origine).

Les classes *bvg-n* contiennent des graphes à degrés fixés et correspondent aux classes si_x-b_y-n où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible, $y \in \{3, 6, 9\}$ le degré et $n \in \{100, 200, 400, 800\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *bvg-n* contient 90 instances.

Les classes *bvgm-n* contiennent des graphes obtenus en perturbant légèrement des graphes à degrés fixés et correspondent aux classes si_x-$b_y m$-n où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible, $y \in \{3, 6, 9\}$ le degré et $n \in \{100, 200, 400, 800\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *bvgm-n* contient 90 instances.

Les classes *m4D-n* contiennent des graphes correspondant à des maillages réguliers 4D et correspondent aux classes si_x-*m4D-n* où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible et $n \in \{81, 256, 526, 1296\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *m4D-n* contient 30 instances.

Les classes *m4Dr-n* contiennent des graphes correspondant à des maillages irréguliers 4D et correspondent aux classes si_x-$m4Dr_r$-n où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible, $r \in \{2, 4, 6\}$ le degré d'irrégularité et $n \in \{81, 256, 526, 1296\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *m4Dr-n* contient 90 instances.

Les classes *r-p-n* contiennent des graphes générés aléatoirement et correspondent aux classes si_x-$rand$-r_p-n où $x \in \{.2, .4, .6\}$ donne la taille du graphe motif par rapport au graphe cible, $p \in \{.01, .05, .1\}$ la probabilité d'ajouter une arête entre 2 nœuds et $n \in \{100, 200, 400, 600\}$ le nombre de nœuds des graphes cibles. Ainsi, chaque classe *r-p-n* contient 90 instances.

5.2 Approches comparées

LAD Le filtrage *LAD* a été implémenté en C et a été intégré dans une procédure de recherche par construction d'un arbre de recherche. A chaque nœud de l'arbre, le prochain nœud motif à affecter est choisi selon l'heuristique *MinDom*, *i.e.*, on choisit le nœud motif qui a le plus petit nombre de nœuds cibles dans son domaine. Un point de choix est créé pour chaque nœud cible appartenant au domaine de la variable à affecter, et ces différents points de choix sont explorés par ordre croissant de valeur. A chaque nœud de l'arbre de recherche, le filtrage *LAD* est combiné avec *GAC (AllDiff)*.

ILF(k) L'implémentation originale de *ILF(k)* était en Gecode. Nous considérons ici une implémentation en C qui utilise les mêmes structures de données et les mêmes heuristiques d'ordre que *LAD* et qui est aussi combinée avec *GAC(AllDiff)*. Cette nouvelle implémentation est beaucoup plus efficace que l'originale. Par exemple, les instances de la classe *sf5-8-1000* sont résolues en 0.19 secondes avec la nouvelle implémentation de *ILF(1)* tandis qu'elles étaient résolues en 11.2 secondes avec l'ancienne implémentation. Nous donnons les résultats obtenus pour $k \in \{1, 2, 4\}$. Nous ne donnons pas de résultats pour $k > 4$ car cela n'améliore jamais les résultats.

Abscon *Abscon* est un solveur de CSP générique écrit en Java par Lecoutre et Tabary (voir [10] pour plus de détails). Nous considérons deux variantes de ce solveur : *Abscon(FC)*, qui correspond à *FC(Edges)* et *FC(Diff)*, et *Abscon(AC)*, qui correspond à *AC(Edges)* combinée à un filtrage des contraintes de différence intermédiaire entre *AC(Diff)* et *GAC(AllDiff)*. Les deux variantes utilisent l'heuristique d'ordre *MinDom* pour choisir la prochaine variable à affecter.

Vflib *Vflib* [2, 3] est une bibliothèque C++ dédiée à la résolution des problèmes d'isomorphisme de graphes et de sous-graphe. Elle effectue une propagation simple par vérification en avant des contraintes d'arête et de différence, mais cette propagation est limitée aux nœuds qui sont adjacents à des nœuds déjà appariés. Elle utilise des heuristiques d'ordre de variable et de valeur spécifiques : les variables et les valeurs sont choisies de sorte que le sous-graphe induit par les nœuds correspondants soit connexe (sauf si le motif ou la cible sont composés de différentes composantes connexes).

5.3 Résultats expérimentaux

Toutes les expérimentations ont été réalisées sur la même machine, à savoir un Intel Xeon E5520 cadencé à 2.26 GHz, et avec la même limite de temps, à savoir une heure pour chaque instance.

Considérons tout d'abord le problème consistant à chercher toutes les solutions à une instance, ce qui permet une comparaison qui dépend moins des heuristiques d'ordre de valeurs. Pour cela, nous avons écarté les instances qui ont trop de solutions : nous n'avons considéré que les classes de la base "scalefree" ainsi que les plus petites classes de la base "vflib". La table 1

	Vflib		*Abscon(FC)*		*Abscon(AC)*		*ILF(1)*		*ILF(2)*		*ILF(4)*		*LAD*	
	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps
sf5-8-200	16	72.45	**20**	2.04	**20**	1.75	**20**	**0.00**	**20**	0.02	**20**	0.03	**20**	0.02
sf5-8-600	0	-	**20**	138.10	**20**	135.01	**20**	**0.07**	**20**	0.15	**20**	0.15	**20**	0.29
sf5-8-1000	0	-	**20**	1651.11	**20**	1631.88	**20**	**0.19**	**20**	0.55	**20**	0.59	**20**	0.83
sf20-300-300	0	-	16	386.87	16	474.11	**20**	**0.35**	**20**	5.95	**20**	8.24	**20**	2.56
si20-300-300	0	-	6	823.20	5	1046.91	**20**	132.33	19	30.42	19	48.75	**20**	**27.77**
bvg-100	**90**	**0.02**	**90**	1.99	**90**	2.78	**90**	0.04	**90**	0.07	**90**	0.13	**90**	0.75
bvgm-100	89	6.55	89	11.06	**90**	16.57	**90**	**0.48**	**90**	0.49	**90**	**0.48**	**90**	0.53
m4D-81	**30**	0.09	**30**	1.08	**30**	1.04	**30**	0.03	**30**	0.05	**30**	0.05	**30**	**0.02**
m4Dr-81	**90**	1.65	**90**	3.70	**90**	2.40	**90**	**0.18**	**90**	0.19	**90**	0.20	**90**	**0.18**
r0.01-100	21	83.60	23	121.98	28	322.67	**29**	**158.35**	**29**	170.63	**29**	170.53	**29**	180.24
r0.05-100	2	513.01	22	28.60	**23**	56.78	**23**	135.81	22	107.18	22	108.99	**23**	**19.73**
r0.1-100	0	-	25	64.91	28	218.38	28	217.17	28	242	28	243.12	**29**	**148.38**
Tous	338	13.83	451	118.72	460	144.86	480	34.41	478	31.09	478	32.07	**481**	**22.34**

TABLE 1 – Comparaison pour le problème consistant à chercher toutes les solutions : nombre d'instances résolues (nb) et temps moyen correspondant.

		Abscon		*ILF*			*LAD*
class	(#sol)	*FC*	*AC*	*k=1*	*k=2*	*k=4*	
sf5-8-200	(1.10)	3076	108	5	0	0	0
sf5-8-600	(1.00)	418	418	4	0	0	0
sf5-8-1000	(1.05)	557	557	7	0	0	0
sf20-300-300	(4.45)	397844	29338	38	13	7	0
si20-300-300	(0.00)	913730	61191	15342	62	22	27
bvg-100	(218)	10037	2862	461	391	391	0
bvgm-100	(145855)	8977	95618	641	379	222	1
m4D-81	(1253)	1904	327	701	669	652	23
m4Dr-81	(30642)	22920	23592	1356	1304	1300	12
r0.01-100	(57291325)	38853	6749220	10621	6717	6175	60
r0.05-100	(6062230)	233044	615882	2857279	539522	539167	5243
r0.1-100	(30501838)	819714	1985225	2227792	2224579	2224408	320067

TABLE 2 – Comparaison du nombre de nœuds échecs (moyenne sur les instances résolues) ; la colonne #sol donne le nombre moyen de solutions des instances.

Vflib	*Abscon*		*ILF*			*LAD*
	FC	*AC*	*k=1*	*k=2*	*k=4*	
468	647	662	698	699	699	**728**
73.72	72.51	54.25	30.85	31.12	30.77	**14.57**
-	1202372	324075	297107	182588	159493	13560

TABLE 3 – Comparaison pour le problème consistant à chercher une solution, sur les instances de la classe LV : la première ligne donne le nombre d'instances résolues, la deuxième le temps et la troisième le nombre de nœuds échecs (moyennes sur les instances résolues).

montre que pour ces classes, *LAD* a résolu 1 (resp. 3, 3, 23, 29 et 143) instance de plus que *ILF(1)* (resp. *ILF(2), ILF(4), Abscon(AC), Abscon(FC)* et *Vflib*. *LAD* est plus lent que *ILF* pour les classes *sf-5-8-** et *bvg*, mais ces instances sont en fait très faciles et *LAD* les résout en moins d'une seconde. En revanche, pour les classes plus difficiles comme *si20-300-300*, *r0.05-100* et *r0.1-100*, *LAD* est significativement plus rapide que *ILF*. Pour toutes les classes, *LAD* et *ILF* sont plus rapides que *Abscon* (notons cependant qu'*Abscon* est implémenté en Java alors que les autres approches sont implémentées en C, qui est généralement sensiblement plus rapide que Java). *Vflib* est compétitif pour les classes *bvg-100*, *m4D-81* et *m4Dr-81*, mais il n'est pas compétitif du tout pour les autres classes.

La table 2 compare les nombres de nœuds échecs de chaque approche sauf *Vflib* (pour lequel nous ne disposons pas de cette information). Pour certaines classes, comme *sf5-8-**, *LAD* et *ILF* ont un nombre comparable de nœuds échec, et cela correspond aux classes qui sont plus rapidement résolues par *ILF* que par *LAD*. Cependant, pour d'autres classes, comme *r*-100*, *LAD* explore beaucoup moins de nœuds que *ILF*. Le nombre de nœuds échecs pour *ILF* comme pour *LAD* est sensiblement plus petit que pour *Abscon*.

Afin de comparer le passage à l'échelle des différentes approches et les évaluer sur un plus grand nombre d'instances, nous considérons maintenant le problème consistant à chercher uniquement la première solution. La table 3 compare les différentes approches sur la classe *LV*, contenant 793 instances de natures très variées, dont certaines s'avèrent vraiment difficiles. Pour cette classe, *LAD* a résolu 29 (resp. 29, 30, 66, 81 et 260) instances de plus que *ILF(4)* (resp. *ILF(2)*, *ILF(1)*, *Abscon(AC)*, *Abscon(FC))* et *Vflib*). *Abscon(AC)* et *ILF(1)* ont un nombre de nœuds échecs comparable, et en ont environ 3 fois moins que *Abscon(FC). ILF(2)* et *ILF(4)* ont moins de nœuds échecs mais cette réduction de l'espace de recherche n'est pas suffisante pour leur permettre d'être significativement meilleurs qu'*ILF(1)*. Le nombre de nœuds échecs de *LAD* est bien plus petit (plus de 20 fois plus petit que *Abscon(AC)* et *ILF(1)*).

La table 4 compare les différentes approches pour les classes les plus faciles de la base "vflib" et montre que *LAD* est la seule approche capable de résoudre

	Vflib		*Abscon(FC)*		*Abscon(AC)*		*ILF(1)*		*ILF(2)*		*ILF(4)*		*LAD*	
	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps
bvg-200	**90**	**0.00**	**90**	0.68	**90**	0.78	**90**	**0.00**	**90**	**0.00**	**90**	**0.00**	**90**	0.14
bvg-400	**90**	**0.00**	**90**	2.85	**90**	2.99	**90**	0.01	**90**	0.01	**90**	0.01	**90**	1.06
bvg-800	**90**	**0.02**	**90**	54.13	**90**	54.86	**90**	0.03	**90**	0.04	**90**	0.05	**90**	8.41
bvgm-200	**90**	**0.00**	**90**	0.95	**90**	0.73	**90**	**0.00**	**90**	**0.00**	**90**	**0.00**	**90**	0.01
bvgm-400	**90**	**0.01**	89	3.20	89	1.66	**90**	1.55	**90**	**0.01**	**90**	**0.01**	**90**	0.04
bvgm-800	**90**	**0.03**	**90**	12.02	**90**	12.07	**90**	0.06	**90**	0.04	**90**	**0.03**	**90**	0.19
m4D-256	29	0.00	**30**	2.88	**30**	1.73	**30**	**0.01**	**30**	**0.01**	**30**	**0.01**	**30**	0.04
m4D-526	23	4.11	**30**	159.76	**30**	164.90	29	9.61	**30**	32.93	29	30.72	**30**	**1.71**
m4D-1296	20	0.05	23	252.73	23	242.47	29	52.93	29	61.21	29	73.33	**30**	**5.67**
m4Dr-256	**90**	**0.00**	**90**	7.91	**90**	1.44	**90**	0.23	**90**	1.05	**90**	2.24	**90**	0.06
m4Dr-526	**90**	**0.01**	89	23.47	89	23.35	89	14.02	89	18.31	89	19.08	**90**	0.33
m4Dr-1296	**90**	**0.06**	89	193.27	89	188.44	**90**	6.41	**90**	5.46	**90**	5.43	**90**	1.63
Tous	882	0.12	890	41.95	890	40.60	897	4.25	898	5.55	897	6.04	**900**	**1.43**

TABLE 4 – Comparaison pour le problème consistant à chercher une solution sur les instances faciles de "vflib" (nb = nombre d'instances résolues en moins d'une heure et temps = temps moyen pour les instances résolues).

	Vflib		*Abscon(FC)*		*Abscon(AC)*		*ILF(1)*		*ILF(2)*		*ILF(4)*		*LAD*	
	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps	nb	temps
r0.01-200	3	1735.93	28	192.14	**30**	0.99	28	27.48	28	44.09	28	46.49	**30**	**0.04**
r0.01-400	0	-	20	33.14	29	69.68	14	175.83	14	228.78	14	214.85	**30**	**45.58**
r0.01-600	0	-	23	226.38	23	236.14	12	428.14	9	1069.96	7	806.96	**29**	**113.51**
r0.05-200	0	-	25	266.77	28	142.80	28	125.57	28	198.68	28	198.66	**30**	**38.28**
r0.05-400	0	-	22	632.84	24	647.48	**25**	519.04	**25**	500.54	**25**	**494.12**	17	1190.88
r0.05-600	0	-	**14**	**1915.65**	**14**	1936.98	13	1505.51	5	2319.68	5	2304.85	1	2100.61
r0.1-200	0	-	27	143.36	**29**	**309.54**	26	320.52	26	357.70	26	351.10	21	646.31
r0.1-400	0	-	**6**	**1764.63**	**6**	1972.18	5	1950.67	5	1917.68	5	2070.81	1	961.35
r0.1-600	0	-	0	-	0	-	0	-	0	-	0	-	0	-
Tous	3	1735.93	165	443.14	**183**	**409.55**	151	414.03	140	447.36	138	426.67	159	268.48

TABLE 5 – Comparaison pour le problème consistant à chercher une solution sur les instances *r-p-n* de "vflib" (nb = nombre d'instances résolues en moins d'une heure et temps = temps moyen pour les instances résolues).

toutes les instances de ces classes. *Vflib* est très efficace et montre de bonnes propriétés de passage à l'échelle pour certaines de ces classes. De fait, *Vflib* utilise des heuristiques d'ordre de variables et de valeurs qui ne sont pas utilisées par les autres approches : à chaque itération, il choisit d'apparier le couple de nœuds (u, v) tel qu'à la fois u et v soient adjacents à des nœuds déja appariés (dans la mesure du possible). Ces heuristiques peuvent expliquer les très bonnes performances de *Vflib* sur certaines instances quand le but est de trouver juste une solution et que l'instance est satisfiable.

Enfin, la table 5 compare les différentes approches sur les classes aléatoires *r-p-n* de la base "vflib", et montre que les approches comparées exhibent des propriétés de passage à l'échelle différentes : quand la probabilité p d'ajouter une arête est 0.01, *LAD* est meilleur qu'*Abscon* qui est meilleur qu'*ILF*, mais quand cette probabilité augmente, *Abscon* devient meilleur qu'*ILF* qui est meilleur que *LAD*. En fait, plus les graphes ont de nœuds et sont denses, et plus les performances de *LAD* sont dégradées. Cela provient notamment du fait que la complexité du filtrage *LAD* est $\mathcal{O}(n_p \cdot n_t \cdot d_p^2 \cdot d_t^2)$: le degré est 10 fois plus grand (en moyenne) quand $p = 0.1$ que quand $p = 0.01$. En conclusion, quand les graphes sont relativement peu denses, il est intéressant de filtrer avec *LAD* tandis que quand les graphes sont plus denses, il est plus intéressant d'utiliser un filtrage tel que *AC(Edges)*.

6 Conclusion

Nous avons introduit un nouvel algorithme de filtrage pour le problème d'isomorphisme de sous-graphes. Ce filtrage propage le fait que les différents nœuds adjacents à un même nœud motif doivent être appariés à des nœuds cibles qui sont tous différents et qui sont tous adjacents à un même nœud cible. Ce filtrage est plus fort que *LV2002* et assure la même cohérence que la variante la plus forte de *ILF(k)*, *i.e.*,

quand l'étiquetage initial intègre complètement les réductions de domaines et quand les extensions d'étiquetages sont itérées jusqu'à atteindre un point fixe. Cependant, cette cohérence est assurée à un moindre coût en mettant à jour les couplages incrémentalement, au lieu de les recalculer à partir de rien à chaque fois, et en ne mettant à jour que les couplages qui ont été impactés par une réduction de domaine au lieu de recalculer tous les couplages.

Nous avons montré expérimentalement sur un jeu d'essai représentatif de près de 2000 instances que ce nouveau filtrage est capable de résoudre plus d'instances et qu'il réduit drastiquement l'espace de recherche de sorte que de nombreuses instances sont résolues sans énumération. Cependant, ce filtrage est moins bon que la cohérence d'arc pour les graphes les plus denses. Cette procédure de filtrage pourrait être facilement intégrée dans un langage de programmation par contraintes. En particulier, nous avons en projet de l'intégrer dans notre système pour apparier des graphes basé sur les contraintes [9] qui est implémenté au dessus de *Comet* [6].

Nous avons également en projet d'améliorer LAD en étudiant différentes stratégies pour choisir, à chaque itération, le prochain couple de nœuds (u, v) enlevé de S. Dans les expérimentations rapportées ici, nous avons considéré une stratégie simple de *dernier entré premier servi (LIFO)* étant donné que S est implémentée par une pile. Cependant, nous pourrions utiliser une file de priorité ordonnant les couples en fonction du nombre d'arêtes ayant été enlevées dans le graphe bipartie $G_{(u,v)}$.

Remerciements à Yves Deville, pour les nombreuses et enrichissantes discussions sur ce travail, à Jean-Christophe Luquet, qui a implémenté une première version de *ILF* pendant son stage de DUT, et à Christophe Lecoutre, qui m'a guidée avec beaucoup de patience dans l'utilisation d'Abscon. Ce travail a été effectué dans le contexte du projet Sattic (Anr grant Blanc07-1_184534).

Références

[1] Christian Bessiere and Romuald Debruyne. Theoretical analysis of singleton arc consistency and its extensions. *Artif. Intell.*, 172(1) :29–41, 2008.

[2] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the vf graph matching algorithm. In *ICIAP '99 : Proceedings of the 10th International Conference on Image Analysis and Processing*, page 1172, Washington, DC, USA, 1999. IEEE Computer Society.

[3] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, pages 149–159, 2001.

[4] P. Foggia, C. Sansone, and M. Vento. A database of graphs for isomorphism and sub-graph isomorphism benchmarking. In *3rd IAPR-TC15 Workshop on Graph-based Representations in Pattern Recognition*, 2001.

[5] M. Garey and D. Johnson. *Computers and Intractability.* Freeman and Co., New York, 1979.

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

[7] J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4) :225–231, 1973.

[8] J. Larrosa and G. Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical. Structures in Comp. Sci.*, 12(4) :403–422, 2002.

[9] V. le Clément, Y. Deville, and C. Solnon. Constraint-based graph matching. In *15th Conference on Principles and Practice of Constraint Programming (CP)*, volume 5732 of *LNCS*, pages 274–288. Springer, 2009.

[10] C. Lecoutre and S. Tabary. Abscon 112 : towards more robustness. In *3rd International Constraint Solver Competition (CSC'08)*, pages 41–48, 2008.

[11] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inf. Sci.*, 19(3) :229–250, 1979.

[12] R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28 :225–233, 1986.

[13] J.-C. Regin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th Conf. American Assoc. Artificial Intelligence*, volume 1, pages 362–367. Amer. Assoc. Artificial Intelligence, 1994.

[14] J.-C. Régin. *Développement d'Outils Algorithmiques pour l'Intelligence Artificielle. Application à la Chimie Organique.* PhD thesis, 1995.

[15] C. Solnon. *AllDifferent*-based filtering for subgraph isomorphism. *Artificial Intelligence (to appear)*, 2010.

[16] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1) :31–42, January 1976.

[17] S. Zampelli, Y. Deville, and C. Solnon. Solving subgraph isomorphism problems with constraint programming. *Constraints (to appear)*, 2010.

Simulation du comportement d'un opérateur en situation de combat naval

Isabelle Toulgoat[1], **Pierre Siegel**[2], **Yves Lacroix**[3]

[1]DCNS, [2]Université de Provence,[3]Laboratoire Systèmes Navals Compexes

isabelle.toulgoat@dcnsgroup.com, siegel@cmi.univ-mrs.fr, yves.lacroix@univ-tln.fr

Résumé

Cet article porte sur une application qui permet de simuler le comportement d'un officier dans un sous-marin. Nous nous sommes intéressés plus particulièrement à ses réactions en cas de détection d'un sous-marin adverse. Cette application a été implémentée en Prolog.

Dans les simulations de combat naval, les performances opérationnelles de navires militaires sont estimées pour un scénario donné. Dans les modèles courants, les réactions de l'opérateur sont prédéfinies. Ces réactions ne sont pas réalistes : la décision de l'opérateur peut conduire à des réactions inattendues. Cet article présente une méthode pour modéliser le comportement d'un opérateur dans les simulations. Cette méthode permet de raisonner sur des informations incertaines et révisables : un opérateur a une vue partielle de son environnement et il doit réviser ses décisions avec l'arrivée et le changement d'informations. Notre méthode utilise une logique non-monotone : les règles de comportement sont formalisées avec la logique des défauts, à laquelle nous ajoutons une gestion du temps. Nous utilisons des préférences pour gérer le choix entre plusieurs règles, avec une technique simple de probabilité.

Cette méthode a permis de modéliser les réactions d'un officier dans un scénario faisant intervenir deux sous-marins adverses. Cette application, implémentée en Prolog, a été interfacée avec un atelier de simulation de combat naval de DCNS.

Abstract

This article deals with a study case which models the officer behavior in a submarine, and more especially in a case of an adversary detection. This application has been implemented in Prolog.

Naval action's simulations estimate the operational performance of warships or submarines for a given scenario. In common models, the operator's reactions are predefined. This is not realistic : the operator's decision can produce unexpected reactions.

This article presents a method to model operator decision in simulations. This method allows to reason about incomplete, revisable and uncertain information : an operator has only partial information about his environment and must revise his decisions. Our method uses a non-monotonic logic : the rules of behavior are formalized with the default logic, to which we added a consideration of time. Our method uses preferences to manage the choices between different rules, with simple probability techniques.

This application, implemented in Prolog, has been interfaced with DCNS simulator framework and applied to a scenario involving two adverse submarines.

1 Introduction

Le but des simulations de combat naval est d'évaluer les performances opérationnelles de navires militaires ou de forces navales dans un scénario et un théâtre d'opérations données. L'atelier de simulation de DCNS, ATANOR, permet de modéliser des navires et des équipements du système de combat et de dérouler un scénario en temps simulé en faisant interagir les différents modèles [13].

Dans cet atelier de simulation, le comportement est modélisé avec des réseaux de Pétri [8]. Un réseau de Pétri est constitué de places, qui modélisent les différents états de l'équipement et de transitions entre ces places. Les transitions entre ces places sont activées par des évènements internes et externes. Une marque détermine la place courante du réseau de Pétri à un instant donné. Une seule place peut être active à la fois, ce qui interdit les fonctionnements parallèles au sein de l'équipement.

La modélisation des règles de comportement avec un réseau de Pétri donne des réactions prédéfinies et automatiques. Or, dans une situation tactique complexe, l'analyse et la décision d'un opérateur peut conduire à une réaction inattendue. De plus, un inconvénient de la modélisation par réseau de Pétri est de recommencer à implémenter une nou-

velle machine à états si l'on veut ajouter une nouvelle action [5].

Le but de ce travail est de développer un système qui permet de modéliser les lois de comportement d'un opérateur dans les simulations de performances. Nous avons travaillé sur un cas d'application faisant intervenir deux sous-marins adverses, et plus spécialement sur les réactions de l'officier en cas de détection de son adversaire. Un des objectifs est de pouvoir interfacer ce système de modélisation du comportement avec l'atelier de simulation ATANOR : après avoir récupérer les données d'ATANOR sur le sous-marin et son environnement, le système doit pouvoir renvoyer les instructions de manoeuvres pour le sous-marin à ATANOR.

Ce système doit répondre à plusieurs exigences :

- permettre de modéliser les lois de comportement de l'opérateur.
- permettre de raisonner sur des informations incomplètes, révisables et incertaines. En effet, un opérateur a seulement une vue partielle de son environnement. Dans le cas d'un sous-marin, aveugle en immersion, les seules informations proviennent des sonars passifs. Ces informations sont incertaines et incomplètes : le sous-marin peut facilement perdre la détection, l'officier a seulement accès à une estimation de la trajectoire de son adversaire .... Les décisions de l'officier doivent pouvoir être révisées avec l'arrivée de nouvelles informations [3] [11].
- choisir entre plusieurs propositions lorsqu'il propose plusieurs actions possibles pour une même situation.
- permettre l'addition de nouvelles règles de comportement, sans avoir à modifier la représentation des connaissances et sans avoir à remettre en question les règles précédemment établies (à l'inverse des réseaux de Pétri dans lesquels les modifications sont compliquées).
- être capable de raisonner sur des règles générales, sans avoir à compiler de manière très précise toutes les informations. Il n'est pas nécessaire pour l'utilisateur de décrire tous les cas de figures.

Ce travail est demandé par la société DCNS pour des applications militaires : nous avons besoin d'un programme simple et robuste. Nous utilisons pour cela la logique non-monotone la plus connue : la logique des défauts. Nous y ajoutons une prise en compte du temps : nous avons les données sur le sous-marin à l'instant t, et nous en déduisons les instructions au temps suivant $t + 1$. Nous utilisons les préférences pour choisir entre les différentes actions, avec une technique simple de probabilités. Ce travail a été implémenté en Prolog et interfacé avec l'atelier de simulation ATANOR. Une grande partie de ce travail a été présenté à la conférence NMR [14].

Dans cet article, nous présenterons le cas d'application avec quelques règles de comportement. Ensuite, nous présenterons les limites de la logique classique et pourquoi nous utilisons une logique non-monotone. Nous expliquerons la formalisation des règles en logique des défauts. Nous utilisons seulement des défauts normaux et des clauses de Horn dans le but d'avoir un programme simple. Cependant, nous pouvons étendre ce travail à d'autre cas d'application avec des règles plus compliquées. Ensuite, nous expliquons le choix entre les extensions grâce aux préférences et à une technique simple de probabilité. Finalement, nous présentons l'interface réalisée avec ATANOR et des résultats.

2 Cas d'application : détection et pistage d'un sous-marin

Dans un scénario faisant intervenir deux sous-marins, nous modélisons la décision d'un officier de quart dans un sous-marin, en fonction des évènement perçus sur la situation tactique. Nous avons défini des règles de comportement en interrogeant des sous-mariniers.

Voici quelque exemples de règles :

- Règle 1 : Tant que le sous-marin n'a pas de détection, il poursuit une trajectoire aléatoire de recherche dans sa zone de patrouille. Une trajectoire aléatoire de recherche est définie de la façon suivante : le sous-marin avance tout droit et, de temps en temps, il change de cap. Le sous-marin est sourd dans son baffle (à l'arrière du sous-marin, la réception des sonars est amoindrie pour diverses raisons), cette manoeuvre lui permet de vérifier qu'il n'est pas pisté. Le but de cette manoeuvre est de quadriller le plus largement possible la zone, afin d'avoir le plus de chance de détecter un intrus.

  Remarque : c'est une règle de changement minimal [3] [6] [17] . Cette règle est appliquée tant que le sous-marin n'a pas de nouvelles informations.
- Règle 2 : Si le sous-marin détecte un autre sous-marin, l'officier engage les actions suivantes :
  - Manoeuvre d'anti-collision : il manoeuvre de façon à mettre l'ennemi dans un certain gisement (angle entre la direction de détection du but et la ligne de foi du sous-marin qui détecte).
  - Manoeuvre d'élaboration de la solution : il manoeuvre dans le but de confirmer ses informations sur la trajectoire de l'ennemi.
  - Ralliement du poste de pistage : une fois que l'officier s'est assuré que le but ne l'a pas détecté, il peut rallier son baffle, position dans laquelle il ne sera pas détecté.
  - Pistage : une fois que le sous-marin se trouve dans le baffle du but, il peut commencer le pistage. Pour cela, il effectue des tronçons tout en restant dans la baffle.
- Règle 3 : Si les deux sous-marins sont trop proches,

l'officier doit effectuer un dérobement. Il s'éloigne afin de ne pas être contre-détecté et de garder une distance de sécurité.
- Règle 4 : Si le sous-marin est un sous-marin diesel et que quelques heures se sont écoulées depuis la dernière charge de batterie, il doit se mettre au schnorchel. Pour cela, il remonte à l'immersion periscopique et utilise son tube d'air.
- Règle 5 : Si le sous-marin perd la détection, l'officier rallie la dernière position connue de son adversaire. Si il le retrouve, il peut reprendre ses actions de pistage. Sinon, il reprend une trajectoire aléatoire.
- Règle 6 : Avec le sonar à évitement d'obstacle MOAS (Mine and Obstacle Avoidance System), le sous-marin peut détecter des mines, des gros rochers ou encore des falaises. Si le sous-marin détecte un gros rocher, il change de cap afin de placer le rocher de côté.

Ces règles peuvent être en compétition : à un même moment, le sous-marin peut avoir besoin d'effectuer plusieurs actions. Par exemple, il peut avoir besoin de se mettre au schnorchel et également de poursuivre son action de pistage. L'officier doit alors gérer ces choix.

3 La logique classique et ses limites

Les logiques classiques, comme la logique mathématique ou encore le calcul propositionnel, sont des logiques monotones : si on ajoute une information ou une formule E' à une formule E, tout ce qui était déductible de E sera déductible à partir de E $\cup$ E'. Cette propriété de monotonie va être gênante si l'on veut raisonner sur des informations incomplètes, incertaines et révisables. En effet, dans ce cadre, il sera courant d'invalider des conclusions précédemment établies lorsque l'on ajoute de nouvelles informations ou lorsque les informations sont changées [15].

Par exemple, prenons la règle : "Généralement, un sous-marin, qui n'a pas de détection, fait une trajectoire aléatoire de recherche". A première vue, on peut exprimer ce type d'information avec la logique du premier ordre :
Règle 1 :

$\forall x, \neg detection(x) \rightarrow trajectoire\_aleatoire(x)$

Cette formulation est cohérente si la seule information connue est "Le sous-marin n'a pas de détection". Mais si on ajoute la règle : "Si quelques heures se sont écoulées depuis la dernière charge de batterie, le sous-marin doit se mettre au schnorchel", elle est définie de la façon suivante :

Règle 2 : $\forall x, Tlc(x) \geq 5 \rightarrow schnorchel(x)$, avec Tlc le temps écoulé depuis la dernière charge et $schnorchel$ l'action de se mettre au schnorchel.

Il est difficile de gérer des règles générales avec un nombre important d'exceptions [11]. De plus, nous ne pouvons pas réviser le raisonnement et les conclusions. Sachant que le sous-marin n'a pas de détection, nous déduisons qu'il doit faire la trajectoire aléatoire. Mais si nous savons que quelques heures se sont écoulées depuis la dernière charge de batterie, nous en déduisons que le sous-marin doit se mettre au schnorchel. Nous obtenons deux conclusions qui ne sont pas consistantes : le sous-marin ne peut pas en même temps exécuter ces deux actions.

De plus, nous avons besoin d'une logique pour raisonner sur des informations incertaines. Dans le cas d'un sous-marin, aveugle en immersion, les seules informations viennent des sonars passifs, elles sont incertaines et incomplètes [9].

4 La logique non-monotone et la logique des défauts

Une logique non monotone permet d'éliminer la propriété de monotonie de la logique classique : si un raisonnement donne des conclusions avec une base de connaissance donnée, ces conclusions peuvent être revues avec l'arrivée de nouvelles connaissances. Cela permet de prendre en compte les informations incomplètes, incertaines et révisables. Cette logique présente une similitude naturelle avec le raisonnement humain : par manque d'information ou manque de temps, une personne peut raisonner avec des connaissances partielles et réviser les conclusions au besoin lorsqu'elle a plus d'informations.

La logique des défauts, introduite par Ray Reiter [10] en 1980, est la logique non-monotone la plus utilisée. Elle formalise le raisonnement par défaut : des conclusions sensées peuvent être faites en l'absence de preuve contraire. Une théorie des défauts $\Delta = (D, W)$ est composée d'un ensemble de faits W, qui sont des formules issues du calcul propositionnel ou bien de la logique du premier ordre et d'un ensemble de défauts D, qui sont des règles d'inférences à contenu spécifique. Les défauts permettent de gérer l'incertitude. Rappelons la définition d'un défaut et la définition d'une extension :

Définition : Défaut Un défaut est une expression de la forme : $\frac{A(X) : B(X)}{C(X)}$, où A(X), B(X) et C(X) sont des formules et X est un ensemble de variables.

A(X) est le prérequis, B(X) est la justification et C(X) est le conséquent.

Le défaut $\frac{A(X) : B(X)}{C(X)}$ signifie : si A(X) est vérifié, si il est possible que B(X) soit vrai (B(X) est consistant), alors C(X) est vrai.

Si $B(X) = C(X)$, le défaut est dit normal (la justification et le conséquent sont identiques). Les défauts normaux signifient " Les A sont des B, sauf exceptions ", "Normalement, les A sont B".

Définition : Extension L'utilisation des défauts augmente les formules déduites de la base de connaissances W. Pour cela, on calcule les extensions qui sont définies de la façon suivante :

E est une extension de Δ si et seulement si $E = \cup_{i=0,\infty} E_i$, avec

$$E_0 = W \text{ et pour } i \geq O,$$
$$E_{i+1} = Th(E_i) \cup \{C/(\frac{A:B}{C}) \in D, A \in E_i, \neg B \notin E\}$$

où $Th(E_i)$ désigne l'ensemble des théorèmes obtenus de façon monotone à partir de $E_i : Th(E_i) = \{w/E_i \vdash w\}$.

Il est important de noter que E apparaît dans la définition de E_{i+1}. Il n'est donc pas possible de construire E avec un algorithme incrémental.

Lorsque l'on travaille avec des défauts normaux, l'extension est définie de la façon suivante :

E est une extension de Δ si et seulement si $E = \cup_{i=0,\infty} E_i$, avec

$$E_0 = W \text{ et pour } i \geq O,$$
$$E_{i+1} = Th(E_i) \cup \{C/(\frac{A:C}{C}) \in D, A \in E_i, \neg C \notin E_i\}$$

où $Th(E_i)$ désigne l'ensemble des théorèmes obtenus de façon monotone à partir de $E_i : Th(E_i) = \{w/E_i \vdash w\}$.

Pour notre cas d'application, nous utilisons seulement des défauts normaux, mais nous pourrions étendre notre travail aux autres défauts.

5 Formalisation des règles avec la logique des défauts

5.1 Prise en compte du temps

Pour formaliser les règles de comportement, on utilise la logique des défauts, à laquelle on ajoute une prise en compte du temps. En effet, nous avons les données sur le sous-marin à l'instant t et nous déduisons les instructions pour le sous-marin au temps suivant $t + 1$, en fonction de l'état du sous-marin et des nouvelles informations. Pour introduire le temps, nous avons commencé par travailler avec l'article écrit par Cordier et Siegel [3]. Il traite du problème de révision : comment garder à jour une base de connaissances dynamique (qui évolue au court du temps) et comment réviser les informations.

Nous avons besoin de prendre le temps en considération dans les définitions des faits W et des défauts D de la logique des défauts $\Delta = (D, W)$ [12].

5.2 Définition des faits avec prise en compte du temps

L'ensemble des faits W est défini avec les formules de la logique propositionnelle ou la logique du premier ordre. Nous utilisons uniquement les clauses de Horn. Elles permettent d'écrire deux type de règles :

- les clauses de Horn avec un littéral positif :
  $(g(t) \vee \neg f_1(t) \vee \neg f_2(t) \vee \ldots \vee \neg f_k(t))$, où les $f_i(t)$ et $g(t)$ sont des littéraux positifs au temps t. Cette formule peut également s'écrire avec une implication : $(f_1(t) \wedge f_2(t) \wedge \ldots \wedge f_k(t)) \rightarrow g(t)$.
  Ce type de règles permet de définir les règles qui sont toujours vraies, ce sont les règles classiques des systèmes experts. Par exemple, nous formalisons la règle : " Si le sous-marin est en trajectoire aléatoire, il doit tourner d'une angle compris entre α et β", de la façon suivante :
  $trajectoire\_aleatoire(X_t) \rightarrow tourner(X_t, (\alpha, \beta))$
- les clauses de Horn sans littéral positif sont écrites de la façon suivante : $(\neg f_1(t) \vee \neg f_2(t) \ldots \vee \neg f_k(t))$, qui est équivalent à $\neg(f_1(t) \wedge f_2(t) \ldots \wedge f_k(t))$. Nous utilisons ces règles pour définir les exclusions mutuelles deux à deux, cela correspond aux prédicats qui ne peuvent pas être exécutés en même temps :
  $(\neg f_1(t) \vee \neg f_2(t))$, qui est équivalent à $\neg(f_1(t) \wedge f_2(t))$.
  Nous pouvons définir une règle telle que : "Le sous-marin ne peut pas faire en même temps une trajectoire aléatoire et aller au schnorchel" : $\neg(trajectoire\_aleatoire(X_t) \wedge schnorchel(X_t))$.

5.3 Définition des défauts avec prise en compte du temps

Les défauts D sont des règles d'inférence à contenu spécifique, qui prennent en compte l'incertitude. Ils expriment le fait que, si il n'y a pas de contradiction à exécuter une action, le sous-marin peut le faire. Nous utilisons seulement les défauts normaux. Ils permettent de formaliser des règles telles que : "Si le sous-marin n'a pas de détection, il fait une trajectoire aléatoire", de la façon suivante :

$$\frac{\neg detection(X_t) : trajectoire\_aleatoire(X_{t+1})}{trajectoire\_aleatoire(X_{t+1})}.$$

Ce défaut signifie : "Si le sous-marin n'a pas de détection au temps t et si il lui est possible de faire une trajectoire aléatoire au temps $t + 1$, il fait une trajectoire aléatoire au temps $t + 1$".

Les défauts nous permettent de définir les comportements généraux du sous-marin (se mettre au schnorchel, éviter la collision, pister ...). Ensuite, l'ensemble des faits permet de spécifier, pour chaque comportement, l'action à réaliser (changement de cap, vitesse, immersion) ainsi que que les exculsions mutuelles.

5.4 Le calcul d'extension

Nous utilisons le calcul d'extension pour étudier tous les défauts et retenir ceux qui répondent au problème. Chaque

extension est une manoeuvre possible pour le sous-marin : en fonction de l'état du sous-marin au temps t, une extension donne une manoeuvre possible pour le sous-marin au temps $t + 1$. Les défauts normaux permettent d'assurer l'existence d'au moins une extension. Généralement, nous aurons plusieurs extensions pour une même base de connaissance.

Nous pourrions utiliser les ASP (Answer set programming) [7] pour calculer les extensions. Afin de garder un programme simple, nous avons préféré implémenter notre propre calcul d'extension. Les défauts normaux et les clauses de Horn nous permettent d'implémenter de façon simple le calcul des extensions avec le langage Prolog. Nous appelons notre programme NoMROD, pour Non-Monotonic Reasoning for Operator Decision.

6 Sélection des extensions avec des préférences

Le but de cette partie est de simuler la décision de l'officier. Dans une situation tactique, la décision d'un opérateur est un aspect clé. A chaque instant, l'officier doit choisir une action. Nous définissons une méthode qui permet de choisir entre les différentes extensions. Cette méthode permet de simuler différents types de comportements, qui dépendent du caractère de l'officier.

Cette méthode de sélection a été définie en trois étapes :

1. la définition de principes généraux auxquels doit répondre la sélection de l'extension.
2. la définition d'une fonction de poids pour chaque extension. Cette fonction de poids prend en compte le comportement de l'officier à deux niveaux :
   - Niveau 1 : l'importance de chaque défaut en fonction de différents critères. Pour cela, on définit des coefficients de préférences $C_1, \ldots, C_n$, qui permettent de définir des préférences en fonction des différents critères. Ces coefficients décrivent l'importance de chaque action.
   - Niveau 2 : le caractère de l'officier avec des coefficients de caractères $\beta_1, \ldots, \beta_n$. Ces coefficients définissent l'importance que l'officier accorde à chaque critère. Grâce à ces coefficients de caractères, nous pouvons modéliser différents officiers : prudent, téméraire ...

   Les principes de sélection de l'extension ainsi que la fonction de poids ont été développés pour notre cas d'application mais peuvent être facilement généralisés.
3. la définition d'une méthode pour sélectionner une extension, en ajoutant une part d'aléatoire dans le choix de l'extension.

6.1 Les principes de sélection d'une extension

Premièrement, nous étudions des principes et les exigences auxquels la sélection de l'extension doit répondre :

1. choisir les extensions les plus intéressantes et laisser les autres de côté.
   Exemple : NoMROD propose deux extensions :
   – faire une trajectoire aléatoire de recherche.
   – aller au schnorchel.
   La trajectoire aléatoire de recherche est une règle de changement minimal : elle est appliquée tant que le sous-marin n'a pas de nouvelles informations. L'officier choisira donc d'aller au schnorchel.
2. choisir les extensions obligatoires pour la survie de l'équipage.
   Par exemple, l'officier peut repousser la montée au schnorchel, si il est en train d'exécuter une autre action importante (par exemple le pistage). Lorsque cette règle est vérifiée, l'officier a approximativement trente minutes de batterie. Quand cette réserve sera pratiquement vide, l'officier sera obligé d'aller au schnorchel.
3. gérer les choix entre plusieurs extensions.
   Exemple : NoMROD propose deux extensions :
   – éviter la collision avec un sous-marin.
   – éviter la collision avec un gros rocher.
   Ces deux comportements sont très importants pour la sauvegarde du sous-marin. NoMROD doit être capable de choisir entre les extensions de même importance.
4. respecter le changement minimal : tant que le sous-marin n'a pas de nouvelles informations, il reste dans le même état, il ne change pas de comportement. On doit donner plus d'importance à une action déjà commencée.
   Avec cette règle, l'officier persistera dans ses choix, il n'oscillera pas entre plusieurs comportements. Cependant, NoMROD doit être capable de stopper une action si une autre devient obligatoire.
   Example : NoMROD propose deux extensions :
   – pister l'ennemi.
   – monter au schnorchel.
   Supposons que le système choisisse le pistage. Ces deux extensions seront proposées tant que la montée au schnorchel n'aura pas été effectuée. Le meilleur choix pour l'officier est de poursuivre le pistage, et lorsque le schnorchel devient obligatoire (le sous-marin a juste assez de batterie pour monter à la surface), l'officier doit effectuer cette action.
5. le sous-marin ennemi ne doit pas deviner quelle action notre officier choisira.

Ces principes sont des principes de bon sens, qui sont facilement généralisables à d'autre cas d'application.

6.2 La fonction de poids pour les extensions

Nous définissons une fonction de poids pour donner des poids aux extensions. Ces poids quantifient l'importance des extensions. Nous utilisons une méthode d'aide à la décision multi-critères, qui sert à modéliser les préférences d'un décideur. Ces méthodes servent à résoudre des problèmes décisonnels complexes, où plusieurs critères doivent être pris en compte dans le choix [1],[16]. Il existe plusieurs catégories de méthodes d'aide à la décision multi-critères. Nous utilisons la théorie de l'utilité multi-attribut, qui permet d'agréger les différents points de vue en une fonction unique. Elle est basée sur l'axiome suivant :

Tout décideur essaye inconsciemment de maximiser une fonction $U = U(g_1, \ldots, g_n)$, qui agrège tous les points de vue à prendre en compte (avec g_i les critères).

Nous utilisons une fonction d'aggrégation simple : une somme pondérée.

Chaque extension utilise des défauts. Pour quantifier l'importance des extensions, nous introduisons des coefficients de préférences sur les défauts.

6.2.1 Les coefficients de préférences sur les défauts

Chaque défaut est un comportement général. Pour spécifier l'importance des défauts, nous leurs attribuons des coefficients de préférences.

De façon similaire, les préférences sont utilisées dans des systèmes de raisonnement non-monotone. Par exemple, Brewka [2] a défini une logique des défauts avec des priorités et une définition d'ordre dans lequel les défauts doivent être appliqués. Delgrande et al [4] ont présenté une classification des préférences pour les approches basées sur des raisonnements non-monotones.

Nous définissons différents coefficients de préférences $C_1, \ldots, C_n$ pour spécifier l'importance des défauts en fonction de différents critères. Par exemple, nous pouvons spécifier l'importance des défauts pour la sauvegarde du sous-marin, pour l'efficacité dans la mission du sous-marin, pour l'obéissance aux ordres, pour des critères écologiques ...

Pour chaque défaut D_j, nous attribuons les valeurs de ces coefficients $C_{1j} \ldots C_{kj}, \ldots, C_{nj}$. Le coefficient C_{kj} définit l'importance du défaut j par rapport au critère k. Nous fixons arbitrairement ces coefficients entre 0 et 1000.

Par exemple, on définit quatre défauts : $\{D_1, D_2, D_3, D_4\}$, et deux coefficents : la sauvegarde du sous-marin $C_{sauvegarde}$ et l'efficacité dans la mission du sous-marin $C_{efficacite}$. Pour chaque défaut, on définit la valeur des coefficients de sauvegarde et d'efficacité (cf tableau 1).

TAB. 1 – Valeurs des coefficients pour chaque défaut.

Défauts	$C_{sauvegarde}$	$C_{efficacite}$
D_1	500	600
D_2	10	800
D_3	1000	900
D_4	50	60

TAB. 2 – Scores des extensions.

Scores	E_1	E_2
$Score_{sauvegarde}$	550	1060
$Score_{efficacite}$	660	1760

6.2.2 Fonction d'utilité

Chaque extension E_i utilise des défauts : $E_i = \{D_1 \ldots, D_j, \ldots, D_m\}$. Dans notre cas, chaque extension utilise au moins un défaut.

Pour chaque extension E_i, nous calculons les scores $\{Score_1(E_i), \ldots, Score_k(E_i), \ldots, Score_n(E_i)\}$, qui correspondent respectivement aux critères $1, \ldots, n$.

Un score $Score_k(E_i)$ est la somme des coefficients C_k de chaque défaut utilisé dans l'extension E_i : $Score_k(E_i) = \sum_{j=1}^{m} C_{kj}$.

Nous supposons que NoMROD propose p extensions. Pour chaque $Score_k(E_i)$ pour une extension E_i, nous calculons une fonction d'utilité $\mu_k(E_i)$. Nous voulons que la somme des fonctions d'utilité μ_k soit égale à 1 : $\sum_{j=1}^{p} \mu_k(E_j) = 1$. Le score $Score_k(E_i)$ est divisé par la somme de tous les $Score_k(E_j)$ des p extensions proposées par le système : $\mu_k(E_i) = \dfrac{Score_k(E_i)}{\sum_{j=1}^{p} Score_k(E_j)}$.

Reprenons les défauts donnés en exemple dans le paragraphe 6.2.1.

On suppose que l'on doit choisir entre deux extensions : $E_1 = \{D_1, D_4\}$ et $E_2 = \{D_2, D_3, D_4\}$.

On calcule les scores des extensions (cf tableau 2) et les fonctions d'utilité (cf tableau 3).

6.2.3 Le caractère de l'officier

Nous voulons modéliser le caractère de l'officier. En fonction de son caractère, l'officier adoptera des tactiques différentes. Par exemple, un officier prudent donnera plus d'importance aux comportements qui assurent la sauvegarde du sous-marin. Un officier téméraire favorisera les comportements importants pour la réussite de la mission

TAB. 3 – Fonctions d'utilité.

μ	E_1	E_2
$\mu_{sauvegarde}$	$\frac{550}{1610}$	$\frac{1060}{1610}$
$\mu_{efficacit}$	$\frac{660}{2420}$	$\frac{1760}{2420}$

du sous-marin.

Pour modéliser ces caractères, nous définissons des coefficients de caratères $\beta_1, \ldots, \beta_n$, de telle façon que la somme de ces coefficients soit égale à 1 : $\sum_{k=1}^{n} \beta_k = 1$. Le coefficient β_k correspond à l'importance accordé par l'officier au critère k.

Définissons, par exemple, un officier plutôt prudent. En reprenant l'exemple du paragraphe 6.2.1, on doit attribuer deux coefficients de caractère : un coefficient pour la sauvegarde $\beta_{sauvegarde}$ et un coefficient pour l'efficacité $\beta_{efficacite}$. Un officier prudent va préférer favoriser la sauvegarde de son sous-marin plutôt que l'efficacité dans la réussite de la mission, prenons par exemple : $\beta_{sauvegarde} = 0.6$ et $\beta_{efficacite} = 0.4$.

6.2.4 Fonction de poids

Finalement, nous obtenons la fonction de poids de chaque extension E_i, en faisant la somme des fonctions d'utilité μ_k, pondérées par les coefficients de caractères β_k :

$$P(E_i) = \sum_{k=1}^{n} \beta_k \mu_k(E_i).$$

La somme des fonctions de poids des extensions a la propriété suivante : $\sum_{i=1}^{p} P(E_i) = 1$

En reprenant l'exemple précédent, on obtient les fonctions de poids suivantes :

$P(E_1) = \beta_{sauvegarde} * \mu_{sauvegarde}(E_1) + \beta_{efficacit} * \mu_{efficacit}(E_1)$

$P(E_2) = \beta_{sauvegarde} * \mu_{sauvegarde}(E_2) + \beta_{efficacit} * \mu_{efficacit}(E_2)$

Finalement, on a $P(E_1) = 0.31$ et $P(E_2) = 0.69$.

6.3 Choix aléatoire d'une extension

Dans une situation tactique, la décision d'un opérateur est un aspect clé, qui peut conduire à des réactions inattendues. En effet, chaque opérateur possède ses propres réactions. Dans notre cas d'application, le choix ne doit donc pas toujours être déterministe.

De plus, avec des réactions attendues, le sous-marin ennemi pourrait facilement deviner les réactions de l'officier.

Nous avons donc besoin d'une méthode qui prend en compte les réactions inattendues. Pour ces raisons, nous ne choisissons pas toujours les extensions avec un poids maximum. Nous préférerons introduire d'avantage d'incertitude avec un choix aléatoire. Cependant, ce choix aléatoire doit être cohérent avec les principes qui guident la décision de l'officier et avec les principes de sélection d'une extension définis au paragraphe 6.1.

6.3.1 Choix aléatoire

Le choix aléatoire est basé sur un tirage aléatoire : on a p extensions : $E_1, \ldots, E_p$, et les fonctions de poids respectives : $P(E_1), \ldots, P(E_p)$, telles que $\sum_{i=1}^{p} P(E_i) = 1$. Chaque fonction de poids $P(E_i)$ correspond à la probabilité pour l'extension d'être choisie.

Exemple 1 : Nous devons choisir entre deux extensions :
- E_1 avec la fonction de poids $P(E_1) = 0.1$.
- E_2 avec la fonction de poids $P(E_2) = 0.9$.

Nous avons une chance sur dix de choisir l'extension E_1, et neuf chances sur dix de choisir l'extension E_2. Avec le choix aléatoire, nous pouvons gérer les choix entre plusieurs extensions (principe 3).

6.3.2 Correction de la fonction de choix des extensions

Le tirage aléatoire est réaliste si nous avons à choisi entre des extensions qui ont des poids proches. Par contre, il nous pose des problèmes dans certains cas. En effet, si nous avons une extension avec un poids très important, il semble plus logique de choisir celle là (exemple 1). Nous avons le même problème dans le cas suivant (exemple 2). NoMROD propose six extensions : cinq extensions avec une même fonction de poids à 0.1 et une extension avec un poids de 0.5. Le tirage aléatoire donne autant de chance d'être choisies aux cinq extensions avec la fonction de poids à 0.1 : $5 * 0.1 = 0.5$, qu'à l'extension avec la fonction de poids à 0.5.

Nous devons modifier la fonction de poids, dans le but de donner un poids plus important aux extensions importantes et un poids moins important aux autres. Pour cela, nous appliquons une correction aux fonctions de poids : la fonction puissance $f(x) = x^k$, avec $k > 1$. La correction est appliquée de la façon suivante :

- Nous devons choisir entre p extensions : $E_1, \ldots, E_p$, avec les fonctions de poids respectives : $P(E_1), \ldots, P(E_p)$, telles que $\sum_{i=1}^{p} P(E_i) = 1$.
- Nous appliquons la fonction puissance $f(x) = x^k$, avec $k > 1$: $P(E_1)^k, \ldots, P(E_p)^k$. Plus la puissance

k est importante, plus les extensions avec des petits poids seront réduites.
– La somme des fonctions de poids est ensuite ramenée à 1 : nous divisons par la somme des fonctions de poids de toutes les extensions $\frac{P(E_j)^k}{\sum_{i=1}^{p} P(E_i)^k}$.

Cette correction donne plus d'importance aux extensions avec des fonctions de poids importantes, et moins d'importance aux autres. Pour le moment, nous ne fixons par les valeurs de la puissance k, nous voulons tester différentes valeurs.

Appliquons cette correction à l'exemple 2. Nous prenons la fonction puissance $f(x) = x^2$. La fonction de poids 0.1 devient $0.1^2 = 0.01$ et la fonction de poids à 0.5 devient $0.5^2 = 0.25$.

On ramène la somme des fonctions de poids à 1. La somme des fonctions de poids avec correction est $\sum_{i=1}^{6} P(E_i) = 0.3$.

On obtient cinq extensions avec la fonction de poids $\frac{0.1^2}{0.3} = 0.04$ et une extension avec la fonction de poids $\frac{0.5^2}{0.3} = 0.8$. Avec la correction, on donne plus de chance à l'extension avec un poids important d'être choisi.

6.3.3 Filtrage des extensions avec de fonctions de poids trop faibles

Pour être sûr de choisir l'extension la plus intéressante et de laisser les autres de côté (principe 1), nous éliminons les extensions avec des fonctions de poids trop petites. On fixe un seuil : les extensions avec une fonction de poids inférieure à ce seuil sont supprimées. Pour le moment, nous ne fixons pas cette valeur de seuil : nous voulons tester différentes valeurs (nous pourrions par exemple fixer cette valeur en fonction de la fonction de poids maximale).

Maintenant que nous avons défini les fonctions de poids pour les extensions, nous définissons une règle qui prend en compte le changement minimal.

6.4 Prise en compte du changement minimal

Pour respecter le changement minimal (principe 4), nous définissons une règle générale de changement minimal. Pour cela, on garde en mémoire le comportement du sous-marin en cours (trajectoire aléatoire, l'évitement de la collision, le pistage, ...) au temps t. On appelle ce comportement $Comportement(t)$. Chaque comportement est le conséquent d'un défaut D_i :

$$D_i = \frac{Prerequis(t) : Comportement(t+1)}{Comportement(t+1)}.$$

Avec la règle du changement minimal, nous voulons donner plus de chance à une action déjà commencée. Pour cela, nous définissons le défaut suivant :

$$D_{ch\_min} = \frac{Comportement(t) \wedge Prerequis(t) : Comportement(t+1)}{Comportement(t+1)}.$$

Cette règle signifie : " Si le prérequis du comportement précédent est toujours vrai au temps t, et si il est possible de rester dans ce comportement au temps $t+1$, le sous-marin peut garder le même comportement au temps $t+1$".

Cette règle donne plus de chance à une action déjà commencée et permet à l'officier de persister dans ses choix.

7 Interface avec l'atelier de simulation et résultats

Une interface a été réalisée entre NoMROD et l'atelier de simulation ATANOR. ATANOR envoie à NoMROD les informations sur :
– le sous-marin auquel on applique les règles de comportement (cap, vitesse, immersion, position, ...)
– le sous-marin ennemi (détection, position estimée, vitesse estimée, ...).

NoMROD compile les règles de comportement, sélectionne une extension et renvoie les instructions de cap, vitesse et immersion à ATANOR.

Sur la figure 1, nous avons une exécution de NoMROD, interfacée avec l'atelier de simulation ATANOR. Cette exécution simule un scénario d'un peu moins de trois heures.

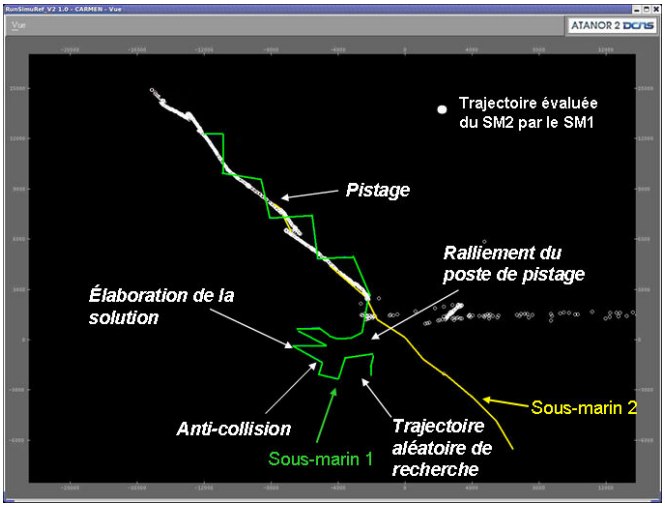


FIG. 1 – Détection et pistage d'un sous-marin

On appelle Sous-marin 1, le sous-marin auquel on applique les règles de comportement de NoMROD. Et on note Sous-marin 2 le sous-marin ennemi, dont le comportement est défini par ATANOR. Le but du sous-marin ennemi est de traverser la zone de patrouille sans être détecté.

Dans ce scénario, le sous-marin 1 fait une trajectoire aléatoire de recherche, car il n'a pas de détection. Lorsqu'il détecte l'ennemi, l'officier enclenche les actions suivantes : évitement de la collision, élaboration de la solution, ralliement du poste de pistage, pistage. Au début, la trajectoire évaluée du sous-marin 2 par le sous-marin 1 est éloignée de la trajectoire réelle. La manoeuvre d'élaboration de la solution permet d'obtenir une meilleure estimation de la trajectoire du sous-marin ennemi.

Nous obtenons un programme efficace. Pour simuler un scénario d'environ 2h40, le temps de calcul utilisé par ATANOR et NoMROD est d'environ 6 secondes et le programme NoMROD ne prend que 20 % de ce temps de calcul.

Ce scénario a été validé par des sous-mariniers, qui retrouvent bien les actions qu'ils engagent en cas de détection d'un sous-marin ennemi.

8 Conclusion

La logique des défauts permet de formaliser les règles de comportement d'un officier dans un sous-marin, en prenant en compte les informations incomplètes, incertaines et révisables. La formalisation en logique des défauts permet d'écrire des règles générales, les défauts, sans avoir à prendre en compte les règles précédemment établies. Nous avons juste à prendre en compte l'écriture des exclusions mutuelles entre les différents comportements.

Le système obtenu compile les informations valables et donne toutes les possibilités d'action avec les extensions.

Pour simuler le choix de l'officier, nous avons défini des fonctions de poids pour chaque extension avec des coefficients de préférence sur les défauts et des coefficients de caractère. Nous ajoutons ensuite une part supplémentaire d'aléatoire dans le choix des extensions.

Nous devons maintenant travailler sur une méthode générale pour attribuer la valeur des coefficients de préférence sur les défauts et les coefficients de caractères (par exemple par apprentissage). Nous voudrions également tester d'autres fonctions de poids pour les extensions (par exemple l'intégrale de Choquet).

Références

[1] S. Ben Mena. Introduction aux méthodes multicritères d'aide à la décision. *Biotchnol.Agron.Soc.envion.*, pages 83–93, 2000.

[2] G. Brewka. Adding priorities and specificity to default logic. In L.Pereira and D.Pearce, editors, *Lecture Notes in Artificial Intelligence*, pages 247–260. European Workshop on Logics in Artificial Intelligence (JELIA'94), 1994.

[3] M.O. Cordier and P. Siegel. A temporal revision model for reasoning about world change. In *Second International Conference on Principles of Knowledge Representation and Reasoning*, pages 732–739, 1992.

[4] J. Delgrande, T. Shaub, and H. Tompits. A classification and survey of preference handling approaches in non-monotonic reasoning. *Computational Intelligence*, 20(2) :308–334, 2004.

[5] Ferber. *Les systèmes multi-agents. Vers une intelligence collective.* InterEditions, 1995.

[6] M. Ginsberg and D.E Smith. Reasoning about action 1 : a possible worlds approach. *Readings in Non Monotonic Reasoning*, 1987.

[7] P. Nicolas, L. Garcia, and I. Stephan. Possibilistic stable models. In *International Joint Conferences on Artificial Intelligence*, 2005.

[8] C.A. Petri. Fundamentals of a theory of asynchronous information flow. In *1st IFIP World Computer Congress*, 1962.

[9] Jr. Prouty. *Displaying uncertainty : a comparison between submarine subject matter experts*. PhD thesis, Naval postgraduate school, Monterey, California, 2007.

[10] R. Reiter. A logic for default reasoning. *Artificial intelligence*, 1980.

[11] L. Sombé. *Raisonnement sur des informations incomplètes en intelligence artificielle*. Teknea, 1989.

[12] I. Toulgoat. Modélisation du processus de décision d'un opérateur dans les simulations de combat naval. In *Journées d'Intelligence Artificielle Fondamentales, Session doctorants,Marseille*, 2009.

[13] I. Toulgoat, J. Botto, Y. De lassus, and C Audoly. Modeling operator decision in underwater warfare performance simulations. In *Conference UDT, Cannes.*, 2009.

[14] I. Toulgoat, P. Siegel, Y. Lacroix, and J. Botto. Operator decision in naval action's simulations. In *13th International Workshop on Non-Monotonic Reasoning*, 2010.

[15] I. Toulgoat, P. Siegel, Y. Lacroix, and J. Botto. Operator decision modeling in a submarine. In *9th International Conference on Computer and IT Applications in the Maritime Industries*, pages 65–75, 2010.

[16] P. Vincke. *L'aide multicritère à la décision*. Ellipses, 1989.

[17] M. Winslett. Reasoning about actions using a possible model approach. In *Proceedings of the 7th National Conference of AI*, pages 89–93, 1988.