AfIA

Association française
pour l'Intelligence Artificielle

JFPC

Journées Francophones
de Programmation par Contraintes

PFIA 2023

Table des matières

Élise Vareilles, ISAE SUPAERO
Éditorial 5

Comité de programme 6

JFPC 1 - 3 juillet 2023, 10h30-12h 7

Amel Hidouri, Said Jabbour, Badran Raddaoui
Sur l'énumération des motifs High Utility Fréquents : Une approche basée sur l'IA symbolique 8

Auguste Burlats, Gilles Pesant
Exploiter l'entropie en programmation par contraintes 10

Thibault Falque, Jean Marie Lagniez, Christophe Lecoutre, Romain Wallon
Approximer pour mieux résoudre 13

Pierre Siegel, Andrei Doncescu, Vincent Risch, Sylvain Sené
Representation of Gene Regulation Networks by Hypothesis Logic Based Boolean Systems ... 23

JFPC 2 - 4 juillet 2023, 10h30-12h 25

Suruthy Sekar, Gael Glorian, Guillaume Perez, Wijnand Suijlen, Eric Monfroy, Arnaud Lallouet
Explications paresseuses Ad hoc des contraintes Element et Sum 26

Anthony Blomme, Daniel Le Berre, Anne Parrain, Olivier Roussel
Compresser des arbres de recherche UNSAT à l'aide d'un système de cache 38

Mohamed-Bachir Belaid, Nassim Belmecheri, Arnaud Gotlieb, Nadjib Lazaar, Helge Spieker
Approche générique pour l'acquisition de contraintes qualitatives 46

Matthieu Py, Arnauld Tuyaba
Utilisation de SAT pour résoudre le problème SALBP avec minimisation du pic de consommation 48

JFPC 3 - 4 juillet 2023, 14h40 - 16h10 52

Djawad Bekkoucha, Abdelkader Ouali, Justine Reynaud, Bruno Cremilleux, Patrice Boizumault, Aymeric Beauchamp
Extraction de Motifs d'Intervalles Fermés en utilisant la Programmation Par Contraintes 53

Frederic Koriche, Christophe Lecoutre, Anastasia Paparrizou, Hugues Wattez
Identification de la meilleure heuristique en satisfaction de contraintes 62

Vianney Coppé, Xavier Gillard, Pierre Schaus
Accélération de l'algorithme de séparation et évaluation pour les diagrammes de décision grâce à la mémoïsation 71

Vinasétan Ratheil Houndji, Généreux Akotenou, Afis Kousse, Klaus Bonou Selegbe
GrailSolver, à la dernière itération de l'obtention du graal de la programmation 76

JFPC 4 - 4 juillet 2023, 16h30 - 18h00 80

Grégoire Menguy, Sebastien Bardin, Nadjib Lazaar, Arnaud Gotlieb
Analyse de Code Automatique : Revisiter l'Inférence de Préconditions via l'Acquisition de Contraintes 81

Auguste Burlats, Pierre Schaus, Cristel Pelsser
Placement optimal de moniteurs dans un réseau pour la tomographie booléenne 83

Augustin Delecluse, Charles Thomas, Pierre Schaus
Variables de séquence pour le problème du voyageur de commerce avec fenêtres de temps 88

Arnaud Malapert, Margaux Schmied, Davide Fissore, Marie Pelleau, Ambre Picard Machetto
Jouer avec des Cryptarithmes en Programmation par Contraintes 93

JFPC 5 - 5 juillet 2023, 10h30 - 12h00 102

Yousra El Ghazi, Djamal Habet, Cyril Terrioux
Conception des lignes d'un réseau de transport maritime à l'aide de la PPC 103

Khadidja Fellah, Soumaya Fellah, Bouabdellah Kechar
Optimisation de la consommation d'énergie et de la collecte de données dans les réseaux de capteurs sans fil 113

Julien Rouzot, Christian Artigues, Philippe Garnier, Emmanuel Hebrard, Pierre Lopez
Une contrainte globale pour l'ordonnancement des transferts de données dans les missions spatiales 115

Éditorial

Journées Francophones de Programmation par Contraintes

Les Journées Francophones de Programmation par Contraintes (JFPC) sont organisées à l'initiative de l'Association Française de Programmation par contraintes (AFPC). Elles constituent le principal congrès francophone centré autour des problèmes de satisfaction de contraintes (CSP), de satisfiabilité (SAT) et de programmation logique sous contraintes (CLP). Les JFPC regroupent aussi des thématiques liées comme la recherche opérationnelle (RO), les méta-heuristiques, l'analyse par intervalles, etc. De part ses applications, la programmation par contraintes s'ouvre à de nombreuses communautés connexes, en particulier la robotique et la bio-informatique.

Les JFPC se veulent un lieu convivial de rencontres, de discussions et d'échanges entre doctorants, chercheurs confirmés et industriels. Historiquement, les JFPC sont issues de la fusion entre les Journées Francophones de Programmation Logique avec Contraintes (JFPLC) qui existaient depuis 1992 (voire 1982, si on considère son prédécesseur, le Séminaire Programmation en Logique) et des Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets (JNPC) qui existaient depuis 1995 (pendant les deux premières années sous le nom de CNPC). Ces dix-huitièmes Journées Francophones de Programmation par Contraintes ont eu lieu du 03 au 05 juillet 2023 à Strasbourg dans le cadre de la Plate-Forme Intelligence Artificielle (PFIA'23). Auparavant, les JFPC avaient pendant quelques années été organisées conjointement avec les JFPLC (avant la fusion) et plus tard avec les JIAF (avant que ces derniers ne rejoignent la PFIA).

Le comité de programme a été composé en essayant de représenter le plus de lieux possibles et en intégrant un certain nombre de jeunes chercheurs (doctorants en toute fin de thèse ou ATER) en plus des chercheurs expérimentés. La présence de jeunes chercheurs dans le comité de programme se justifie par l'état d'esprit général de JFPC : être un lieu de rencontres entre tous les chercheurs de la communauté où les plus jeunes peuvent prendre de l'expérience dans un cadre convivial et bienveillant. Le passé a montré que les chercheurs débutants sont très soucieux de relire les articles avec un soin particulier, ce qui compense leur manque d'expérience. Je remercie tous les membres du comité de programme du temps qu'ils ont passé et de la qualité de leur relecture.

Cette année, dix-neuf soumissions (douze articles longs, un article court et six résumés d'articles déjà publiés dans une conférence internationale de renom l'an passé) ont été relues, toutes ont été acceptées. C'est à nouveau un taux d'acceptation exceptionnel qui nous permettra de discuter autour des problématiques actuelles de la PPC. Merci aux membres du comité de lecture pour leurs remarques bienveillantes et leurs recommandations pertinentes apportées aux jeunes chercheurs.

En plus des présentations orales des articles que vous retrouverez dans ces actes, nous avons eu pendant la conférence deux exposés invités industriels : celui de Siham ESSODAIGUI, Responsable de service Intelligence Artificielle Appliquée chez Groupe Renault, France sur "Application de la PPC à l'industrie automobile : passé, présent et futur", et celui de Jean-Guillaume Fages, Co-fondateur, COSLING, France sur "La Programmation Par Contraintes au service de l'industrie".

Il me reste à remercier particulièrement toute l'équipe du comité d'organisation de PFIA qui nous ont accueilli avec bienveillance et souplesse, l'encadrement rigoureux et très rodé de l'organisation ne prenant jamais le dessus sur leur souci d'essayer de satisfaire au maximum nos demandes et nos particularités.

Élise Vareilles, ISAE SUPAERO

Comité de programme

Présidence

— Élise Vareilles, ISAE-SUPAERO, Toulouse, France.

Membres

— Quentin Cappart, École Polytechnique de Montréal, Canada ;
— Clément Carbonnel, LIRM Montpellier, France ;
— Mohamed Sami Cherif, Université Aix-Marseille, France ;
— Thi-Bich-Hanh Dao, Université d'Orléans, France ;
— Jean-Guillaume Fages, Cosling, France ;
— Hao Jin-Kao, Université d'Angers, France ;
— Jean-Marie Lagniez, CRIL Lens, France ;
— Arnaud Lallouet, Huawai Technologies, France ;
— Olivier Lhomme, IBM, France ;
— Chu-Min Li, Université de Picardie Jules Verne, France ;
— Giovanii Lo Bianco, Université de Toronton, Canada ;
— Xavier Lorca, IMT Mines Albi, France ;
— Ndiaye Samba Ndjojh, Liris, France ;
— Bertrand Neveu, LIGM Imagine École des Ponts ParisTech, France ;
— Abdelkader Ouali, Université de Caen, France ;
— Eric Piette, Maastricht Université ;
— Cédric Pralet, ONERA Toulouse, France ;
— Charles Prud'Homme, IMT Atlantique, France ;
— Nicolas Prcovic, Université Aix-Marseille, France ;
— Matthieu Py, Université Aix-Marseille, France ;
— Abdourahim Sylla, G-SCOP Grenoble, France ;
— Julien Vion, LAMIH, France ;
— Romain Wallon, CRIL Lens, France.

JFPC 1 - 3 juillet 2023, 10h30-12h

Sur l'énumération des motifs High Utility Fréquents : Une approche basée sur l'IA symbolique

Amel Hidouri[1], Said Jabbour[1], Badran Raddaoui[2]

[1] CRIL CNRS UMR 8188

[2] Télécom SudParis, Institut Polytechnique de Paris, SAMOVAR

Résumé

Le problème de la fouille des motifs high utility permet l'extraction des motifs qui ont les valeurs d'utilité (profit) les plus élevées en fonction d'un seuil d'utilité donné.
D'autre part, la satisfiabilité propositionnelle (SAT) est un modèle de représentation et de résolution pour représenter et résoudre différent problèmes en fouille de données. Dans ce travail [1], nous présentons un modèle basé sur SAT pour trouver efficacement les itemsets high utility fréquents avec un seuil d'utilité minimum appliqué à chaque transaction où l'itemset apparaît. Plus précisément, nous réduisons le problème de l'extraction d'itemsets high utility fréquents à celui de l'énumération des modèles de formules en logique propositionnelle.

Mots-clés

Fouille de données, Motifs High Utility, Satisfiabilité Propositionnelle

1 Introduction

L'objectif de la fouille des itemsets profitables (HUIM) est de trouver les itemsets ayant une utilité supérieure à un seuil. Le mesure d'utilité peut être évaluée en termes de profit, du coût, ou toute autre mesure de préférence définie par l'utilisateur. Cependant, la mesure de l'intérêt d'un itemset doit principalement refléter non seulement l'importance des items en termes de profit, mais aussi leur occurrence dans les données afin de prendre des décisions plus cruciales. Cependant, ces deux axes de recherche sont généralement considérés séparément. Plus précisemment, les deux cadres ont été conçus avec des objectifs différents, soit pour extraire les motifs qui se produisent fréquemment tout en ignorant leur profit, soit pour calculer motifs qui produisent les gains les plus élevés comme seul critère tout en évitant le mesure de fréquence. Dans ce dernier cas, les itemsets non fréquents pourraient être considérés comme des itemsets à haute utilité. Ces dernières années, on peut être plus intéressé à trouver en même temps des articles fréquemment achetés avec une valeur de gain élevée.
Dans la littérature, un certain nombre de méthodes de fouille de motifs fréquents à haute utilisé a été proposé. `FCHUIM` et `HU-FIMi` combinent la mesure d'utilité et celle de la fréquence pour trouver tels itemsets. Une autre forme spécialisée de ces motifs se concentre sur la mesure d'utilité appliquée localement. Par conséquent, les transactions impliquant l'itemset avec la valeur d'utilité la plus faible ne sont pas considérées comme des couvertures pour ce motif car aucune information pertinente n'est ajoutée à l'utilité globale de l'itemset dans la base de données. `2P-UF` et `FUFM` sont deux algorithmes permettant de trouver ces motifs.
Dans cet article, nous proposons une approche basée sur la logique propositionnelle pour l'extraction plus efficace d'itemsets fréquents à haute utilité à partir de bases de données de transactions dans le cas où la métrique de fréquence est appliquée aux transactions dans lesquelles l'itemset apparaît, et également lorsque cette métrique est limitée aux transactions avec l'utilité la plus élevée.

1. La version anglaise de cet article est publié dans la conférence (CP 2022 (27 :1-27 :17)), le premier auteur est un étudiant.

2 Préliminaires

Fouille de Motifs Étant donné l'ensemble $\Omega = \{a_1, a_2, \ldots, a_n\}$ de n items distincts. Une base données de transactions est un ensemble de transactions $= \{T_1, T_2, \ldots, T_m\}$ où chaque transactions $T_i \in D$ un ensemble d'items. Un motif X est un sous-ensemble non vide de Ω, i.e., $X \subseteq \Omega$. Le support d'un itemset X est défini par : $\texttt{Supp}(X, D) = |\{i \in [1..m] | T_i \in D \ and \ X \subseteq T_i\}|$. Chaque item dans une transaction est associé à une utilité *interne* $w_{int}(a, T_i)$ et une utilité *externe* $w_{ext}(a)$. Ainsi, l'utilité d'un item a dans une transaction $T_i \in D$ est $u(a, T_i) = w_{int}(a, T_i) \times w_{ext}(a)$. L'utilité d'un itemset X dans une transaction est $u(X, T_i) = \sum_{a \in X} u(a, T_i)$, l'utilité d'un itemset X dans D est $u(X) = \sum_{T_i \in D \mid X \subseteq T_i} u(X, T_i)$. Un itemset X est dit high utility (HUI) dans une base de données D s'il a une valeur d'utilité supérieure à un seuil d'utilité minimum donné θ.

2.1 Logique propositionnelle et SAT

Soit $\mathcal{L}$ un langage propositionnel construit inductivement à partir d'un ensemble fini `PS` de variables propositionnelles, les constantes booléennes $\top$ (*true* ou 1) et $\bot$ (*false* ou 0) ainsi que les connecteurs logiques classiques $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$. Nous utilisons les lettres x, y, z, etc. pour désigner les éléments de `PS`. Les formules propositionnelles de $\mathcal{L}$ sont notées Φ, Ψ, etc. Un littéral est une variable proposi-

tionnelle (x) de PS ou sa négation ($\neg x$). Une clause est une disjonction (finie) de littéraux. Pour toute formule propositionnelle Φ du langage $\mathcal{L}$, $\mathcal{P}(\Phi)$ dénote les symboles de PS apparaissant dans Φ.

3 Modélisation en SAT

Rappelons tout d'abord que la tâche HUIM traditionnelle a été récemment réduite à SAT [3] . Dans ce travail, nous étendons l'encodage proposé pour la fouille des motifs fréquents de haute utilité. Notre formulation est constituée d'un ensemble de contraintes comme résumé dans la Figure 1. Plus précisément, étant donnée une table de transactions, l'encodage proposé associe à chaque item a (resp. identifiant de transaction i), une variable propositionnelle dénommée p_a (resp. q_i). La première formule propositionnelle (1) représente la couverture de l'itemset candidat. Cette contrainte (1), appelée contrainte de couverture, permet d'exprimer qu'une transaction T_i ne contient pas l'itemset X, si un des items de X n'est pas dans T_i. La contrainte (2) permet de forcer les itemsets recherchés à être fermés. Cette contrainte est la même que l'encodage classique des itemets fréquents. Elle stipule que pour un item a, si toutes les transactions qui ne contiennent pas a n'appartiennent pas à la couverture de l'itemset recherché X, alors a doit être dans X. Finalement, la contrainte sur l'utilité d'un itemset X dans D est exprimée en utilisant l'inéquation linéaire (3). Cette contrainte exige que l'utilité de l'itemset recherché doit être supérieure ou égale à un seuil fixé θ. La contrainte 4 impose que l'itemset soit fréquent. Donc, la formule $\Phi^{\texttt{fhuim}}$ = (1) $\wedge$ (4) $\wedge$ (3) encode le problème de la fouille des motifs fréquents de haute utilité. $\Phi^{\texttt{fhuim}}$ peut être toujours étendu avec la contrainte 2 pour les itemsets clos.

La deuxième partie de cet article est consacrée à la fouille d'un autre type de motifs dit les motifs fréquents à haute utilité locale (FLHUI). Ainsi, un FLHUI est un motif fréquent et de haute utilité dans chaque transaction là ou il apparait. Contrairement à l'encodage précédent , nous introduisons un ensemble de trois variable propositionnelles $\{p_a,\ a \in \Omega\}$, $\{q_i,\ i \in [1..m]\}$, and $\{r_i, i \in [1..m]\}$. Ensuite, pour limiter la mesure de fréquence aux transactions avec la valeur d'utilité la plus élevée, nous considérons la nouvelle contrainte (5). Ensuite, pour contraindre l'itemset candidat à être fréquent, c'est-à-dire à être couvert par au moins δ transactions, nous ajoutons la contrainte de cardinalité (6)).

Afin d'énumérer tous les modèles de la formule Φ_{fhuim} et Φ_{flhuim}, nous proposons deux approches fondées sur DPLL légèrement modifié pour l'énumération de tous les modèles en employant une stratégie de décomposition afin d'améliorer l'efficacité. L'idée principale est d'éviter d'encoder toute la base de données en faveur de la résolution de nombreux sous-problèmes indépendants de petite taille plutôt qu'un seul gros problème. Formellement, étant donné l'ensemble des items $\Omega = \{a_1, \ldots, a_n\}$, l'ensemble de modèles de la formule Φ_{huim} peut être divisé en $\{E_1, \ldots, E_n\}$ avec E_1 est un sous-ensemble contenant l'item a_1, E_2 est

$$\bigwedge_{i=1}^{m}(\neg q_i \leftrightarrow \bigvee_{a\in\Omega\setminus T_i} p_a) \tag{1}$$

$$\bigwedge_{a\in\Omega}(p_a \vee \bigvee_{a\notin T_i} q_i) \tag{2}$$

$$\sum_{i=1}^{m}\sum_{a\in T_i} u(a,T_i)\times(p_a \wedge q_i) \geq \theta \tag{3}$$

$$\sum_{i=1}^{m} q_i \geq \delta \tag{4}$$

$$\bigwedge_{i=1}^{m}(r_i \leftrightarrow q_i \wedge (\sum_{a\in T_i} u(a,T_i)\, p_a \geq \theta')) \tag{5}$$

$$\sum_{i=1}^{m} r_i \geq \delta \tag{6}$$

FIGURE 1 – Encodage de HUIM en SAT

un sous-ensemble contenant l'item a_2 mais pas l'item a_1, E_n contient a_n mais pas les items $a_1, \ldots, a_{n-1}$. Plus généralement, E_i est obtenu en enumérant les modèles de la sous-formule Φ_i en propageant $p_{ai} = \top$ et $p_{ak} = \bot\ \forall\ 1 \leq k < i$. En d'autres termes, $\Phi_i = \Phi \wedge p_{a_i} \wedge \bigwedge_{1\leq k<i} \neg p_{a_k}$.

4 Résultats expérimentaux

Nous comparons la performance de nos algorithmes SATFHUIM et SATFLHUIM avec les algorithmes spécialisés HU−FIMi[1] et FUFM[2]. Les tests sont effectués sur des instances réelles prises de SPMF. Les expérimentations montrent que nos méthodes sont très compétitives avec l'état de l'art et passent à l'échelle sur des grandes instances. Notons que les performances de tous les algorithmes dépendent des caractéristiques de l'instance ainsi que les valeurs du seuil d'utilité et de la fréquence.

Références

[1] R Uday Kiran, T Yashwanth Reddy, Philippe Fournier-Viger, Masashi Toyoda,545 P Krishna Reddy, and Masaru Kitsuregawa. Efficiently finding high utility-frequent itemsets using cutoff and suffix utility. In Pacific-Asia Conference on Knowledge Discovery and Data Mining, pages 191–203. Springer, 2019

[2] Vid Podpecan, Nada Lavrac, and Igor Kononenko. A fast algorithm for mining utility-frequent itemsets. Constraint-Based Mining and Learning, page 9, 2007.

[3] Amel Hidouri, Said Jabbour, Badran Raddaoui, Boutheina Ben Yaghlane, Mining Closed High Utility Itemsets based on Propositional Satisfiability, Data & Knowledge Engineering, Volume 136, 2021.

Exploiter l'Entropie pour la Programmation Par Contraintes

A. Burlats[1], G. Pesant[2]

[1] UCLouvain, ICTEAM, Belgique, auguste.burlats@uclouvain.be
[2] Polytechnique Montréal, Canada, gilles.pesant@polymtl.ca

Résumé

L'introduction de la propagation de croyances dans la programmation par contraintes permet d'estimer la probabilité qu'une paire variable-valeur appartienne à une solution. Ces probabilités marginales nous permettent non seulement de développer des heuristiques de branchement mais aussi d'appliquer le concept d'entropie à la programmation par contraintes. Nous explorons comment les entropies des variables et du problème améliorent la recherche d'une solution à un problème combinatoire. Nous évaluons notre proposition sur un ensemble varié de problèmes.

Mots-clés

Programmation par contraintes, entropie, heuristique de branchement.

1 Introduction

La Programmation Par Contraintes (PPC) est une approche efficace pour la résolution de problèmes combinatoires. Elle permet en effet une forte réduction de l'espace de recherche en exploitant les contraintes du problème et les algorithmes d'inférence liés à celles-ci pour filtrer les paires variables-valeurs enfreignant les contraintes. Mais l'amplitude de cette réduction est fortement liée à l'ordre dans lequel sont fixées les variables du problème. Des heuristiques robustes et généralisables pour ordonner les variables sont donc cruciales. L'introduction de la Propagation De Croyances (PDC) en PPC permet d'estimer la probabilité qu'une paire variable-valeur fasse partie d'une solution du problème [5]. Ces probabilités marginales permettent de développer des heuristiques d'ordonnancement de variable [1] mais aussi d'appliquer le concept d'entropie à la PPC comme nous allons le développer dans cet article [1].

2 Adapter l'entropie à la PPC

2.1 Distributions marginales

Nous considérons $P = \langle X, D, C\rangle$, un problème de satisfaction de contraintes (PSC). Ce problème est défini par $X = \{x_1, x_2, \ldots, x_n\}$ est un ensemble fini de variables, $D = \{D(x_1), D(x_2), \ldots, D(x_n)\}$ est un ensemble fini de domaines et $C = \{c_1, c_2, \ldots, c_m\}$ est un ensemble fini de contraintes.

Une solution $\mathbf{s} = (v_1, v_2, \ldots, v_n)$ à P va affecter à chaque variable $x_i \in X$ une valeur v_i appartenant à son domaine $D(x_i)$ de manière à ce que toutes les contraintes de C soient respectées. Notons S^P l'ensemble de toutes les solutions de P et notons $\mathbf{s}[x]$ la valeur affectée à x dans la solution $\mathbf{s}$. On définit alors

$$\theta_x^P(v) = \frac{|\{\mathbf{s} \in S^P : \mathbf{s}[x] = v\}|}{|S^P|}$$

comme étant la proportion de solutions où x prend la valeur v [2]. Cette quantité est appelée la *marginale* de la paire variable-valeur (x, v), en référence à la probabilité marginale que x prenne la valeur v dans une solution choisie uniformément au hasard dans S. Ici nous supposons que S n'est pas vide, i.e. que P est *satisfiable*, dans le cas contraire toutes les marginales seront nulles.

2.2 Entropie

Les distributions marginales des variables permettent de quantifier l'incertitude que l'on a sur la valeur qu'une variable x devrait prendre à travers la notion d'*entropie*. Nous définissons cette *entropie* $H(x)$ à partir l'entropie de Shannon [6] :

$$H(x) = -\sum_{v \in D(x)} \theta_x(v) \log(\theta_x(v)).$$

Cette quantité non négative reflète l'incertitude sur la valeur que devrait prendre la variable x : une entropie nulle signifie que $\theta_x(v) = 1$ pour une valeur v et donc que $\theta_x(v') = 0\ \forall v' \neq v$, i.e. que $x = v$ dans toutes les solutions ; à l'inverse, l'entropie maximale $\log(|D(x)|)$ est atteinte lorsque $\theta_x(v) = \frac{1}{|D(x)|}\ \forall v \in D(x)$ i.e. que la distribution marginale est uniforme. L'entropie normalisée (aussi appelée efficacité) de x divise son entropie par le logarithme de la cardinalité de son domaine, hormis dans le cas où son domaine ne possède qu'une valeur : son entropie (normalisée) est dans ce cas nulle. Nous définissons l'entropie du problème P comme

$$H(P) = \frac{\sum_{x \in X\ :\ |D(x)|>1} \frac{H(x)}{\log(|D(x)|)}}{|X|}.$$

Cette définition correspond à la moyenne des entropies normalisées et est donc comprise entre 0 et 1 inclus. Bien sûr notre définition de l'entropie repose sur des marginales dont nous ne connaissons pas la valeur exacte. La PDC nous permet de calculer une estimation de ces marginales [5].

1. Cet article est un résumé de celui publié dans les actes de la conférence CPAIOR 2023 [3].

2. Pour simplifier les notations, l'exposant P ne sera généralement pas noté.

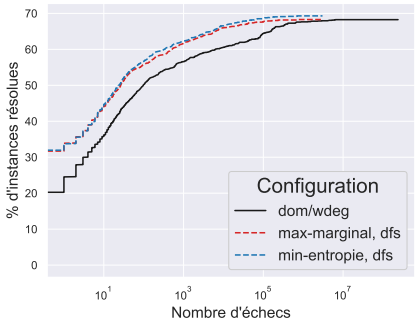


FIGURE 1 – % d'exemplaires résolus contre le nombre d'échecs pour trois heuristiques de branchement.

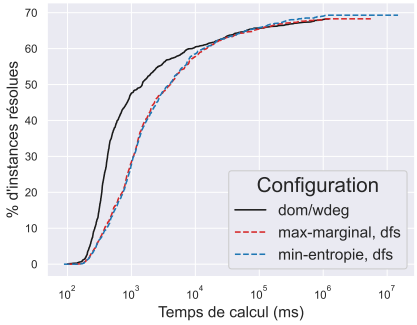


FIGURE 2 – % d'exemplaires résolus contre le temps de calcul pour trois heuristiques de branchement.

3 Exploiter l'entropie

L'introduction des notions d'entropies des variables et du problème nous permettent d'explorer plusieurs approches pour accélérer la recherche d'une solution ou pour optimiser l'usage de la PDC [3]. En particulier ici l'entropie des variables est un support puissant pour faire de meilleures décisions de branchement : plus faible est l'entropie d'une variable, plus faible est l'incertitude concernant la valeur que cette variable devrait prendre. Il est donc judicieux de la fixer en priorité. Nous proposons l'heuristique d'ordonnancement de variable *min-entropie* qui sélectionne la variable présentant la plus faible entropie et lui affecte en priorité la valeur de son domaine avec la marginale la plus élevée. Il faut noter que, dans le cas où les distributions marginales sont uniformes, la variable avec la plus faible entropie est celle avec le plus petit domaine. Ainsi, *min-entropie* peut être vue comme une généralisation de l'heuristique standard *plus petit domaine*.

4 Évaluation

Dans cette section nous cherchons à évaluer la qualité de notre stratégie de recherche, *min-entropie*. Pour la situer dans l'état de l'art, nous la comparons avec l'heuristique *dom/wdeg* [2] ainsi qu'avec une autre heuristique exploitant les distributions marginales, *max-marginal* [1] qui choisit de fixer la variable présentant la marginale la plus élevée. Pour chacune des heuristiques une recherche en profondeur a été exécutée pour trouver une solution et dans le cas de *dom/wdeg* des relances ont été exécutées au cours de la recherche afin d'en améliorer les performances (la première relance se fait après 50 échecs, et cette valeur est multipliée par 2 à chaque relance).

Les heuristiques ont été évaluées sur un jeu 1407 exemplaires de XCSP3 [3] et du concours Minizinc [4]. Elles ont été exécutées sur un serveur avec deux Intel E5-2683 v4 Broadwell @ 2.1GHz. Le solveur utilisé est MiniCPBP [5], un solveur basé sur MiniCP [4] et appliquant la PDC. Chaque test disposait de 12 Go de mémoire et le temps limite a été fixé à 20 minutes.

La figure 1 compare les performances des trois heuristiques. Les résultats sont présentés sous la forme de profils de performance : chaque point d'un graphe montre la proportion d'exemplaires (en ordonnée) résolus avec un nombre d'échecs inférieur ou égal à la valeur en abscisse. La métrique utilisée est le nombre d'échecs rencontrés au cours de la recherche d'une solution. Plus ce nombre est faible plus l'heuristique est une stratégie intéressante. On peut observer que les heuristiques exploitant les distributions marginales (*min-entropie* et *max-marginal*) montrent de meilleures performances car leurs courbes croissent plus vite que celle de *dom-wdeg*. *Min-entropie* semble être légèrement meilleure que *max-marginal*. Mais ces performances sont à relativiser : la PDC implique un surcoût conséquent sur le temps de calcul, comme cela peut être vu sur la figure 2, montrant des profils de performances en utilisant le temps de calcul comme métrique. On peut y observer que sur la majorité des problèmes, *dom/wdeg* égalise ou surpasse *min-entropie* lorsque l'on mesure le temps de calcul. Malgré cela *min-entropie* est capable de résoudre plus d'exemplaires que *dom/wdeg*. Mais l'usage de la PDC n'est pas optimisé et de nombreuses pistes peuvent être explorées pour réduire son coût. Certaines, non évoquées dans cet article, exploitent l'entropie du problème $H(P)$ pour réduire le nombre d'itérations de la PDC et pour améliorer la précision des estimations des distributions marginales.

5 Conclusion

Nous avons étudié l'entropie des PSC et de leurs variables à domaine fini, rendue possible par l'estimation des distributions marginales grâce à l'application de la combinaison de la PPC et de la PDC. Nos expériences sur 1407 exemplaires de 14 problèmes différents ont montré qu'exploiter l'entropie en choisissant de fixer la variable avec la plus faible entropie à chaque étape de la recherche est une heuristique d'ordonnancement de variable intéressante. Mais le coût de la PDC est important et il faut donc optimiser cette dernière pour que cette heuristique soit intéressante pour une large variété de problèmes.

3. http://www.xcsp.org/instances/
4. https://www.minizinc.org/challenge2022/mznc2022_probs.tar.gz
5. https://github.com/PesantGilles/MiniCPBP

Références

[1] Behrouz Babaki, Bilel Omrani, and Gilles Pesant. Combinatorial Search in CP-Based Iterated Belief Propagation. In Helmut Simonis, editor, *Principles and Practice of Constraint Programming*, pages 21–36, Cham, 2020. Springer International Publishing.

[2] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In Ramón López de Mántaras and Lorenza Saitta, editors, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, pages 146–150. IOS Press, 2004.

[3] Auguste Burlats and Gilles Pesant. Exploiting Entropy in Constraint Programming. In *Proc. CPAIOR*, Lecture Notes in Computer Science. Springer, 2023.

[4] L. Michel, P. Schaus, and P. Van Hentenryck. MiniCP : a Lightweight Solver for Constraint Programming. *Mathematical Programming Computation*, 13(1) :133–184, 2021.

[5] Gilles Pesant. From Support Propagation to Belief Propagation in Constraint Programming. *Journal of Artificial Intelligence Research*, 66 :123–150, 2019.

[6] Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27(3) :379–423, July 1948.

Approximer pour mieux résoudre

T. Falque[1,2], JM. Lagniez[2], C. Lecoutre[2], R. Wallon[2]

[1] Exakis Nelite
[2] CRIL, Univ Artois & CNRS

{falque,lagniez,lecoutre,wallon}@cril.fr

Résumé

Cet article propose des méthodes de sous-approximation pour améliorer la résolution de problèmes de satisfaction de contraintes (CSP) et d'optimisation sous contraintes (COP). Plus précisément, nous proposons d'utiliser différentes mesures pour sélectionner des contraintes à supprimer, et ainsi réduire le problème jusqu'à ce que celui-ci devienne « facile ». La solution trouvée pour le sous-problème est ensuite utilisée pour guider la recherche sur le problème original. Nous évaluons les performances de notre méthode sur un ensemble de problèmes issus de la dernière compétition XCSP3, et comparons les résultats avec des méthodes classiques de résolution. Les résultats obtenus montrent que notre approche améliore les performances du solveur, permettant même de trouver des solutions à des problèmes qui ne sont pas résolus autrement.

Mots-clés

programmation par contraintes, optimisation sous contraintes, recherche heuristique, sous-approximation

Abstract

In this paper, we introduce sub-approximation methods to improve the solving of constraint satisfaction (CSP) and contraint optimization (COP) problems. In particular, we propose to use different measures to select constraints to remove, so as to reduce the problem until it becomes "easy". The solution that is found for the sub-problem is then used to guide the search on the original problem. We evaluate the performance of our method on a set of problems from the latest XCSP3 competition, and compare the results with classical solving approaches. The obtained results show that our approach improves the performance of the solver, even allowing to find solutions for problems that are not otherwise solved.

Keywords

constraint programming, constraint optimization, heuristic search, subapproximation

1 Introduction

La programmation par contraintes (PPC) est un puissant paradigme permettant de résoudre des problèmes de satisfaction ou d'optimisation combinatoires. Il permet d'exprimer des contraintes et fonctions objectifs complexes, faisant de ce paradigme un outil pouvant être utilisé dans de nombreuses applications, telles que l'ordonnancement, la planification ou encore l'allocation de ressources [17]. De nombreuses techniques ont été développées pour résoudre ce type de problèmes [14], et ont été implantées dans différents solveurs [22, 19, 15]. Bien que ces solveurs soient souvent très efficaces en pratique, la résolution de problèmes sous contraintes est NP-difficile : il n'est donc pas toujours possible de trouver des solutions exactes en un temps raisonnable.

Partant de cette observation, des méthodes incomplètes ont été proposées, comme par exemple celles utilisant la recherche locale [11]. Cependant, ces techniques ne sont pas capables de démontrer l'incohérence d'un problème donné en entrée, ni l'optimalité des solutions qu'elles peuvent trouver. Pour combler ce manque, des approches hybrides, combinant des méthodes complètes et incomplètes, ont été proposées (voir par exemple [13]). Dans cet article, nous proposons également d'exploiter des méthodes incomplètes pour guider un solveur complet (plus précisément, ACE), afin de résoudre plus efficacement des problèmes d'optimisation. Notons que notre technique peut aisément être appliquée à des problèmes de satisfaction, mais nous ne traitons pas ce cas ici.

Plus précisément, notre approche consiste à exécuter le solveur sur un problème donné pour un nombre limité de *runs*, afin de détecter si le problème est difficile. Si aucune solution n'est trouvée entre temps, le problème est réduit en retirant certaines de ses contraintes. Plusieurs réductions sont proposées, pour retirer soit une contrainte à la fois, soit un *groupe* complet de contraintes, comme défini par exemple dans le format XCSP3 [4]. Nous introduisons également différentes mesures pour sélectionner dynamiquement quelles contraintes retirer, soit en utilisant les propagations et conflits qu'elles génèrent, soit en considérant leur score wdeg. La réduction est appliquée jusqu'à ce

que le sous-problème devienne « suffisamment facile » pour en trouver rapidement une solution. Cette solution est ensuite utilisée comme heuristique de sélection de choix de valeurs pour guider la recherche afin de trouver une solution du problème original.

Nous avons implanté l'approche proposée dans le solveur `ACE`, et nous l'avons évalué empiriquement sur des instances de la dernière compétition `XCSP3`. Nos expérimentations montrent que l'approche proposée peut être bénéfique, même pour des instances difficiles pour lesquelles le solveur ne parvient initialement pas à trouver de solutions pendant la limite de temps fixée.

Le reste de cet article est organisé de la manière suivante. La section 2 introduit les techniques de résolution de la PPC. La section 3 propose un rapide tour d'horizon des travaux liés aux méthodes incomplètes pour la PPC. La section 4 décrit nos approches en détails. La section 5 présente nos résultats expérimentaux, et la section 6 nous permet de conclure et d'envisager différentes perspectives d'étude.

2 Préliminaires techniques

Un *réseau de contraintes* (CN, pour *Contraint Network*) est un ensemble fini de variables et de contraintes. Chaque variable x peut prendre une valeur dans un ensemble fini appelé *domaine* de x, et noté $\texttt{dom}(x)$. Chaque contrainte c est définie par une relation sur un ensemble de variables appelé *portée* (ou *scope*) de c, et noté $\texttt{scp}(c)$. Une *solution* d'un CN est une affectation de valeurs à toutes ses variables satisfaisant chacune de ses contraintes. Un CN est dit *cohérent* (ou *satisfiable*) s'il admet au moins un solution, et le *problème de satisfaction de contraintes* (CSP) associé consiste à déterminer si un CN possède une solution.

Un *réseau de contraintes sous optimisation* (CNO) est un réseau de contraintes associé à une fonction objectif `obj` qui associe toute solution du CN à une valeur dans $\mathbb{Z}$. Sans perte de généralité, nous pouvons considérer que `obj` doit être minimisée. Une solution S d'un CNO est une solution du CN associé. S est dite *optimale* s'il n'existe aucune solution S' telle que $\texttt{obj}(S') < \texttt{obj}(S)$. Étant donné un *problème d'optimisation sous contraintes* (COP), la tâche associée est de trouver une solution optimale du CNO associé. Notons que CN et CNO sont souvent appelés *instances* CSP et COP, respectivement.

La recherche par retours-arrières est une procédure complète classique pour la résolution d'instances CSP et COP. Elle alterne des affectations (et réfutations) de variables ainsi qu'un mécanisme de propagation de contraintes pour filtrer l'espace de recherche. Une approche possible pour explorer cet espace est le *branchement binaire*. Typiquement, un arbre binaire de recherche $\mathcal{T}$ est construit : à chaque nœud interne de $\mathcal{T}$, (i) une paire (x, v) est sélectionnée, où x est une variable fixée et v est une valeur dans $\texttt{dom}(x)$, et (ii) deux cas (branches) sont considérés, correspondant à l'affectation $x = v$ et à la réfutation $x \neq v$. Une autre appoche consiste à essayer successivement toutes les valeurs possibles pour la variable x, et donc de considérer d branches à partir du nœud x, où d représente la taille de $\texttt{dom}(x)$ (cette approche est appelée *d-way branching*). Après chaque affectation, la propagation de contraintes est appliquée pour éliminer des valeurs du domaine des variables, comme dans MAC [18] qui applique des propagations de manière à maintenir l'arc-cohérence. L'ordre dans lequel les variables sont choisies durant le parcours en profondeur d'abord de l'espace de recherche est déterminé par une *heuristique de choix de variables*, comme par exemple `dom/wdeg` [3], une heuristique générique classique. De plus, une *heuristique de choix de valeurs* détermine l'ordre dans lequel les valeurs à affecter sont choisies. Dans les problèmes de type COP, une approche fréquemment utilisée consiste à affecter en priorité les valeurs obtenues dans la dernière solution trouvée. Cette technique est appelée *solution saving* [23, 6].

La recherche par retours-arrières pour les problèmes COP est fondée sur la résolution de CSP. Elle consiste à ajouter une *contrainte d'objectif* $\texttt{obj} < \infty$ au réseau de contraintes (même si elle est initialement trivialement satisfaite), et de mettre à jour sa limite chaque fois qu'une nouvelle solution est trouvée. Cela signifie qu'à chaque nouvelle solution S, son coût $B = \texttt{obj}(S)$ est calculé, et la contrainte d'objectif devient $\texttt{obj} < B$. Une séquence de solutions est alors générée, chacune étant meilleure que les précédentes (la cohérence est prouvée en tenant compte de la contrainte d'objectif) jusqu'à ce qu'il n'en existe plus (l'incohérence est finalement prouvée avec la borne de la dernière solution), garantissant ainsi l'optimalité de la dernière solution.

Les politiques de redémarrage jouent un rôle essentiel dans les solveurs de contraintes modernes, comme elles permettent de contrer la distribution à longue traînée des temps d'exécution des solveurs SAT, CSP et COP [8]. Une politique de redémarrage correspond typiquement à une fonction $\texttt{restart} : \mathbb{N}^+ \rightarrow \mathbb{N}^+$ qui indique le nombre maximal d'étapes autorisées pour une tentative donnée d'exécution de l'algorithme de recherche, appelée *run*. Cela signifie qu'une recherche avec retours-arrières pilotée par une politique de redémarrage construit une séquence d'arbres binaires de recherche $\langle \mathcal{T}_1, \mathcal{T}_2, \ldots \rangle$, où $\mathcal{T}_j$ est l'arbre de recherche exploré au *run* j. Notons que le *cutoff*, à savoir le nombre maximal d'étapes autorisées au cours d'un *run*, peut correspondre au nombre de retours-arrières, au nombre de mauvaises décisions [2], où à toute autre mesure pertinente. Le *cutoff* d'une stratégie de redémarrage peut être fixe, auquel cas $\texttt{restart}(j)$ prend la même valeur quelque soit le *run* j, ou dynamique, auquel cas `restart` augmente régulièrement la valeur du *cutoff* (par exemple, géométriquement [24]), garantissant ainsi l'exploration de la totalité de l'espace des solutions partielles.

3 Travaux connexes

L'approche CEGAR (*Counter-Example-Guided Abstraction Refinement*) est une approche incrémentale pour la résolution de problèmes de décision et d'optimisation combinatoires. Elle a été initialement développée pour la vérification formelle, et a été largement utilisée pour d'autres applications telles que, entre autres, les formules booléennes quantifiées (QBF) [12], les réseaux de contraintes qualitatifs (QCN) [7], le problème de formation d'équipes (PTF) [20] ou encore le problème des cycles hamiltoniens [21]. La pierre angulaire de CEGAR consiste à remplacer le problème à résoudre par une abstraction supposément plus facile à résoudre. Il y a généralement deux types d'abstractions : sur-abstraction et sous-abstraction. Les sur-abstractions sont des représentations plus contraintes que le problème que l'on souhaite résoudre. Par conséquent, si l'on prouve que l'abstraction est cohérente, alors le problème lui-même est cohérent. Sinon, l'abstraction est affinée jusqu'à ce que l'on prouve sa cohérence, ou qu'elle devienne équisatisfiable au problème original. Dans ce dernier cas, si l'abstraction est incohérente, alors le problème initial l'est aussi. Les sous-abstractions fonctionnent de manière similaire, hormis le fait qu'elles démarrent avec un problème moins contraint, ce qui implique que le processus s'arrête dès que l'on prouve que le problème est incohérent ou que l'abstraction est équisatisfiable au problème initial.

Contrairement à CEGAR, notre approche démarre avec le problème original, et non une abstraction de celui-ci. Ainsi, ce n'est clairement pas une approche CEGAR par sous-abstraction, puisque le problème le plus contraint que nous considérons est le problème lui-même, et lorsque le problème devient cohérent, il le devient sur un problème moins contraint. Ce n'est pas non plus une approche CEGAR par sur-abstraction, puisque les étapes de raffinement consistent à affaiblir l'approximation.

Dans cet esprit, la technique que nous proposons est plus proche des travaux présentés dans [9, 1]. Dans [9], les auteurs proposent une combinaison synergique entre un solveur CSP complet et une procédure de recherche locale stochastique (SLS) permettant de résoudre des CSP. Plus précisément, cette procédure est guidée vers les sous-parties les plus difficiles du CSP, ce qui permet d'obtenir des informations cruciales pour guider la recherche complète. Les informations collectées sont les variables FAC (*Falsified in All Constraints*), qui sont les variables apparaissant dans toutes les contraintes falsifiées par une certaine interprétation, et donc dans au moins une contrainte de chaque noyau minimal (c'est-à-dire, un sous-ensemble de contraintes incohérent). Ces variables sont ensuite choisies en priorité par l'heuristique de branchement. De plus, la meilleure interprétation trouvée jusque là par la SLS est utilisée comme heuristique de choix de valeurs. L'approche proposée dans [1] fonctionne de manière similaire, tout en ayant été adaptée pour les solveurs SAT.

Le principal inconvénient des approches proposées dans [9, 1] est que les performances des solveurs SLS sont généralement mauvaises sur des instances structurées, rendant ainsi ces approches peu efficaces sur ce type de problèmes. Notre approche évite cet écueil en utilisant un solveur complet pour trouver une solution du problème.

Enfin, nos travaux sont également liés à ceux présentés dans [10]. Les auteurs de cet article proposent de calculer des sous-ensembles maximaux cohérent (MSS) ainsi que des sous-formules minimales incohérentes (MUS) en utilisant une procédure de type SLS. Pour cela, ils exploitent cette procédure pour calculer un score pour les clauses qui sont souvent falsifiées durant l'exploration de l'espace de recherche. Ces scores sont ensuite utilisés pour calculer un MUS, en retirant les clauses ayant les scores les plus faibles, ou pour calculer un MSS, en retirant les clauses ayant les plus grands scores.

D'une certaine manière, notre approche essaye également de calculer un MSS en retirant incrémentalement certaines contraintes. Cependant, comme le but est différent, le processus ne se termine pas lorsqu'une sous partie cohérente de la formule a été trouvée, mais lorsque la cohérence ou l'incohérence du problème original a été prouvée. De plus, la manière dont nous sélectionnons les contraintes à supprimer est totalement différente.

4 Techniques d'approximation

Cette section décrit les méthodes d'approximation que nous proposons pour résoudre des problèmes de satisfaction (CSP) et d'optimisation (COP) sous contraintes. Notre approche est fondée sur une combinaison de la recherche par retours-arrières (voir la section 2) et de techniques de relaxation.

Par relaxation, nous faisons ici référence à un processus ignorant temporairement certaines contraintes afin de simplifier le problème. Si une solution d'un sous-problème est obtenue, nous restaurons les contraintes ignorées, avant d'utiliser cette solution pour guider la recherche sur le problème précédent. Sinon, nous ignorons des contraintes supplémentaires, et répétons le processus jusqu'à ce que le sous-problème devienne « suffisamment facile » pour pouvoir facilement en trouver une solution. Ces étapes successives d'approximation-restauration sont gérées par une stratégie d'approximation s. Nous notons $P = p_0$ le problème original à résoudre, et p_i le i-ème sous-problème. Cette stratégie est associée à une politique de suppression de contraintes r ainsi qu'à une mesure m utilisée pour sélectionner l'ensemble des contraintes à retirer, noté δ (qui peut être un singleton). Nous notons s_r^m la stratégie s utilisant la politique de suppression r et la mesure m.

4.1 Boucle d'approximation

Indépendamment du choix des stratégies, l'algorithme 1 décrit la boucle principal de notre approche.

Algorithme 1 : Boucle principale d'approximation.
entrée : Un solver ayant préalablement chargé une instance CSP/COP.
sortie : La réponse trouvée par le solveur.

```
1  state ← NORMAL
2  result ← state.solve(solver)
3  while result = UNKNOWN do
4  |  state ← state.nextState()
5  |  result ← state.solve(solver)
6  |  while (result = SAT) & (state ≠ NORMAL) do
7  |  |  state ← state.previousState()
8  |  |  solution ← solver.solution()
9  |  |  result ← state.solve(solver, solution)
10 |  end
11 end
12 return result
```

Au départ (lignes 1-2), le solveur essaie de résoudre le problème original pendant un nombre limité de *runs* (dans notre cas, 50 *runs*). Comme illustré dans l'algorithme 2, l'appel à la méthode `solve` délègue simplement la résolution au solveur. Si une solution est trouvée au cours de cette exécution, l'exécution du solveur se poursuit normalement, et aucune approximation n'est réalisée.

Algorithme 2 : Méthode `solve` pour l'état NORMAL.
entrée : Un solver ayant préalablement chargé une instance CSP/COP.
sortie : La réponse éventuellement trouvée par le solveur.

```
1 updateRestartLimits(solver)
2 result ← solver.solve()
3 return result
```

Sinon, le solveur retourne la valeur UNKNOWN, et le solveur est ensuite exécuté sur des sous-problèmes de l'instance originale. Plus précisément, le solveur passe dans l'état APPROXIMATION (lignes 4-5). Dans cet état, l'ensemble des contraintes à supprimer est sélectionné, et ces contraintes sont ignorées au moment d'invoquer la méthode `solve` (voir l'algorithme 3). Cette fois encore, le nombre de *runs* est limité (mais il est augmenté à chaque itération dans `updateRestartLimits`), de sorte que le résultat peut rester UNKNOWN [1]. Dans ce cas, la boucle à la ligne 3 se poursuit, et une nouvelle approximation est calculée, jusqu'à l'obtention d'une solution.

Algorithme 3 : Méthode `solve` pour l'état APPROX.
entrée : Un solver ayant préalablement chargé une instance CSP/COP.
sortie : La réponse éventuellement trouvée par le solveur.

```
1 constraints ← nextConstraintsToRemove(solver)
2 foreach c ∈ constraints do
3 |  ignore(c)
4 end
5 updateRestartLimits(solver)
6 result ← solver.solve(solLimit = 1)
7 return result
```

Lorsque qu'une solution est trouvée, la deuxième boucle commence (ligne 6), dans laquelle les problèmes précédents sont résolus dans l'ordre inverse (ligne 7). Afin de les résoudre, la dernière solution S ayant été trouvée pour un sous-problème est utilisée comme heuristique de choix de valeurs, comme dans le cas du *solution saving* (lignes 8-9). Plus précisément, la première valeur choisie au moment d'affecter une variable est celle affectée à cette variable dans S, à moins qu'elle ne soit plus dans le domaine de la variable (dans ce cas, l'heuristique `first` est appliquée). Le processus se termine lorsqu'une solution du problème original est trouvée (ce problème est atteint lorsque l'état redevient NORMAL).

4.2 Stratégies d'approximation

Décrivons maintenant en détails comment notre approche passe d'une sous-approximation à une autre. Dans cet article, nous proposons deux variantes : `PROGRESSIVEAPPROX` and `AGGRESSIVEAPPROX`. La stratégie choisie modifie le comportement de `previousState` (ligne 7 de l'algorithme 1) de la manière suivante.
Dans `PROGRESSIVEAPPROX`, l'idée principale consiste à générer un problème p_i en supprimant un ensemble de contraintes δ_i jusqu'à obtenir une solution S_i de p_i. La solution est ensuite utilisée pour résoudre le problème p_{i-1}. Si une solution est trouvée pour p_{i-1}, nous utilisonscette solution pour résoudre p_{i-2}, et ainsi de suite. Si aucune solution n'est trouvée durant le nombre de *runs* limité sur p_{i-1}, nous calculons un nouvel ensemble de contraintes à supprimer et répétons le processus. Un exemple de cette approche est illustré à la figure 1. La première étape consiste à résoudre le problème original p_0 en un nombre fixé de *runs*. Puis, à l'étape 2, un ensemble de contraintes δ_1 est supprimé pour obtenir le problème p_1, qui est ensuite résolu (étape 3). Si aucune solution n'est trouvée, le processus continue et un deuxième ensemble de contraintes δ_2 est supprimé pour obtenir le problème p_2 (étape 4). Le problème p_2 est ensuite résolu (étape 5), et la solution est ensuite

1. Quand il n'est plus possible de retirer plus de contraintes, par exemple s'il n'en reste plus qu'une, le nombre de *runs* devient illimité.

utilisée pour résoudre le problème p_1 après avoir restauré les contraintes de δ_2 (étape 6). Enfin, l'ensemble de contraintes δ_1 est restauré (étape 7), et la solution obtenue pour le problème p_1 est utilisée pour résoudre le problème original.

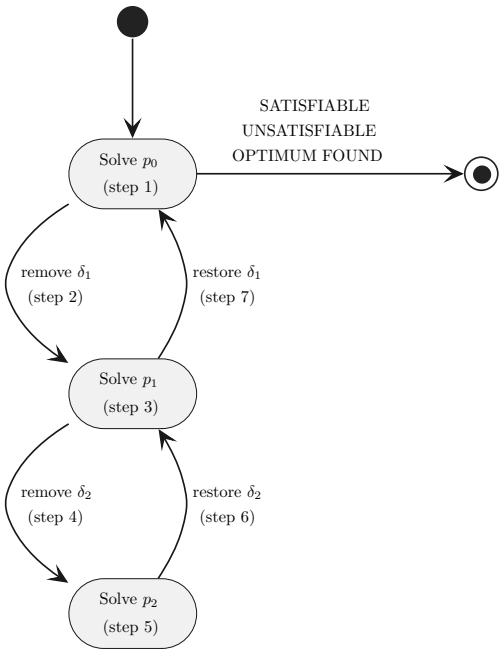


FIGURE 1 – Un exemple de la stratégie `PROGRESSIVEAPPROX`.

Pour la stratégie `AGGRESSIVEAPPROX`, nous suivons les mêmes étapes que dans la stratégie `PROGRESSIVEAPPROX`, mais le problème original est cette fois-ci résolu à chaque fois qu'une solution de l'un des sous-problèmes est obtenue. Une exemple de cette stratégie est illustré à la figure 2. Observons que, lorsque la solution du sous-problème p_2 est obtenue à l'étape 5, toutes les contraintes (à savoir, celles de δ_1 et de δ_2) sont restaurées, et la solution obtenue pour p_2 est utilisée pour résoudre le problème original.

4.3 Politiques de suppression et mesures

Dans cet article, nous proposons deux stratégies de suppression principales. La première consiste à retirer une contrainte à la fois, tandis que la seconde exploite la capacité du format `XCSP3` à préserver la structure originale de problème, en regroupant les contraintes en groupes ou en blocs. En particulier, lorsque les contraintes sont regroupées en blocs, elles partagent une relation sémantique, alors que, lorsqu'elles sont reliées en groupes, elles partagent une relation syntaxique.

Afin de sélectionner quelles contraintes devraient être supprimées, nous utilisons différentes mesures dynamiques qui évoluent durant la recherche afin de sélectionner quel ensemble de contraintes devrait être le prochain à être supprimé. La première mesure que nous

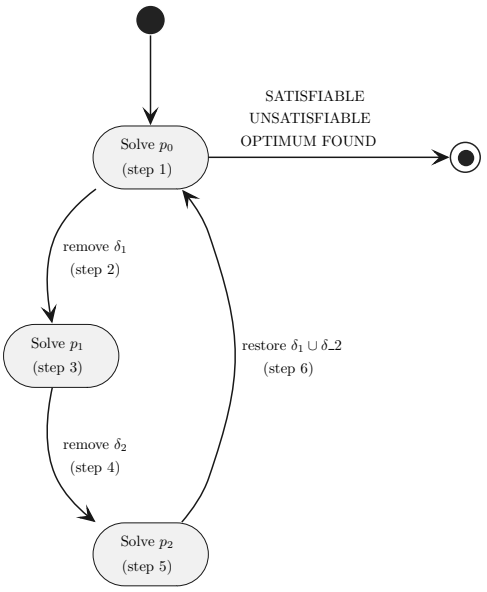


FIGURE 2 – Un exemple de la stratégie `AGGRESSIVEAPPROX`.

proposons, notée `neff`, est le nombre de fois qu'une contrainte a eu un effet lors du filtrage : cette valeur est incrémentée après l'appel au propagateur, s'il a provoqué un conflit ou une réduction du domaine de la variable. De manière similaire, nous proposons la mesure `nback`, dans laquelle nous comptons seulement le nombre de fois que la contrainte a provoqué un conflit, et donc un retour-arrière (intuitivement, réaliser de nombreux retours-arrières peut ralentir la recherche, comme ils ne permettent pas toujours d'explorer de nouvelles parties de l'espace de recherche). Une autre mesure possible est de considérer l'heuristique classique `wdeg`, qui choisit en priorité la variable ayant le plus grand « degré pondéré ». Cette heuristique maintient un score pour chaque contrainte que nous proposons d'utiliser comme mesure.

Dans tous les cas, nous considérons l'ensemble de contraintes qui maximise la mesure choisie. Pour une contrainte c, nous notons cette mesure $\nu(c)$. Pour un ensemble de contraintes δ, nous définissons la mesure $N(\delta) = \sum_{c \in \delta} \nu(c)$ et $\bar{N}(\delta) = \frac{1}{|\delta|} \sum_{c \in \delta} \nu(c)$ (autrement dit, $\bar{N}(\delta)$ représente la moyenne des mesures).

5 Expérimentations

Cette section présente nos résultats expérimentaux obtenus pour les différentes stratégies d'approximation sur différents ensembles d'instances. Pour évaluer les performances de notre approche, nous avons implanté les stratégies dans un projet spécifique utilisant le solveur `ACE` [2] – le nouveau nom d'`AbsCon` – en guise d'oracle. En particulier, nous avons utilisé l'adaptateur

2. `https://github.com/xcsp3team/ace`

Universe[3] de ACE : aceurancetourix[4]. Nous avons comparé notre implantation avec la version originale de ACE. Par souci d'espace, nous présentons dans cette section uniquement les 4 meilleures combinaisons de stratégies. Une analyse complète, réalisée avec Metrics[5] est disponible à l'adresse https://gitlab.com/productions-tfalque/articles/approximation/experiments/-/tree/jfpc2023.
Tous les solveurs ont été exécutés sur un cluster de machines équipées de 128 Go de RAM et de deux processeurs quadricœur Intel XEON E5-2637 (3.5 GHz). Conformément à la configuration imposée lors de la dernière compétition, la limite de temps a été fixée à 40 minutes et la limite de mémoire à 32 Go.
Pour la réalisation de nos expérimentations, nous avons considéré des problèmes d'optimisation issus de la bibliothèque XCSP [5, 16], et en particulier de la dernière compétition XCSP'22. Cet ensemble d'instances, noté $\mathcal{I}_{\texttt{full}}$, comporte 250 instances réparties en 19 familles de problèmes. Nous notons $\mathcal{I}_{\texttt{hard}}$ le sous-ensemble de $\mathcal{I}_{\texttt{full}}$ comportant uniquement des instances difficiles. Une instance est considérée difficile lorsque ACE n'a trouvé aucune solution en moins de 5 minutes. Il y a 11 instances difficiles, réparties dans 4 familles. Parmi elles, 2 instances de la famille MultiAgentPathFinding ne peuvent pas être représentées par ACE en raison de domaines trop grands. Nous les avons donc retiré de notre étude. Il reste donc 9 instances issues de 3 familles dans $\mathcal{I}_{\texttt{hard}}$, qui correspondent toutes à des problèmes de minimisation.
Le tableau 1 aggrège les résultats obtenus par les différentes stratégies pour chaque famille d'instances. Nous pouvons voir que nos approches peuvent résoudre jusqu'à 4 instances supplémentaires par rapport à ACE. La plus grande amélioration est apportée par l'approche $\texttt{PROGRESSIVE}^{\texttt{nback}}_{\texttt{constraint}}$. Pour obtenir plus d'informations sur les gains obtenus, le tableau 2 détaille les résultats des différentes stratégies sur les instances de $\mathcal{I}_{\texttt{hard}}$. En particulier, nous pouvons observer que les différentes approches utilisant l'approximation sont capables de trouver des bornes là où ACE n'y parvient pas. Le diagramme à la figure 3 complète cette observation : en effet, il illustre clairement le fait que notre approche parvient à trouver une première solution pour les instances considérées au moins aussi rapidement que ACE, et parfois significativement plus rapidement sur les instance de $\mathcal{I}_{\texttt{full}}$.
Pour évaluer le coût de notre approche, la figure 4 propose un diagramme de dispersion comparant ACE et l'approche $\texttt{PROGRESSIVE}^{\texttt{nback}}_{\texttt{constraint}}$. Sur ce diagramme, nous pouvons voir que notre approche peut se révéler plus lente que ACE. Cela peut s'expliquer par le fait qu'il est nécessaire d'exécuter plusieurs fois le solveur avant de trouver une première solution, là où ACE ne le fait qu'une seule fois. Néanmoins, le coût de notre approche reste négligeable dans une large majorité des cas, ce que nous pouvons également constater dans les résultats donnés dans le tableau 1 : dans la plupart des cas, le nombre d'instances résolues dans chaque famille par nos approches est au moins égal à celui du nombre d'instances résolues par ACE.
Remarquons que ACE est très bon pour trouver rapidement des solutions sur les instances de $\mathcal{I}_{\texttt{full}}$, ce qui explique pourquoi il y a peu d'instances difficiles. Par construction, notre approche est plus adaptée à de tels problèmes, comme elle requiert plusieurs appels aux solveurs avant de réellement résoudre le problème original. De plus, notre approche n'apporte en pratique aucun avantage sur les instances incohérentes. En effet, il est clair que trouver une solution d'un sous-problème dans ce cas ne pourra pas aider à trouver une solution du problème original. En théorie, il est toutefois possible de prouver l'incohérence du problème original en prouvant celle de l'un de ses sous-problèmes, mais ce cas ne s'est pas produit dans nos expérimentations.

6 Conclusion

Dans cet article, nous avons présenté une approche fondée sur l'approximation pour résoudre des problèmes d'optimisation plus efficacement. Nous avons introduit différentes techniques pour supprimer des contraintes du problème original, le rendant plus facile à résoudre afin d'obtenir rapidement une solution du sous-problème obtenu. Cette solution est ensuite utilisée pour guider la recherche d'une solution du problème original. Nos expérimentations montrent que notre approche permet souvent de réduire le temps nécessaire pour trouver une solution du problème, et pour certaines instances, d'améliorer la borne de la meilleure solution obtenue.
En guise de perspective, nous souhaitons développer de nouvelles stratégies pour mieux identifier quelles contraintes devraient être supprimées, afin d'améliorer les performances de notre approche actuelle. Une autre perspective est de mieux exploiter *l'ensemble* des solutions trouvées dans les sous-problèmes considérés, plutôt que de seulement exploiter la dernière. Par exemple, une approche possible serait de développer une heuristique de choix de valeurs utilisant la fréquence d'apparition des valeurs affectées par les différentes solutions. De plus, lorsque la solution d'un sous-problème ne satisfait pas toutes les contraintes du problème original, nous pourrions également exploiter des formes « d'explications » de cette non-satisfaction. Cette information pourrait être ajoutée aux sous-problèmes pour guider le processus d'approximation. Enfin, notre approche permet en pratique d'identifier les sous-parties les plus difficiles du problème original. En tirant partie de cette information à la manière des techniques proposées dans [9, 1], il pourrait être possible d'obtenir des informations supplémentaires utiles pour la résolution du problème complet.

3. https://github.com/crillab/universe
4. https://github.com/crillab/aceurancetourix
5. https://github.com/crillab/metrics

family	ACE	PROGRESSIVE$^{\text{wdeg}}_{\text{constraint}}$	AGGRESSIVE$^{\text{wdeg}}_{\text{constraint}}$	AGGRESSIVE$^{\text{nback}}_{\text{constraint}}$	PROGRESSIVE$^{\text{nback}}_{\text{constraint}}$
AircraftLanding	**13**	**13**	**13**	**13**	**13**
CVRP	9	**10**	**10**	**10**	**10**
ClockTriplet	**10**	**10**	**10**	**10**	**10**
CoinsGrid	**10**	**10**	**10**	**10**	**10**
CyclicBandwidth	**12**	**12**	**12**	**12**	**12**
DC	**26**	**26**	**26**	**26**	**26**
EchelonStock2	**10**	**10**	**10**	**10**	**10**
Filters	**8**	**8**	**8**	**8**	**8**
ItemsetMining	**15**	**15**	**15**	**15**	**15**
MultiAgentPathFinding	**18**	**18**	**18**	**18**	**18**
NurseRostering	**19**	**19**	**19**	**19**	**19**
NursingWorkload	7	8	8	8	**10**
Rcpsp	**10**	**10**	**10**	**10**	**10**
Rlfap	**25**	**25**	**25**	**25**	**25**
Spot5	**10**	**10**	**10**	**10**	**10**
Tal	**10**	**10**	**10**	**10**	**10**
Triangular	**10**	**10**	**10**	**10**	**10**
WarOrPeace	**10**	**10**	**10**	**10**	**10**
Warehouse	**9**	**9**	**9**	**9**	**9**
Total	241	243	243	243	245

TABLE 1 – Nombre d'instances résolues dans chaque famille par chacune des approches. Les valeurs en gras mettent en évidence le(s) solveur(s) résolvant le plus d'instances de cette famille.

input	ACE	PROGRESSIVE$^{\text{wdeg}}_{\text{constraint}}$	AGGRESSIVE$^{\text{wdeg}}_{\text{constraint}}$	AGGRESSIVE$^{\text{nback}}_{\text{constraint}}$	PROGRESSIVE$^{\text{nback}}_{\text{constraint}}$
CVRP-A-n33-k6	-	1792 (37.77)	1792 (41.06)	**1792 (36.7)**	1792 (37.12)
NurseRostering-20	-	-	-	-	-
NursingWorkload-15zones	-	-	-	-	-
NursingWorkload-3zones0	106683 (1482.95)	-	-	-	**106845 (162.22)**
NursingWorkload-3zones1	-	118706 (269.62)	118706 (240.73)	**118718 (55.88)**	118718 (61.55)
NursingWorkload-3zones2	-	-	-	-	**117392 (201.57)**
NursingWorkload-3zones3	-	**123131 (89.35)**	123787 (213.46)	122799 (90.32)	120967 (150.16)
NursingWorkload-3zones4	-	-	-	-	-
NursingWorkload-6zones	-	-	-	-	-

TABLE 2 – Première borne trouvée par les différentes stratégies pour chaque instance difficile. Le nombre entre parenthèses montre le temps nécessaire pour trouver cette borne (en secondes). Les valeurs en gras mettent en évidence le premier solveur à trouver une borne sur l'instance considérée.

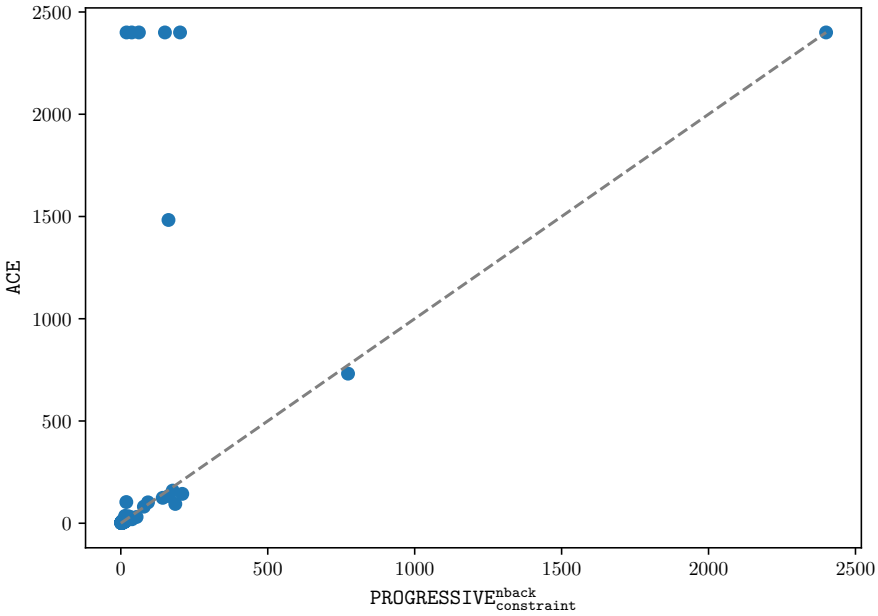


FIGURE 3 – Diagramme de dispersion comparant le temps nécessaire à ACE et à l'approche $\texttt{PROGRESSIVE}^{\texttt{nback}}_{\texttt{constraint}}$ pour trouver une première solution. Chaque point du diagramme représente une instance, et son ordonnée correspond au temps (en secondes) mis par ACE pour trouver la première solution de cette instance, tandis que son abscisse correspond au temps (en secondes) mis par l'approche $\texttt{PROGRESSIVE}^{\texttt{nback}}_{\texttt{constraint}}$ pour trouver une première solution sur cette même instance.

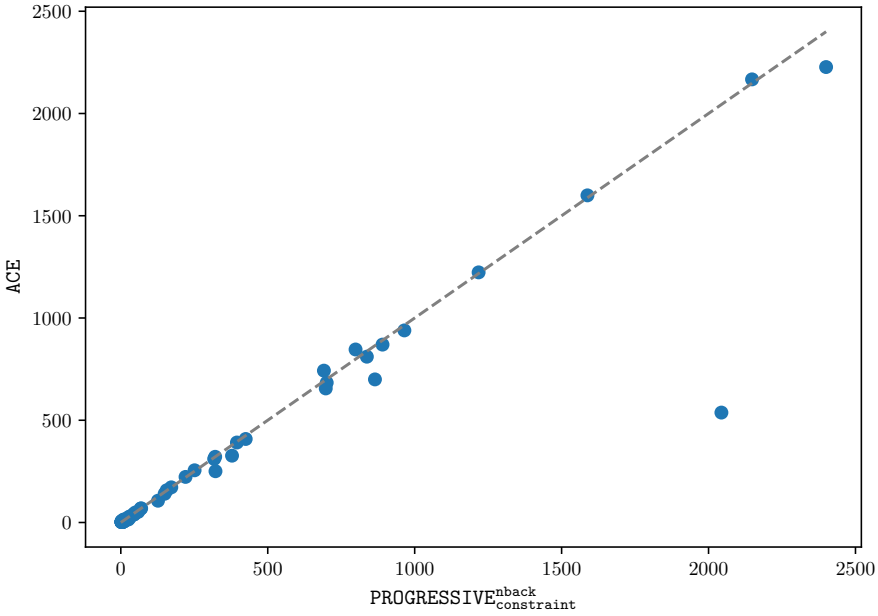


FIGURE 4 – Diagramme de dispersion comparant le temps d'exécution de ACE et de l'approche $\texttt{PROGRESSIVE}^{\texttt{nback}}_{\texttt{constraint}}$. Chaque point du diagramme représente une instance, et son ordonnée correspond au temps d'exécution (en secondes) de ACE sur cette instance, tandis que son abscisse correspond au temps d'exécution (en secondes) de $\texttt{PROGRESSIVE}^{\texttt{nback}}_{\texttt{constraint}}$ sur cette même instance.

Références

[1] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. Boosting local search thanks to cdcl. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 17th International Conference, LPAR-17, Yogyakarta, Indonesia, October 10-15, 2010. Proceedings*, volume 6397 of *Lecture Notes in Computer Science*, pages 474–488. Springer, 2010.

[2] C. Bessiere, B. Zanuttini, and C. Fernandez. Measuring search trees. In *Proceedings of ECAI'04 workshop on Modelling and Solving Problems with Constraints*, pages 31–40, 2004.

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

[4] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette. XCSP3 : an integrated format for benchmarking combinatorial constrained problems. *CoRR*, abs/1611.03398, 2016.

[5] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3-core : A format for representing constraint satisfaction/optimization problems. *CoRR*, abs/2009.00514, 2020.

[6] E. Demirovic, G. Chu, and P. Stuckey. Solution-based phase saving for CP : A value-selection heuristic to simulate local search behavior in complete solvers. In *Proceedings of CP'18*, pages 99–108, 2018.

[7] Gael Glorian, Jean-Marie Lagniez, Valentin Montmirail, and Michael Sioutis. An incremental sat-based approach to reason efficiently on qualitative constraint networks. In John N. Hooker, editor, *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings*, volume 11008 of *Lecture Notes in Computer Science*, pages 160–178. Springer, 2018.

[8] C. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1) :67–100, 2000.

[9] Éric Grégoire, Jean-Marie Lagniez, and Bertrand Mazure. A CSP solver focusing on fac variables. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 493–507. Springer, 2011.

[10] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Boosting a complete technique to find MSS and MUS thanks to a local search oracle. In Manuela M. Veloso, editor, *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 2300–2305, 2007.

[11] H.H. Hoos and E. Tsang. Local search methods. In *Handbook of Constraint Programming*, chapter 5, pages 135–167. Elsevier, 2006.

[12] Mikolás Janota, William Klieber, João Marques-Silva, and Edmund M. Clarke. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234 :1–25, 2016.

[13] Tony Lambert. *Hybridation de méthodes complètes et incomplètes pour la résolution de CSP. (Hybridization of complet and incomplete methods tosolve CSP).* PhD thesis, University of Nantes, France, 2006.

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

[15] Christophe Lecoutre. Ace, a generic constraint solver. *CoRR*, abs/2302.05405, 2023.

[16] Christophe Lecoutre and Nicolas Szczepanski. PYCSP3 : modeling combinatorial constrained problems in python. *CoRR*, abs/2009.00326, 2020.

[17] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

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

[19] M. Sanchez, S. Bouveret, S. de Givry, F. Heras, P. Jegou, J. Larrosa, S. Ndiaye, E. Rollon, T. Schiex, C. Terrioux, G. Verfaillie, and M. Zytnicki. Max-CSP competition 2008 : toulbar2 solver description. In *[?]*, pages 63–70, 2008.

[20] Nicolas Schwind, Emir Demirovic, Katsumi Inoue, and Jean-Marie Lagniez. Partial robustness in team formation : Bridging the gap between robustness and resilience. In Frank Dignum, Alessio Lomuscio, Ulle Endriss, and Ann Nowé, editors, *AAMAS '21 : 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, pages 1154–1162. ACM, 2021.

[21] Takehide Soh, Daniel Le Berre, Stéphanie Roussel, Mutsunori Banbara, and Naoyuki Tamura. Incremental sat-based method with native boolean cardinality handling for the hamiltonian cycle problem. In Eduardo Fermé and João Leite, editors, *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, volume 8761 of *Lecture Notes in Computer Science*, pages 684–693. Springer, 2014.

[22] The Choco Team. Choco : an open source Java constraint programming library. In *[?]*, pages 8–14, 2008.

[23] J. Vion and S. Piechowiak. Une simple heuristique pour rapprocher DFS et LNS pour les COP. In *Proceedings of JFPC'17*, pages 39–45, 2017.

[24] T. Walsh. Search in a small world. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, IJCAI '99, page 1172–1177, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

Representation of Gene Regulation Networks by Hypothesis Logic Based Boolean Systems

Pierre Siegel[1], Andrei Doncescu[2], Vincent Risch[1], Sylvain Sené[3]

[1] Aix Marseille University, LIS
[2] French West Indies University, Lamia
[3] Université publique,

Résumé d'un article paru dans Journal of Supercomputing, 79(4) : 4556-4581 (2023), Springer.

Résumé

Les systèmes dynamiques booléens (SDB) sont des ensembles finis de propositions qui interagissent en temps discret. Ils sont en particulier utilisés pour la modélisation des réseaux de signalisation génétique (RSG). L'étude d'un SDB se concentre principalement sur la recherche de ses configurations stables, cycles limites et cycles instables. En utilisant une logique modale $\mathcal{H}$ on obtient de nouvelles représentations des SDB et des RSG. Ces représentations utilisent une restriction de $\mathcal{H}$ qui, permet d'utiliser des algorithmes SAT pour calculer les configuration et les cycles.

Mots-clés

Réseaux génétiques, Systèmes Dynamiques Booléens, Logique non-monotone, Logique modale, logique des Hypothèses, SAT

.

1 Logique des Hypothèses

La logique des hypothèses $\mathcal{H}$ [4, 5] a été définie pour étendre la *Logique des défauts* (DL) [3, 1]. C'est une logique bi-modale avec deux opérateurs modaux L et [H]. L'opérateur [L] a les axiomes du système modal T et [H] ceux du système modal K. Une interprétation intuitive est que, si f est une formule alors Lf dit que f est prouvé. Le dual H de [H] est défini par H$f = \neg$[H]$\neg f$ et on dit que Hf est une hypothèse.

Un défaut $\frac{A:B}{C}$ de DL, peut être traduit dans $\mathcal{H}$ par la formule L$A \wedge$ H$B \rightarrow$ LC dont la signification intuitve est : "Si A est prouvé et B est une hypothèse valide alors C est prouvé." On montre dans [4, 5] que DL est un fragment de $\mathcal{H}$. Cette formule donne aussi une traduction de la clause Prolog (et de la règle ASP) C :– A, not(B).

Si F est un ensemble de formules de $\mathcal{H}$, une *extension* de F est obtenue en ajoutant à F un ensemble maximal consistant d'hypothèses. Dans DL certaines théories consistantes peuvent ne pas avoir d'extensions. Dans $\mathcal{H}$ elles ont toujours des extensions dont certaines, les *extensions fantômes*, n'ont pas de contrepartie dans DL. Ce sont ces extensions fantômes qui vont permettre de représenter les SDB.

Ici on utilise une restriction du langage de $\mathcal{H}$, suffisante pour représenter les SDBs et les réseaux de gênes : on interdit la composition des modalités. Cette restriction permet de traduire tout ensemble fini de formules de $\mathcal{H}$ en un ensemble fini de formules du calcul propositionnel et donc d'utiliser des algorithmes SAT.

2 Représentation des réseaux de gènes dans $\mathcal{H}$

Les systèmes biologiques, peuvent être représentés par un ensemble d'éléments en interaction (gènes, protéines, enzymes métaboliques, ...), dont les états changent au cours du temps. Dans ce contexte, les *voies de signalisation* sont des interdépendances spécifiques entre les gènes/protéines, comme réponses des cellules à des signaux chimiques ou à des réactions de changements environnementaux. L'ensemble des gènes impliqués dans ces voies définissent les réseaux de régulation génétique. Les systèmes biologiques, étant rarement stables, le problème est de représenter leur évolution, et en particulier les *cycles* du système.

Pour représenter un réseau de gènes en utilisant $\mathcal{H}$, on considère que si i est une protéine/gêne :

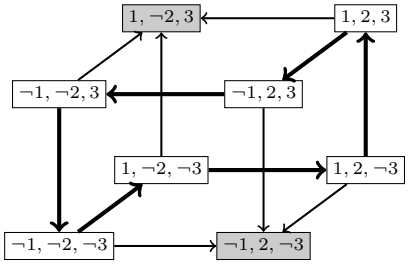
- i dit que i est présente dans la cellule et $\neg i$ qu'elle est absente.
- Li dit que i est produite (activée) par la cellule et $\neg$Li que i n'est pas produite.
- L$\neg i$ dit que i est détruite (inhibée) par la cellule (et $\neg$L$\neg i$ qu'elle n'est pas détruite).
- Hi (resp. $\neg$Hi) dit que la cellule donne (resp. ne donne pas) l'autorisation d'essayer de produire i.
- H$\neg i$ (resp. $\neg$H$\neg i$) dit que la cellule donne (resp. ne donne pas) l'autorisation d'essayer de détruire i.

On note que même si Hi donne l'autorisation de tenter de produire (ou détruire) i, cette production (ou destruction) n'est effective qu'en fonction du contexte, c'est à dire l'ensemble de toutes les interactions.

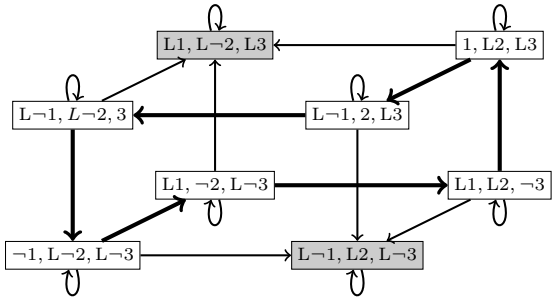
En utilisant $\mathcal{H}$, le role d'une extension est de donner un ensemble consistant d'autorisations. Cette représentation permet de démontrer des propriétés classique des réseaux de gênes

3 Systèmes Dynamiques Booléens

Un SDB décrit l'évolution des interactions dans un réseau booléen, sur un ensemble $V = \{1, ..., n\}$ de variables booléennes. Une *configuration* $x = (x_1, ..., x_n)$ du réseau est une affectation d'une valeur de vérité $x_i \in \{0, 1\}$ à chaque élément i de V. L'ensemble de toutes les configurations (les interprétations d'un point de vue logique) est $X = \{0, 1\}^n$. La *dynamique* d'un SDB donne l'évolution des éléments de V dans un temps discret. Elle est modélisée par une *fonction de transition* $f : X \to X$ et un mode d'actualisation μ. Une dynamique est *asynchrone* si pour tout x , $f(x)$ diffère de x par au plus un x_i. Une dynamique asynchrone est représentée par un *graphe de transition asynchrone* ATG) $\mathscr{G}(f) = (X, T(f))$. C'est un graphe orienté dont l'ensemble des sommets est X et l'ensemble des arcs est l'ensemble des transitions asynchrones effectives. La figure ci dessous donne l'ATG de $f(x_1, x_2, x_3) = (\neg x_2, \neg x_3, x_1)$. On y retrouve en gris deux *états stables*, et un *cycle instable* donné par les arcs foncés.

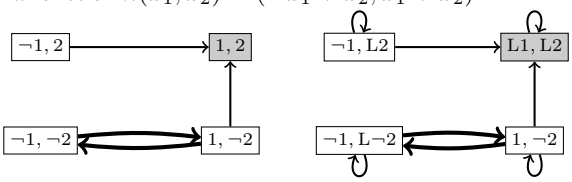


On peut traduire un SDB en $\mathcal{H}$ ainsi : toute fonction de transition locale f_i est traduite par l'ensemble $TR(f_i)$ de deux formules : $TR(f_i) = \{\mathrm{H}f_i(x) \to \mathrm{L}i, \mathrm{H}f_i(\neg x) \to \mathrm{L}\neg i\}$. La traduction $TR(f)$ de f est l'union des traduction des f_i. On démontre alors des liens formels entre un SDB et sa traduction. Les démonstrations utilisent la *sémantique de Kripke* [2] des logiques modales. La figure ci dessous représente le *modèle de Kripke*, pour l'opérateur L, de la traduction de $f(x_1, x_2, x_3) = (\neg x_2, \neg x_3, x_1)$. On remarque que pour dessiner ce modèle on part de la figure de l'ATG de f ci dessus. Ensuite, pour tout $i \in \{1, 2, 3\}$ un sommet s contient $\mathrm{L}i$ si tous les arcs qui partent de s laissent i inchangé.



Dans ce modèle de Kripke, tous les noeuds représentent des extensions de $TR(f)$. Pour une extension E, une variable i est libre si ni Li ni $L\neg i$ n'est dans E. Le *degré de liberté* de E est son nombre de variables libres. Il correspond au nombre d'arcs qui partent de E dans le modèle de Kripke. On démontre alors que les états stables d'un ATG f correspondent aux extensions de degré de liberté 0 de $TR(f)$. Les autres états corespondent aux extensions fantômes. De plus on montre que tout cycle stable de f, de longueur n correspond à en ensemble d'extensions de n de degré 1 de $TR(f)$. Les cycles instables correspondent à des ensembles d'extensions dont une au moins est de degré supérieur à 1.

Ci dessous, on représente l' ATG et le modèle de Kripke de la fonction $h(x_1, x_2) = (\neg x_1 \vee x_2, x_1 \vee x_2)$



4 Algorithme SAT

Il faut maintenant calculer les extensions d'un ensemble F de formules de $\mathcal{H}$. Les logiques modales peuvent être traduites en logique du premier ordre. Comme les modalités peuvent être composées ($LLL..i$), on obtient un univers de Herbrand infini ce qui interdit d'utiliser SAT. Mais, pour représenter les SDB, il est inutile de composer les modalités. L'ensemble fini des variables booléenes $V = \{1, ..., n\}$ est fini et les Li et Hi suffisent. On peut alors renommer chacun de ces éléments par une variable propositionnelle. De plus les opérateurs L et [H] ont les axiomes des logiques modales T et K. Or ces axiomes n'utilisent pas de composition de modalité. Ils peuvent donc être traduits par un ensemble fini de clauses propositionnelles. Ceci ouvre la voie à l'utilisation d'algorithmes SAT pour calculer les extensions d'un ensemble F de formules de $\mathcal{H}$,
On remarque que si on voulait utiliser le système $S4$ en ajoutant l'axiome (4) $\vdash \mathrm{L}f \to \mathrm{LL}f$ on obtiendrait des compositions de modalités. Il en est de même pour l'axiome (5) du système S5.

Références

[1] Delgrande, J.P., Schaub, T. : Expressing default logic variants in default logic. Journal of Logic and Computation **15**, 593–621 (2005)

[2] Kripke, S.A. : Semantical analysis of modal logic I Normal modal propositional calculi. Mathematical Logic Quarterly **9**, 67–96 (1963)

[3] Reiter, R. : A logic for default reasoning. Artificial Intelligence **13**, 81–132 (1980)

[4] Schwind, C., Siegel, P. : A modal logic for hypothesis theory. Fundamenta Informaticae **21**, 89–101 (1994)

[5] Siegel, P., Schwind, C. : Modal logic based theory for non-monotonic reasonning. Journal of Applied Nonclassical Logic **3**, 73–92 (1993)

JFPC 2 - 4 juillet 2023, 10h30-12h

Explications paresseuses Ad hoc des contraintes Element et Sum

Suruthy Sekar[1], Gael Glorian[1], Guillaume Perez[1], Wijnand Suijlen[1], Eric Monfroy[2] et Arnaud Lallouet[1]

[1] Huawei Technologies, Boulogne-Billancourt, France
[2] LERIA, Université d'Angers, Angers, France

[1]{suruthy.sekar2, gael.glorian, guillaume.perez, wijnand.suijlen, arnaud.lallouet}@huawei.com
[2]eric.monfroy@univ-angers.fr

Abstract

L'hybridation CP/SAT combine les forces de deux paradigmes, la programmation par contraintes (CP) et la satisfaction Booléenne (SAT). Elle permet d'exploiter les algorithmes de propagation dédiés de CP et de les coupler au puissant mécanisme d'apprentissage de clauses de SAT. Contrairement à la génération paresseuse de clauses, les solveurs basés sur l'explication paresseuse extraient une explication SAT après chaque conflit au sein des propagations du cœur CP. Dans cet article, nous proposons un algorithme d'explication générique de contrainte et de nouvelles explications Ad hoc pour les contraintes Element et Sum (ainsi que certaines variantes). Les expérimentations menées sur l'ensemble des instances des différentes compétitions XCSP3 montrent une amélioration de la taille moyenne des clauses générées grâce aux explications Ad hoc ainsi qu'un temps de résolution diminué.

1 Introduction

L'utilisation conjointe des techniques de résolution CP et SAT afin de résoudre des problèmes combinatoires est une technique fréquemment mise en avant afin d'améliorer l'efficacité des solveurs CP. La pierre angulaire de ces techniques d'hybridation est l'utilisation de clauses pour renforcer, voire remplacer, la propagation de contraintes CP classique usuellement effectuée par des algorithmes de filtrage dédiés. En particulier, cette hybridation permet d'utiliser la méthode *CDCL* (*Conflict Driven Clause Learning*) [15], une puissante technique tirée de SAT, à l'intérieur même d'un environnement CP de plus haut niveau. Nous distinguons trois variantes principales de l'hybridation CP/SAT, dépendant du moment où les clauses SAT sont générées :

— La traduction complète vers SAT [18, 22] puis l'utilisation d'un solveur SAT sous-jacent. À partir d'un modèle CP, le solveur le traduit entièrement vers SAT. Il est possible choisir de préserver la satisfiabilité ou la propagation. De notables efforts ont été mobilisés afin de contenir l'explosion combinatoire qui en découle.

— Les solveurs basés sur les clauses paresseuses (*LCG* pour *Lazy Clause Generation*) [16, 17, 3], quant à eux, ne vont plus véritablement réaliser la propagation de contraintes. En effet, ces solveurs vont uniquement générer les clauses nécessaires au moteur SAT pour imiter le comportement de ces propagateurs. Cela réduit notablement le nombre de clauses générées.

— Les solveurs basés sur les explications paresseuses [6, 9, 7], enfin, génèrent les clauses nécessaires uniquement lors d'un conflit, afin de l'expliquer. Pour ce faire, il est nécessaire de stocker des informations suffisantes afin de reconstruire les clauses demandées.

Dans cet article, nous nous intéressons à cette dernière variante. Dans ce contexte, nous proposons de nouveaux algorithmes dédiés afin d'extraire les raisons des propagations et des conflits sur les contraintes globales *Element* et *Sum* ainsi que certaines variantes (par exemple, *Weighted Sum*). L'article est structuré comme suit : dans un premier temps, nous présentons les paradigmes CP et SAT ainsi que le contexte d'hybridation avec explications paresseuses. Puis, nous introduisons nos algorithmes d'extraction de raisons (appelés *raisonneurs*) sur les différentes formes des contraintes *Element* et *Sum*. Nous poursuivons par une étude expérimentale comparant ces algorithmes dédiés aux algorithmes génériques et enfin, nous concluons et présentons des pistes de travaux futurs.

2 Préliminaires

Dans cette section, nous introduisons les notations et définitions que nous utiliserons pour présenter nos contributions, à savoir, des algorithmes d'extraction de raisons dans le contexte d'hybridation CP/SAT avec explications paresseuses. Pour cela, nous présentons sommairement le problème de satisfiabilité booléenne, puis le problème de satisfaction de contraintes enfin nous terminons cette section par la représentation CP/SAT hybride.

Le problème de satisfiabilité booléenne (SAT). Un littéral est une variable booléenne x qui peut être munie d'un signe afin d'exprimer la négation $\neg x$. Une clause est une disjonction de littéraux. Une formule sous forme normale conjonctive (CNF) est une conjonction de clauses construites sur un ensemble fini de variables booléennes

$\mathcal{B}$. Une clause qui ne contient qu'un seul littéral est dite unitaire. Une solution d'une formule CNF est une interprétation qui assigne une valeur (0 ou 1) à chaque variable booléenne et qui satisfait toutes les clauses. Le problème SAT est un problème de décision qui consiste à déterminer si une formule booléenne donnée admet un modèle, qui par définition est une interprétation valide.
En SAT, l'inférence est régie par la *propagation unitaire*, déterminée en prenant le point fixe du conditionnement de la formule considérée sous une décision. Si le problème vide est dérivé, le problème initial est satisfiable. En revanche, si la clause vide est dérivée, le problème est non satisfiable. Dans ce cas, le conflit est analysé et une *clause assertive* est ajoutée au problème, empêchant la survenue de ce conflit dans le futur. De plus, un retour-arrière (intelligent) sur un niveau de décision précédent est effectué sur le problème. L'ensemble de ces décisions et propagations sont stockées dans une pile appelée *trail* afin de faciliter le retour-arrière.

Le problème de satisfaction de contraintes (CSP). Un réseau fini de contraintes P est un couple $(\mathcal{X}, \mathcal{C})$, où $\mathcal{X}$ est un ensemble fini de variables et $\mathcal{C}$ un ensemble fini de contraintes. À chaque variable $x \in \mathcal{X}$ est associé un domaine (courant) totalement ordonné, noté $\texttt{dom}(x)$ qui contient un ensemble fini de valeurs qui peuvent être assignées à x. Le domaine à un instant précis t est noté $\texttt{dom}^t(x)$. Le domaine initial d'une variable est noté $\texttt{dom}^0(x)$. Afin de simplifier les algorithmes que nous allons présenter, nous introduisons la notation spéciale $\texttt{dom}_\Delta(x) = \texttt{dom}^0(x) \setminus \texttt{dom}(x)$ qui capture les valeurs supprimées du domaine de la variable x depuis la racine de l'arbre de recherche. Nous représentons la borne inférieure (resp. supérieure) d'un domaine $\texttt{dom}(x)$ par $\texttt{lb}(x)$ (resp. $\texttt{ub}(x)$) [1]. Chaque contrainte $c \in \mathcal{C}$ porte sur un ensemble fini ordonné de variables $\texttt{scp}(c)$ appelé la portée de c. L'arité d'une contrainte c correspond au nombre de variables sur lesquelles la contrainte porte ($|\texttt{scp}(c)|$). Une contrainte est définie formellement par une relation notée $\texttt{rel}(c)$, qui contient l'ensemble des tuples autorisés pour les variables de sa portée, c'est-à-dire les combinaisons qui satisfont c. Un tuple τ peut être perçu comme l'instanciation d'un sous-ensemble de variables de $\mathcal{X}$. Nous notons $\tau[x]$ la valeur de la variable x dans le tuple τ. Une solution d'un réseau de contraintes P est une instanciation de $\mathcal{X}$ telle que toutes les contraintes $c \in \mathcal{C}$ soient satisfaites. Le problème CSP est un problème de décision qui consiste à déterminer si un réseau de contraintes donné admet une solution.
Les solveurs CP fonctionnent en limitant itérativement l'espace de solution et en filtrant l'espace de recherche en appliquant des propriétés locales de cohérence jusqu'à ce qu'un point fixe soit atteint. Une forme classique de cohérence est la cohérence d'arc généralisée [14] (*GAC*, *Generalized Arc Consistency*). Pour une contrainte $c \in \mathcal{C}$, GAC garantit que chaque couple (x, a), avec $x \in \texttt{scp}(c)$ et $a \in \texttt{dom}(x)$, admet un support sur c. Un support est un tuple $\tau \in \prod_{x \in \texttt{scp}(c)} \texttt{dom}(x)$ tel que $\tau[x] = a$ et $\tau \in \texttt{rel}(c)$.

Représentation CP/SAT hybride. Les solveurs hybrides CP/SAT combinent les représentations CP et SAT, où la représentation SAT duplique celle de CP afin de permettre un apprentissage de conflit plus fin. Il existe trois représentations SAT possibles : l'encodage *direct* ; *d'ordre* et *log*. Dans le cas de l'encodage direct, un littéral $[\![x = a]\!]$ est créé pour chaque valeur $a \in \texttt{dom}(x)$. Si ce littéral est vrai, alors la valeur a est assignée à la variable x. En revanche, si ce littéral est faux, l'inverse est vrai et nous avons $\neg[\![x = a]\!]$ (ou encore $[\![x \neq a]\!]$). Ces deux littéraux peuvent être représentés par une variable booléenne. En ce qui concerne l'encodage d'ordre, nous avons cette fois un littéral $[\![x \leq a]\!]$ qui, lorsqu'il est vrai, encode le fait que $x \leq a$. Si ce littéral est faux, nous avons donc $\neg[\![x \leq a]\!]$ (ou encore $[\![x > a]\!]$). Pour l'encodage log, enfin, nous considérons une représentation binaire des valeurs des domaines. En effet, s'il est nécessaire d'utiliser l bits pour représenter l'ensemble des valeurs du domaine de x, alors nous utilisons l variables booléennes $\{x_{[i]} \mid i \in [0..l-1]\}$, chacune d'elles représentant le i^{e} chiffre de la représentation binaire.
Notons que ces encodages nécessitent des règles supplémentaires assurant l'intégrité de la représentation SAT. Dans le cas de l'encodage direct, nous avons besoin des règles implicites de domaines *At Least One* (ALO) et *At Most One* (AMO) [2]. Pour une variable $x \in \mathcal{X}$, la règle ALO s'exprime avec la clause $\{[\![x = a]\!] \mid a \in \texttt{dom}(x)\}$. La règle AMO, quant à elle, est représentée par les clauses $\{\neg[\![x = a]\!] \vee \neg[\![x = b]\!] \mid a, b \in \texttt{dom}(x), a < b\}$. L'intégrité de la représentation pour l'encodage d'ordre est assurée par l'ensemble de clauses $\{\neg[\![x \leq a]\!] \vee \neg[\![x \leq a+1]\!] \mid a \in \texttt{dom}(x) \setminus \texttt{ub}(x)\}$. L'encodage log en revanche ne nécessite des règles d'intégrité uniquement si des valeurs négatives ont besoin d'être représentées. Lorsque plusieurs représentations coexistent, il est nécessaire d'ajouter des clauses de liaison entre celles-ci [17].
En génération paresseuse de clauses, la propagation CP se doit d'être modélisée par des clauses SAT qui produiront le même résultat que les propagateurs CP, mais par propagation unitaire. Pour éviter l'explosion exponentielle du nombre de variables booléennes, la plupart des solveurs utilisent des encodages de contraintes avantageux pour certaines contraintes (voir [19] pour la contrainte *Cardinality*).
Dans les solveurs utilisant les explications paresseuses, contrairement aux traductions SAT et aux solveurs basés sur la génération paresseuse de clauses, les clauses générées n'interviennent pas dans la propagation directe des contraintes, mais sont uniquement utilisées lors de l'analyse de conflit et de la propagation unitaire. Ainsi, les encodages qui introduisent de nouvelles variables ne

1. Ces notations peuvent aussi être munies d'un instant t. Par exemple, la borne inférieure d'une variable x au niveau de décision 0 est exprimée $lb^0(x)$

2. Ces règles sont implicites dans la représentation ensembliste des domaines d'un réseau de contraintes. En effet, un domaine vide entraîne un conflit, de même que l'assignation à une valeur précédemment supprimée, ou encore, à deux valeurs en même temps, est impossible.

sont pas simples à utiliser car ces nouveaux littéraux n'ont jamais été propagés durant la recherche. Il existe cependant deux encodages pour contraintes génériques, définies par une table de valeurs, qui n'ont pas besoin de variables supplémentaires : l'encodage *direct* ou *conflit* [20] et l'encodage *support* [11, 4]. Nous nous intéressons au sein de cet article uniquement à l'encodage direct. Pour une contrainte c, l'encodage basé sur le conflit consiste à ajouter une clause $\{\neg[\![x = \tau[x]]\!] \mid x \in \texttt{scp}(c)\}$ pour chaque tuple $\tau \notin \texttt{rel}(c)$. Malheureusement, la propagation d'un tel encodage est équivalent à la propagation d'un « *forward checking* » et non de l'arc consistance [4].

3 Explications génériques paresseuses

Lors de la découverte d'un conflit dans une propagation CP, le solveur construit une contrainte empêchant la survenue ultérieure du même conflit, qui sera ajoutée au solveur. Cette contrainte est exprimée sous la forme d'une clause SAT. Pour ce faire, une première *explication* du conflit est générée et traduite en SAT [3]. Nous définissons cette explication de conflit comme l'ensemble des suppressions intervenues dans le domaine de variables qui à elles seules *sont suffisantes* pour déduire le conflit, par propagation. Cette première explication est ensuite simplifiée en utilisant la résolution, via un processus nommé analyse de conflit. Cette dernière utilise la résolution pour substituer les suppressions du niveau de décision courant, causées par propagation, par l'explication de cette propagation. Nous appelons *raison* ce type d'explication, qui est définie comme étant l'ensemble des valeurs filtrées du domaine des variables qui *sont suffisantes* pour déduire la propagation.

Dans les solveurs utilisant les explications paresseuses, l'explication initiale utilisée pour l'analyse de conflit est générée au moment où cette explication est sollicitée. Puis la résolution utilise les raisons intervenues à un moment antérieur à celui où celles-ci sont sollicitées. Dans les deux cas cependant, le but est d'identifier avec précision pourquoi une propagation a lieu ou un conflit est détecté. Cela implique intuitivement d'« inverser » le comportement du propagateur de la contrainte considérée. À la manière d'un solveur SAT, les valeurs supprimées par décision ou propagation sont stockées dans le *trail* (afin de construire le graphe d'implication). Cependant, après une affectation de variable, la suppression des valeurs restantes du domaine de cette variable sont ajoutées dans le trail, appliquant la règle AMO.

Un propagateur étant un algorithme de filtrage spécifiquement dédié à une contrainte, il est naturel que les fonctions calculant les explications et raisons le soient également. Le but étant qu'elles puissent être les plus courtes et impactantes possible [4]. Néanmoins, il existe une forme d'explication et de raison générique, qui peut s'appliquer à n'importe quelle contrainte. En effet, il est possible de considérer l'ensemble des suppressions des domaines des variables appartenant à la portée de la contrainte. Ainsi, il est certain que les suppressions ayant effectivement causé la propagation ou le conflit sont incluses dans cette explication générique, aux dépens de son optimalité.

Explication générique de conflit. Soit c une contrainte et T le trail au moment où survient un conflit. Une clause assertive valide pour un conflit causé par c est $\{\neg[\![x = a]\!] \mid x \in \texttt{scp}(c) \wedge a \in \texttt{dom}_\Delta(x)\}$. La taille d'une telle explication, en nombre de littéraux, est dans le pire cas $|\texttt{scp}(c)| \times d$ avec $d = |\texttt{dom}(x)|, x \in \texttt{scp}(c)$. En effet, si toutes les variables de la portée de la contrainte considérée sont assignées alors nous considérons toutes les valeurs supprimées de toutes les variables comme explication.

Raison générique de propagation. Soit c une contrainte et T le trail au moment où survient une propagation. Une raison valide pour la propagation de $\neg[\![x = a]\!]$ causée par c est $\{\neg[\![y = b]\!] \mid y \in \texttt{scp}(c) \setminus x, b \in \texttt{dom}_\Delta(y)\}$. La taille d'une telle raison, en nombre de littéraux, est dans le pire cas $(|\texttt{scp}(c)| - 1) \times d$ avec $d = |\texttt{dom}(x)|, x \in \texttt{scp}(c)$. En effet, si toutes les autres variables que celle dont on cherche la raison d'une propagation de la portée de la contrainte considérée sont assignées alors nous considérons toutes les valeurs supprimées de toutes les autres variables comme raison.

Pour simplifier les notations ainsi qu'éviter d'alourdir les algorithmes présentés, l'ordre des propagations n'est pas représenté. Bien sûr, en pratique, nous considérons uniquement les suppressions ayant lieu avant le littéral dont nous cherchons la raison. Pour cela, nous pouvons considérer que la notation $\texttt{dom}_\Delta(x)$ inclut cet ordre. Par exemple, celle-ci peut être lue dans un raisonneur portant sur la propagation de $\ell = \neg[\![x = a]\!]$ de la manière suivante : $\texttt{dom}_\Delta(x) = \texttt{dom}^0(x) \setminus \texttt{dom}^l(x)$ où $\texttt{dom}^l(x)$ serait le domaine de la variable x avant la propagation de ℓ au lieu du domaine courant.

Ainsi, construire des explications et raisons de taille moindre est avantageux, car plus efficace pour la propagation unitaire. Les algorithmes que nous proposons dans la suite du papier sont dans la lignée des travaux de [6] et permettent de construire des explications et raisons dédiées aux contraintes *Element* et *Sum*. Nous avons choisi d'étendre ces travaux à ces deux contraintes particulières afin de compléter les contraintes apparaissant dans la *MiniTrack* des différentes compétitions *XCSP*[1, 2].

4 La contrainte Element

La première contrainte pour laquelle nous proposons des algorithmes d'explication et de raison Ad hoc dans le contexte des explications paresseuses est la contrainte *Element* [10]. Cette contrainte globale impose $A[Index] = Result$ avec

- A, un tableau de variables ;
- $Index$, une variable ;
- $Result$, qui peut être soit une constante ou une

3. En pratique, la traduction peut être omise selon la représentation utilisée [9].

4. Pas toujours minimales en pratique, car bien trop coûteux d'un point de vue de la complexité des algorithmes mis en place.

variable, en fonction de la variante.

Afin de proposer des algorithmes dédiés à cette contrainte, nous nous basons sur les règles de propagation GAC de [5]. Une contrainte $element(Index, A, Result)$ est GAC si et seulement si :

$$Index = i \quad \Rightarrow \quad \texttt{dom}(A[i]) \subseteq \texttt{dom}(Result) \qquad (1)$$

$$i \in \texttt{dom}(Index) \quad \Rightarrow \quad \texttt{dom}(A[i]) \cap \texttt{dom}(Result) \neq \emptyset \qquad (2)$$

$$\texttt{dom}(Result) \quad \subseteq \quad \bigcup_{i \in \texttt{dom}(Index)} \texttt{dom}(A[i]) \qquad (3)$$

Dans la suite, nous présentons les algorithmes d'explication de conflits et d'extraction de raison de propagation pour les deux variantes : où $Result$ est une variable, ou une constante.

4.1 Element constant

Dans le cas où $Result$ est une constante, les règles de propagation peuvent être réduites. En effet, celui-ci n'ayant pas de domaine, les deux premières règles de propagation peuvent être simplifiées et la troisième règle peut être mise de côté. Pour la version constante nous obtenons donc deux règles de propagation :

$$Index = i \quad \Rightarrow \quad A[i] = Result \qquad (4)$$

$$i \in \texttt{dom}(Index) \quad \Rightarrow \quad Result \in \texttt{dom}(A[i]) \qquad (5)$$

L'équation 4 implique que si le domaine de la variable $Index$ ne comporte plus qu'un élément, nous pouvons alors appliquer la contrainte et assigner la valeur pointée à $Result$. L'équation 5, qui peut se réécrire $Result \notin dom(A[i]) \Rightarrow i \notin dom(Index)$ permet de décrire que dès lors que la valeur $Result$ n'est plus dans le domaine d'un élément de A, l'index de cet élément peut immédiatement être retiré du domaine de la variable $Index$.

Explication d'un conflit. Ainsi, commençons par identifier les deux cas de conflit pouvant survenir avec la contrainte *Element* sous la variante constante. Premièrement, il est possible que le domaine de la variable $Index$ devienne vide, auquel cas la contrainte est insatisfiable. De plus, il est aussi possible que le domaine de la variable $Index$ devienne unitaire, supposons avec la valeur val. Cela force la règle de propagation de l'équation 4 à se mettre en place. Dans ce cas, le fait que le domaine de $A[val]$ ne contienne plus la valeur $Result$ est un autre cas de conflit.

Algorithm 1 Explication Element Constant

Input : A, le tableau de variables
$Index$, une variable
$Result$, une constante

Output : Un ensemble de littéraux qui explique le conflit

1: $cl \leftarrow \{\neg[\![Index = a]\!] \mid a \in \texttt{dom}_\Delta(Index)\}$
2: **for all** $i \in \texttt{dom}(Index)$ **do**
3: **if** $Result \in \texttt{dom}_\Delta(A[i])$ **then**
4: $cl \leftarrow cl \cup \{\neg[\![A[i] = Result]\!]\}$
5: **end if**
6: **end for**
7: **return** cl

Au sein de l'algorithme 1, les deux cas de conflit faisant appel à la variable $Index$, nous commençons par stocker l'ensemble des suppressions intervenues dans cette variable, voir ligne 1. Il nous reste donc à expliquer pourquoi les valeurs éventuellement restantes du domaine d'$Index$, ne peuvent faire partie d'une solution. Cela est dû à la suppression de la valeur $Result$ du domaine des variables pointées par les valeurs restantes du domaine d'$Index$ (lignes 2-6). La taille d'une telle explication, en nombre de littéraux, est dans le pire cas $\max(|\texttt{dom}(Index)|, |A|)$, égal à $|\texttt{dom}(Index)|$ dans une contrainte bien formée.

Raison d'une propagation. Afin de créer un raisonneur sur un propagateur quelconque, il faut identifier les cas de modifications des domaines des variables de la portée de la contrainte considérée. Comme indiqué par les règles de propagation, il n'est possible de filtrer des valeurs qu'au sein de la variable $Index$ et du tableau A. Le premier cas est la suppression de valeurs pour la variable $Index$. Ces suppressions surviennent lorsque le domaine d'une variable pointée par le domaine de la variable $Index$ dans le tableau A ne contient plus la valeur $Result$. Le second cas est l'assignation d'une variable du tableau A à la valeur $Result$ lorsque la variable $Index$ est assignée.

Algorithm 2 Raison Element Constant

Input : ℓ, le littéral dont on cherche la raison
A, le tableau de variables
$Index$, une variable
$Result$, une constante

Output : Un ensemble de littéraux formant la raison de ℓ

1: **if** $\ell = \neg[\![Index = i]\!]$ **then**
2: $cl \leftarrow \{\neg[\![A[i] = Result]\!]\}$
3: **else** ▷ // $\ell = [\![\texttt{A[i]} = \texttt{Result}]\!]$
4: $cl \leftarrow \{[\![Index = i]\!]\}$
5: **end if**
6: **return** cl

En effet, dans l'algorithme 2, dans le cas où le littéral dont nous cherchons la raison serait une valeur supprimée a du domaine de la variable $Index$, la raison est la valeur $Result$ qui a été filtrée du domaine de $A[a]$ (ligne 2). De plus, si le domaine d'$Index$ est réduit à une seule valeur, supposons i, nous pouvons appliquer $A[i] = Result$ ce qui mène à la suppression des valeurs restantes du domaine de $Result$, à l'exception de i. La taille d'une telle raison, en nombre de littéraux, est dans le pire cas 1.

4.2 Element variable

Les règles de propagation utilisées dans la variante de la contrainte *Element* où $Result$ est une variable sont les règles originales présentées dans les équations 1, 2 et 3. Nous allons donc généraliser les algorithmes proposés dans la section 4.1 au cas où $Result$ est une variable.

Explication d'un conflit. En nous basant sur l'explication de conflit de la version constante et des règles de propagation énoncées précédemment, nous pouvons déduire que l'ensemble retourné par l'algorithme

produisant l'explication d'un conflit de la version variable de la contrainte *Element* doit contenir l'ensemble des suppressions dans la variable $Index$; l'ensemble des suppressions dans la variable $Result$; ainsi que l'ensemble des suppressions de $A[i]$, qui apparaissent dans le domaine de la variable $Result$, dont l'indice i apparaît dans les valeurs restantes du domaine de la variable $Index$.

Algorithm 3 Explication Element Variable

Input : A, le tableau de variables
$Index$, une variable
$Result$, une variable
Output : Un ensemble de littéraux qui explique le conflit

1: $cl \leftarrow \{\neg[\![Index = a]\!] \mid a \in \texttt{dom}^s(Index)\}$
2: $cl \leftarrow cl \cup \{\neg[\![Result = a]\!] \mid a \in \texttt{dom}^s(Result)\}$
3: **for all** $i \in \texttt{dom}(Index)$ **do**
4: $\quad cl \leftarrow cl \cup \{\neg[\![A[i] = a]\!] \mid a \in \texttt{dom}^s(A[i]) \wedge a \in dom(Result)\}$
5: **end for**
6: **return** cl

Dans l'algorithme 3, nous commençons par ajouter directement à l'explication les suppressions des variables $Index$ et $Result$ (ligne 1-2). Ensuite, pour chaque valeur encore disponible dans le domaine courant de la variable $Index$, nous ajoutons à l'ensemble de littéraux qui explique le conflit les valeurs supprimées du tableau de variables A qui sont encore présentes dans le domaine courant de la variable $Result$ (line 3-4). En d'autres termes, nous pointons les valeurs supprimées qui auraient pu empêcher le conflit d'avoir lieu. Cet algorithme, dans le pire cas, produit des explications de taille $\texttt{max}(|\texttt{dom}(Index)|, |A|) + |\texttt{dom}(Result)| - 1$, en nombre de littéraux.

Raison d'une propagation. Trois types de propagation peuvent survenir pour supprimer une valeur de la variable $Index$, de la variable $Result$ ou d'une des variables du tableau A. Ces trois cas doivent être traités indépendemment dans le raisonneur.

L'algorithme 4 implémente les trois cas de raisons. Les lignes 1 à 3 traitent le cas où le littéral ℓ à expliquer est $\neg[\![Index = i]\!]$. Le domaine de cette variable est modifié uniquement en suivant la règle suivante : nous supprimons au sein de la variable $Index$ les valeurs v pour lesquelles il n'y a pas de valeurs j tel que $j \in A[v] \cap Result$. Nous devons donc considérer toutes les valeurs supprimées dans le domaine de la variable $A[i]$ qui apparaissent dans le domaine initial de la variable $Result$. De plus, cela doit être fait dans l'autre sens, c'est-à-dire que nous devons prendre en compte les valeurs supprimées du domaine de la variable $Result$ qui apparaissent dans le domaine initial de la variable $A[i]$. Les lignes 4 à 6 traitent le deuxième cas, où $\ell = \neg[\![Result = a]\!]$. Une valeur est supprimée du domaine de la variable $Result$ lorsqu'elle n'apparaît plus dans aucun domaine des variables du tableau A. Cela implique deux ensembles de littéraux. Le premier concerne les valeurs supprimées dans le domaine de la variable $Index$ que nous devons considérer (ligne 5). En effet, si ces valeurs

Algorithm 4 Raison Element Variable

Input : ℓ, le littéral dont on cherche la raison
A, le tableau de variables
$Index$, une variable
$Result$, une variable
Output : Un ensemble de littéraux formant la raison de ℓ

1: **if** $\ell = \neg[\![Index = i]\!]$ **then** ▷ // Rule 2
2: $\quad cl \leftarrow \{\neg[\![A[i] = b]\!] \mid b \in \texttt{dom}^s(A[i]) \cap \texttt{dom}^0(Result)\}$
3: $\quad cl \leftarrow cl \cup \{\neg[\![Result = b]\!] \mid b \in \texttt{dom}^s(Result) \cap \texttt{dom}^0(A[a])\}$
4: **else if** $\ell = \neg[\![Result = a]\!]$ **then** ▷ // Rule 3
5: $\quad cl \leftarrow \{\neg[\![Index = i]\!] \mid i \in \texttt{dom}^s(Index)\}$
6: $\quad cl \leftarrow cl \cup \{\neg[\![A[i] = a]\!] \mid i \in \texttt{dom}^0(Index) \wedge a \in \texttt{dom}^s(A[i])\}$
7: **else** ▷ // $\ell = \neg[\![\texttt{A[i]} = \texttt{a}]\!]$, Rule 1
8: $\quad cl \leftarrow \{[\![Index = i]\!]\}$
9: $\quad cl \leftarrow cl \cup \{\neg[\![Result = a]\!] \quad | \; a \in \texttt{dom}^s(Result)\}$
10: **end if**
11: **return** cl

n'avaient pas été supprimées, a aurait possiblement été valide. Le second permet de prendre en compte les valeurs dans les domaines des variables du tableau A qui avaient permis à ℓ de rester valide (ligne 6). Enfin, les lignes 7 à 10 concernent le dernier cas, où $\ell = \neg[\![\texttt{A[i]} = \texttt{a}]\!]$. Le domaine d'une variable du tableau A ne peut être modifié que si la variable $Index$ est affectée à une valeur qui pointe vers celle-ci. Nous devons donc considérer la valeur assignée à la variable Index (ligne 9). Ensuite un second littéral est requis dans le domaine de la variable $Result$. Lorsque la variable $Index$ est réduite à un singleton i alors les domaines de $Result$ et de $A[i]$ sont réduits à leur intersection. Cela signifie que les valeurs supprimées de chacune des variables qui étaient communes dans leur domaine initial seront retirées dans le domaine de l'autre. Nous allons chercher dans les valeurs supprimées de la variable $Result$ celle qui est égale à la valeur du littéral dont on veut la raison (ligne 10).

L'algorithme 4, dans le pire cas, produit des raisons de taille $\texttt{max}(|\texttt{dom}(Result)|, |\texttt{dom}(Index)|)$, en nombre de littéraux.

5 La contrainte Sum

La seconde catégorie de contraintes pour laquelle nous avons établi des algorithmes d'extraction d'explication de conflit et de raison Ad hoc est la contrainte globale *Sum* [21] et ses variantes. Cette dernière peut être munie de coefficients ainsi que d'un opérateur. Pour cet article nous nous limitons aux opérateurs $=$, $\leq$ ainsi que $\geq$ car ceux-ci sont étroitement liés. Commençons par décrire la contrainte *Sum* de manière générique ainsi que les variantes considérées.

Soit un ensemble x_i, $i \in \{1, \ldots, n\}$ de variables et une constante $\texttt{limit} \in \mathbb{N}$, la contrainte globale *Sum* assure

que :

$$\sum_{i=1}^{n} x_i \odot \texttt{limit} \quad \text{où} \odot \in \{=, \leq, \geq\}.$$

Nous distinguons deux catégories de variantes, avec coefficients et sans coefficients. La version avec coefficients se décrit de la manière suivante : soient deux ensembles x_i et c_i, $i \in \{1, \ldots, n\}$, où les x_i représentent des variables, les c_i les coefficients (ici constants) et une constante $\texttt{limit} \in \mathbb{N}$. La contrainte globale *Sum* assure que :

$$\sum_{i=1}^{n} c_i \times x_i \odot \texttt{limit} \quad \text{où} \odot \in \{=, \leq, \geq\}.$$

Ces deux grandes catégories ainsi que les trois opérateurs sélectionnés impliquent donc six variantes que nous nommerons de la sorte, `(W)SumOP` où `W` implique la présence des coefficients, et `OP` sera l'un des trois opérateurs sous forme textuelle abrégée : `EQ,GE,LE`.

$\sum_{i=1}^{n} x_i = lim$ (SumEQ)	$\sum_{i=1}^{n} c_i \times x_i = lim$ (WSumEQ)
$\sum_{i=1}^{n} x_i \geq lim$ (SumGE)	$\sum_{i=1}^{n} c_i \times x_i \geq lim$ (WSumGE)
$\sum_{i=1}^{n} x_i \leq lim$ (SumLE)	$\sum_{i=1}^{n} c_i \times x_i \leq lim$ (WSumLE)

Tout au long de cette section, pour des raisons de redondance et de place, nous présentons uniquement les algorithmes sur les variantes `SumEQ` et `WSumEQ`. En effet, la filtrage de `(W)SumEQ` revient à appliquer `(W)SumLE` puis `(W)SumGE` tour à tour. De ce fait, chaque algorithme et règle de propagation concernant `(W)SumEQ` pourra être écourté afin de convenir à `(W)SumLE` et à `(W)SumGE` [5].

Afin de décrire d'une manière simple les algorithmes de cette section, il nous faut introduire une notation dédiée au calcul de la somme de bornes. En effet, par souci de simplicité et de généricité, notons $\texttt{lb}_c(x_i, c_i)$ (resp. $\texttt{ub}_c(x_i, c_i)$) la fonction qui renvoie la borne minimum (resp. maximum) d'une variable x_i conditionnée par un coefficient c_i (qui sera de 1 dans les cas des sommes non pondérées `SumOP`). Formellement :

$$\texttt{lb}_c(x_i, c_i) = \min(c_i \times \texttt{lb}(x_i), c_i \times \texttt{ub}(x_i)) \qquad (6)$$

$$\texttt{ub}_c(x_i, c_i) = \max(c_i \times \texttt{lb}(x_i), c_i \times \texttt{ub}(x_i)) \qquad (7)$$

Par abus de langage, et tout au long de cette section, nous parlerons de borne inférieure (resp. supérieure) de la somme à propos du résultat de $\texttt{lb}_c(x_i, c_i))$ (resp. $\texttt{ub}_c(x_i, c_i))$).

Ainsi, le propagateur de `(W)SumEQ` que nous utilisons peut être illustré par deux règles de propagation qui vont chacune retirer une valeur a du domaine d'une variable x.

5. Les lignes des algorithmes concernant l'une ou l'autre des variantes sont indiquées pour chaque algorithme.

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} \texttt{lb}_c(x_i, c_i) > \texttt{limit} - a \times c_j \Rightarrow x_j \neq a \qquad (8)$$

$$\sum_{\substack{i=1 \\ i \neq j}}^{n} \texttt{ub}_c(x_i, c_i) < \texttt{limit} - a \times c_j \Rightarrow x_j \neq a \qquad (9)$$

Pour la première règle (équation 8), notons que si le fait d'additionner une valeur a (pondérée selon le cas) d'une variable quelconque de x et l'ensemble des bornes inférieures des autres variable de x nous donne un résultat strictement supérieur à `limit`, nous avons un cas de propagation. Cette unique règle de propagation est suffisante pour la variante `(W)SumLE`.

Similairement, la seconde règle de propagation (équation 9) dénote que le fait qu'additionner une valeur a (pondérée selon le cas) d'une variable quelconque de x et l'ensemble des bornes supérieures des domaines des autres variables de la portée de la contrainte donne un résultat strictement inférieur à `limit` est un cas de propagation. En effet, s'il n'est pas possible d'atteindre `limit` dès lors que l'on fixe $x = a$, cela nous permet de supprimer a du domaine de la variable x considérée. Cette règle seule de propagation est suffisante pour la variante `(W)SumGE`.

Explication d'un conflit. Les cas de conflit pour une somme de ce type sont liés aux bornes de la somme encadrant la limite. En effet, si la limite (resp. la borne inférieure de la variable `limit`) est supérieure à la somme des plus grandes valeurs possibles (pondérées pour `WSumEQ`) des variables, alors un conflit a lieu. De manière duale, nous pouvons décrire le cas où la limite serait inférieure à la somme des plus petites valeurs.

Algorithm 5 Explication de (W)SumEQ

Input : $\texttt{limit} \in \mathbb{N}$, une constante
x, les variables de la contrainte
c, les coefficients de la contrainte
$n = |x|(= |c|)$, un entier

Output : Un ensemble de littéraux qui explique le conflit

1: $cl \leftarrow \emptyset$
2: $min \leftarrow \sum_{i=1}^{n} \texttt{lb}_c(x_i, c_i)$
3: $max \leftarrow \sum_{i=1}^{n} \texttt{ub}_c(x_i, c_i)$
4: **if** $max < \texttt{limit}$ **then**
5: **for all** $i \in \{1, \ldots, n\}$ **do**
6: $cl \leftarrow cl \cup \{\neg[\![x_i = a]\!] \mid a \in \texttt{dom}_\Delta(x_i) \wedge a \times c_i > \texttt{ub}_c(x_i, c_i)\}$
7: **end for**
8: **else if** $min > \texttt{limit}$ **then**
9: **for all** $i \in \{1, \ldots, n\}$ **do**
10: $cl \leftarrow cl \cup \{\neg[\![x_i = a]\!] \mid a \in \texttt{dom}_\Delta(x_i) \wedge a \times c_i < \texttt{lb}_c(x_i, c_i)\}$
11: **end for**
12: **end if**
13: **return** cl

Afin de reconstruire l'explication de la variante (W) SumEQ, nous nous basons sur les cas de conflit décrits plus tôt. Ces derniers sont représentés lignes 4 à 7 et lignes 8 à 12 au sein de l'algorithme 5. Dans le premier cas, si la somme des bornes supérieures est strictement inférieure à `limit`, l'explication de ce conflit est intuitivement l'ensemble des bornes supérieures ($\texttt{ub}_c(x_i, c_i)$). Autrement dit, il est suffisant de prendre l'ensemble des suppressions ayant mené aux bornes supérieures courantes, c'est-à-dire l'ensemble des valeurs supprimées qui sont supérieures à la borne supérieure au moment du conflit. Il en va de même pour le deuxième cas de conflit. Il est également possible de tronquer cet algorithme afin de l'adapter aux variantes (W) SumGE (resp. (W) SumLE) en ne considérant que les lignes 1, 3 à 7 et 13 (resp. 1, 2 et 8 à 13).

Raison d'une propagation. Afin de décrire les raisonneurs des variantes de la contrainte globale *Sum* et similairement aux notations `lb` et `ub`, nous pouvons munir les notations $\texttt{lb}_c$ et $\texttt{ub}_c$ (équations 6 et 7) d'un instant t. Par exemple : si nous voulons connaître la borne supérieure d'une variable d'indice i avant qu'un littéral ℓ soit propagé, nous pouvons l'exprimer par $\texttt{lb}_c^{\ell}(x_i, c_i)$.

Algorithm 6 Raison de (W)SumEQ

Input : $\ell = \neg[\![x_j = v]\!]$, le littéral dont on cherche la raison
`limit` $\in \mathbb{N}$, une constante
x, les variables de la contrainte
c, les coefficients de la contrainte
$n = |x|(= |c|)$, un entier

Output : Un ensemble de littéraux formant la raison de ℓ

1: $cl \leftarrow \emptyset$
2: $min \leftarrow \sum_{i=1, i\neq j}^{n} \texttt{lb}_c^{\ell}(x_i, c_i)$
3: $max \leftarrow \sum_{i=1, i\neq j}^{n} \texttt{ub}_c^{\ell}(x_i, c_i)$
4: **if** $max + v \times c_j <$ `limit` **then**
5: **for all** $i \in \{1, \ldots, n\} \setminus \{j\}$ **do**
6: $cl \leftarrow cl \cup \{\neg[\![x_i = a]\!] \mid a \in \texttt{dom}_{\Delta}(x_i) \wedge a \times c_i > \texttt{ub}_c^{\ell}(x_i, c_i)\}$
7: **end for**
8: **else if** $min + v \times c_j >$ `limit` **then**
9: **for all** $i \in \{1, \ldots, n\} \setminus \{j\}$ **do**
10: $cl \leftarrow cl \cup \{\neg[\![x_i = a]\!] \mid a \in \texttt{dom}_{\Delta}(x_i) \wedge a \times c_i < \texttt{lb}_c^{\ell}(x_i, c_i)\}$
11: **end for**
12: **end if**
13: **return** cl

Dans le cas de la variante (W) SumEQ, il existe deux règles de propagation ayant pu déclencher la propagation du littéral ℓ. Déterminer la règle de propagation responsable permet d'en construire la raison. Pour ce faire, nous sommons l'ensemble des bornes inférieures et supérieures (éventuellement pondérées), des variables de la portée, en omettant précisément la variable impliquée dans ℓ. En effet, dans le cas d'une contrainte globale, la suppression d'une valeur ne peut expliquer la suppression d'une autre valeur si celles-ci proviennent de la même variable.

L'idée est de supposer $x_j = a$ et trouver la raison de pourquoi il est impossible, sous cette hypothèse, de satisfaire la contrainte. Premièrement, il est possible qu'additionner l'ensemble des bornes supérieures et a soit strictement inférieur à `limit`; dans ce cas, la raison de cette propagation est l'ensemble des valeurs supérieures à la borne supérieure *au moment* où a eu lieu la propagation de ℓ, de chaque variable de la portée de la contrainte, à l'exception de la variable impliquée dans ℓ. Ce cas de figure apparaît aux lignes 4 à 7 de l'algorithme 6. La deuxième possibilité concerne le cas où additionner l'ensemble des bornes inférieures et a est strictement supérieur à `limit`. Dans ce cas, la raison de ℓ est l'ensemble des valeurs inférieures à la borne inférieure au moment de la propagation de ℓ, de chaque variable de la portée de la contrainte, exception faite pour la variable concernée par ℓ. Ce cas de figure est traité aux lignes 8 à 12 de l'algorithme 6.

A l'image de l'explication, il est possible d'adapter cet algorithme pour traiter le cas des variantes (W) SumGE en considérant uniquement les lignes 1, 3 à 7 et 13. Le cas des variantes (W) SumLE est traité en ne considérant que les lignes 1, 2 et 8 à 13.

Bien que l'algorithme 6 soit correct, en pratique il est très coûteux de calculer les bornes des domaines des variables à un moment t à chaque appel de la fonction construisant la raison, notamment sur des domaines de grande taille. Ces appels se comptent rapidement au nombre de plusieurs millions. Il est généralement nécessaire de parcourir une grande partie du domaine d'une variable afin de calculer $\texttt{lb}_c^{\ell}(x_i, c_i)$ ou $\texttt{ub}_c^{\ell}(x_i, c_i)$, ce qui pourrait annuler le gain d'avoir des raisons plus courtes dû à la complexité de calcul de ces bornes. En effet, le raisonneur générique (présenté Section 3) a une complexité dans le pire cas en $\mathcal{O}(nd)$ avec n la taille de la portée de la contrainte considérée et d la plus grande taille de domaine. L'algorithme 6 a une complexité dans le pire cas en $\mathcal{O}(nd^2)$. C'est pourquoi, nous choisissons de relaxer la fonction calculant la raison afin de gagner en complexité temporelle. À la place de $\texttt{lb}_c^{\ell}(x_i, c_i)$ (resp. $\texttt{ub}_c^{\ell}(x_i, c_i)$), utilisons les fonctions $\texttt{lb}_c(x_i, c_i)$ (resp. $\texttt{ub}_c(x_i, c_i)$). Remarquons que $\texttt{lb}_c(x_i, c_i) \geq \texttt{lb}_c^{\ell}(x_i, c_i)$ puisque la construction de la raison intervient uniquement dans la même branche que la propagation. En effet, une borne inférieure ne peut qu'augmenter au fur et à mesure de la recherche au sein de la même branche de l'arbre. Pour la même raison, notons que $\texttt{ub}_c(x_i, c_i) \leq \texttt{ub}_c^{\ell}(x_i, c_i)$.

Le fait de ne plus utiliser les bornes au moment de la propagation ne nous permet plus d'identifier aisément la règle de propagation en cause : il faut donc la déduire.

$$\sum_{i=1, i\neq j}^{n} (\texttt{lb}_c(x_i, c_i) + x_j \times c_j) < \texttt{limit} \tag{10}$$

$$\sum_{i=1, i\neq j}^{n} (\texttt{lb}_c^{\ell}(x_i, c_i) + x_j \times c_j) < \texttt{limit} \tag{11}$$

Ainsi, (10) implique (11). Cette implication exclut de fait que la raison cherchée soit causée par la règle de propagation (8). Mais cette propagation ayant nécessairement eu lieu, nous pouvons en déduire que la règle de propagation utilisée est (9) et en déduire la raison concernée, qui correspond ici aux lignes 5 à 7 de l'algorithme 6.

Similairement, lorsque $\sum_{i=1,i\neq j}^{n}(\texttt{ub}_c(x_i,c_i)+x_j\times c_j) > \texttt{limit}$, cela implique que $\sum_{i=1,i\neq j}^{n}(\texttt{ub}_c^{\ell}(x_i,c_i)+x_j\times c_j) > \texttt{limit}$. Nous pouvons en déduire que la règle de propagation mise en cause est (8), ce qui correspond aux lignes 9 à 11 de l'algorithme 6.

Notons cependant que (11) $\not\Longrightarrow$ (10). En effet, il peut exister certains cas où l'augmentation de la borne inférieure entre le moment où la propagation a lieu et celle où la raison est demandée est suffisante pour ne plus permettre d'identifier la règle de propagation responsable. Ce cas sera traité en prenant l'explication générique, c'est-à-dire l'ensemble des suppressions des domaines.

Algorithm 7 Raison relaxée de (W)SumEQ

Input : $\ell = \neg[\![x_j = v]\!]$, le littéral dont on cherche la raison
$\texttt{limit} \in \mathbb{N}$, une constante
x, les variables de la contrainte
c, les coefficients de la contrainte
$n = |x|(= |c|)$, un entier

Output : Un ensemble de littéraux formant la raison de ℓ

1: $cl \leftarrow \emptyset$
2: $min \leftarrow \sum_{i=1,i\neq j}^{n} \texttt{lb}_c(x_i,c_i)$
3: $max \leftarrow \sum_{i=1,i\neq j}^{n} \texttt{ub}_c(x_i,c_i)$
4: **if** $min + v \times c_j < \texttt{limit}$ **then**
5: **for all** $i \in \{1,\dots,n\} \setminus \{j\}$ **do**
6: $cl \leftarrow cl \cup \{\neg[\![x_i = a]\!] \mid a \in \texttt{dom}_\Delta(x_i) \wedge a \times c_i > \texttt{ub}_c(x_i,c_i)\}$
7: **end for**
8: **end if**
9: **if** $max + v \times c_j > \texttt{limit}$ **then**
10: **for all** $i \in \{1,\dots,n\} \setminus \{j\}$ **do**
11: $cl \leftarrow cl \cup \{\neg[\![x_i = a]\!] \mid a \in \texttt{dom}_\Delta(x_i) \wedge a \times c_i < \texttt{lb}_c(x_i,c_i)\}$
12: **end for**
13: **else**
14: **for all** $i \in \{1,\dots,n\} \setminus \{j\}$ **do**
15: $cl \leftarrow cl \cup \{\neg[\![x_i = a]\!] \mid a \in \texttt{dom}_\Delta(x_i)\}$
16: **end for**
17: **end if**
18: **return** cl

6 Expérimentations

Afin de tester nos algorithmes et constater si effectivement les explications et raisons Ad hoc accélèrent la résolution de problème de satisfaction, nous les avons implémentées dans le solveur *Nacre* [9]. Ce solveur hybride basé sur les explications paresseuses a l'avantage d'avoir été conçu comme un outil extensible, où l'ajout d'algorithmes d'explication et de raisonneur Ad hoc est rendu aisé. Il s'agit également d'un solveur pertinent, puisqu'il a terminé à la première place de la catégorie *Minitrack - CSP* de la compétition *XCSP* 2018 [13, 8] et 2019 [12]. Nous avons donc choisi d'utiliser les instances de cette même compétition [6]. En particulier, nous nous intéressons aux instances des *Minitrack - CSP* des années 2017, 2018, 2019 et 2022.

	Generic		Ad hoc	
Family (# instances)	SAT	UNS	SAT	UNS
Langford (33)	12	3	12	**5**
Quasigroup-all (148)	10	15	**11**	15
Quasigroup-3 (37)	2	3	**3**	3
Quasigroup-4(37)	2	4	2	4
Quasigroup-5(37)	4	4	4	4
Quasigroup-7 (37)	2	4	2	4
Total	22	18	**23**	**20**
Total par méthode	40		**43**	

TABLE 1 – Comparaison du nombre d'instances résolues pour la contrainte Element constant. La ligne Quasigroup-all est la somme du détail de cette famille.

Element.
Nous considérons pour ces expérimentations l'ensemble des instances des *Minitrack - CSP* des années 2017, 2018 et 2019 contenant au moins une contrainte *Element* ; l'année 2022 ne contenant pas d'instance de la sorte. Les expérimentations ont été lancées sur un ordinateur équipé de 4 cœurs, cadencés à 3,3GHz et disposant de 64Go de RAM. La limite temporelle a été fixée à 2400 secondes et la limite spatiale à 15500 Mo, s'inspirant de la configuration des compétitions XCSP.

Le tableau 1 montre en gras les valeurs montrant le solveur implémentant les explications et raisons ad hoc résolvant plus d'instances que le solveur implémentant les explications et raisons génériques. De plus, le solveur Ad hoc résout 3 instances de plus, ce qui ici représente un gain d'environ 8%. Toutefois, nous n'avons pas constaté d'amélioration en termes de nombre d'instances résolues pour la contrainte *Element* variable.

Sum.
A l'instar de la contrainte *Element*, nous considérons également l'ensemble des instances des *Minitrack - CSP* des années 2017, 2018, 2019 et 2022 qui contiennent au moins une contrainte globale *Sum*, peu importe la variante. Cela nous donne un total de 158 instances. Ces expérimentations ont été effectuées sur une machine muni d'un processeur *Dual Intel Xeon E5-2683 v4* de 16×2 cœurs cadencé à 2,10 GHz et équipé de 256 Go de mémoire.

6. https://www.xcsp.org/competitions/

	# Instances	Time (sec.)	PAR1	PAR2	PAR10
VBS	87	22,092	22,092	22,092	22,092
Ad hoc	82	29,571	29,571	41,571	137,571
Generic	82	32,176	32,176	44,176	140,176

TABLE 2 – Résultats contrainte globale *Sum* sur les instances des compétitions *XCSP* résolues par au moins une des méthodes.

Afin de mesurer le gain éventuel des explications et raisons Ad hoc pour la contrainte *Sum*, nous comparons les deux versions de Nacre dont les sommes sont traitées soit de manière générique soit de manière Ad hoc. Par souci de simplicité, nous nommerons "Générique" la version traitant la contrainte *Sum* avec des raisons et explications génériques, et "Ad hoc" la version utilisant les algorithmes d'explications et de raisons Ad hoc présentés section 5 pour cette même contrainte. Nous fixons la limite temporelle à 2400 secondes et la limite spatiale à 15500 Mo, à l'image de la compétition *XCSP*.

Analyse des résultats.

Afin de simplifier la présentation et l'analyse des résultats, nous regroupons ceux-ci par famille. De plus, les instances non résolues par les deux méthodes à la fois, c'est-à-dire celle qui n'ont pas de résultat après la limite temporelle de 2400 secondes, sont mises de côté.

Les résultats obtenus en termes de temps cumulé de résolution sont présentés dans le tableau 2. Nous appelons *VBS* (pour *Virtual Best Solver*) un solveur théorique tel que, pour chaque instance, le *VBS* choisisse toujours la meilleure méthode disponible parmi celles testées. De plus, les colonnes *PARx* indiquent le temps cumulé de résolution en ajoutant la limite temporelle multiplié par x pour les instances non résolues. Ces colonnes permettent d'avoir une précision plus fine sur le temps de résolution.

Ainsi, notons que bien que la méthode Ad hoc résolve exactement le même nombre d'instances que la méthode générique, elle les résout cependant plus rapidement. En effet, la méthode Ad hoc résout l'ensemble des instances considérées en 29571 secondes contre 32176 secondes pour le solveur implémentant les explications et raisons génériques pour la contrainte Sum, ce qui représente une amélioration de 8,1% du temps de résolution.

Notons également que le solveur Ad hoc fonctionne nettement mieux sur les instances insatisfiables que sur les instances satisfiables. Le gain en temps est de 24,7% sur les instances UNSAT alors qu'il n'est que de 3,3% sur les instances SAT, comme nous pouvons le voir dans le tableau 3 et qui est confirmé par le nuage de points en figure 1.

En nous intéressant à la décomposition par famille des chiffres constatés, elle aussi présente en tableau 3, nous pouvons observer une certaine disparité entre les différentes familles. Celles gagnant le plus à utiliser les explications et raisons Ad hoc pour la contrainte `Sum` sont les familles *MagicSquare*, *Pb* et *Primes*. A l'inverse, d'autres familles voient leur temps de résolution impacté par ces méthodes

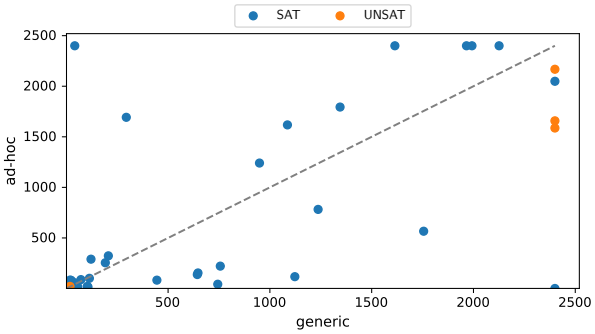


FIGURE 1 – Nuage de points représentant les instances SAT et UNSAT

Ad hoc : il s'agit des familles *MagicHexagon*, *MarketSplit*, *MultiKnapsack* et *Wwtpp*. Cette disparité peut partiellement être expliquée par un surcoût imposé par nos fonctions Ad hoc pour la contrainte *Sum* comparées aux fonctions génériques d'explication et raison. En effet, alors que la fonction d'explication et raison générique parcourt chaque domaine une unique fois afin de construire l'explication ou la raison, notre algorithme impose de sommer l'ensemble des bornes inférieures et supérieures une première fois avant de parcourir chaque domaine à nouveau. Nous avons vu que l'accès à cette borne, notamment lorsqu'il s'agit d'une borne à un moment antérieur est très coûteux.

C'est précisément pour pallier cet inconvénient que nous avons mis en place l'algorithme 7, implémentant les raisons relaxées, qui échangent une partie de la précision des clauses générées contre un temps moindre passé à calculer les bornes. Dans l'optique de mettre à l'épreuve cet algorithme, nous nous sommes d'abord penchés sur les instances de la *Minitrack - CSP*. Cependant, aucune de ces instances ne gagne réellement à utiliser la version relaxée des raisons de la contrainte *Sum*. Néanmoins, d'autres familles prouvent le potentiel de cette relaxation : c'est le cas notamment de la famille MagicSequence, une famille d'instances contenant uniquement des contraintes de type *Cardinality* et *Sum*. Les résultats des expérimentations

Problem (# Instances)	Generic		Ad hoc	
	Time (sec.)	Avg Size	Time (sec.)	Avg Size
DiamondFree (6)	54.41	1047.48	57.18	1280.52
Kakuro (6)	0.69	90.19	1.07	110.88
MagicHexagon (6)	1721.31	380.72	4932.75	451.77
MagicSquare (5)	2592.01	45.91	255.71	49.45
MarketSplit (6)	4654.9	136.52	5795.94	136.69
MultiKnapsack (9)	5158.06	187.1	6552.28	186.11
Pb (27)	16404.29	12638.00	9983.97	8827.52
Primes (13)	1290.9	2233.93	294.8	2095.98
Sat (6)	0.64	102.23	0.64	102.23
Wwtpp (3)	299.25	20473.7	1697.44	29045.0
SAT Instances (80)	24953.69	-	24133.35	-
UNSAT Instances (7)	7222.77	-	5438.43	-
Total	32176.46	-	29571.78	-

TABLE 3 – Temps cumulé de résolution des instances et taille moyenne des clauses, groupé par famille.

	Generic	Ad hoc	Relaxed ad hoc
MagicSequence-300-ca	1.31	17.24	2.79
MagicSequence-600-co	12.57	339.39	33.88
MagicSequence-800-ca	43.54	922.81	93.59
MagicSequence-1000-ca	99.12	2400	219.06
MagicSequence-1000-co	100.09	2400	212.3

TABLE 4 – Temps de résolution des instances de la famille MagicSequence

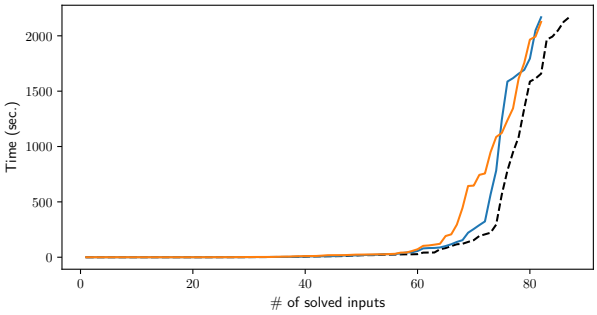


FIGURE 2 – Cactus plot entre les méthodes Générique (orange) et Ad hoc (bleu). Le VBS apparaît en ligne pointillée.

lancées sur cette famille sont visibles en tableau 4. Nous voyons que la méthode relaxée améliore nettement le temps de résolution, bien que sans jamais rattraper le solveur Générique. Notons que dans le nom des instances de cette famille, le chiffre suivant "MagicSequence" représente la taille des domaines. Ainsi, nous voyons nettement que plus le domaine est grand, plus l'écart entre le solveur Générique et le solveur Ad hoc est conséquent. Cet écart important est considérablement amenuisé grâce aux raisons relaxées de la contrainte *Sum*. Il est donc important de rester prudent lors de l'insertion d'explications Ad hoc.

7 Conclusion & Perspectives

Cet article s'inscrit dans le contexte de l'hybridation CP/SAT utilisant les explications paresseuses et traite en particulier la génération de clauses SAT lors de la survenue d'un conflit au sein d'un solveur CP. L'algorithme de génération de ces clauses est dépendant de la contrainte provoquant le conflit ou la propagation. C'est pourquoi, nous avons proposé dans cet article des algorithmes Ad hoc d'explication et de raison pour les contraintes globales *Element* et *Sum* ainsi que plusieurs de leurs variantes. De plus, et afin de pallier le coût important des raisonneurs à fin grain de la contrainte *Sum*, nous en avons proposé une relaxation. Les algorithmes Ad hoc pour la contrainte *Sum* ainsi que pour la contrainte *Element* se révèlent efficaces puisqu'il sont responsables respectivement de la résolution de 8% d'instances supplémentaires pour *Element* constant et un gain de temps de 8% sur la résolution pour *Sum*, toutes variantes confondues. De plus, l'élaboration d'une version relaxée des raisonneurs de *Sum* nous a permis de mettre en évidence une nette amélioration du temps de résolution des instances de la famille MagicSequence, par rapport au solveur Ad hoc, sans toutefois rattraper les explications génériques.

Cependant, cette piste nécessite des expérimentations plus poussées, ce que nous envisageons dans le futur. Une autre piste de réflexion concerne la meilleure efficacité de notre solveur Ad hoc sur les instances insatisfiables. En effet, ceci pourrait profiter aux instances de satisfaction de contraintes avec une fonction à optimiser, où il est en général nécessaire de prouver l'insatisfiabilité du problème pour une certaine valeur de la fonction objectif pour prouver l'optimalité d'une instanciation.

Une autre perspective intéressante pourrait être de proposer un meilleur algorithme d'explication et de raison génériques sous la forme notamment d'un générateur de raisonneur. Nous avons vu qu'une raison revient à "inverser" un propagateur. Ainsi, il serait théoriquement possible de pouvoir faire cette inversion, étant donnés les propagateurs.

Références

[1] Gilles Audemard, Frédéric Boussemart, Christophe Lecoutre, Cédric Piette, and Olivier Roussel. Xcsp^3 and its ecosystem. *Constraints An Int. J.*, 25(1-2) :47–69, 2020.

[2] Frédéric Boussemart, Christophe Lecoutre, Gilles Audemard, and Cédric Piette. Xcsp3-core : A format for representing constraint satisfaction/optimization problems. *CoRR*, abs/2009.00514, 2020.

[3] Thibaut Feydy and Peter J. Stuckey. Lazy clause generation reengineered. In Ian P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009.

[4] Ian P. Gent. Arc consistency in SAT. In Frank van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence, ECAI'2002, Lyon, France, July 2002*, pages 121–125. IOS Press, 2002.

[5] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*, volume 4204 of *Lecture Notes in Computer Science*, pages 182–197. Springer, 2006.

[6] Ian P. Gent, Ian Miguel, and Neil C. A. Moore. Lazy explanations for constraint propagators. In Manuel Carro and Ricardo Peña, editors, *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, volume 5937 of *Lecture Notes in Computer Science*, pages 217–233. Springer, 2010.

[7] Gael Glorian. *Hybridation de techniques d'apprentissage de clauses en programmation par contraintes.* PhD thesis, Artois University, Arras, France, 2019.

[8] Gael Glorian. Nacre. In Christophe Lecoutre and Olivier Roussel, editors, *Proceedings of the 2018 XCSP3 Competition*, pages 85–85, 2019.

[9] Gael Glorian, Jean-Marie Lagniez, and Christophe Lecoutre. NACRE - A nogood and clause reasoning engine. In Elvira Albert and Laura Kovács, editors, *LPAR 2020 : 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 of *EPiC Series in Computing*, pages 249–259. EasyChair, 2020.

[10] Pascal Van Hentenryck and Jean-Philippe Carillon. Generality versus specificity : An experience with AI and OR techniques. In Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith, editors, *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988*, pages 660–664. AAAI Press / The MIT Press, 1988.

[11] Simon Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artif. Intell.*, 45(3) :275–286, 1990.

[12] Christophe Lecoutre and Olivier Roussel. Xcsp3 competition, 2019. *URL : http ://www. cril. univ-artois. fr/XCSP19/[cited 22.02. 2022].*

[13] Christophe Lecoutre and Olivier Roussel. Proceedings of the 2018 xcsp3 competition. *arXiv preprint arXiv :1901.01830*, 2018.

[14] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1) :99–118, 1977.

[15] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[16] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007.

[17] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints An Int. J.*, 14(3) :357–391, 2009.

[18] Steven D. Prestwich. CNF encodings. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 75–100. IOS Press, 2021.

[19] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, pages 827–831. Springer, 2005.

[20] Toby Walsh. SAT v CSP. In Rina Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*, volume 1894 of *Lecture Notes in Computer Science*, pages 441–456. Springer, 2000.

[21] Tallys H. Yunes. On the sum constraint : Relaxation and applications. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*, pages 80–92. Springer, 2002.

[22] Neng-Fa Zhou and Håkan Kjellerstrand. The picat-sat compiler. In Marco Gavanelli and John H. Reppy, editors, *Practical Aspects of Declarative Languages - 18th International Symposium, PADL 2016, St. Petersburg, FL, USA, January 18-19, 2016. Proceedings*, volume 9585 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2016.

Compresser des arbres de recherche UNSAT à l'aide d'un système de cache

Anthony Blomme[1], Daniel Le Berre[1], Anne Parrain[1], Olivier Roussel[1]

[1] Univ. Artois, CNRS, Centre de Recherche en Informatique de Lens (CRIL), F-62300 Lens, France
{blomme, leberre, parrain, roussel}@cril.fr

5 mai 2023

Résumé

Afin de fournir aux utilisateurs de solveurs SAT des preuves d'incohérence petites et faciles à comprendre, nous présentons des techniques de mise en cache pour identifier des sous-preuves redondantes et réduire la taille des arbres de preuve UNSAT. Dans un arbre de recherche, nous élaguons les branches correspondant à des sous-formules qui ont été prouvées incohérentes auparavant dans l'arbre. Pour ce faire, nous utilisons un cache inspiré par les compteurs de modèles et nous l'adaptons au cas des formules incohérentes. Nous discutons de l'implémentation de ce cache dans un solveur CDCL et un solveur DPLL. Cette approche peut réduire drastiquement l'arbre de preuve UNSAT de plusieurs instances des compétitions SAT'02 et SAT'03.

Mots-clés

SAT, IA Explicable, Solveur.

Abstract

In order to provide users of SAT solvers with small, easily understandable proofs of unsatisfiability, we present caching techniques to identify redundant subproofs and reduce the size of some UNSAT proof trees. In a search tree, we prune branches corresponding to subformulas that were proved unsatisfiable earlier in the tree. To do so, we use a cache inspired by model counters and we adapt it to the case of unsatisfiable formulas. The implementation of this cache in a CDCL and a DPLL solver is discussed. This approach can drastically reduce the UNSAT proof tree of several benchmarks from the SAT'02 and SAT'03 competitions

Keywords

SAT, Explainable AI, Solver.

1 Introduction

Depuis deux décennies, les solveurs SAT ont été fréquemment utilisés pour résoudre des problèmes NP-complets, et sont donc devenus courants dans de nombreuses applications [1]. Cependant, avec la hausse de la complexité des programmes informatiques, il y a un besoin toujours plus important que ces programmes soient capables de fournir des explications à leurs résultats. Ce besoin concerne naturellement tout type de solveur et donc les solveurs SAT dans notre cas. De ce fait, il est maintenant attendu que les solveurs SAT fournissent des explications car ceux-ci ne peuvent plus être utilisés comme de simples boîtes noires. Quand une instance est cohérente, le modèle trouvé peut être donné comme explication, et compressé en le réduisant à un impliquant premier [5]. Quand l'instance est incohérente, donner une explication est plus difficile car il faut alors montrer qu'aucune solution ne peut être trouvée. Certaines formes d'explication ont été proposées pour prouver l'incohérence d'une formule. Par exemple, nous pouvons envisager de donner un *MUS* (pour *Minimal Unsatisfiable Subset*) [9] à l'utilisateur, ce qui donne l'origine de l'incohérence, ou encore un certificat d'incohérence exprimé dans un format particulier tel que DRAT [17]. Ce dernier enregistre les étapes importantes d'un solveur et peut ensuite être contrôlé par un vérificateur indépendant. Cependant, ces techniques peuvent présenter un intérêt limité pour l'utilisateur car, dans le premier cas, il n'y a aucune garantie qu'un MUS soit plus petit que la formule complète et, dans le second cas, un certificat peut avoir un nombre exponentiel d'étapes. Dans les deux cas, ces formes de preuves ne peuvent pas être facilement comprises par l'utilisateur.

Dans cet article, nous nous focalisons sur des formules incohérentes. Notre but est de compresser significativement l'arbre de recherche d'un solveur afin d'obtenir une preuve suffisamment petite pour pouvoir être donnée comme explication à l'utilisateur. Nous nous concentrons sur la reconnaissance de motifs réguliers en raison de leur impact potentiellement important sur la taille de l'arbre, et aussi parce qu'ils peuvent être expliqués individuellement et indépendamment à l'utilisateur. Un cache peut être utilisé pour reconnaître ces motifs. Si la sous-formule courante a déjà été explorée et prouvée incohérente, alors la branche peut être élaguée. Notre objectif est de réduire la taille de l'explication, et cela ne nous dérange pas de passer du temps sur cette tâche si cela nous permet d'obtenir une bonne compression. Par conséquent, nous n'excluons pas des techniques coûteuses comme les oracles NP, tant qu'elles nous permettent de réduire la taille de la preuve.

Cet article est organisé de la façon suivante. Dans la section 2, nous introduisons certaines notions fondamentales. Dans la section 3, après un exemple dédié au problème de pigeons (PHP), nous discutons de l'intégration d'un cache

pour formules incohérentes dans les solveurs SAT, en considérant deux architectures de solveur. Ensuite, nous présentons quelques résultats expérimentaux dans la section 4. Enfin, nous concluons et nous présentons quelques pistes de recherche.

2 Préliminaires

Une *variable* booléenne v peut être soit vraie soit fausse. Un *littéral* est soit une variable v soit sa négation $\neg v$. Une *clause* est une disjonction (ou un ensemble) de littéraux et une formule sous *Forme Normale Conjonctive* (CNF) est une conjonction (ou un ensemble) de clauses. Une *affectation* est une fonction d'un ensemble de variables vers les valeurs de vérité 0 (pour *faux*) ou 1 (pour *vrai*). Une clause est satisfaite par une affectation si elle contient au moins un littéral l assigné à vrai. Une formule est satisfaite par une affectation si et seulement si toutes ses clauses sont satisfaites. Décider s'il existe une affectation qui satisfait une formule CNF donnée est connu comme le *problème de la cohérence booléenne* (SAT), qui est NP-complet [2]. La formule est *SAT* s'il est possible de trouver une telle affectation et elle est *UNSAT* sinon. Soit une affectation I, $F_{|I}$ désigne la formule simplifiée par I : les clauses satisfaites sont supprimées de la formule et les littéraux falsifiés sont supprimés des clauses restantes. Une *clause unitaire* est une clause c qui contient uniquement un littéral non falsifié l, qui doit donc être assigné à vrai. La clause c peut ensuite être considérée comme la *raison* de l'affectation de l et va être désignée $reason(l)$. Appliquer cette opération jusqu'à ce qu'il ne reste plus de clauses unitaires est appelé la *propagation unitaire*. Étendre une affectation avec un littéral sans raison est appelé une *décision*. Il est possible d'associer un *niveau de décision* à chaque littéral propagé ou décidé. Celui-ci est défini comme le nombre de décisions prises par le solveur avant l'affectation du littéral considéré. Une fonction $DL(l)$ fournit le niveau auquel le littéral l a été décidé ou propagé.

Les solveurs SAT sont des programmes informatiques capables de résoudre le problème SAT. Les premiers solveurs SAT capables de résoudre ce problème reposaient sur l'architecture *Davis Putnam Logemann Loveland* (DPLL) [4, 3]. Il y a deux décennies, une nouvelle architecture appelée *Conflict Driven Clause Learning* (CDCL) est apparue adaptée à la résolution de problèmes structurés et a fait des solveurs SAT des oracles couramment utilisés pour résoudre des problèmes NP-complets [1]. Les solveurs SAT explorent un arbre de recherche, dans lequel un chemin de la racine aux feuilles est une affectation partielle, et les feuilles correspondent à des clauses falsifiées (que l'on nomme conflit) quand la formule est incohérente. Alors que les approches DPLL explorent un arbre binaire en branchant sur les valeurs de vérité des variables, les solveurs CDCL utilisent l'analyse de conflit et l'apprentissage de clause pour diriger la recherche [11].

3 Redondance au cours de la recherche

Détecter des sous-arbres communs dans un arbre de recherche n'est pas nouveau : par exemple dans les compteurs de modèles [16, 14] les sous-arbres communs sont utilisés pour mettre en cache des nombres de modèles déjà calculés. Dans notre cas, nous voulons implémenter une idée similaire, mais en ciblant les formules incohérentes. Dans ce contexte, le cache contiendra des formules prouvées UNSAT, que nous espérons retrouver pendant la recherche. Un élément du cache est appelé une *entrée*. Le temps nécessaire à la réalisation de la compression n'est pas important à ce stade. Nous étudions en priorité les capacités de compression.

3.1 Exemple introductif

Les problèmes de pigeons sont des problèmes incohérents classiques connus pour être difficiles pour les solveurs et présentant de nombreuses symétries [8]. Le problème consiste à assigner $n+1$ pigeons à n pigeonniers avec les contraintes qu'un pigeon doit être associé à un pigeonnier et qu'un pigeonnier ne peut pas accueillir plus d'un pigeon. Pour ce problème, nous définissons les variables $x_{i,k}$, avec $i \in \{1, \dots n+1\}$ et $k \in \{1, \dots n\}$, qui indiquent que le pigeon i est associé au pigeonnier k. La première contrainte peut donc être encodée en utilisant une clause de taille n pour chaque pigeon : $C_{1,n} = \bigwedge_{1 \leq i \leq n+1}(x_{i,1} \vee \dots \vee x_{i,n})$. Pour la seconde, nous devons créer toutes les exclusions mutuelles entre deux pigeons et pour un pigeonnier spécifique : $C_{2,n} = \bigwedge_{1 \leq i < j \leq n+1} \bigwedge_{1 \leq k \leq n}(\neg x_{i,k} \vee \neg x_{j,k})$. Avec ces considérations, un problème de pigeons pour une valeur n (PHP_n) est défini comme $PHP_n = C_{1,n} \wedge C_{2,n}$. Quand la variable $x_{1,k}$ est assignée à $vrai$ et propagée, nous nous retrouvons avec un problème de pigeons de taille $n-1$. Cela se produit quand on explore les n façons de placer le premier pigeon. Une fois que le premier sous-problème PHP_{n-1} a été exploré, il peut être ajouté au cache et les $n-1$ autres sous-problèmes peuvent être reconnus. Cette méthode peut être répétée récursivement jusqu'à rencontrer le problème PHP_2. Seulement deux branches, une décision et sa négation, sont nécessaires pour complètement explorer ce dernier. La figure 1 illustre l'imbrication des sous-problèmes de pigeons.

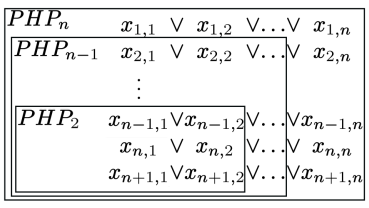


FIGURE 1 – Problème de pigeons de taille n.

Considérons par exemple le problème PHP_4 et une heuristique qui décide négativement la première variable non assignée. Cette heuristique va d'abord assigner $\neg x_{1,1}$, $\neg x_{1,2}$ et $\neg x_{1,3}$. Après ces trois décisions, la clause $x_{1,1} \vee x_{1,2} \vee$

$x_{1,3} \vee x_{1,4}$ va propager $x_{1,4}$. Cette dernière affectation va également propager les littéraux $\neg x_{2,4}$, $\neg x_{3,4}$, $\neg x_{4,4}$ et $\neg x_{5,4}$. Nous devons maintenant explorer le problème PHP_3. L'heuristique va ensuite décider $\neg x_{2,1}$ puis $\neg x_{2,2}$ et la clause $x_{2,1} \vee x_{2,2} \vee x_{2,3} \vee x_{2,4}$ va propager $x_{2,3}$. Cela va aussi propager $\neg x_{3,3}$, $\neg x_{4,3}$ et $\neg x_{5,3}$. Nous arrivons alors au problème PHP_2. Ce dernier sera entièrement exploré en décidant $\neg x_{3,1}$ puis en inversant cette décision. Les deux branches vont mener à un conflit. Comme nous avons prouvé que le problème PHP_2 est UNSAT, nous pouvons l'enregistrer dans le cache. En inversant les décisions $\neg x_{2,2}$ puis $\neg x_{2,1}$, nous allons obtenir à chaque fois un problème PHP_2 basé sur différentes variables. En consultant le cache, nous savons que nous avons déjà exploré ce problème à un renommage de variables près et nous pouvons directement conclure que ces deux branches sont incohérentes. Nous pouvons maintenant stocker le problème PHP_3 et un comportement similaire va se produire en inversant les décisions $\neg x_{1,3}$, $\neg x_{1,2}$ et $\neg x_{1,1}$. Après cela, la recherche va s'arrêter et l'instance va être considérée incohérente. La figure 2 montre l'arbre de recherche obtenu avec cette méthode. On remarque qu'il contient une branche unique avec toutes les décisions. Au final, nous avons 5 détections d'isomorphisme pour un total de 7 branches.

Avec cette méthode, il est possible d'avoir un total de $\sum_{y=2}^{n-1} y = ((n-2)(n+1))/2$ détections d'isomorphisme pour PHP_n. En ajoutant les deux branches de PHP_2, nous avons un total de $(((n-2)(n+1))/2) + 2$ branches.

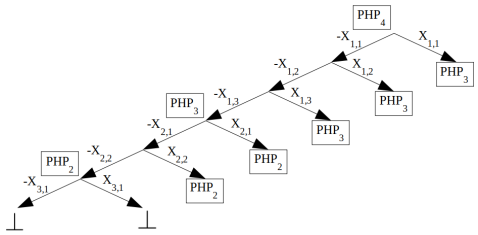


FIGURE 2 – Branche unique attendue en résolvant le problème PHP_4. Les propagations ont été omises.

3.2 Cache pour UNSAT

Pour généraliser le résultat du problème de pigeons, nous avons besoin de trouver une façon de détecter qu'une sous-formule donnée a déjà été trouvée dans l'arbre de recherche. Les compteurs de modèles [7] utilisent ce genre de fonctionnalité pour éviter de calculer plusieurs fois le nombre de modèles d'une même sous-formule (cela inclut le cas UNSAT, pour lequel le nombre de modèles est 0). Pour ce faire, ils utilisent une représentation normalisée de la sous-formule. Celle implémentée dans le compteur de modèles Cachet [14] assure que deux sous-formules avec les mêmes clauses sont considérées identiques même si ces clauses ne sont pas dans le même ordre ou si elles n'ont pas les mêmes indices. Cependant, cette technique ne va pas fonctionner sur notre exemple du problème de pigeons. En effet, dans ce cas, les sous-formules ne sont pas identiques, elles sont basées sur différentes variables. Il nous faut alors supporter la notion d'égalité modulo un renommage. Il y a aussi un problème spécifique lié à la mise en cache de formules UNSAT : une formule est UNSAT si elle contient une sous-formule UNSAT. Nous ne cherchons donc plus seulement des formules identiques, mais aussi des formules sous-sommées par une entrée du cache. Dans ce contexte, le cache ne peut plus être implémenté avec un dictionnaire. Nous devons vérifier séquentiellement toutes les entrées pour lesquelles la formule considérée a au moins autant de clauses de chaque taille. Ces deux fonctionnalités (inclusion et renommage) peuvent être implémentées en résolvant un problème NP-complet de sous-isomorphisme de graphe quand nous interrogeons le cache. Pour ce faire, nous encodons les formules sous forme de graphes de manière classique : les littéraux correspondent à des nœuds de la même couleur et les clauses correspondent à des nœuds dont la couleur dépend de leur taille. Une arête connecte les littéraux opposés et une clause est reliée à chacun de ses littéraux. La figure 3 montre un exemple de cette représentation. Chaque couleur est ici représentée par une forme différente.

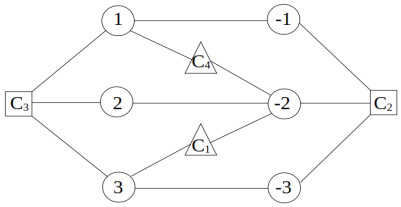


FIGURE 3 – Graphe correspondant à $F = (\neg x_2 \vee x_3) \wedge (\neg x_3 \vee \neg x_2 \vee \neg x_1) \wedge (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2)$. Les littéraux sont représentés par des cercles, les clauses binaires/ternaires par des triangles/carrés.

3.3 Sources d'incohérence

Un nœud de l'arbre de recherche est identifié par l'interprétation I des variables menant à ce nœud. Quand la sous-formule $F_{|I}$ obtenue à un nœud est incohérente, notre but est de l'enregistrer dans le cache. Cependant, nous ne voulons pas stocker la sous-formule complète dans le cache, mais seulement un sous-ensemble incohérent de $F_{|I}$. En d'autres termes, nous voulons enregistrer un sous-ensemble incohérent de $F_{|I}$, mais pas nécessairement un sous-ensemble minimal (MUS) car cela serait trop coûteux et vraisemblablement peu rentable. Les solveurs SAT modernes sont capables de fournir une source d'incohérence d'une formule F quand elle est UNSAT (ce que l'on nomme un noyau UNSAT [18]). Toutefois, un tel noyau est généralement donné à la fin de la recherche et il correspond à la formule complète. En revanche, nous devons générer localement un noyau incohérent pour tout nœud de l'arbre de telle sorte que $F_{|I}$ est incohérent. Pour obtenir ce sous-ensemble incohérent, il nous faut collecter les clauses identifiées comme conflits ou utilisées dans les propagations menant à ces conflits. Dans la suite de l'article, nous appellerons *sources* d'une sous-formule incohérente

$F_{|I}$ les clauses *initiales* de F utilisées par le solveur pour prouver l'incohérence de $F_{|I}$. Cet ensemble sera désigné par $S(F, I)$. Ces sources peuvent facilement être obtenues en récoltant récursivement la raison de chaque propagation menant aux conflits. Ce processus est en essence similaire à l'analyse de conflit des solveurs CDCL, sauf que l'on ne réalise aucune étape de résolution. Un point important est que les sources ne peuvent contenir que des clauses de la formule initiale. Dans un solveur CDCL, si une clause apprise apparaît dans les sources, elle est remplacée par l'ensemble des clauses initiales qui l'ont générée. Les sources peuvent être définies formellement comme suit.

Définition 1 *Nous définissons d'abord la source d'une clause $S(C)$. Quand C est une clause initiale, $S(C) = \{C\}$. Quand L est une clause apprise, $S(L)$ est l'ensemble des clauses initiales de F qui apparaissent dans la dérivation de L par résolution.*
Soit $F_{|I}$ une sous-formule incohérente et $\{I_1, \dots I_m\}$ l'ensemble des branches développées par le solveur pour prouver cette incohérence. Chaque $F_{|I_j}$ contient un conflit C_j. Nous définissons $S_0(F, I_j) = \{C_j\}$ et $S_{i+1}(F, I_j) = S_i(F, I_j) \cup \{S(reason(l)) | l \in c \wedge c \in S_i(F, I_j) \wedge DL(l) \geq DL(I_j)\}$. Cette séquence a un point fixe désigné $S(F, I_j)$. Enfin, les sources $S(F, I)$ de $F_{|I}$ sont définies comme $S(F, I) = \cup_j S(F, I_j)$.

Par construction, $S(F, I)_{|I}$ est incohérente car elle contient toutes les clauses initiales utilisées par le solveur pour prouver l'incohérence de $F_{|I}$. Nous avons aussi $S(F, I)_{|I} \subseteq F_{|I}$. Par conséquent $S(F, I)_{|I}$ est un noyau incohérent de $F_{|I}$. Dans un solveur DPLL, $S(F, I)$ peut être obtenu en collectant les sources des deux nœuds fils $S(F, I \cup \{l\})$ et $S(F, I \cup \{\neg l\})$ et en ajoutant les clauses propagées par l, un littéral apparaissant dans les sources des nœuds fils. Ce point sera discuté dans la section 3.5. Dans un solveur CDCL, les sources sont obtenues en collectant toutes les clauses utilisées dans l'analyse de conflit, et en remplaçant chaque clause apprise par ses sources (i.e. les clauses collectées au niveau du conflit qui a généré cette clause apprise).

3.4 Le cas CDCL

L'architecture CDCL est actuellement l'approche état-de-l'art pour la résolution pratique de SAT [11]. Il est donc logique de mettre en œuvre le cache sur cette architecture. Cependant, cela pose quelques soucis. Un solveur CDCL explore l'arbre de recherche de manière non chronologique, puisque chaque fois qu'il apprend une nouvelle clause, le solveur revient au niveau de décision ayant propagé un littéral grâce à cette clause (ce n'est pas toujours le cas des solveurs CDCL implémentant un retour en arrière chronologique [13]). L'exemple de la figure 4 illustre ce comportement. Considérons la formule propositionnelle $F' = \{x \vee y \vee z, \neg x \vee y\} \cup F$. Nous ne savons pas si F est cohérente ou non. Supposons que le solveur prenne d'abord des décisions sur des variables différentes de x, y, z, puis la décision $\neg z$ qui supprime le littéral z dans la première clause. Après cela, d'autres décisions sont prises et le solveur décide ensuite $\neg y$, ce qui supprime le littéral y dans les deux premières clauses. À ce moment-là, comme nous devons satisfaire à la fois x et $\neg x$, un conflit va être dérivé par propagation unitaire. Après l'analyse de conflit, le solveur apprend la clause $y \vee z$, qui est la résolvante des deux premières clauses, retourne à la décision $\neg z$ et propage directement le littéral y, qui satisfait les deux premières clauses. Notez que la recherche non chronologique ignorera les nœuds entre les décisions $\neg y$ et $\neg z$. Ainsi, la cohérence des sous-formules correspondantes demeure inconnue. En d'autres termes, l'étape de retour en arrière dans un solveur CDCL n'indique pas que les sous-formules associées à des nœuds inexplorés sont incohérentes. C'est ce que notre exemple illustre : la décision y aurait pu être prise à n'importe quel niveau entre les décisions $\neg z$ et $\neg y$. Dans ce cas, les deux premières clauses auraient été satisfaites et la cohérence de la sous-formule n'aurait plus dépendu que de la cohérence de F. Cela a une conséquence sur la façon dont nous remplissons notre cache : nous ne pouvons ajouter une entrée au cache que quand nous somme sûrs que la formule simplifiée courante est UNSAT et que cela a été prouvé par le solveur. Dans un solveur CDCL, cela ne se produit qu'aux feuilles de l'arbre, quand nous rencontrons une clause conflictuelle. Les sources d'incohérence sont calculées en parcourant le graphe d'implication à partir de la clause conflit. Comparé à l'analyse de conflit classique, ce calcul produit un sous-ensemble des clauses initiales. Les clauses apprises sont remplacées par les clauses initiales nécessaires à les produire. D'une certaine manière, les sources que nous calculons « déplient » les clauses apprises en clauses initiales. En pratique, au fur et à mesure que la recherche progresse, la taille des sources trouvées augmente. Les entrées du cache sont créées en calculant les sources des feuilles que l'on simplifie en utilisant les décisions courantes et la propagation unitaire. Pendant cette simplification, nous excluons les littéraux propagés par une clause apprise, sinon nous obtiendrions toujours une clause vide. Nous avons réalisé quelques expérimentations initiales avec ce cadre. Pour les instances de problème de pigeons, nous avons pu retrouver un petit arbre similaire à celui montré à la figure 2 en adaptant l'heuristique pour qu'elle branche d'abord positivement sur les variables apparaissant le plus fréquemment dans la formule (plutôt que négativement comme l'heuristique de base de Minisat).

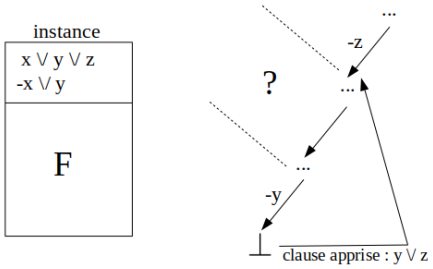


FIGURE 4 – Exemple illustrant le problème induit par la recherche non chronologique des solveurs CDCL.

Par contre, cette heuristique n'est pas performante sur les instances considérées. De plus, l'utilisation de notre cache était limitée à une étape de post-traitement, car CDCL a besoin d'une raison pour revenir en arrière. Changer la façon dont la raison est calculée modifie l'exploration de l'espace de recherche et peut donc agrandir l'arbre de recherche final. Par conséquent, nous avons aussi considéré l'approche DPLL qui ne souffre pas de ce problème.

3.5 Le cas DPLL

Dans un solveur de type DPLL, lorsque les deux fils d'un nœud correspondant à une interprétation I ont été explorés (une branche pour la décision l, une autre pour la décision $\neg l$) et que les deux sont incohérents, nous savons que $F_{|I}$ est aussi incohérent et les sources $S(F, I)$ peuvent être obtenues comme présenté dans la section 3.3. $S(F, I)$ simplifié par I est incohérent et ajouté au cache. Quand un nouveau nœud identifié par l'interprétation I est exploré, la première étape est de vérifier dans le cache s'il y a une entrée E qui est inclue dans la formule $F_{|I}$ courante à un renommage de littéraux près. S'il existe E dans le cache et σ un renommage des littéraux tel que $\sigma(E) \subseteq F_{|I}$, alors $F_{|I}$ est nécessairement incohérente puisque E est incohérente. Ce test peut être traduit en problème de sous-isomorphisme de graphe (voir section 3.2). Si on en a trouvé un, l'ensemble F' des clauses de $F_{|I}$ qui correspondent à des clauses de E est facilement obtenu en faisant correspondre des nœuds à des clauses. $F'_{|I}$ est incohérente mais en général F' peut être cohérente. En effet, F' doit être complété avec les clauses nécessaires pour propager les littéraux effacés dans F' au niveau de décision courant pour obtenir une formule incohérente. Prenons par exemple un problème de pigeons P encodé par les clauses $\{C_1, C_2, \ldots, C_n\}$ et considérons la formule F définie par $\{\neg x \vee y, \neg x \vee \neg y, x \vee C_1, C_2, \ldots, C_n\}$. Supposons aussi que le problème de pigeons P soit déjà présent dans le cache. À partir de F, si on branche sur y, $\neg x$ est propagé, et la formule simplifiée contient maintenant P qui est reconnu comme une entrée du cache. Les clauses de F correspondant à P sont $F' = \{x \vee C_1, C_2, \ldots, C_n\}$. Si on branche sur $\neg y$, nous obtenons aussi $F' = \{x \vee C_1, C_2, \ldots, C_n\}$ de la même manière. Cependant, F' est cohérente car la première clause de P peut être neutralisée par x. Pour retrouver une formule incohérente, nous devons ajouter toutes les clauses utilisées pour propager $\neg x$ sur les deux branches, ce qui signifie que nous devons ajouter $\{\neg x \vee y, \neg x \vee \neg y\}$ à F' pour obtenir une formule incohérente, qui est la source $S(F, \emptyset)$ et donc F peut maintenant être ajoutée comme entrée du cache. Il est à souligner que les vérifications du cache ont un coût élevé : nous résolvons plusieurs fois un problème NP-complet. Cependant, comme notre objectif n'est pas d'accélérer le temps de résolution mais plutôt de réduire la taille de l'arbre de recherche, il est acceptable de passer un long moment à chercher dans le cache si, au final, l'arbre généré est suffisamment petit. Quand une nouvelle entrée est ajoutée dans le cache, nous pouvons utiliser le plus grand niveau de décision présent dans les sources pour réaliser le retour en arrière. L'idée ici est d'éviter de revenir sur des décisions non impliquées dans le conflit. Ces nœuds nous donneraient la même entrée à ajouter au cache que l'entrée courante. Nous pouvons alors revenir au niveau de décision trouvé de cette manière. Si nous revenons à une décision qui n'a pas encore été inversée, alors nous inversons cette décision. Sinon nous ajoutons une nouvelle entrée au cache et nous répétons cette procédure. À titre d'exemple, considérons une formule F qui contient les clauses $\{a \vee b \vee c, a \vee b \vee \neg c, a \vee \neg b \vee c, a \vee \neg b \vee \neg c\}$ et l'interprétation $I = \langle \neg a, x, y, z \rangle$. Alors les décisions b et $\neg b$ vont toutes les deux mener à un conflit, ce qui signifie que $F_{|I}$ est incohérente de même que $S(F, I)_{|I} = \{b \vee c, b \vee \neg c, \neg b \vee c, \neg b \vee \neg c\}$. Tant que a est falsifié, ces clauses incohérentes restent dans la formule et nous pouvons donc revenir au niveau de la décision $\neg a$. Notez que même si cette technique de retour en arrière est similaire à l'analyse de conflit des solveurs CDCL, il y a tout de même quelques différences. Tout d'abord nous ne réalisons aucune étape de résolution, et donc aucune coupe dans le graphe d'implication (UIP). Ensuite nous ne sommes autorisés à passer que les sous-arbres dont nous savons qu'ils sont incohérents. Une autre différence est le fait que les clauses utilisées durant une analyse de conflit sont une explication de la clause apprise alors que les sources sont une explication de l'incohérence de la sous-formule.

4 Résultats expérimentaux

Nous avons implémenté l'approche proposée à l'aide de Minisat [6]. Nous avons désactivé la simplification de la base de clauses pour garder les clauses initiales pendant toute la recherche. Nous avons aussi désactivé les redémarrages qui construisent une séquence d'arbres de recherche. Pour l'approche DPLL, nous avons également désactivé l'apprentissage de clause et l'analyse de conflit. Cette dernière est remplacée par une procédure dédiée à la collecte des sources. L'activité d'une variable, qui est mise à jour pour chaque nouvelle clause apprise dans un solveur CDCL classique, est ici mise à jour à chaque fois qu'une nouvelle clause est ajoutée aux sources dans notre contexte. Pour l'approche CDCL, nous avons à la fois la procédure d'analyse de conflit et la procédure de collecte de sources (l'heuristique et l'apprentissage de clauses sont inchangées par rapport à Minisat). Le Glasgow Subgraph Solver (abrégé en GSS) [12] est appelé pour calculer des sous-isomorphismes de graphes, et donc questionner notre cache. Nous avons utilisé des instances des compétitions SAT'02 (partie *submitted*) [15] et SAT'03 (parties *handmade* et *industrial*) [10]. Nous avons choisi ces instances parce que nous avions besoin d'instances « faciles » pour Minisat, comme notre approche a une complexité de calcul élevée. Un résumé des résultats obtenus peut être trouvé dans la table 1. Minisat est évidemment bien plus efficace que nos approches avec du post-traitement ou un cache intégré (décrites plus tard) à cause du coût élevé de notre cache. Le DPLL avec post-traitement est clairement moins efficace que le CDCL avec post-traitement sur ces instances. Par contre, intégrer directement le cache à l'intérieur d'un solveur DPLL four-

TABLE 1 – Résumé de nos expérimentations. Pour chaque compétition, nous donnons le nombre d'instances connues pour être UNSAT et le nombre d'instances résolues par Minisat en 1 minute (instances faciles). Ensuite, nous donnons le nombre d'instances résolues dans chaque expérimentation. Les nombres entre parenthèses indiquent le nombre d'arbres de recherche avec une branche unique trouvés.

Compétition	#UNSAT	Minisat (1min)	DPLL (15min)		CDCL (15min)
			Post-traitement	Cache intégré	Post-traitement
SAT'02	381	276	40 (4)	106 (42)	78 (11)
SAT'03	198	78	15 (13)	87 (53)	39 (28)

nit des résultats significativement meilleurs que le CDCL avec post-traitement. Cela permet même de résoudre des instances que Minisat ne peut pas résoudre en 4 heures (e.g. les instances de la famille Urquhart).

4.1 Post-traitement des traces

Dans cette section, nous nous intéressons au potentiel de compression de notre approche. Il est donc nécessaire de comparer l'arbre avec et sans cache. La seule façon de faire est dans un premier temps de résoudre le problème et stocker l'arbre de recherche et dans un second temps de lancer le mécanisme de cache sur l'arbre stocké. Ainsi, il est facile de comparer l'arbre initial et l'arbre obtenu en utilisant le cache. En pratique, stocker l'arbre de recherche peut mener à des fichiers de très grande taille, c'est pourquoi nous simulons le post-traitement directement dans le solveur. Nous imposons un temps limite de 2 secondes pour chaque appel au GSS lorsque nous essayons d'identifier un sous-isomorphisme de graphe. Les deux approches DPLL et CDCL ont été testées avec un temps limite de 15 minutes sur nos instances. Certains résultats individuels de ces expérimentations sur quelques familles d'instances peuvent être trouvés dans la table 2. Nous comparons les nombres de conflits, donc de branches, des deux arbres de recherche. Le rapport entre ces deux nombres représente le pouvoir de compression de notre approche. La taille d'une instance est la somme des longueurs de ses clauses. Une distribution des rapports obtenus par les deux approches peut être trouvée dans la table 3. Le rapport de compression peut être très bon (plus petit que 10^{-3}), notamment pour l'approche CDCL. Malheureusement, cela n'arrive que pour un petit sous-ensemble des instances, principalement les familles marg, Urquhart et xor_chain, qui sont très structurées. Pour l'approche DPLL, le post-traitement s'est comporté principalement comme attendu et décrit dans la section 3.1 pour les problèmes de pigeons. La seule différence vient de l'heuristique utilisée dans Minisat, qui décide négativement la première variable et décide ensuite négativement les variables en commençant par la dernière et dans l'ordre décroissant. Donc, après que le problème PHP_{n-1} a été ajouté dans le cache, il est reconnu $n-1$ fois avec la première variable assignée (ce problème a été ajouté juste avant PHP_{n-1}). Ce problème est aussi reconnu quand on inverse la première décision. Ce comportement crée une branche additionnelle et donc le nombre de branches trouvé diffère de un comparé au nombre de branches attendu. Concernant les instances des compétitions SAT, pour les familles marg, Urquhart et xor_chain, nous avons souvent obtenu un arbre de recherche avec une branche unique comme montré dans la figure 2. Pour ces instances, quand le solveur ajoute une nouvelle entrée au cache après une certaine décision, elle est souvent reconnue après la négation de cette décision. Cela nous permet d'élaguer de nombreuses branches et cela explique les bons rapports obtenus. Ce n'est pas strictement le cas pour certaines instances (e.g. marg2x6.cnf et x1_16.cnf) mais nous avons obtenu des arbres très courts pour elles. Par exemple, la figure 5 montre l'arbre de recherche obtenu par notre approche quand le cache est utilisé dans les deux approches DPLL et CDCL. Les boîtes en violet représentent l'ajout d'une nouvelle entrée dans le cache et les boîtes en vert correspondent à la détection d'une entrée. Le label « i x » (« isomorphisme x«) signifie que l'élément enregistré à « cache x » a été reconnu.

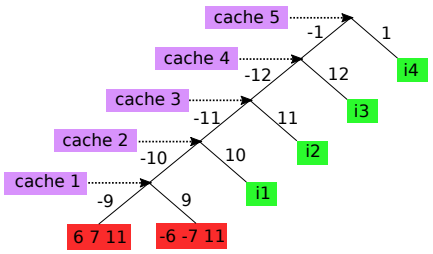


FIGURE 5 – Arbre de recherche pour marg2x2.cnf avec le cache.

4.2 Solveur avec cache intégré

Comme seconde expérimentation, nous avons utilisé le cache pendant la recherche elle-même. Nous n'avons pu réaliser qu'une implémentation pour DPLL, puisque générer une clause conflit depuis une correspondance du cache est une question ouverte à ce stade. Nous avons considéré les mêmes instances que précédemment et toujours avec un temps limite de 15 minutes par instance. Un extrait pertinent des résultats est donné dans la table 4. Pour un total de 95 instances, le solveur développe un arbre avec une branche unique comme présenté à la figure 2. La bonne compression précédemment obtenue pour les familles marg, Urquhart et xor_chain est toujours obtenue, également sur des instances plus grandes. Nous avons observé que le solveur DPLL avec cache intégré peut résoudre en moins de 15 minutes des instances que Minisat ne parvient pas à résoudre en plus de 4 heures. C'est le cas de certaines instances Urquhart de SAT'02 par exemple. Arrêter la recherche dès qu'une entrée du cache est détectée

TABLE 2 – Résultats expérimentaux concernant le pouvoir de compression de notre approche. Pour chaque instance, nous fournissons sa taille en nombre de littéraux, et le nombre de conflits trouvés sans et avec cache pour les deux approches DPLL et CDCL. Un tiret indique un dépassement de temps limite. Le rapport de compression est le nombre de conflits avec cache divisé par le nombre de conflits sans cache.

		DPLL (post-traitement)			CDCL (post-traitement)		
Instance	Taille	Conflits (sans cache)	Conflits (cache)	Rapport de compression	Conflits (sans cache)	Conflits (cache)	Rapport de compression
PHP_7	448	$6.8\ 10^3$	23	$3.4\ 10^{-3}$	$5.6\ 10^3$	853	$1.5\ 10^{-1}$
PHP_{12}	2,028	-	-	-	-	-	-
marg2x6.sat03-1444	528	$5.2\ 10^5$	21	$4.0\ 10^{-5}$	$3.0\ 10^4$	20	$6.6\ 10^{-4}$
marg3x3add8.sat03-1449	1,056	-	-	-	$1.8\ 10^5$	32	$1.8\ 10^{-4}$
Urquhart-s3-b9	1,240	$5.1\ 10^5$	20	$3.9\ 10^{-5}$	$1.9\ 10^4$	21	$1.1\ 10^{-3}$
Urquhart-s3-b3	2,152	-	-	-	$1.6\ 10^6$	29	$1.8\ 10^{-5}$
x1_16	364	$6.0\ 10^4$	18	$3.0\ 10^{-4}$	$2.2\ 10^3$	20	$9.1\ 10^{-3}$
x1_24	556	-	-	-	$2.0\ 10^5$	78	$3.9\ 10^{-4}$
3col20_5_6	646	33	20	$6.0\ 10^{-1}$	27	27	1
3col40_5_5	1,286	756	198	$2.6\ 10^{-1}$	118	72	$6.1\ 10^{-1}$
homer06	1,800	$5.1\ 10^5$	195	$3.8\ 10^{-4}$	-	-	-
homer17	3,718	-	-	-	-	-	-

TABLE 3 – Distribution des rapports pour les techniques de post-traitement pour les approches DPLL et CDCL.

Rapport	Non résolu	[1 ;0.75[	[0.75 ;0.5[	[0.5 ;0.25[	[0.25 ;10^{-1}[	[10^{-1} ;10^{-2}[	[10^{-2} ;10^{-3}[	$\leq 10^{-3}$
DPLL	524	4	5	11	11	5	6	13
CDCL	462	38	13	5	3	6	7	45

permet de résoudre bien plus d'instances que le DPLL initial, notamment pour les familles testées dans la première expérimentation. Cependant, comme la taille du cache ne fait qu'augmenter au fur et à mesure de la recherche, essayer de reconnaître une entrée du cache peut devenir très coûteux, même avec le temps limite de 2 secondes. De plus, comme les sous-formules peuvent aussi être très grandes, trouver un isomorphisme peut prendre plus de temps que la limite imposée. Pour des grandes instances, certains appels au GSS ont peut être été annulés et il se peut que nous soyons passés à côté de certains isomorphismes, et donc compressions. Cela se produit par exemple pour les problèmes de pigeons plus grands que PHP_{16}.

5 Conclusion

Notre objectif dans ce travail est d'élaguer le plus possible les branches d'un arbre de recherche UNSAT pour réduire sa taille. Pour ce faire, nous avons proposé un cache inspiré par ce qui existe déjà pour les compteurs de modèles. L'idée est d'enregistrer des sous-formules UNSAT et d'essayer de les reconnaître plus tard dans l'arbre de recherche pour éviter d'explorer plusieurs sous-parties identiques de l'arbre. Nous avons présenté une méthode syntaxique basée sur la détection de sous-isomorphismes de graphe. Nous avons vu qu'il est possible d'obtenir des rapports plutôt bons et des preuves courtes et même un arbre de recherche avec une branche unique pour certaines familles d'instances, notamment celles avec beaucoup de symétries ou similarités mais il n'est toujours pas clair si cette approche peut fonctionner sur un large éventail d'instances. Nous avons proposé une implémentation de ce cache sur les deux architectures DPLL et CDCL. Si cette dernière présente des résultats prometteurs, elle ne passe pour l'instant pas à l'échelle car nous n'avons pu l'implémenter que comme un post-traitement. L'intégration du cache directement dans le solveur DPLL a permis de réduire drastiquement beaucoup plus d'arbres de recherche, en incluant des problèmes que Minisat n'est pas capable de résoudre. Malheureusement, générer une clause conflit depuis une correspondance du cache est une question ouverte à ce stade. Nous pouvons envisager quelques pistes pour améliorer notre approche. Tout d'abord, nous avons considéré une approche basée sur la gestion d'un cache mais nous n'avons pas implémenté la possibilité de supprimer les entrées qui semblent inutiles. Cette fonctionnalité est disponible dans les compteurs de modèles pour éviter que la mémoire ne dépasse une certaine limite. Cela pourrait être un bon ajout à notre approche. Nous n'avons considéré que deux heuristiques (celle de Minisat et une variante). D'autres heuristiques pourraient être essayées, comme celles utilisées dans les compteurs de modèles. Concernant la détection de sous-isomorphismes de graphe, il pourrait être intéressant de collecter certaines informations pendant un appel au GSS et essayer de les réutiliser dans de futurs appels. De plus, nous nous intéressons à la détection d'entrées du cache dans lesquelles certains littéraux sont falsifiés, comme celles-ci sont aussi incohérentes. En effet, l'une des raisons pour lesquelles le CDCL ne produit pas souvent un arbre avec une branche unique est que la sous-formule courante n'est pas directement une entrée du cache mais une avec des littéraux falsifiés. Finalement, nous recherchons d'autres formes de redondance pour compresser des arbres de recherche UNSAT dans un cadre plus général.

TABLE 4 – Résultats expérimentaux quand le cache est utilisé pendant la recherche. Pour chaque instance, nous donnons les nombres de conflits, d'entrées dans le cache, d'appels au GSS qui ont trouvé un isomorphisme ainsi que le nombre total d'appels et le nombre d'appels abandonnés (i.e. qui ont dépassé le temps limite de deux secondes). Le nombre entre parenthèses indique le nombre d'entrées différentes du cache reconnues par isomorphisme. Nous fournissons aussi le temps passé par le solveur (sans détection d'isomorphisme) et le temps cumulé de tous les appels au GSS. Tous les temps sont en secondes.

	DPLL (cache intégré)						
Instance	Conflits	Taille du cache	Sous-isomorphismes de graphe	Appels	Abandons	Temps (recherche)	Temps (GSS)
PHP_7	23	22	21 (6)	21	0	0.007	0.180
PHP_{12}	68	67	66 (11)	66	0	0.071	5.728
PHP_{16}	122	121	120 (15)	120	0	0.274	63.741
marg2x6.sat03-1444	21	20	17 (17)	18	0	0.004	0.162
marg3x3add8.sat03-1449	26	25	22 (22)	24	0	0.024	0.813
marg6x6.sat03-1456	86	85	84 (84)	84	0	0.134	7.446
Urquhart-s3-b9	20	19	18 (18)	18	0	0.009	0.175
Urquhart-s3-b3	29	28	27 (27)	27	0	0.024	0.486
Urquhart-s5-b5	94	93	92 (91)	101	0	0.292	36.967
x1_16	18	17	14 (14)	42	0	0.005	0.419
x1_24	25	24	23 (23)	23	0	0.037	0.779
x2_80.sat03-1605	395	394	393 (318)	2,257	121	0.919	427.492
3col20_5_6	12	11	6 (3)	31	0	0.004	0.178
3col40_5_5	357	319	235 (41)	52,583	0	1.451	564.310
homer06	111	105	98 (27)	420	40	0.495	116.096
homer17	363	348	352 (92)	1,691	211	3.249	712.465

Remerciements

Le premier auteur est en partie financé par la région Hauts-de-France. Ce travail a été soutenu par le projet CPER Data de la région Hauts-de-France.

Références

[1] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability - Second Edition*. 2021.

[2] Stephen A. Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM*, pages 151–158, 1971.

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

[4] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3) :201–215, 1960.

[5] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *FMCAD 2013*, pages 46–52, 2013.

[6] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT 2003*, pages 502–518, 2003.

[7] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting. In *Handbook of Satisfiability - Second Edition*, pages 993–1014. 2021.

[8] Armin Haken. The intractability of resolution. *Theor. Comput. Sci.*, 39 :297–308, 1985.

[9] Alexey Ignatiev, Alessandro Previti, Mark H. Liffiton, and João Marques-Silva. Smallest MUS extraction with minimal hitting set dualization. In *CP 2015*, pages 173–182, 2015.

[10] Daniel Le Berre and Laurent Simon. The essentials of the SAT 2003 competition. In *SAT 2003*, pages 452–467, 2003.

[11] João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability - Second Edition*, pages 133–182. 2021.

[12] Ciaran McCreesh, Patrick Prosser, and James Trimble. The glasgow subgraph solver : Using constraint programming to tackle hard subgraph isomorphism problem variants. In *ICGT 2020*, pages 316–324, 2020.

[13] Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In *SAT 2018*, pages 111–121, 2018.

[14] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *SAT 2004*, 2004.

[15] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch. The SAT2002 competition. *Ann. Math. Artif. Intell.*, 43(1) :307–342, 2005.

[16] Marc Thurley. sharpsat - counting models with advanced component caching and implicit BCP. In *SAT 2006*, pages 424–429, 2006.

[17] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr. Drat-trim : Efficient checking and trimming using expressive clausal proofs. In *SAT 2014*, pages 422–429, 2014.

[18] Lintao Zhang and Sharad Malik. Validating SAT solvers using an independent resolution-based checker : Practical implementations and other applications. In *DATE 2003*, pages 10880–10885, 2003.

Approche générique pour l'acquisition de contraintes qualitatives *

Mohamed-Bachir Belaid,[1] Nassim Belmecheri,[2] Arnaud Gotlieb,[2] Nadjib Lazaar[3] and Helge Spieker[2]

[1]NILU, Norwegian Institute for Air Research, Kjeller, Norway.
[2]Simula Research Laboratory, Oslo, Norvége
[3] LIRMM, Université de Montpellier, CNRS, Montpellier, France

Abstract

Many planning, scheduling or multi-dimensional packing problems involve the design of subtle logical combinations of temporal or spatial constraints. On the one hand, the precise modelling of these constraints, which are formulated in various relation algebras, entails a number of possible logical combinations and requires expertise in constraint-based modelling. On the other hand, active constraint acquisition (CA) has been used successfully to support non-experienced users in learning conjunctive constraint networks through the generation of a sequence of queries. In this paper, we propose GEQCA*, which stands for Generic Qualitative Constraint Acquisition, an active CA method that learns qualitative constraints via the concept of qualitative queries.* GEQCA *combines qualitative queries with time-bounded path consistency (PC) and background knowledge propagation to acquire the qualitative constraints of any scheduling or packing problem. We prove soundness, completeness and termination of* GEQCA *by exploiting the jointly exhaustive and pairwise disjoint property of qualitative calculus and we give an experimental evaluation that shows (i) the efficiency of our approach in learning temporal constraints and, (ii) the use of* GEQCA *on real scheduling instances.*

Résumé

De nombreux problèmes de planification et d'ordonnancement impliquent la création de combinaisons subtiles de contraintes temporelles ou spatiales. La modélisation précise de ces contraintes, qui sont formulées dans diverses algèbres de relations, nécessite une expertise en modélisation basée sur les contraintes et implique un grand nombre de combinaisons logiques possibles. L'acquisition active de contraintes (AC) a été utilisée avec succès pour aider les utilisateurs non expérimentés à apprendre les réseaux de contraintes conjonctives par la génération d'une séquence de requêtes. Dans cet article, nous proposons une méthode d'AC appelée GEQCA *pour Generic Qualitative Constraint Acquisition, qui permet d'apprendre les contraintes qualitatives en utilisant des requêtes qualitatives.* GEQCA *combine les requêtes qualitatives avec la cohérence de chemin limitée dans le temps (PC pour Path Consistency) et la propagation des connaissances de base pour acquérir les contraintes qualitatives. Nous prouvons la correction, la complétude et la terminaison de* GEQCA*. Nous présentons également une évaluation expérimentale qui montre l'efficacité de notre approche dans l'apprentissage des contraintes temporelles ainsi que l'utilisation de* GEQCA *sur des instances réelles d'ordonnancement.*

Introduction

Le raisonnement sur le temps et l'espace est essentiel pour résoudre de nombreux problèmes pratiques, tels que la planification automatisée [4] et l'ordonnancement [2]. Dans ce contexte, le calcul qualitatif fournit un cadre algébrique qui établit des relations entre des paires d'entités à l'aide d'un langage qui est *exhaustif et disjoint par paires*. Des exemples de calculs qualitatifs incluent (sans s'y limiter) l'algèbre des points [6] ou l'algèbre des intervalles d'Allen [1] pour raisonner sur les tâches temporelles, et le calcul des connexions de régions (RCC) [5] pour raisonner sur les relations topologiques entre les régions spatiales. Dans ce contexte, les techniques de satisfaction de contraintes et de programmation par contraintes (CP) sont des cadres pratiques pour modéliser et résoudre des réseaux de contraintes qualitatives. Pour faciliter la modélisation des problèmes de programmation par contraintes, Bessiere et al. ont introduit un cadre permettant d'apprendre les modèles de contraintes par un apprentissage passif à partir d'un ensemble d'exemples d'affectations étiquetées ou par un apprentissage actif avec des requêtes spécifiques permettant de classer les affectations complètes.

Cet article présente le concept de l'acquisition générique de contraintes qualitatives (GEQCA), un nouvel algorithme d'acquisition active de contraintes pour apprendre tout type de contraintes qualitatives entre entités. L'algorithme GEQCA combine des requêtes qualitatives, une cohérence de chemin limitée dans le temps, une heuristique dédiée et une propagation des connaissances de base pour acquérir des contraintes. L'algorithme est conçu pour répondre aux limitations des algorithmes d'acquisition de contraintes existants, telles que l'incapacité de traiter les disjonctions, le contrôle du nombre de requêtes et la connaissance limitée du contexte. L'objectif de GEQCA est de faciliter la modélisation et la résolution de réseaux de contraintes complexes dans des situations pratiques telles que les problèmes d'ordonnancement.

*Cette présentation se base sur des résultats publiés à AAAI 2022 [3].

GEQCA : Acquisition de contraintes via des requêtes qualitatives

GEQCA est un algorithme générique conçu pour apprendre des contraintes qualitatives. Il est basé sur un nouveau concept appelé "requête qualitative", où l'utilisateur doit confirmer si une relation atomique est valide entre une paire de variables d'entité données. GEQCA prend en entrée un vocabulaire de variables d'entités, un langage de relations atomiques, des connaissances de base et un timeout comme paramètre. L'algorithme commence avec un réseau contenant uniquement des contraintes universelles entre entités, puis itère sur les paires d'entités pour réduire l'ensemble des relations possibles à un ensemble localement consistant avec ce que l'utilisateur a en tête. GEQCA utilise une procédure de propagation pour réduire automatiquement les relations incompatibles avec l'état courant de l'apprentissage sans avoir besoin de passer par l'utilisateur.

Experiments

Notre évaluation expérimentale de GEQCA porte sur l'algèbre d'Allen appliquée à des entités temporelles. Le langage Γ utilisé contient les 13 relations atomiques connues de cette algèbre.

Le tableau 1 présente l'effort fourni par l'utilisateur pour résoudre 5 instances de planification en utilisant GEQCA, avec une heuristique de sélection des paires que nous avons introduite dans ce travail. Nous avons utilisé les instances RCPSP3, disponibles publiquement, en considérant la structure du problème incluant la durée des tâches, les exigences en ressources et les capacités des sources, noté K_1. De plus, certaines contraintes peuvent déjà être connues de l'utilisateur, telles que la contrainte globale cumulative et la contrainte de délai. Nous appelons K_2 le background knowledge incluant la contrainte cumulative et la contrainte de délai. Nous avons également utilisé une limite de temps (`cutoff`) d'une heure et noté T_{max} le temps d'attente maximum entre deux requêtes. Chaque instance de planification est caractérisée par le nombre de tâches (par exemple, `sch_30_1` fait référence à l'instance numéro 1 avec 30 `TIs`).

La première observation est que l'utilisation de GEQCA avec des connaissances sur la structure du problème K_1 permet de réduire considérablement l'effort de l'utilisateur (en moyenne une réduction de 38%). La deuxième observation est que l'effort de l'utilisateur est également réduit lorsqu'il utilise des connaissances qui portent sur des contraintes connues telles que la contrainte *cumulative* et les contraintes de délai. Nous observons une réduction de 41% en utilisant K_1 pour la propagation. L'utilisation de $K_1 \wedge K_2$ apporte une amélioration faible mais non significative (en moyenne une réduction de 41% au lieu de 38%). En outre, en termes de temps CPU, le temps d'attente entre deux requêtes peut atteindre le seuil d'une heure sous $\mathcal{K}= K_1 \wedge K_2$. Cela s'explique par la procédure *solve*, qui peut prendre plus d'une heure pour essayer de prouver la cohérence d'une relation avec le réseau appris.

Instance	$\mathcal{K}$					
	$\varnothing$		K_1		$K_1 \wedge K_2$	
	`eF`	T_{max}	`eF`	T_{max}	`eF`	T_{max}
`sch_30_1`	95%	0.79	55%	0.91	53%	1.18
`sch_30_2`	98%	0.62	52%	0.90	48%	393.08
`sch_60_1`	99%	4.80	65%	8.51	62%	13.00
`sch_60_2`	99%	7.72	64%	8.08	61%	12.55
`sch_60_3`	98%	5.94	59%	9.66	57%	3,600

TABLE 1 – Effort de l'utilisateur `eF` avec GEQCA agissant sur les instances RCPSP (avec `cutoff` $= 3,600s$, T_{max} en secondes).

Conclusion

Cet article, publié à AAAI 2022, présente un nouvel algorithme d'apprentissage actif appelé GEQCA pour apprendre des réseaux qualitatifs via des requêtes qualitative. L'algorithme utilise la propriété JEPD du calcul qualitatif et la cohérence du chemin sur les contraintes temporelles pour prouver la convergence et minimiser le nombre de requêtes nécessaires. Les résultats montrent que GEQCA est une approche appropriée et efficace pour des applications pratiques.

Remerciements

Ce travail a reçu un financement du projet T-LARGO et AutoCSP du Conseil norvégien de la recherche, accord de subvention n° 274786 et n° 324674. Ainsi que du programme de recherche et d'innovation Horizon 2020 de l'Union européenne (projet TAILOR).

Références

[1] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11) :832–843, November 1983.

[2] Roman Barták, Miguel Salido, and Francesca Rossi. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing*, 21 :5–15, 02 2008.

[3] Mohamed-Bachir Belaid, Nassim Belmecheri, Arnaud Gotlieb, Nadjib Lazaar, and Helge Spieker. Geqca : Generic qualitative constraint acquisition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 3690–3697, 2022.

[4] Said Belhadji and Amar Isli. Temporal Constraint Satisfaction Techniques in Job Shop Scheduling Problem Solving. *Constraints*, 3(2) :203–211, June 1998.

[5] David A Randell, Zhan Cui, and Anthony G Cohn. A spatial logic based on regions and connection. *KR*, 92 :165–176, 1992.

[6] Marc B Vilain and Henry A Kautz. Constraint propagation algorithms for temporal reasoning. In *AAAI*, volume 86, pages 377–382, 1986.

Utilisation de SAT pour résoudre le problème SALBP avec minimisation du pic de consommation

Matthieu Py, Arnauld Tuyaba

Université Clermont Auvergne, Mines Saint-Etienne, CNRS, LIMOS, F-63000 Clermont-Ferrand, France

{matthieu.py, arnauld.tuyaba}@uca.fr

Résumé

Dans cet article, on s'intéresse au problème SALB3PM (Simple Assembly Line Balancing Problem with Power Peak Minimization). On propose une résolution de ce problème en utilisant une séquence de problèmes SAT à résoudre, avec appels à un solveur SAT pour résoudre chaque sous-problème. La pertinence de cette approche est vérifiée expérimentalement par comparaison avec les résultats existants sur quelques instances issues de la littérature. Ces travaux sont une extension de [4].

Mots-clés

Ordonnancement, SALB3PM, Modélisation, Programmation par contraintes, SAT, contraintes énergétiques

1 Présentation du problème SALB3PM

Le problème SALB3PM (Simple Assembly Line Balancing Problem with Power Peak Minimization) est un problème d'optimisation qui consiste, étant donné un ensemble de tâches et un ensemble de machines, à construire l'ordonnancement des tâches sur les différentes machines de manière à minimiser le pic d'énergie consommé par l'ordonnancement obtenu. Les données du problème SALB3PM sont les suivantes :

- Un ensemble O de n tâches, où chaque tâche $j \in O$ a une durée de traitement t_j et une consommation énergétique W_j.
- Un ensemble de machines M numérotées de 1 à m.
- Un temps de cycle c : après c unités de temps, les machines sont arrêtées. L'horizon temporel est découpé en périodes et est noté $T = \{0, ..., c-1\}$. On note l'ensemble $T^j = \{0, ..., c-t_j\}$ l'ensemble des périodes où peut commencer la tâche $j \in O$ pour pouvoir terminer avant la fin du temps de cycle.
- Un ensemble de précédences P : si une tâche $i \in O$ précède une tâche $j \in O$, ce qui est noté $i \prec j$, alors :
  - soit la tâche i est affectée à une machine de plus petit numéro que la tâche j,
  - soit les tâches i et j sont affectées à la même machine mais la tâche i est ordonnancée avant la tâche j.

L'objectif du problème SALB3PM est de décider d'affecter chaque tâche sur une machine et d'ordonnancer les tâches affectées sur les mêmes machines (c'est à dire déterminer leur période de lancement), en minimisant le pic d'énergie de la solution obtenue, c'est à dire en minimisant la consommation maximale énergétique des tâches ordonnancées simultanément.

2 Modélisation linéaire existante

Dans l'article [3], le problème SALB3PM est modélisé sous la forme du programme linéaire en nombres entiers présenté dans la figure 1. Les variables de décision sont les suivantes :

- Les variables d'affectation $X_{j,k}$. Étant donnée une tâche $j \in O$ et une machine $k \in M$, la variable $X_{j,k}$ vaut 1 si et seulement si la tâche j est affectée à la machine k.
- Les variables d'ordonnancement $S_{j,t}$. Étant donnée une tâche $j \in O$ et une période $t \in T$, la variable $S_{j,t}$ vaut 1 si et seulement si la tâche j débute à la période t.
- La variable W_{max} est une borne supérieure sur le pic de consommation énergétique.

Les lignes du programme linéaire en nombres entiers modélisent la fonction objectif et les contraintes du problème :

1. On doit minimiser le pic de consommation énergétique.
2. Chaque tâche doit être affectée à exactement une machine.
3. La somme des durées des tâches affectées à chaque machine ne peut pas dépasser le temps de cycle.
4. Si une tâche i précède une tâche j ($i \prec j$), alors la tâche i ne peut pas être affectée à une machine de plus grand numéro que la tâche j.
5. Chaque tâche doit débuter à une et une seule période.
6. Si une tâche i précède une tâche j ($i \prec j$) et que les deux tâches sont affectées à la même machine, alors la tâches j débute forcément après la tâche i.
7. Il est impossible d'avoir deux tâches en cours d'exécution sur la même machine à une période donnée.

$$min \quad W_{max} \tag{1}$$

$$s.t. \quad \sum_{k \in M} X_{j,k} = 1 \qquad \forall j \in O \tag{2}$$

$$\sum_{j \in O} t_j \times X_{j,k} \leq c \qquad \forall k \in M \tag{3}$$

$$X_{j,k} \leq \sum_{h \in M : h \leq k} X_{i,h} \qquad \forall i, j \in O : i \prec j, k \in M \tag{4}$$

$$\sum_{t \in T^j} S_{j,t} = 1 \qquad \forall j \in O \tag{5}$$

$$S_{j,t} \leq \sum_{\tau=0}^{t-t_i} S_{i,\tau} + 2 - X_{i,k} - X_{j,k} \qquad \forall i, j \in O : i \prec j, k \in M, t \in T^j \tag{6}$$

$$X_{i,k} + X_{j,k} + \sum_{\tau=t-t_i+1}^{t} S_{i,\tau} + \sum_{\tau=t-t_j+1}^{t} S_{j,\tau} \leq 3 \qquad \forall i, j \in O, k \in M, t \in T \tag{7}$$

$$\sum_{j \in O} W_j \times (\sum_{\tau=t-t_j+1}^{t} S_{j,\tau}) \leq W_{max} \qquad \forall t \in T \tag{8}$$

$$X_{i,k}, \;\; S_{i,t} \in \{0,1\}, \;\; W_{max} \in \mathbb{Z}^+ \qquad \forall i \in O, k \in M, t \in T$$

FIGURE 1 – Modélisation de SALB3PM sous forme de PLNE [3]

8. Le pic de consommation énergétique est plus grand que la consommation maximale énergétique des tâches ordonnancées sur une même période.

3 Approche SAT

On modélise le problème SALB3PM sous forme d'un problème de satisfaisabilité booléenne (problème SAT). Comme le problème SAT est un problème de décision, et non un problème d'optimisation, on modélise initialement le problème de faisabilité qui consiste à trouver une solution qui respecte toutes les contraintes du problème et qui est présenté dans la figure 2. Les variables de décisions sont les suivantes, où les deux premiers jeux de variables de décision sont les mêmes que pour la modélisation linéaire :

— Les variables d'affectation $X_{j,k}$. Étant donnée une tâche $j \in O$ et une machine $k \in M$, la variable $X_{j,k}$ vaut 1 si et seulement si la tâche j est affectée à la machine k.

— Les variables d'ordonnancement $S_{j,t}$. Étant donnée une tâche $j \in O$ et une période $t \in T$, la variable $S_{j,t}$ vaut 1 si et seulement si la tâche j débute à la période t.

— Les variables d'activité $A_{j,t}$. Étant donnée une tâche $j \in O$ et une période $t \in T$, la variable $A_{j,t}$ vaut 1 si et seulement si la tâche j est active à la période t.

Les premières lignes de la formule propositionnelle modélisent les contraintes du problème :

1. Chaque tâche doit être affectée à au moins une machine.
2. Chaque tâche doit être affectée à au plus une machine.
3. Si une tâche i précède une tâche j ($i \prec j$), alors la tâche i ne peut pas être affectée à une machine de plus grand numéro que la tâche j.
4. Chaque tâche doit débuter à au moins une période.
5. Chaque tâche doit débuter à au plus une période.
6. Une tâche ne peut pas débuter si elle ne peut pas terminer avant la fin du temps de cycle.
7. Il est impossible d'avoir deux tâches en cours d'exécution sur la même machine à une période donnée.
8. Si une tâche débute à une période donnée, alors elle est forcément active ensuite pendant la durée de son exécution.
9. Si une tâche i précède une tâche j ($i \prec j$), alors la tâche j ne peut pas être affectée à la même machine i et commencer après le début de la tâche i.

On peut constater, par exemple en comparant la ligne (2) du programme linéaire et les lignes (2) et (3) de la formule propositionnelle, que le modèle SAT est beaucoup plus gros que le modèle linéaire. Pour cette raison, on effectue un prétraitement pour calculer un ensemble de décisions *évidentes* à cause de la liste des précédences et de la durée des tâches. Ces prétraitements servent à rajouter de nouvelles contraintes permettent de réduire la taille de l'espace de recherche mais elles servent surtout à éliminer des contraintes du modèle initial (si une des contraintes (2) à (9) est en contradiction avec une des contraintes (10) à (12), alors elle n'est pas intégrée au modèle) :

10 Si une chaîne de précédences empêche une tâche j d'être affectée sur une machine k, on interdit cette affectation (on le note $ip(j,k) = 1$)

$$\bigvee_{k \in M} X_{j,k} \qquad \forall j \in O \quad (1)$$

$$\overline{X_{j,k_1}} \vee \overline{X_{j,k_2}} \qquad \forall j \in O, k_1, k_2 \in M : k_1 \neq k_2 \quad (2)$$

$$\overline{X_{j,k}} \vee \overline{X_{i,h}} \qquad \forall i, j \in O : i \prec j, k, h \in M : k < h \quad (3)$$

$$\bigvee_{t \in T^j} S_{j,t} \qquad \forall j \in O \quad (4)$$

$$\overline{S_{j,t_1}} \vee \overline{S_{j,t_2}} \qquad \forall j \in O, t_1, t_2 \in T^j : t_1 \neq t_2 \quad (5)$$

$$\overline{S_{j,t}} \qquad \forall j \in O, t \in T : t \notin T^j \quad (6)$$

$$\overline{X_{i,k}} \vee \overline{X_{j,k}} \vee \overline{A_{j,t}} \vee \overline{A_{j,t}} \qquad \forall i, j \in O : i \neq j, k \in M, t \in T \quad (7)$$

$$\overline{S_{j,t}} \vee A_{j,t+\epsilon} \qquad \forall j \in O, t \in T^j, \epsilon \in [0, t_j - 1] \quad (8)$$

$$\overline{X_{i,k}} \vee \overline{X_{j,k}} \vee \overline{S_{i,t_1}} \vee \overline{S_{j,t_2}} \qquad \forall i, j \in O : i \prec j, k \in M, \; t_1 \in T^i, t_2 \in T^j : t_1 > t_2 \quad (9)$$

$$\overline{Xj,k} \qquad \forall j \in O, k \in M : ip(j,k) \quad (10)$$

$$\overline{Xj,k} \vee \overline{S_{j,t}} \qquad \forall j \in O, k \in M : p(j,k), t \in T^j : ip(j,k,t) \quad (11)$$

$$A_{j,t} \qquad \forall t \in [c - t_i, t_i - 1] \quad (12)$$

$$\bigvee_{j \in C} \overline{A_{j,t}} \qquad \forall C \in \mathbb{C}, t \in T \quad (13)$$

FIGURE 2 – Modélisation des contraintes de SALB3PM sous forme de formule propositionnelle

11 Si une chaîne de précédences empêche une tâche j d'être affectée sur une machine k à une période t, on interdit d'avoir simultanément cette affectation et cet ordonnancement (on le note $ip(j,k,t) = 1$)

12 Si une tâche a une durée qui dépasse la moitié du temps de cycle, elle est forcément active aux dates médianes

Une fois que l'on a écrit le modèle mathématique précédent, sa résolution par un algorithme de résolution du problème SAT permet de trouver un ordonnancement faisable.

Pour ensuite déterminer l'ordonnancement avec un pic d'énergie minimum, on relance, autant de fois que possible, notre algorithme en intégrant au modèle des contraintes supplémentaires. Ces contraintes correspondent à l'interdiction d'exécuter en même temps un ensemble de tâches qui l'étaient dans la dernière solution calculée et dont la somme des énergies est supérieure au pic d'énergie de la meilleure solution rencontrée depuis le début de l'exécution de l'algorithme. Ces contraintes correspondent à la ligne (13) du modèle et l'ensemble $\mathbb{C}$ contient en permanence l'ensemble des ensembles de tâches que l'on ne souhaite plus voir ordonnancées en même temps. Après ajout de ces contraintes, on relance la résolution du problème de faisabilité.

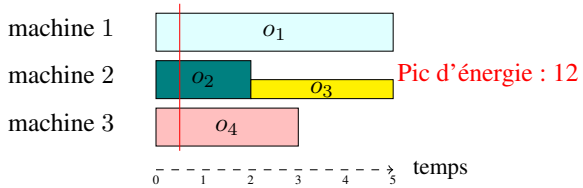
Lorsqu'il n'est plus possible de trouver une solution pour le problème de faisabilité, la meilleure solution rencontrée jusqu'à cet instant est une solution optimale pour le problème SALB3PM.

4 Exemple illustratif

On considère 4 tâches, 3 machines, un temps de cycle $c = 5$ et des précédences lexicographiques $o_1 \prec o_2 \prec o_3 \prec o_4$:

Tâche	o_1	o_2	o_3	o_4
Durée	5	2	3	3
Energie	4	4	2	4

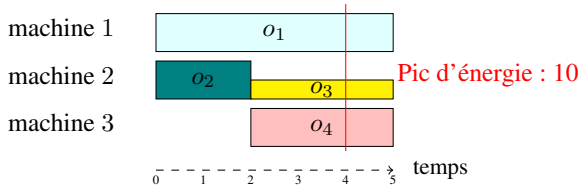
Tout d'abord, on crée le modèle initial et, à la première itération, le solveur SAT trouve la solution suivante :



La solution proposée a un pic d'énergie de 12, causé par l'ordonnancement des tâches o_1, o_2 et o_4. On ajoute donc les contraintes suivantes au modèle, pour empêcher désormais ces trois tâches d'être actives à une même période :

$$\overline{A_{o_1,t}} \vee \overline{A_{o_2,t}} \vee \overline{A_{o_4,t}} \qquad \forall t \in T$$

On relance ensuite le solveur SAT et, à la deuxième itération, le solveur SAT trouve la solution suivante :



Instance	Données			Résultat PLNE			Résultat SAT		
	n	m	c	Pic	Temps	Statut	Pic	Temps	Statut
mertens-1	7	6	6	**104**	**0.01 s**	Optimal	**104**	0.227 s	Optimal
mertens-2	7	2	18	**49**	**0.09 s**	Optimal	**49**	0.238 s	Optimal
bowman-1	8	5	20	**164**	**0.05 s**	Optimal	**164**	0.226 s	Optimal
jaeschke-1	9	8	6	**248**	**0.02 s**	Optimal	**248**	0.21 s	Optimal
jaeschke-2	9	3	18	**78**	**0.16 s**	Optimal	**78**	0.279 s	Optimal
jackson-1	11	8	7	**179**	**0.08 s**	Optimal	**179**	0.254 s	Optimal
jackson-2	11	2	94	**65**	1.05 s	Optimal	**65**	**0.502 s**	Optimal
mansoor-1	11	4	48	**145**	0.69 s	Optimal	**145**	**0.379 s**	Optimal
mansoor-2	11	2	94	**77**	5.03 s	Optimal	**77**	**0.563 s**	Optimal
mitchell-1	21	8	14	**221**	1.44 s	Optimal	**221**	**0.6 s**	Optimal
mitchell-2	21	3	39	**85**	427.94 s	Optimal	**85**	**9.884 s**	Optimal
roszieg-1	25	10	14	**254**	104.36 s	Optimal	**254**	**7.8 s**	Optimal
roszieg-2	25	4	32	**117**	163.59 s	Optimal	**117**	**5.653 s**	Optimal
heskiaoff-1	28	8	138	≤ 290	$\geq$ 3600 s		**251**	**196.693 s**	Optimal
heskiaoff-2	28	3	342		$\geq$ **3600 s**		$\leq$ **107**	$\geq$ **3600 s**	
buxey-1	29	14	25	$\leq$ **292**	$\geq$ **3600 s**		$\leq$ **292**	$\geq$ **3600 s**	
buxey-2	29	7	47	≤ 350	$\geq$ 3600 s		**172**	**137.46 s**	Optimal
sawyer-1	30	14	25	**395**	157 s	Optimal	**395**	**2.044 s**	Optimal
sawyer-2	30	7	47		$\geq$ 3600 s		**214**	**257.916 s**	Optimal
gunther-1	35	14	40	**394**	2297 s	Optimal	**394**	**2.131 s**	Optimal
gunther-2	35	9	54		$\geq$ 3600 s		**295**	**6.081 s**	Optimal

TABLE 1 – Comparaison des approches PLNE et SAT sur quelques instances

La solution proposée a un pic d'énergie de 10, causé par l'ordonnancement des tâches o_1, o_3 et o_4. On ajoute donc les contraintes suivantes au modèle, pour empêcher désormais ces trois tâches d'être actives à une même période :

$$\overline{A_{o_1,t}} \vee \overline{A_{o_3,t}} \vee \overline{A_{o_4,t}} \qquad \forall t \in T$$

On relance ensuite le solveur SAT et, à la troisième itération, le solveur SAT trouve que le problème ne possède pas de solution. La meilleure solution proposée jusqu'à maintenant, qui est ici la solution proposée à la deuxième itération, est donc la solution optimale, avec un pic d'énergie de 10.

5 Expérimentations

Nous avons expérimenté notre algorithme sur plusieurs instances de la littérature, avec pour chaque instance les résultats obtenus en implémentant le modèle existant sur CPLEX [3] et les résultats obtenus avec notre algorithme. Le problème SAT sous-jacent est résolu grâce au solveur Sat4j [1]. On donne, pour chaque instance, le nombre de tâches n, de machines m, le temps de cycle c et, pour chaque méthode, le pic d'énergie de la meilleure solution calculée, la durée d'exécution (au plus 1 heure) et son statut (optimal ou non). On constate dans la table 1 que l'approche proposée permet de trouver la solution optimale sur plus d'instances (19 instances sur 21) qu'avec l'approche ILP connue (15 sur 21).

6 Conclusion

Dans cet article, on a étudié le problème SALB3PM. On a proposé une approche qui fait appel itérativement au problème SAT pour résoudre ce problème. Cette approche a été comparée avec une approche par programmation linéaire en nombres entiers existante, et donne des résultats préliminaires prometteurs. Ces résultats préliminaires doivent être confirmés en offrant le même niveau de finesse de modélisation pour les deux approches et en comparant avec les récentes améliorations du modèle linéaire [2]. Il faudra réfléchir également à comment réduire la taille du modèle SAT initial et comment mieux gérer la fonction objectif. Enfin, il serait intéressant d'essayer d'hybrider ces approches entre elles ou avec d'autres approches heuristiques [5].

Références

[1] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7 :59–64, 2010.

[2] Paolo Gianessi and Xavier Delorme. A new ILP Model for a Line Balancing Problem with Minimization of Power Peak. In *31st European Conference on Operational Research (EURO-2021)*, 2021.

[3] Paolo Gianessi, Xavier Delorme, and Oussama Masmoudi. Simple Assembly Line Balancing Problem with Power Peak Minimization. In *IFIP International Conference on Advances in Production Management Systems (APMS)*, 2019.

[4] Matthieu Py and Arnauld Tuyaba. Résolution du problème SALB3PM grâce à une approche SAT. In *24ème édition du congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF)*, 2023.

[5] Arnauld Tuyaba, Laurent Deroussi, Nathalie Grangeon, and Sylvie Norre. Prise en compte de la consommation énergétique dans l'équilibrage de lignes d'assemblage. In *24ème édition du congrès annuel de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF)*, 2023.

JFPC 3 - 4 juillet 2023, 14h40 - 16h10

Extraction de Motifs d'Intervalles Fermés en utilisant la Programmation Par Contraintes

D. Bekkoucha[1], A. Ouali[1], J. Reynaud[1], B. Crémilleux[1], P. Boizumault[1], A. Beauchamp[2]

[1] Normandie Univ, UNICAEN, ENSICAEN, CNRS, GREYC, 14000 Caen, France
[2] Université d'Orléans, LIFO

[1]prénom.nom@unicaen.fr
[2]prénom.nom@univ-orleans.fr

Résumé

De nombreuses approches en programmation par contraintes (PPC) ont été proposées pour extraire des motifs à partir des données binaires. Ces approches nécessitent une étape préalable de discrétisation binaire pour extraire des motifs sur des données numériques. Cette transformation entraîne souvent une perte d'information ou d'efficacité. A contrario, nous proposons une approche déclarative en définissant une modélisation PPC afin d'extraire directement des motifs d'intervalles fermés à partir des données numériques. La binarisation de ces données est effectuée à la volée afin de préserver l'information originelle. De plus, le modèle proposé peut facilement être étendu avec les modèles existants afin d'extraire des motifs hétérogènes. Les expérimentations menées sur différents jeux de données montrent l'efficacité de notre approche par rapport aux approches déclaratives et sa capacité à découvrir des motifs intéressants.

Mots-clés

Motifs d'intervalles fermés, Programmation par contraintes, Structures de patrons, Motifs hétérogènes.

Abstract

Recent years have seen the rise of constraint programming (CP) to address pattern mining tasks. To deal with numerical data, many of the approaches require a prior discretization of the attributes by using a scaling procedure. Transforming numerical attributes into binary ones leads either to a loss of information or of efficiency. By contrast, we propose to mine on the fly closed interval patterns directly from numerical data in a declarative and efficient way without a binarization step beforehand. Our approach takes advantage of the general CP framework to directly model and mine Closed Interval Patterns from numerical data. Interestingly, we show how the CP model that we define can be easily extended to other data mining tasks such as mining heterogeneous patterns. This is possible by combining our model with other CP models. Experiments conducted on various datasets demonstrate the computational effectiveness of our proposition and its ability to discover meaningful patterns.

Keywords

Closed Interval Patterns, Constraint Programming, Pattern Structures, Heterogenous patterns.

1 Introduction

L'extraction de motifs à partir de données numériques reste un défi majeur, même si différentes approches existent [18, 17] la plupart des approches reposent sur une étape de discrétisation binaire (binarisation) préalable à l'extraction des motifs. Le principe est de diviser l'ensemble des valeurs d'un attribut d'un jeu de données en plusieurs intervalles de valeurs appelés *bins*. Ce processus peut être réalisé à l'aide de techniques supervisées ou non-supervisées qui exploitent des informations telles que des classes, des tests statistiques, l'entropie, etc. [4] ou suivant une approche bayésienne [1]. Les *bins* sont ensuite utilisés comme attributs binaires (i.e. des items) sur l'ensemble des objets pour définir et extraire des motifs. Malheureusement, ces techniques sont susceptibles d'engendrer une perte d'information due à la discrétisation des données numériques. Pour pallier ce problème, Kaytoue et al. [8] ont proposé `MinIntChange`, une méthode dédiée à l'extraction directe (i.e. sans discrétisation) de motifs d'intervalles fermés. Les motifs extraits du jeu de données binaires construit par `MinIntChange` couvrent tous les motifs d'intervalles possibles du jeu de données numériques d'origine. Cependant `MinIntChange` considère uniquement l'extraction de motifs d'intervalles de motifs numériques alors que l'aspect déclaratif de notre méthode permet de traiter d'autres types de motifs comme les motifs hétérogènes.

Cet article porte sur la tâche générale d'extraction de motifs dans des données numériques afin de découvrir de nouvelles connaissances. Notre travail est régi par les deux objectifs suivants. Premièrement, nous voulons conserver l'ensemble de l'information originelle exprimée par les données numériques. Deuxièmement, nous nous plaçons dans un cadre déclaratif et plus précisément dans celui de la Programmation Par Contraintes (PPC) afin de concevoir une méthode capable de s'adapter aux différentes variantes de cette tâche générale. Le paradigme déclaratif permet de spécifier des tâches de fouille de motifs de manière

simple en utilisant des primitives générales de satisfaction de contraintes [10, 7]. Il est plus facile de combiner ces primitives (et d'en incorporer de nouvelles) pour écrire des requêtes traitant de tâches de fouille complexes. Suivre ces deux principes conduit à bénéficier du paradigme déclaratif pour traiter des données numériques.

Dans cet article, nous présentons une nouvelle méthode déclarative pour découvrir des motifs à partir de données numériques, appelée `CP4CIP` (Constraint Programming For Closed Interval Patterns). Nous définissons un modèle PPC pour exprimer les motifs d'intervalles. Nous démontrons comment les motifs d'intervalles fermés peuvent être directement extraits des données numériques avec ce modèle. Une idée clé de notre proposition est d'effectuer le processus de binarisation à la volée selon les valeurs des données. Cela permet de préserver l'information originelle des données numériques.

À notre connaissance `CP4CIP` est la première méthode déclarative capable de découvrir des motifs directement à partir de données numériques. Un autre point important est que `CP4CIP` peut être facilement étendu pour découvrir d'autres types de motifs y compris à partir de données hétérogènes. À titre d'illustration, nous montrons comment une simple combinaison de `CP4CIP` et de `CP4IM` [15] (un modèle PPC reconnu pour extraire des motifs ensemblistes sur les données binaires) permet d'exploiter efficacement des motifs hétérogènes combinant des données numériques et binaires. L'extraction de tels motifs revient à considérer conjointement un contexte numérique et un contexte binaire. Cela illustre la flexibilité de la PPC pour traiter différents types de motifs à partir de données mixtes.

Ce article est organisé comme suit. La section 2 présente les travaux connexes tandis que la section 3 rappelle les notions de base en fouille de données numériques et en programmation par contraintes. La section 4 détaille nos modèles en PPC pour extraire des motifs d'intervalles fermés et des motifs hétérogènes. La section 5 décrit les expérimentations et reporte les résultats. Enfin, nous concluons l'article dans la section 6.

2 Travaux connexes

La fouille de motifs dans des données numériques a commencé par l'extraction des règles d'association quantitatives [18, 17] et différents travaux se sont succédés depuis [16]. Beaucoup d'entre eux sont fondés sur une approche en deux étapes où chaque attribut numérique est discrétisé selon certaines fonctions d'intérêt, comme par exemple le support ou les valeurs de classe. Ensuite, des motifs sont extraits des données discrétisées. Cette approche est univariée (un attribut est discrétisé sans prendre en compte les autres attributs) et conduit à une perte d'information. Toujours dans le domaine des règles d'association, la méthode `QuantMiner` [16] utilise un algorithme génétique pour découvrir de bons intervalles pour les attributs numériques dans les règles d'association en optimisant à la fois le support et la confiance.

Les tuiles et les sous-groupes sont des types de motifs très communs en fouille de données. Kontonasios et al. [9] utilisent un modèle d'entropie maximale pour extraire d'une manière itérative les tuiles intéressantes en fonction d'un modèle de fond. Cependant, ce modèle requiert des valeurs attendues suivant certaines connaissances préalables alors que, dans notre travail, nous considérons des valeurs absolues par rapport à des seuils donnés. Dans le domaine de la découverte de sous-groupes, Nguyen et Vreeken [14] proposent un processus de discrétisation dont l'objectif est de maximiser la qualité moyenne des motifs. Dans [11, 19], une approche basée sur la longueur minimale de description est utilisée pour découvrir des motifs utiles et renvoyer un ensemble de motifs non redondants se chevauchant avec des bornes bien définies. En considérant la notion de motifs à intervalles fermés, la méthode `OSMIND` [13] trouve des sous-groupes optimaux suivant une mesure d'intérêt dans des données purement numériques. Meeng et Knobbe [12] effectuent une comparaison approfondie des méthodes existantes pour traiter les attributs numériques dans les sous-groupes. Notre travail tire avantage du principe des intervalles fermés et ne se limite pas aux sous-groupes.

Afin de chercher à la volée des intervalles intéressants sur des données numériques, Kaytoue et al. ont proposé `MinIntChange` [8], une méthode pour extraire d'une manière exhaustive des motifs numériques en utilisant le cadre de l'Analyse Formelle de Concepts (AFC). Cependant cette méthode est dédiée aux motifs d'intervalles fermés. À notre connaissance, il n'existe pas de méthode déclarative pour découvrir des motifs directement à partir de données numériques. En revanche, différentes approches fondées sur la PPC ont été proposées pour extraire des motifs dans un contexte binaire. Pour comparer `CP4CIP` avec l'état de l'art des méthodes déclaratives, nous utilisons `CP4IM` [15] avec une méthode de binarisation des données numériques sans perte d'information *interordinal scaling* (voir la section 3.1). Le modèle `CP4CIP` diffère de `CP4IM` en deux points. Premièrement, la couverture d'un motif d'intervalles candidat est effectuée à la volée pendant le processus de résolution. Deuxièmement, la relation de fermeture est exprimée en utilisant un encodage différent de celui de `CP4IM`. En PPC, certains travaux utilisent des contraintes globales pour capturer les relations cachées entre un ensemble de variables afin d'améliorer l'efficacité. En considérant les motifs fermés, une contrainte globale a été proposée par [10] pour extraire des motifs fermés a partir de données binaires. Dans cet article, nous nous concentrons sur la partie modélisation en utilisant une solution prête à l'emploi pour les tâches d'extraction de motifs numériques.

3 Préliminaires

Une *base de données numérique* $\mathcal{N}$ est définie par un ensemble d'objets $\mathcal{G}$ dans lequel chaque objet est décrit par un ensemble d'attributs $\mathcal{M}$. Chaque attribut $m \in \mathcal{M}$ possède un domaine $\mathcal{W}_m$ de cardinalité finie contenant toutes les valeurs réelles possibles de m dans $\mathcal{N}$. Le tableau 1 montre un exemple jouet d'un ensemble de

données numériques contenant 5 objets et 3 attributs. Dans ce cadre, une base de données binaires est un cas particulier de base de données numériques ou les valeurs de tous les attributs sont binaires : $\mathcal{W}_m = \{0, 1\}, \forall m \in \mathcal{M}$.

	m_1	m_2	m_3
g_1	182	74	74
g_2	176	99	74
g_3	167	73	28
g_4	190	74	76
g_5	153	76	52

TABLE 1 – Un exemple de base de données numériques.

	Vue 1		Vue 2		
	m1	m2	x	y	z
1	3	5			
2	3	5	×		
3	4	4	×	×	

TABLE 2 – Contexte formel hétérogène composé de deux vues.

Dans les bases de données numériques, un motif est généralement représenté par une conjonction d'intervalles pour restreindre les valeurs sur les attributs numériques. Dans cet article, nous utilisons la notion de motifs d'intervalles comme définie dans [8]. Un *motif d'intervalle* est défini comme un vecteur d'intervalles $\mathcal{I} = \langle [\underline{w_m}, \overline{w_m}] \rangle_{\forall m \in \mathcal{M}}$, ou $\underline{w_m}, \overline{w_m} \in \mathcal{W}_m$. Chaque dimension du vecteur $\mathcal{I}$ correspond à un attribut suivant un ordre canonique sur l'ensemble des attributs $\mathcal{M}$. Un objet g est dans la **couverture** d'un motif d'intervalles $\mathcal{I}$ quand $w_{g,m} \in [\underline{w_m}, \overline{w_m}]$, $\forall m \in \mathcal{M}$. La **fréquence** de $\mathcal{I}$, dénotée par *freq*$(\mathcal{I})$, est le cardinal de la couverture de $\mathcal{I}$.

Par exemple, $\mathcal{I} = \langle [176, 190], [73, 99], [74, 76] \rangle$ est un motif d'intervalles couvrant les objets $\{g_1, g_2, g_4\}$ dans le tableau 1, avec *freq*$(\mathcal{I}) = 3$.

3.1 Fouille de motifs d'intervalles fermés

L'analyse formelle de concepts (AFC) [5] est une théorie appliquée qui se concentre sur l'énumération de collections compactes de motifs fermés afin d'éviter la redondance dans les données binaires. Un contexte formel est défini comme un triplet $(\mathcal{G}, \mathcal{M}, \mathcal{R})$, où $\mathcal{R} \subseteq \mathcal{G} \times \mathcal{M}$ est une relation binaire entre $\mathcal{G}$ et $\mathcal{M}$. On utilise $g \mathcal{R} m$ pour exprimer qu'un objet $g \in \mathcal{G}$ est en relation $\mathcal{R}$ avec un attribut $m \in \mathcal{M}$.

Soit $M \subseteq \mathcal{M}$, un opérateur d'extension $ext(M) = \{g \in \mathcal{G} | \forall m \in M, g \mathcal{R} m\}$ est l'ensemble des objets contenant tous les éléments $\mathcal{M}$. Soit $G \subseteq \mathcal{G}$, un opérateur d'intention $int(G) = \{m \in \mathcal{M} | \forall g \in G, g \mathcal{R} m\}$ est l'ensemble des éléments contenus dans tous les objets G.

Une paire sous la forme $(G = ext(M), M = int(G))$ est un concept formel, *i.e.* un motif fermé.

Interordinal Scaling est une méthode utilisée pour binariser des bases de données numériques afin d'extraire des motifs fermés du contexte binaire résultant, ces motifs sont interprétés comme des motifs d'intervalles fermés dans une seconde étape. Cette approche préserve toute l'information contenue dans les données numériques, en créant des paires d'attributs sous la forme suivante : $m \leq w_{g,m}; m \geq w_{g,m}, \forall m \in \mathcal{M}, \forall g \in \mathcal{G}$. Chaque élément de ces paires est utilisé comme un attribut binaire sur l'ensemble des objets. La valeur de l'attribut sur chaque objet est fixée à 1 si la condition est remplie, sinon la valeur est fixée à 0. Malgré la préservation complète de l'information, cette approche produit un énorme ensemble de données ayant un nombre d'attributs s'élevant à $\sum_{m \in \mathcal{M}} 2|\mathcal{W}_m|$. Dans notre travail, nous nous appuyons sur les notions issues des représentations condensées de motifs [2] telles que les motifs fermés pour calculer efficacement ces motifs qui résument l'ensemble de l'information.

Les structures de motifs d'intervalles [6] sont une généralisation de l'AFC pour traiter des données plus complexes telles que des données numériques, des graphes, des séquences, etc. Une structure de motifs est formellement définie comme un triplet $(\mathcal{G}, (D, \sqcap), \delta)$, où $(D, \sqcap)$ est l'inf-demi-treillis de descriptions donné dans un espace d'attributs de $|\mathcal{M}|$ dimensions et ordonné par l'opérateur de similarité $\sqcap$, et $\delta : \mathcal{G} \rightarrow D$ est une application qui associe chaque objet à sa description. Dans le cadre des structures de motifs d'intervalles, un objet $g \in \mathcal{G}$ est décrit par un vecteur d'intervalles comme détaillé précédemment. Soit $\mathcal{I}_1, \mathcal{I}_2 \in D$ deux motifs d'intervalles tels que $\mathcal{I}_1 = \langle [\underline{v_m^1}, \overline{v_m^1}] \rangle_{\forall m \in \mathcal{M}}$ et $\mathcal{I}_2 = \langle [\underline{v_m^2}, \overline{v_m^2}] \rangle_{\forall m \in \mathcal{M}}$. L'opérateur de similarité $\sqcap$ est appliqué aux descriptions des objets et retourne l'enveloppe convexe définie par :

$$\mathcal{I}_1 \sqcap \mathcal{I}_2 = \langle\ [min(\underline{v_m^1}, \underline{v_m^2}), \quad max(\overline{v_m^1}, \overline{v_m^2})]\ \rangle_{\forall m \in \mathcal{M}}$$

L'intersection de motifs dans l'AFC a les propriétés d'un infimum dans un demi-treillis, d'où l'idée d'ordonner les motifs en fonction de la relation : $\mathcal{I}_1 \sqsubseteq \mathcal{I}_2 \Leftrightarrow \mathcal{I}_1 \sqcap \mathcal{I}_2 = \mathcal{I}_1$. La connexion de Galois entre $(2^{\mathcal{G}}, \subseteq)$ et $(D, \sqcap)$ est définie comme suit :

$$\begin{cases} G^{\square} = \bigsqcap_{g \in G} \delta(g), \text{ pour } G \subseteq \mathcal{G}, & \text{et} \\ d^{\square} = \{g \in \mathcal{G} | d \sqsubseteq \delta(g)\}, \text{ pour } d \in (D, \sqcap) \end{cases}$$

$G^{\square}$ est la description commune à tous les objets de G, et $d^{\square}$ est l'ensemble des objets dont la description est subsumée par d. Un motif d'intervalles fermés, également appelé concept du contexte $(\mathcal{G}, (D, \sqcap), \delta)$ est la paire (G, d) tel que $G^{\square} = d$ et $d^{\square} = G$. Par exemple, $(\{g_1, g_2, g_4, g_5\}, \langle [176, 190], [74, 99], [74, 76] \rangle)$ est un motif d'intervalles fermés dans le tableau 1.

Les structures de motifs hétérogènes [3], sont des structures de motifs permettant de combiner différents types de données. Par exemple, considérons des données binaires ainsi que des données numériques représentées dans le tableau 2. Ensuite, nommons chaque type de données une *vue*. Fondamentalement, chaque vue correspond à un contexte formel ; les concepts hétérogènes résultent

de "l'intersection" des concepts formels de chaque vue. Formellement, étant donné deux structures de motifs $(\mathcal{G}, (D_1, \sqcap^1), \delta_1)$ et $(\mathcal{G}, (D_2, \sqcap^2), \delta_2)$, la structure hétérogène du motif $(\mathcal{G}, (D_1 \times D_2, \sqcap), \delta)$ est définie comme suit :

$$\begin{cases} G^{\square} = (\bigsqcap^1_{g \in G} \delta_1(g), \bigsqcap^2_{g \in G} \delta_2(g)), & \text{pour } G \subseteq \mathcal{G} \\ (d_1, d_2)^{\square} = \{g \in \mathcal{G} | d_1 \sqsubseteq \delta_1(g), d_2 \sqsubseteq \delta_2(g)\}, & \text{pour } d_1 \in (D_1, \sqcap) \text{ et } d_2 \in (D_2, \sqcap) \end{cases}$$

3.2 Programmation par contraintes

Un CSP consiste en un ensemble de variables $X = \{X_1, \ldots, X_n\}$, un ensemble de domaines $\mathcal{D}$ faisant correspondre chaque variable $X_i \in X$ à un ensemble fini de valeurs possibles $\mathcal{D}(X_i)$, et un ensemble de contraintes $\mathcal{C}$ sur X. Une contrainte $c \in \mathcal{C}$ est une relation spécifiant les combinaisons de valeurs autorisées pour ses variables $X(c)$. Une affectation sur un ensemble $Y \subseteq X$ de variables est une correspondance entre les variables de Y et les valeurs de leurs domaines. Une solution est une affectation sur X satisfaisant toutes les contraintes. Les solveurs de PPC utilisent généralement la recherche arborescente par retour arrière pour explorer l'espace de recherche des affectations partielles et tentent de les étendre à des affectations cohérentes dans le but de trouver des solutions. La principale technique utilisée pour accélérer la recherche est la propagation des contraintes par un algorithme de filtrage qui permet de réduire la taille des domaines.

4 Modèle PPC pour l'extraction de motifs d'intervalles fermés

Dans cette section, nous présentons une modélisation PPC permettant d'extraire directement des motifs d'intervalles fermés de données numériques, `CP4CIP` (Constraint Programming For Closed Interval Patterns), qui permet de formuler des requêtes exécutant des tâches de fouille complexes. Dans ce qui suit, nous montrons comment le problème d'extraction de motifs d'intervalles fermés est exprimé sous la forme d'un modèle PPC. L'ensemble des intervalles et l'ensemble des objets d'un motif d'intervalles fermés sont exprimés par les variables et les contraintes décrites dans cette section.

4.1 Motifs d'intervalles

On dénote par $[X_m]$ un intervalle sur un attribut $m \in \mathcal{M}$ tel que $[X_m] = \{w \in \mathcal{W}_m : \underline{X_m} \leq w \leq \overline{X_m}\}$. Dans notre modèle nous définissons deux variables pour chaque attribut : une variable pour la borne inférieure $\underline{X_m}$ et une variable pour la borne supérieure $\overline{X_m}$ où $\mathcal{D}(\underline{X_m}) = \mathcal{D}(\overline{X_m}) = \mathcal{W}_m$. Un motif d'intervalles fermés est trouvé en affectant les variables $\underline{X_m}$ et $\overline{X_m}$ à une valeur dans $\mathcal{W}_m$ tel que $\underline{X_m} \leq \overline{X_m}, \forall m \in \mathcal{M}$.

Exemple. *Le motif* $< [176, 190], [74, 99], [74, 76] >$ *est représenté par l'instanciation* $\{\underline{x_1} = 176, \overline{x_1} = 190, \underline{x_2} = 74, \overline{x_2} = 99, \underline{x_3} = 74, \overline{x_3} = 76\}$.

L'extension d'un motif d'intervalles est exprimée en définissant une variable binaire G_g pour chaque objet $g \in \mathcal{G}$ dans un jeu de données numériques pour chaque objet $g \in \mathcal{G}$, où $\mathcal{D}(G_g) = \{0, 1\}$. La variable G_g est fixée à 1 ssi l'objet g est couvert par le motif. Les variables $\underline{X_m}$, $\overline{X_m}$ définissent le vecteur d'intervalles d'un motif d'intervalles sur chaque attribut $m \in \mathcal{M}$, et les variables G_g son ensemble d'objets correspondant dans $\mathcal{G}$.

4.2 Contraintes

Dans cette sous-section, nous décrivons les contraintes modélisant un motif d'intervalles fermé.

Contraintes de couverture $d^{\square}$. Un objet est couvert par un motif d'intervalles si et seulement si toutes les valeurs de ses attributs se trouvent dans les intervalles. Pour formuler la couverture, nous introduisons dans notre modèle des variables binaires $B_{g,m}$, où $\mathcal{D}(B_{g,m}) = \{0, 1\}$ pour chaque valeur m et chaque objet g dans la base de données tel que la contrainte réifiée (1) est satisfaite.

$$B_{g,m} = 1 \iff \underline{X_m} \leq w_{g,m} \leq \overline{X_m}, \ \forall m \in \mathcal{M}, \forall g \in \mathcal{G} \quad (1)$$

$$G_g = 1 \iff \sum_{m \in \mathcal{M}} B_{g,m} = |\mathcal{M}|, \ \forall g \in \mathcal{G} \quad (2)$$

La contrainte (2) utilise les variables $B_{g,m}$ pour imposer la couverture sur chaque objet. Un objet $g \in \mathcal{G}$ est couvert ssi la valeur de chaque attribut m de l'objet g est délimitée par l'intervalle $[X_m]$.

Exemple. *D'après le contexte formel ci-dessus (table 1), l'instanciation des variables de couverture pour le motif* $< [176, 190], [74, 99], [74, 76] >$ *est la suivante :* $\{G_1 = 1, G_2 = 1, G_3 = 0, G_4 = 1, G_5 = 0\}$

Contraintes de fermeture $G^{\square} = d$ et $d^{\square} = G$. La fermeture exige que chaque intervalle associé à chaque attribut contienne toutes les valeurs des objets couverts, tandis que chaque valeur d'un objet non couvert doit être en dehors de l'intervalle. Soit $\mathcal{W}^{\uparrow}_m$ (resp. $\mathcal{W}^{\downarrow}_m$) la valeur maximale (resp. minimale) sur les objets de l'attribut m, la relation de fermeture peut être exprimée dans notre modèle en introduisant les nouvelles variables $\underline{H}_{g,m}$ et $\overline{H}_{g,m}$, où $\mathcal{D}(\underline{H_{g,m}}) = \mathcal{W}_m \cup \{\mathcal{W}^{\uparrow}_m + 1\}$, et $\mathcal{D}(\overline{H_{g,m}}) = \mathcal{W}_m \cup \{\mathcal{W}^{\downarrow}_m - 1\}$. La valeur supérieure $\{\mathcal{W}^{\uparrow}_m + 1\}$ (resp. valeur inférieure $\{\mathcal{W}^{\downarrow}_m - 1\}$) est ajouté dans le domaine de $\underline{H}_{g,m}$ (resp. $\overline{H}_{g,m}$) afin d'éviter de sélectionner le minimum (resp. le maximum) sur les objets non couverts. La fermeture du motif d'intervalles est exprimée par les contraintes réifiées (3-8).

$$\forall g \in G, m \in M, G_g = 1 \implies \underline{H_{g,m}} = w_{g,m} \quad (3)$$

$$\forall g \in G, m \in M, G_g = 0 \implies \underline{H_{g,m}} = \mathcal{W}^{\uparrow}_m + 1 \quad (4)$$

$$\forall g \in G, m \in M, \underline{X_m} = min(\underline{H_{1,m}}, \underline{H_{2,m}}, ..., \underline{H_{|\mathcal{G}|,m}}) \quad (5)$$

Pour trouver la valeur minimale sur les objets couverts, nous introduisons les contraintes (3-5). Nous utilisons pour chaque attribut m une contrainte minimale sur les variables

de l'ensemble $\{\underline{H_{g,m}}, \forall g \in G\}$. Les variables $H_{g,m}$ des objets non couverts ont une valeur supérieure à toutes les valeurs des données. Ainsi, le minimum ne peut pas être sélectionné sur les objets non couverts.

$$\forall g \in G, m \in M, G_g = 1 \implies \overline{H_{g,m}} = w_{g,m} \quad (6)$$

$$\forall g \in G, m \in M, G_g = 0 \implies \overline{H_{g,m}} = \mathcal{W}_m^{\downarrow} - 1 \quad (7)$$

$$\forall g \in G, m \in M, \overline{X_m} = max(\overline{H_{1,m}}, \overline{H_{2,m}}, ..., \overline{H_{|\mathcal{G}|,m}}) \quad (8)$$

De la même façon, pour trouver la valeur maximale sur les objets couverts (6-8), nous utilisons pour chaque attribut m une contrainte maximale sur l'ensemble des variables $\{\overline{H_{g,m}}, \forall g \in G\}$. Les variables $\overline{H_{g,m}}$ des objets non couverts ont une valeur inférieure à toutes les valeurs des données. Ainsi, le maximum ne peut pas être sélectionné parmi les objets non couverts.
Les contraintes (3-5) obligent tous domaines des variables d'intervalle ($\underline{X_m}$ et $\overline{X_m}$) à être fermées sur les valeurs des objets couverts.

Exemple. *Les domaines des variables de fermeture pour l'attribut m_1 de l'objet g_1 sont $D(\underline{H_{1,1}}) = \{182, 191\}$ et $D(\overline{H_{1,1}}) = \{152, 182\}$. Le tableau 3 illustre les valeurs prises par les variables $\underline{H_{g,m}}$ (en haut) et $\overline{H_{g,m}}$ (en bas) pour le motif d'intervalles $<$ $[176, 190], [74, 99], [74, 76]$ $>$.* Le minimum sur les valeurs $\{\underline{H_{1,1}}, \underline{H_{2,1}}, \underline{H_{3,1}}, \underline{H_{4,1}}, \underline{H_{5,1}}\}$ est égale à 176.
Le maximum sur les valeurs $\{\overline{H_{1,1}}, \overline{H_{2,1}}, \overline{H_{3,1}}, \overline{H_{4,1}}, \overline{H_{5,1}}\}$ est égale à 190. Le même calcul tient sur les autres attributs.

	$\underline{H_{g,1}}$	$\underline{H_{g,2}}$	$\underline{H_{g,3}}$
g_1	182	74	74
g_2	176	99	74
g_3	191	100	77
g_4	190	74	76
g_5	191	100	77

	$\overline{H_{g,1}}$	$\overline{H_{g,2}}$	$\overline{H_{g,3}}$
g_1	182	74	74
g_2	176	99	74
g_3	152	72	27
g_4	190	74	76
g_5	152	72	27

TABLE 3 – Matrice des valeurs des variables de fermeture inférieure (haut) et supérieure (bas)

Autres contraintes. Notre modèle PPC peut être étendu en utilisant de nouvelles contraintes sur les intervalles et/ou les objets. Les avantages de notre modèle sont doubles : (i) les contraintes supplémentaires sont directement exprimées sur les intervalles plutôt que sur les binaires où un effort de modélisation plus important est nécessaire. (ii) d'autres contraintes complexes peuvent être introduites sur la couverture des objets. Une extension du modèle PPC actuel est donnée dans la section suivante.

4.3 Extension du modèle pour la fouille de motifs hétérogènes

Dans cette section, nous montrons comment `CP4CIP` peut être étendu de façon naturelle pour extraire des motifs hétérogènes tels que ceux présentés dans la section 3.1. Cette tâche est réalisée en combinant le modèle `CP4IM` [15], qui est restreint aux données binaires, avec notre modèle `CP4CIP`. Pour un jeu de données binaires $\mathcal{B}$ composé d'un ensemble d'objets $\mathcal{G}$ et d'un ensemble d'éléments (attributs binaires) $\mathcal{L}$, la modélisation de `CP4IM` utilise deux ensembles de variables booléennes P et G : (i) Variables des items $\{P_1, P_2, ..., P_{|\mathcal{L}|}\}$, où $P_l = 1$ si et seulement si l'item $l \in \mathcal{L}$ est dans l'ensemble des items recherchés ; (ii) Variables d'objet $\{G_1, G_2, ..., G_{|G|}\}$, où $G_g = 1$ ssi l'objet $g \in \mathcal{G}$ est couvert par l'ensemble d'éléments recherché.

$$G_g = 1 \iff \sum_{l \in \mathcal{L}} P_l(1 - \mathcal{B}_{g,l}) = 0, \quad \forall g \in \mathcal{G}. \quad (9)$$

$$(P_l = 1) \iff \sum_{g \in \mathcal{G}} G_g(1 - \mathcal{B}_{g,l}) = 0, \quad \forall l \in \mathcal{L}. \quad (10)$$

`CP4IM` utilise les contraintes (9) et (10) pour faire en sorte que chaque motif solution soit fermé.
Notre modèle peut incorporer directement les contraintes `CP4IM` en utilisant les variables G_g qui sont liées aux variables $\underline{X_m}$ et $\overline{X_m}$ sur les attributs numériques et aux variables P_l sur les attributs binaires.

5 Expérimentations

Dans cette section, nous proposons un protocole expérimental pour évaluer deux aspects de notre approche.
Le premier aspect est l'avantage d'incorporer la binarisation à la volée dans le modèle PPC. À cet égard, nous comparons les performances (en termes de temps CPU) de `CP4CIP` à deux méthodes : (i) `CP4IM`, une méthode basée sur le modèle PPC pour la fouille sur des données binaires [15] en lui appliquant un *interordinal scaling* afin de conserver toute l'information (notons qu'il n'existe pas de concurrent selon le paradigme déclaratif sur des données numériques) ; (ii) les méthodes dédiées `MinIntChange` et `LCM-BIN` après application de la binarisation avec *interordinal scaling*.
Dans un second temps, afin d'illustrer la généricité de notre modèle PPC, nous évaluons la flexibilité de `CP4HP` pour la fouille de motifs hétérogènes. Pour cela, nous considérons une application où nous cherchons à caractériser les variables latentes produites par une analyse ACP sur un ensemble de documents en utilisant leurs mots-clés. `CP4HP` peut-il être utilisé pour extraire des motifs hétérogènes utiles qui pourraient expliquer les variables latentes ?

Jeu de données	Originale	
	#attributs	#objets
AP	5	135
BK	5	96
Cancer	9	116
CH	8	209
LW	10	189
NT	3	130
Yacht	7	308
	Interordinal scaled	
	#IS attributs	densité (%)
AP	1348	50.37
BK	626	50.79
Cancer	1800	50.50
CH	792	51.01
LW	506	51.97
NT	134	52.23
Yacht	644	51.08

TABLE 4 – Caractéristiques des jeux de données

Jeux de données. Nous avons sélectionné plusieurs jeux de données numériques difficiles pour les approches déclaratives et mis a disposition par l'Université de Bilkent [1] et qui ont été utilisés dans [8]. Deux autres jeux de données (Cancer et Yacht) ont également été sélectionnés dans les archives de l'UC Irvine. [2]. Les noms des jeux de données sont indiqués par les abréviations standards utilisées dans la base de données de l'Université Bilkent. Le tableau 4 résume les caractéristiques de chaque ensemble de données. Tous les jeux de données sélectionnés sont de tailles et de types différents. La plupart d'entre eux contiennent des valeurs réelles et l'un d'entre eux des valeurs négatives. Dans le cas où des objets ont des valeurs manquantes, nous supprimons complètement l'objet du jeu de données. Pour traiter efficacement les valeurs réelles dans les jeux de données avec l'API OR-tools, nous avons remplacé chaque valeur réelle par un nombre entier qui préserve l'ordre original.

Protocole expérimental. Nous avons implémenté et résolu les modèles PPC en utilisant directement l'API de OR-tools et son module CP-Solver v9.0 [3] au lieu de passer par un langage de modélisation tel que `MiniZinc`. Toutes les expérimentations ont été menées sur un processeur AMD Opteron 6282SE 2,6 GHz et 512 Go de RAM avec un timeout de 5 heures. Pour chaque jeu de données, nous avons diminué le seuil de fréquence (relative) jusqu'à ce qu'il soit impossible d'extraire tous les motifs d'intervalles fermés dans le temps/mémoire alloué. Le code binaire, bases de données et les résultats sont disponibles ici [4]. Pour les méthodes dédiées, nous avons utilisé le code public de `MinIntChange` et `LCM` v3.

1. `http://funapp.cs.bilkent.edu.tr/DataSets/`
2. `https://archive.ics.uci.edu/ml/datasets.php`
3. `https://github.com/google/or-tools/`
4. `https://github.com/oualiaek/cpforcip.git`

données	Freq. %	Temps CPU (secondes)			
		`CP4IM`	`CP4CIP`	`MinIntChange`	`LCM-BIN`
AP	80	630.41	28.47	5.35	**1.29**
	70	4,804.52	192.84	24.34	**13.12**
	60	14,287.49	559.66	74.70	**55.11**
	50	33,401.62	1,219.80	**135.18**	163.04
	20	TO	5,290.18	**464.30**	1,296.05
BK	80	1,431.73	267.95	15.15	**1.43**
	70	11,599.66	1,728.92	117.03	**13.20**
	60	TO	7,303.11	459.97	**51.25**
	50	TO	18,218.68	1,272.80	**157.22**
	20	TO	TO	5,262.99	**1,806.64**
Cancer	95	129.61	18.47	1.81	**0.06**
	94	361.22	46.17	7.63	**0.13**
	92	4,911.79	479.81	34.65	**1.57**
	90	20,623.85	1,815.01	139.64	**5.11**
	80	TO	TO	TO	**3,637.51**
CH	95	15.81	6.04	0.95	**0.01**
	90	432.80	89.01	10.50	**0.28**
	85	3,435.64	648.77	66.64	**2.35**
	80	TO	2,669.77	215.29	**9.31**
	50	TO	TO	21,278.92	**1,580.62**
LW	80	1,264.93	1,605.48	27.77	**1.55**
	70	10,119.56	9,708.12	162.18	**11.17**
	60	TO	31,266.18	647.13	**45.14**
	50	TO	TO	1,842.20	**144.58**
	20	TO	TO	8,861.99	**1,107.47**
NT	80	0.80	1.81	0.23	**0.01**
	50	6.61	10.81	0.63	**0.03**
	20	26.22	27.26	0.71	**0.16**
	10	40.58	31.85	1.06	**0.19**
	1	60.85	32.77	0.83	**0.35**
Yacht	80	34.07	99.36	0.92	**0.03**
	50	5,644.65	4,562.36	24.33	**3.79**
	40	20,781.77	9,643.05	58.94	**12.86**
	30	TO	18,136.41	125.37	**34.92**
	20	TO	30,097.10	226.56	**97.16**

Table (5) Temps de calcul des quatre méthodes sur les différents jeux de données. Le meilleur temps de calcul pour toutes les méthodes considérées est indiqué en gras. Le meilleur temps CPU pour les méthodes PPC est indiqué sur fond gris.

Extraction de motifs d'intervalles fermés. Dans le tableau 5, nous indiquons le temps CPU nécessaire pour trouver tous les motifs d'intervalles fermés en utilisant différents seuils de fréquence minimale pour chaque jeu de données. Pour les méthodes déclaratives, l'observation générale est que `CP4CIP` surpasse significativement `CP4IM` sur la plupart des jeux de données, en particulier lorsque le jeu de données est grand ou que le seuil de fréquence est bas. En termes de temps CPU où `CP4CIP` surpasse `CP4IM`, nous pouvons noter un facteur d'accélération moyen de 9,37 sur tous les jeux de données. Cependant, nous remarquons que `CP4IM` surpasse `CP4CIP` sur les jeux de données NT et Yacht lorsque le seuil de fréquence est élevé. Si l'on considère le jeu de données NT, on peut voir dans le tableau 4 que le nombre d'attributs binaires après le *interordinal scaling* est relativement faible par rapport aux autres jeux de données. De plus, des valeurs plus élevées des seuils de fréquence combinées à des jeux de données plus petits sont bénéfiques pour `CP4IM` puisqu'il utilise un

plus petit nombre de variables et de contraintes avec moins de solutions candidates. Ces observations illustrent l'avantage d'incorporer la binarisation dans CP4CIP.

Les implémentations de MinIntChange et LCM-BIN étant dans des langages différents (Java pour MinIntChange et C pour LCM-BIN) la comparaison de leurs temps CPU est à relativiser. Les résultats sont fournis pour avoir un aperçu de l'écart entre les méthodes déclaratives et les méthodes dédiées.

À partir de cette évaluation, nous pouvons voir l'intérêt d'utiliser CP4CIP lors de la fouille de jeux de données contenant pour chaque attribut un changement significatif sur les valeurs numériques observées. Plus ce changement est important, plus efficace est CP4CIP comparé a CP4IM. Par conséquent, il est plus efficace d'utiliser les contraintes PPC basées sur le type d'attributs malgré la capacité du modèle CP4CIP à extraire des motifs fermés où un item absent dans un motif est interprété comme [0,0] (l'item n'apparaît pas sur tous les objets couverts) ou [0,1] (l'item apparaît sur certains mais pas tous les objets couverts). Ainsi, il est plus efficace d'étendre CP4CIP en utilisant les contraintes de fermeture de CP4IM puisque les attributs sont déjà binaires. Cela motive l'utilisation de CP4HP qui combine les deux modèles pour exploiter efficacement les attributs hétérogènes.

Extraction de structures de motifs hétérogènes. Afin d'illustrer la généricité et l'intérêt de notre approche, nous considérons une application dans laquelle on cherche à caractériser des variables latentes afin d'expliquer les valeurs qu'elles prennent.

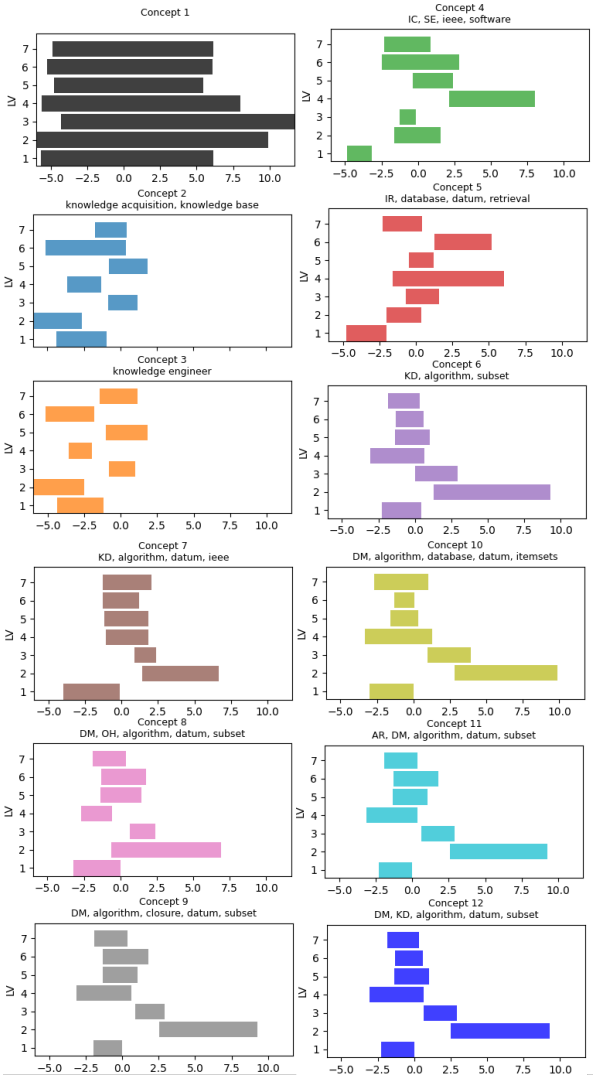
Grâce à des approches comme TF/IDF, nous sommes en mesure de construire une matrice où les lignes correspondent aux documents et les colonnes à un terme. La valeur $0 \leqslant c_{i,j} \leqslant 1$ à la ligne i et à la colonne j dépend de la représentativité du terme t_j pour le document d_i. L'utilisation de l'analyse en composantes principales (ACP) permet de réduire le nombre de dimensions de cette matrice. Chaque nouvelle dimension correspond à une variable latente, connue - dans les informations de recherche - pour représenter un *sujet*.

Malheureusement, ces thèmes ne sont liés à aucune signification réelle. Cependant, en considérant ces vecteurs avec les mots-clés des documents, on peut comprendre le "sens" de certaines parties de l'espace. Par exemple, si tous les documents ayant le mot-clé pattern mining sont dans l'intervalle $[3, 4]$ dans la première dimension, et s'il n'y a aucun autre document dans cet intervalle, on peut supposer que cet intervalle correspond au concept de pattern mining. À cette fin, nous construisons le contexte formel contenant à la fois les variables latentes et les mots-clés. Ensuite, nous profitons de CP4HP pour extraire des concepts hétérogènes. Enfin, ces concepts nous permettent de faire correspondre les mots-clés avec une partie de l'espace latent.

Nous avons utilisé l'API[5] d'ISTEX pour extraire un ensemble de articles de journaux traitant de l'analyse formelle de concepts. Nous avons obtenu 312 articles, ainsi que leurs mots-clés. À partir de ces mots-clés, nous effectuons une ACP afin de représenter chaque article comme un vecteur de 7 dimensions (*i.e.* 7 variables latentes). Ensuite, nous considérons les mots-clés fréquents (*i.e.* apparaissant dans plus de 10 articles, ce qui représente 161 mots-clés) et construisons le contexte hétérogène. Nous avons obtenu un contexte formel hétérogène avec 312 objets, 7 attributs dans la première vue et 161 attributs dans la deuxième vue. Pour chaque attribut de la première vue, le nombre de valeurs uniques est compris entre 304 et 307. La densité de la deuxième vue est de 0.08.

Nous avons utilisé CP4HP et nous avons obtenu des résultats en temps CPU de 186,05 secondes avec 9661 concepts, dont 12 concepts fréquents qui couvrent au moins $3,8\%$ des objets. Ils sont présentés dans la figure 1.

Un rectangle correspond à l'intervalle des valeurs d'une va-



AR : Association Rule ; DM : Datum Mining ; IC : International Conference ; IR : Information Retrieval ; KD : Knowledge Discovery ; OH : Other Hand ; SE : Software Engineering

FIGURE 1 – Concepts hétérogènes fréquents

5. https://api.istex.fr/

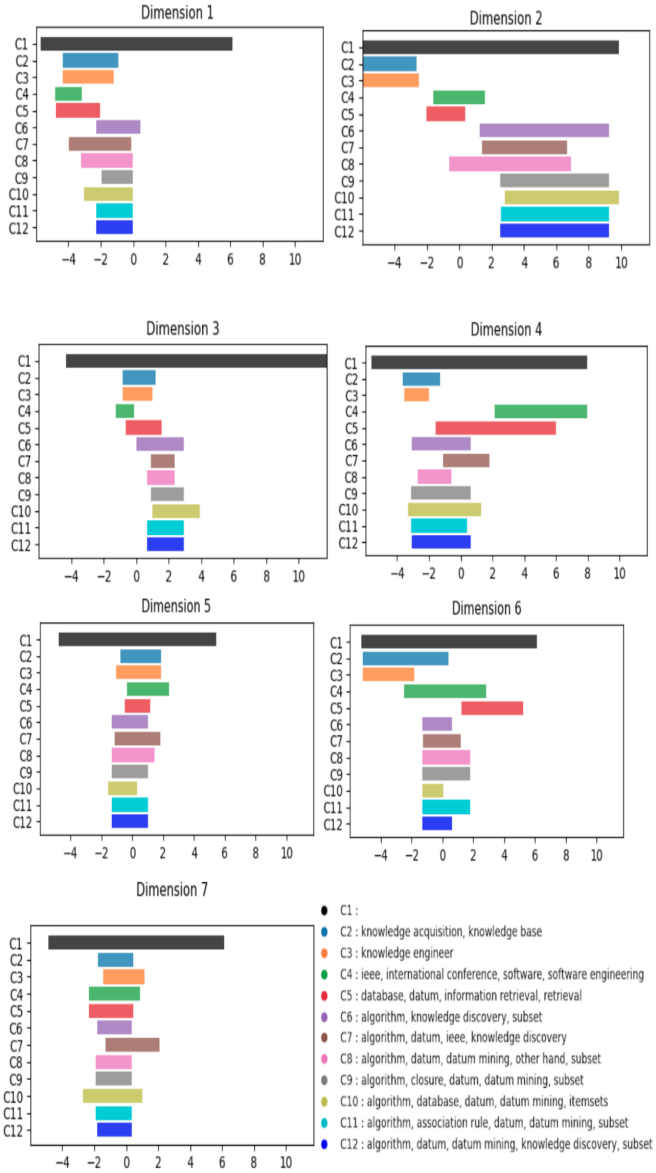


FIGURE 2 – Intervalles de concepts hétérogènes fréquents pour chaque variable latente

riable latente pour ce concept. Par exemple, le concept $C1$ correspond au concept le plus grand : ce concept couvre tous les objets, c'est-à-dire que ses intervalles couvrent toutes les valeurs possibles pour chaque dimension. Il n'y a pas de mot-clé associé à ce concept, car il n'existe pas de mot-clé apparaissant dans tous les articles. La figure fournit un support visuel pour observer les similarités entre les concepts. Avec ces concepts, les similarités dans les valeurs des variables latentes correspondent à des mots-clés similaires. Par exemple, nous observons des similarités entre les concepts qui incluent le mot-clé `algorithme`.

Dans la figure 2, nous comparons les concepts pour chaque dimension (c'est-à-dire le variable latente). Cela nous permet de visualiser la distribution des concepts.

Par exemple, si nous considérons la dimension 5, nous observons que tous les concepts représentés (sauf le "concept supérieur") sont dans $[-2, 2]$, ce qui signifie que cette dimension n'est pas utile pour distinguer les concepts. En revanche, si nous observons la dimension 6, les concepts C6 à C12 ont leurs intervalles entre $-1, 8$ et 2, ce qui n'est le cas d'aucun autre concept. Tous ces concepts ont le mot clé `algorithme`.

Dans la plupart des dimensions nous observons que les concepts fréquents (excepté le concept supérieur) ne couvrent pas la totalité des valeurs observées. Cela signifie que certaines parties de ces intervalles représentent des mots-clés qui n'apparaissent pas dans les concepts fréquents.

6 Conclusion

Cet article présente `CP4CIP`, une méthode déclarative permettant de découvrir des motifs d'intervalles à partir de données numériques sans discrétisation préalable. `CP4CIP` conserve la totalité de l'information originale exprimée par les données numériques en effectuant à la volée le processus de binarisation en fonction des données concrètes. `CP4CIP` tire parti du cadre général de la PPC pour modéliser et exploiter directement les motifs d'intervalles fermés. `CP4CIP` peut être facilement étendu pour découvrir d'autres types de motifs à partir de données variées comme extraire efficacement des motifs hétérogènes mélangeant des données numériques et binaires. Dans le cadre d'un travail futur, nous étudions l'amélioration des modèles PPC actuels afin d'obtenir un meilleur compromis entre flexibilité et efficacité.

7 Remerciements

Le premier auteur est financé par l'ANR et la Région Normandie dans le cadre du projet HAISCoDe.

Références

[1] M. Boullé. MODL : A bayes optimal discretization method for continuous attributes. *Mach. Learn.*, 65(1) :131–165, 2006.

[2] 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 *LNCS*, pages 64–80. Springer, 2005.

[3] Victor Codocedo and Amedeo Napoli. A Proposition for Combining Pattern Structures and Relational Concept Analysis. In *ICFCA*, 2014.

[4] J. Dougherty, R. Kohavi, and M. Sahami. Supervised and unsupervised discretization of continuous features. In *Machine learning : Proceeding of the twelfth international conference*, pages 194–202. M. Kaufmann, 1995.

[5] B. Ganter and R. Wille. *FCA - Mathematical Foundations*. Springer, 1999.

[6] Bernhard Ganter and Sergei O. Kuznetsov. Pattern structures and their projections. In *ICCS*, 2001.

[7] A. Hien, S. Loudni, N. Aribi, Y. Lebbah, M. E. A. Laghzaoui, A. Ouali, and A. Zimmermann. A relaxation-based approach for mining diverse closed patterns. In *ECML-PKDD, 2020.* Springer.

[8] Mehdi Kaytoue, Sergei Kuznetsov, and Amedeo Napoli. Revisiting numerical pattern mining with formal concept analysis. *IJCAI*, 2011.

[9] K.-N. Kontonasios, J. Vreeken, and T. De Bie. Maximum entropy models for iteratively identifying subjectively interesting structure in real-valued data. In *ECML/PKDD*, 2013.

[10] N. Lazaar, Y. Lebbah, S. Loudni, M. Maamar, V. Lemière, C. Bessiere, and P. Boizumault. A global constraint for closed frequent pattern mining. In *Int. Conf. on Principles and Practice of Constraint Programming*, 2016.

[11] T. Makhalova, Sergei O. Kuznetsov, and A. Napoli. Mint : MDL-based approach for Mining INTeresting Numerical Pattern Sets. *Data Min. Knowl. Discov.*, 36 :108–145, 2022.

[12] M. Meeng and A. J. Knobbe. For real : a thorough look at numeric attributes in subgroup discovery. *Data Min. Knowl. Discov.*, 35 :158–212, 2021.

[13] A. Millot, R. Cazabet, and J-F. Boulicaut. Optimal subgroup discovery in purely numerical data. In *PAKDD*, 2020.

[14] H. Vu Nguyen and J. Vreeken. Flexibly mining better subgroups. In *SDM*, 2016.

[15] L. De Raedt, T. Guns, and S. Nijssen. Constraint programming for data mining and machine learning. In *AAAI*, 2010.

[16] A. Salleb-Aouissi, C. Vrain, and C. Nortet. Quantminer : A genetic algorithm for mining quantitative association rules. In *IJCAI*, 2007.

[17] C. Song and T. Ge. Discovering and managing quantitative association rules. In *ACM Int. Conf. on CIKM*, 2013.

[18] R. Srikant and R. Agrawal. Mining quantitative association rules in large relational tables. In *ACM/SIGMOD*, 1996.

[19] J. Witteveen, W. Duivesteijn, A. J. Knobbe, and P. Grünwald. Realkrimp - finding hyperintervals that compress with MDL for real-valued data. In *IDA*, 2014.

Identification de la meilleure heuristique en satisfaction de contraintes*

F. Koriche[1], C. Lecoutre[1], A. Paparrizou[1], H. Wattez[2]

[1] CRIL, Univ. Artois & CNRS

[2] LIX CNRS, École Polytechnique, Institut Polytechnique de Paris

{koriche,lecoutre,paparrizou}@cril.fr, wattez@lix.polytechnique.fr

Résumé

Dans les problèmes de satisfaction de contraintes, l'heuristique de choix de variables occupe une place centrale en sélectionnant les variables sur lesquelles brancher pendant le processus de recherche avec retour-arrière. Comme de nombreuses heuristiques de branchement ont été proposées dans la littérature, un problème clé est d'identifier, parmi un ensemble d'heuristiques candidates, celle qui est la meilleure pour résoudre une instance de satisfaction de contraintes donnée. En se basant sur l'observation que les solveurs de contraintes modernes utilisent des séquences de redémarrage, le problème d'identification de la meilleure heuristique peut être représenté dans le contexte des bandits multi-bras comme un problème d'identification du meilleur bras non stochastique. En d'autres termes, pendant chaque run d'une séquence de redémarrage donnée, l'algorithme du bandit sélectionne une heuristique et reçoit une récompense pour cette heuristique avant de passer au run suivant. L'objectif est d'identifier la meilleure heuristique en utilisant peu de runs, et sans aucune hypothèse stochastique sur le solveur de contraintes. Dans cette étude, nous proposons une variante adaptative du Successive Halving qui exploite la suite de redémarrage universelle de Luby. Nous analysons la convergence de cet algorithme de bandit dans un cadre non stochastique, et nous démontrons son efficacité empirique sur divers benchmarks en satisfaction de contraintes.

1 Introduction

La satisfaction de contraintes est un problème récurrent qui se pose dans de nombreuses applications informatiques dont la bio-informatique, la configuration, la planification, l'ordonnancement et la validation de logiciels. Étant donné un ensemble de variables de décision et un ensemble de contraintes, chacune spécifiant une relation entre certaines variables, le problème de satisfaction de contraintes (CSP) consiste à trouver une affectation de toutes les variables qui satisfait toutes les contraintes. Pour faire face à ce problème NP-complet, les solveurs de contraintes combinent généralement la recherche avec retour-arrière et l'inférence afin d'explorer efficacement l'espace des assignations.

L'*heuristique de choix de variables* joue un rôle clé dans le problème de la satisfaction de contraintes en sélectionnant itérativement la prochaine variable sur laquelle brancher pendant la recherche avec retour-arrière. Le choix de la bonne stratégie de branchement pour une instance de CSP donnée peut accélérer considérablement la résolution, car différents ordonnancements de variables peuvent conduire à des arbres de recherche entièrement différents [6]. Malheureusement, la recherche d'un ordonnancement optimal des variables est irréalisable calculatoirement, car la complexité de la recherche de la variable suivante sur laquelle se brancher pour dériver un arbre de recherche de taille minimale est DP-hard [9]. Pour cette raison, la plupart des tâches de satisfaction de contraintes sont traitées en utilisant des heuristiques de choix de variables existantes, soigneusement conçues par des experts en programmation de contraintes. À ce jour, un large éventail d'heuristiques a été proposé dans la littérature, allant de l'heuristique *statique*, dans laquelle les variables sont ordonnées avant le début de la recherche, à l'heuristique *adaptative* pour laquelle la variable sur laquelle brancher est sélectionnée à l'aide d'informations collectées pendant le processus de recherche (par exemple [4]). En présence d'une telle diversité, une question importante se pose : *Étant donnés une instance CSP et un ensemble d'heuristiques de choix de variables candidates disponibles dans le solveur, quelle heuristique est la meilleure pour résoudre l'instance ?*

Cette question appelle naturellement une approche de type "bandit", comme l'a récemment soutenu [11]. Les problèmes de bandit multi-bras sont des tâches de décision séquentielle dans lesquelles l'algorithme d'apprentissage a accès à un ensemble de bras, et observe la récompense du bras choisi après chaque essai. Dans le contexte de la satisfaction de contraintes, chaque bras est une heuristique candidate et la séquence d'essais peut être dérivée en utilisant des redémarrages [11]. Plus précisément, les solveurs de contraintes modernes utilisent un *suite de redémarrage* qui génère une séquence de « cutoffs » auxquels la recherche par retour-arrière est redémarrée en cas d'échec [7]. Sur la base de ce principe de redémarrage, le solveur explore à chaque « run » un arbre de recherche en utilisant l'heuristique sélectionnée par l'apprenant. Lorsque le cutoff est atteint, le solveur abandonne la recherche en cours et re-

*Article publié à IJCAI'22

commence, tandis que l'apprenant reçoit une récompense pour le bras choisi, qui reflète l'efficacité de l'heuristique correspondante au run en cours.

Conceptuellement, la performance d'un algorithme de bandit réside dans sa capacité à enchaîner *exploration* en acquérant de nouvelles informations sur les bras, et *exploitation* en sélectionnant un bras optimal sur la base des informations disponibles. Dans le cadre standard du *regret cumulé*, cette performance est mesurée par la différence de récompenses cumulées entre le meilleur bras (connu *a posteriori*), et la séquence de bras sélectionnée par l'apprenant [5]. Cette mesure de performance a été utilisée dans [11], avec des algorithmes de bandit à regret minimisé bien connus, tels que `UCB` [2] et `EXP3` [3]. Cependant, lors de la résolution d'une instance de CSP par recherche et redémarrage, la maximisation de la récompense cumulée des heuristiques sélectionnées n'est pas nécessairement la meilleure option. En effet, l'évaluation de la récompense d'une heuristique à un certain run a un *coût* qui dépend du cutoff associée au run. À ce sujet, la plupart des suites de redémarrage utilisées en pratique sont loin d'être uniformes : le cutoff peut varier d'un run à l'autre. Un exemple typique est la suite universelle de *Luby*. [10], dont la séquence de cutoffs $(1, 1, 2, 1, 1, 2, 4, 1, \ldots)$ croît linéairement de manière non monotone. Ainsi, en tenant compte d'une séquence de redémarrage donnée, l'apprenant devrait se concentrer sur l'exploration lors des recherches associées à des cutoffs faibles, et se tourner progressivement vers l'exploitation lors des recherches avec des cutoffs plus importants.

Ces considérations justifient l'utilisation d'un autre modèle de bandit, appelé *exploration pure* ou *identification du meilleur bras*. Ici, l'objectif est de trouver un bras optimal aussi rapidement que possible, et la performance de l'apprenant est généralement mesurée par le nombre d'étapes d'exploration nécessaires pour converger. Bien que la plupart des approches d'exploration pure opèrent dans le contexte stochastique, où les récompenses de chaque bras sont tirées au hasard selon une distribution de probabilité (par exemple [1]), des études récentes ont considéré le modèle plus général *non-stochastique*, où la séquence de récompenses pour chaque bras est convergente, mais on ne sait rien de sa vitesse de convergence [8]. Le dernier modèle est plus approprié pour le problème de satisfaction de contraintes, puisque les récompenses observées pour une certaine heuristique peuvent dépendre des runs au cours desquels l'heuristique a été sélectionnée.

2 Contribution

Avec ces notions en main, nous appelons *identification de la meilleure heuristique* le problème de trouver aussi rapidement que possible une heuristique de choix de variables avec une récompense (asymptotique) optimale, étant donné une instance CSP, un ensemble d'heuristiques candidates, *et* une suite de redémarrage prédéfinie. Dans ce travail, nous nous concentrons sur la suite universelle de Luby mentionnée ci-dessus, qui est régulièrement utilisée en pratique. En se basant sur la structure d'arbre binaire de la séquence de Luby, nous proposons un algorithme d'identification du meilleur bras inspiré de *Successive Halving*. [8]. Notre algorithme, appelé *Adaptive Single Tournament* (`AST`), utilise des tournois à élimination directe pour éliminer successivement la moitié de toutes les heuristiques actuellement restantes, avant de procéder à des runs au cutoff plus important. Nous examinons la convergence de cet algorithme dans le cadre non stochastique, et nous démontrons son efficacité sur divers benchmarks de satisfaction de contraintes. Notamment, en sélectionnant la meilleure heuristique sur de grands cutoffs de la suite de Luby, `AST` surpasse les méthodes de bandit standard `UCB` et `EXP3` proposées dans [11].

Références

[1] Jean-Yves Audibert, Sébastien Bubeck, and Rémi Munos. Best arm identification in multi-armed bandits. In *Proc. of COLT'10*, pages 41–53, 2010.

[2] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2) :235–256, May 2002.

[3] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1) :48–77, 2002.

[4] Frédéric Boussemart, Fref Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proc. of ECAI'04*, pages 146–150, 2004.

[5] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Found. Trends Mach. Learn.*, 5(1) :1–122, 2012.

[6] Ian P. Gent, Ewan MacIntyre, Patrick Presser, Barbara M. Smith, and Toby Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proc. of CP'96*, pages 179–193, 1996.

[7] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reason.*, 24(1) :67–100, 2000.

[8] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Proc. of AISTATS'16*, pages 240–248, 2016.

[9] Paolo Liberatore. On the complexity of choosing the branching literal in DPLL. *Artif. Intell.*, 116(1) :315–326, 2000.

[10] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4) :173–180, 1993.

[11] Hugues Wattez, Frédéric Koriche, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Learning variable ordering heuristics with multi-armed bandits and restarts. In *Proc. of ECAI'20*, pages 371–378, 2020.

Best Heuristic Identification for Constraint Satisfaction

Frederic Koriche[1] , **Christophe Lecoutre**[1] , **Anastasia Paparrizou**[1] and **Hugues Wattez**[2]
[1]CRIL, Univ. Artois & CNRS
[2]LIX CNRS, École Polytechnique, Institut Polytechnique de Paris
{koriche,lecoutre,paparrizou}@cril.fr, wattez@lix.polytechnique.fr

Abstract

In constraint satisfaction problems, the *variable ordering heuristic* takes a central place by selecting the variables to branch on during backtrack search. As many hand-crafted branching heuristics have been proposed in the literature, a key issue is to identify, from a pool of candidate heuristics, which one is the best for solving a given constraint satisfaction task. Based on the observation that modern constraint solvers are using restart sequences, the *best heuristic identification problem* can be cast in the context of multi-armed bandits as a *non-stochastic best arm identification problem*. Namely, during each run of some given restart sequence, the bandit algorithm selects a heuristic and receives a reward for this heuristic before proceeding to the next run. The goal is to identify the best heuristic using few runs, and without any stochastic assumption about the constraint solver. In this study, we propose an adaptive variant of *Successive Halving* that exploits *Luby's universal restart sequence*. We analyze the convergence of this bandit algorithm in the non-stochastic setting, and we demonstrate its empirical effectiveness on various constraint satisfaction benchmarks.

1 Introduction

Constraint satisfaction is a recurring problem that arises in numerous computer science applications including, among others, bio-informatics, configuration, planning, scheduling and software validation. Given a set of decision variables and a set of constraints, each specifying a relation holding among some variables, the Constraint Satisfaction Problem (CSP) is to find an assignment of all variables that satisfies all constraints. To cope with this NP-complete problem, constraint solvers typically interleave backtracking search and inference in order to efficiently explore the space of assignments.

The *variable ordering heuristic* plays a key role in constraint satisfaction by iteratively selecting the next variable to branch on during backtrack search. Choosing the right branching strategy for a given CSP instance can significantly speed up the resolution, because different variable orderings can lead to entirely different search trees [Gent *et al.*, 1996]. Unfortunately, discovering an optimal variable ordering is computationally infeasible, since the complexity of finding the next variable to branch on for deriving a minimal-size search tree is DP-hard [Liberatore, 2000]. For this reason, most constraint satisfaction tasks are handled by using existing variable ordering heuristics, carefully designed by experts in Constraint Programming. To this point, a wide spectrum of hand-crafted heuristics have been proposed in the literature, ranging from *static* heuristics, in which variables are ordered before search starts, to *dynamic* heuristics for which the variable to branch on is selected using information that is collected during the search process (e.g. [Bessiere and Régin, 1996; Smith and Grant, 1998; Boussemart *et al.*, 2004; Refalo, 2004; Michel and Hentenryck, 2012; Habet and Terrioux, 2021; Wattez *et al.*, 2019]). In presence of such a diversity, an important question arises:

Given a CSP instance and a set of candidate (variable ordering) heuristics available in the solver, which heuristic is the best for solving the instance?

This question naturally calls for a "bandit" approach, as recently advocated in [Xia and Yap, 2018; Wattez *et al.*, 2020]. Multi-armed bandit problems are sequential decision tasks in which the learning algorithm has access to a set of arms, and observes the reward for the chosen arm after each trial. In the context of constraint satisfaction, each arm is a candidate heuristic and the sequence of trials can be derived using restarts [Wattez *et al.*, 2020]. More precisely, modern constraint solvers use a *restart scheme* that generates a sequence of "cutoffs" at which backtrack search is restarted if unsuccessful [Gomes *et al.*, 2000]. Based on this restart scheme, the solver explores at each run a search tree using the heuristic selected by the learner. When the cutoff is reached, the solver abandons the current search and restarts, while the learner receives a reward for the chosen arm, which reflects the effectiveness of the corresponding heuristic at the current run.

Conceptually, the performance of a bandit algorithm lies in its ability to interleave *exploration* by acquiring new information about arms, and *exploitation* by selecting an optimal arm based on the available information. In the standard *cumulative regret* setting, this performance is measured by the difference of cumulative rewards between the best arm taken from the benefit of hindsight, and the sequence of arms selected by the learner [Bubeck and Cesa-Bianchi, 2012]. This performance metric was used in [Wattez *et al.*, 2020], together

with well-known low-regret bandit algorithms, such as UCB [Auer *et al.*, 2002a] and EXP3 [Auer *et al.*, 2002b]. However, when solving a CSP instance using search-and-restart, maximizing the cumulative reward of selected heuristics is not necessarily the best option. Indeed, evaluating the reward of a heuristic at some trial has a *cost* that depends on the cutoff associated with the trial. To this point, most restart schemes used in practice are far from uniform: the cutoff may vary from one run to another. A prototypical example is *Luby's universal scheme* [Luby *et al.*, 1993], whose sequence of cutoffs $(1, 1, 2, 1, 1, 2, 4, 1, \ldots)$ grows linearly in a non-monotonic way. Thus, by taking into account a given restart sequence, the learner should focus on exploration at runs associated with low cutoffs, and gradually turn to exploitation at runs with larger cutoffs.

These considerations warrant the use of an alternative bandit setting, called *pure exploration* or *best arm identification*. Here, the goal is to find an optimal arm as quickly as possible, and the performance of the learner is typically measured by the number of exploration steps needed to converge. Although most approaches to pure exploration operate in the stochastic regime, where the rewards of each arm are drawn at random according to a probability distribution (e.g. [Audibert *et al.*, 2010; Kaufmann *et al.*, 2016]), recent studies have considered the more general *non-stochastic* regime, where the sequence of rewards for each arm is convergent, but nothing is known about its rate of convergence [Jamieson and Talwalkar, 2016; Li *et al.*, 2017]. The last regime is more appropriate for constraint satisfaction tasks, since the rewards observed for some heuristic may depend on the trials at which the heuristic has been selected.

With these notions in hand, we call *best heuristic identification* the problem of finding as quickly as possible a variable ordering heuristic with optimal (asymptotic) reward, given a CSP instance, a pool of candidate heuristics, *and* a predefined restart scheme. Our main focus in this paper is the aforementioned Luby's universal scheme, which is regularly used in practice. Based on the binary tree structure of Luby's sequence, we propose a best arm identification algorithm inspired from *Successive Halving* [Karnin *et al.*, 2013; Jamieson and Talwalkar, 2016]. Our algorithm, called *Adaptive Single Tournament* (AST), uses single-elimination tournaments for successively eliminating half of all currently remaining heuristics, before proceeding to trials with larger cutoffs. We examine the convergence of this algorithm in the non-stochastic setting, and we demonstrate its effectiveness on various constraint satisfaction benchmarks. Notably, by selecting the best heuristic on large cutoffs of Luby's sequence, AST outperforms the standard bandit methods UCB and EXP3 advocated in [Wattez *et al.*, 2020].

The rest of the paper is organized as follows. The background about best arm identification is given in Section 2, and the best heuristic identification problem is presented in Section 3, together with an analysis of our algorithm. Comparative experiments are reported in Section 4. Finally, we conclude with some perspectives of research in Section 5.[1]

[1]Proofs of results and additional experiments are available at: https://hal.archives-ouvertes.fr/hal-03678354.

2 Best Arm Identification

For ease of reference, we use $[n]$ to denote the set $\{1, \cdots, n\}$. Multi-armed bandit problems can be formulated as infinitely repeated games between a learning algorithm and its environment. During each trial $t \in \mathbb{N}$, the learner plays an arm i taken from a predefined set $[K]$ of arms, and the environment responds with a reward $r_i(t)$ for this arm. In this study, we assume that for each trial $t \in \mathbb{N}$ and each arm $i \in [K]$, the feedback $r_i(t)$ is generated from a predefined reward function $r_i(\cdot) : \mathbb{N} \to [0, 1]$ for which the limit $\lim_{t\to\infty} r_i(t)$ exists and is equal to ν_i. Since reward functions do not change over time, the environment is "oblivious" to the learner's actions.

This *non-stochastic* setting introduced in [Jamieson and Talwalkar, 2016; Li *et al.*, 2017] lies between two extreme cases: the *stochastic* regime where each reward $r_i(t)$ is an i.i.d. sample from a fixed probability distribution supported, and the *adversarial* regime where each reward $r_i(t)$ is an arbitrary value that can be chosen adaptively by the environment, based on all the arms that the learner has played so far. As observed in [Jamieson and Talwalkar, 2016], the stochastic regime is a special case of the non-stochastic setting, which in turn is a special case of the adversarial regime.

In the *(non-stochastic) best arm identification* problem, the learning algorithm is given an exploration period during which it is allowed to gather information about reward functions. The goal of the learner is to minimize the number of trials required to explore before committing to an optimal arm on all subsequent trials. More formally, suppose without loss of generality that $\nu_1 > \nu_2 \geq \cdots \geq \nu_K$ Then, the performance of the learner is given by the smallest integer $\tau \in \mathbb{N}$ such that for any trial $t \geq \tau$, the learner is playing 1. Now, let

$$\Delta_{\min} = \min_{i=2,\cdots,K} \Delta_i \;\text{ where } \Delta_i = \nu_1 - \nu_i$$

are the *gaps* of suboptimal arms. In the *stochastic* best arm identification setting, these gaps are sufficient to derive bounds on the probability that the learner is selecting the right arm at horizon τ (e.g. [Kaufmann *et al.*, 2016]). However, in the non-stochastic setting, we also need to approximate the convergence rate of reward functions $r_i(\cdot)$. To this end, let

$$\rho_{\max}(t) = \max_{i\in[K]} \rho_i(t) \;\text{ where }\; \rho_i(t) = \sup_{t'\geq t} |\nu_i - r_i(t')|$$

Each "envelope" $\rho_i(\cdot)$ is the point-wise smallest non-increasing function from $\mathbb{N}$ to $[0, 1]$ satisfying $|\nu_i - r_i(t)| \leq \rho_i(t)$ for all t. The quasi-inverse of $\rho_{\max}(\cdot)$ is given by

$$\rho^{-1}_{\max}(v) = \min\{t \in \mathbb{N} : \rho(t) \leq v\}$$

By coupling the smallest gap $\Delta_{\min}$ with the function $\rho^{-1}_{\max}(\cdot)$, which together capture the intrinsic difficulty of the learning problem, we can derive a simple upper bound on the length of the exploration phase required by the learner to converge towards an optimal arm. Namely,

Proposition 1. *There exists a learning algorithm such that, given as input* $[K]$, $\rho^{-1}_{\max}(\cdot)$ *and* $\Delta_{\min}$, *after an exploration period of length*

$$\tau = K + \rho^{-1}_{\max}\left(\frac{\Delta_{\min}}{2}\right)$$

the algorithm always returns the best arm.

Algorithm 1: Adaptive Successive Halving (ASH)

Input: A set of arms $[K]$

Set $T_0 = K\lceil \log_2 K \rceil$
for $p = 0, 1, \ldots$ **do**
  Set $S_0 = [K]$
  for $q = 0, 1, \cdots, \lceil \log_2(K) \rceil - 1$ **do**
    Play each arm in S_q for s_q times, where

$$s_q = \left\lfloor \frac{T_p}{|S_q| \lceil \log_2(K) \rceil} \right\rfloor$$

    Set S_{q+1} as the $\lfloor S_q/2 \rfloor$ best arms in S_q measured according to their s_qth reward
  Play the unique arm in $S_{\lceil \log_2(K) \rceil}$
  Set $T_{p+1} = 2T_p$

Of course, in multi-armed bandit problems, the reward functions $r_i(\cdot)$ are *hidden*, and hence, nothing is *a priori* known about the envelopes $\rho_i(\cdot)$ and the gaps Δ_i. So, the learner has to deal with this issue by devising an exploration strategy whose performance is reasonably close to τ. A common strategy for the best arm identification task is the *Successive Halving* algorithm which was analyzed in both the stochastic setting [Karnin *et al.*, 2013] and the non-stochastic setting [Jamieson and Talwalkar, 2016].

In Algorithm 1, we present an adaptive, parameter-free version of Successive Halving (called ASH) that uses the so-called "doubling trick" for circumscribing the right amount of exploration. More precisely, each iteration of the outer loop consists of an exploration period, specified by the inner loop, followed by an exploitation period during which ASH is playing the unique remaining arm in its pool $S_{\lceil \log_2(K) \rceil}$. The doubling trick is then performed at the end of the outer loop. Based on a simple extension of the analysis given in [Jamieson and Talwalkar, 2016], we get the following result.

Proposition 2. *ASH is playing an optimal arm at the end of any iteration p of the outer loop, whenever*

$$p \geq 2 + \log_2 \left[K\lceil \log_2 K \rceil \left(1 + \rho_{\max}^{-1}\Big(\frac{\Delta_{\min}}{2}\Big) \right) \right]$$

3 Best Heuristic Identification

How can we adapt the paradigm of Successive Halving to constraint satisfaction tasks? This is the purpose of this section; after introducing some technical background about constraint satisfaction, we present a family of algorithms inspired from ASH that identify an optimal variable ordering heuristic.

3.1 Constraint Satisfaction

Recall that any instance P of the *Constraint Satisfaction Problem* (CSP) consists of a finite set of decision variables $\texttt{vars}(P)$, and a finite set of constraints $\texttt{ctrs}(P)$. Each variable x takes values from a finite domain, denoted $\texttt{dom}(x)$. Each constraint c is specified by a relation $\texttt{rel}(c)$ over a set of variables, called the *scope* of c, and denoted $\texttt{scp}(c)$. The *arity* of a constraint c is the size of its scope, and the *degree* of a variable x is the number of constraints in $\texttt{ctrs}(P)$ involving x in its scope. A *solution* to P is the assignment of a value to each variable in $\texttt{vars}(P)$ such that all constraints in $\texttt{ctrs}(P)$ are satisfied. The instance P is *satisfiable* if it admits at least one solution, and it is *unsatisfiable* otherwise.

A standard approach for solving CSP instances is to perform a backtracking search on the space of partial solutions, and to enforce a property called *generalized arc consistency* [Mackworth, 1977] on each decision. The resulting MAC (*Maintaining Arc Consistency*) algorithm [Sabin and Freuder, 1994] partitions the search space using a binary search tree $\mathcal{T}$. For each internal node of $\mathcal{T}$, a pair (x, v) is selected where x is an unfixed decision variable and v is a value in $\texttt{dom}(x)$. Based on this pair, two branches are generated: the assignment $x = v$ and the refutation $x \neq v$. The choice of x is decided by a *variable ordering heuristic*, and the choice of v is determined by some value ordering heuristic, which is usually the lexicographic order over $\texttt{dom}(x)$ by default.

In modern constraint solvers, the above approach is combined with a *restart scheme* that terminates and restarts search at regular intervals. The effect of such restarts is to early abort long runs, which thus saves on the cost of branching mistakes and resulting "heavy-tailed" phenomena [Gomes *et al.*, 2000]. Formally, any restart scheme is a mapping $\sigma : \mathbb{N} \to \mathbb{N}^+$, where $\sigma(t)$ denotes the *cutoff* at which the backtracking search is terminated during the tth run. In practice, $\sigma(t)$ is rescaled to capture a relevant cutoff unit such as, for example, the total number of backtracks [Gomes *et al.*, 2000] or the total number of wrong decisions [Bessiere *et al.*, 2004]. In this study, we concentrate on *Luby's universal scheme* $\sigma_{\text{luby}}(\cdot)$ which uses powers of two: when the cutoff 2^i is used twice, the next cutoff is 2^{i+1}. More formally:

$$\sigma_{\text{luby}}(t) = \begin{cases} 2^{i-1} & \text{if } t = 2^i - 1 \\ \sigma_{\text{luby}}\big(t - 2^{i-1} + 1\big) & \text{if } 2^{i-1} \leq t < 2^i - 1 \end{cases}$$

Based on the MAC algorithm and Luby's restart scheme, the solver builds a sequence of search trees $\langle \mathcal{T}_1, \mathcal{T}_2, \ldots \rangle$, where $\mathcal{T}_t$ is the search tree explored by MAC until it reaches the (possibly rescaled) cutoff $\sigma_{\text{luby}}(t)$. Importantly, the behavior of the constraint solver is generally non-stochastic. Indeed, even if it restarts from the beginning, the solver can memorize some relevant information about previous explorations. For example, the solver may keep in a table the number of constraint checks in the past runs, or it can maintain in a cache the nogoods which have frequently occurred in the search trees explored so far [Lecoutre *et al.*, 2007].

3.2 Learning to Branch

As mentioned in the introduction, various heuristics have been proposed for ordering decision variables during tree search. Examples of such variable ordering heuristics commonly used in constraint solvers include `dom/ddeg` [Bessiere and Régin, 1996], `dom/wdeg` [Boussemart *et al.*, 2004], `impact` [Refalo, 2004], `activity` [Michel and Hentenryck, 2012], `chs` [Habet and Terrioux, 2021] and `cacd` [Wattez *et al.*, 2019]. For instance, `dom/ddeg` branches on the variables with the smallest ratio between the size of their current domain and the current degree formed by counting only unfixed neighboring variables.

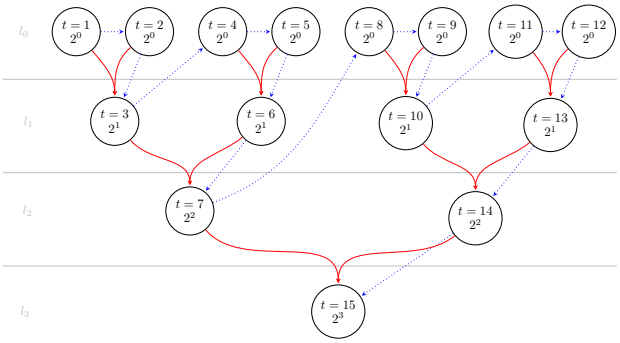


Figure 1: Tree-structured view of Luby's sequence for the first 15 runs. The cutoff sequence is derived by following the blue path, while the behavior of AST is captured by the red paths.

Given a pool $H = \{h_1, \cdots, h_K\}$ of candidate variable ordering heuristics, the problem of identifying the best heuristic for some CSP instance P can be cast as a "best arm identification" task, in which the environment is played by the solver. During each run t of the restart sequence, the learner starts by selecting an index $i \in [K]$. Then, the solver calls the MAC algorithm with the heuristic h_i for building a search tree $\mathcal{T}_t$, which is used to infer a reward $r_t(i) \in [0, 1]$ supplied to the learner. As suggested in [Wattez *et al.*, 2020], this reward can be measured using the (normalized) *pruned tree size*, which captures the ability of the solver to quickly prune large portions of its search space. More formally, let $\texttt{rft}(\mathcal{T}_t)$ be the set of internal nodes where each child is a refutation leaf, and for each such node n, let $\texttt{fut}(n)$ be the set of variables that are left unfixed at n. Then,

$$r_t(i) = \frac{\log_2 \left(\sum_{n \in \texttt{rft}(\mathcal{T}_t)} \prod_{x \in \texttt{fut}(n)} |\texttt{dom}(x)| \right)}{\log_2 \left(\prod_{x \in \texttt{vars}(P)} |\texttt{dom}(x)| \right)}$$

Computing such rewards takes linear space by exploiting the depth-first traversal of the search space performed by the MAC algorithm. However, observing $r_t(i)$ at the end of each run t has a *cost*, which corresponds to the time spent by the constraint solver in exploring the search tree $\mathcal{T}_t$ with the branching heuristic h_i. Because this runtime depends on the tth cutoff in the restart sequence, the learner should take into consideration the solver's restart scheme when trading exploration periods with exploitation phases.

From this perspective, the sequence of cutoffs in Luby's universal scheme $\sigma_{\text{luby}}(\cdot)$ can be viewed as an infinite binary tree organized into layers. This is illustrated in Figure 1 for the first 15 runs. For each layer $l = 0, 1, \cdots$, let t_l be the first run in the sequence with cutoff 2^l. Since by construction t_l is the root of a complete binary tree of size $2^{l+1} - 1$, this suggests using a variant of Successive Halving that starts by exploring 2^l candidate heuristics at the leaves, and successively eliminates half of all currently remaining heuristics until only one heuristic is left at the root t_l. Furthermore, since $t_{l+1} = 2t_l + 1$, this strategy can rely on the doubling trick for reaching the layer l where the best heuristic in H is identified with certainty at t_l.

Algorithm 2: Adaptive Single Tournament (AST)

Input: A set of arms $[K]$, a positive integer $m \geq 1$

Set $S = [K]$
for *each run* $t = 1, 2, \ldots$ **do**
  if $\sigma_{\text{luby}}(t) = 1$ **then**
    Select an arbitrary arm $i \in S$
    Set $S = S \setminus \{i\}$ and **if** $S = \emptyset$ **then** set $S = [K]$
  else
    Let i_{left} be the arm played at run $t - \sigma_{\text{luby}}(t)$
    Let i_{right} be the arm played at run $t - 1$
    Choose $i \in \{i_{\text{left}}, i_{\text{right}}\}$ with best reward r_i
  Play i for m times and set r_i to the mth observed reward at run t

With these notions in hand, we present the *Adaptive Single Tournament* (AST) algorithm that is based on Luby's restart scheme for identifying the best heuristic on some CSP instance. Of course, AST is intended to be used alongside a constraint solver with $H = \{h_1, \cdots, h_K\}$ available branching heuristics. As specified in Algorithm 2, the learner focuses on exploration at cutoff 1, and progressively exploits at larger cutoffs using single tournaments. Based on the tree-structured view of Luby's sequence (Figure 1), the learner explores with equal frequency each arm in $[K]$ at the leaves of the tree. When it reaches an internal node of the tree, the learner selects the best arm among those played at the children of that node. Finally, AST uses a parameter m for testing the behavior of the constraint solver on candidate heuristics. For each run t of the restart sequence, the learner plays m times the arm i selected at t. Correspondingly, the solver performs m backtracking searches with cutoff $\sigma_{\text{luby}}(t)$, and the learner stores the last observed reward.[2]

Proposition 3. *AST is playing an optimal arm at any root node t_l of Luby's sequence, whenever the layer l satisfies:*

$$l \geq 1 + \frac{\lceil \log_2 K \rceil^2}{2} \left\{ 2 + \log_2 \left\lfloor \frac{1}{m} \left(1 + \rho_{\max}^{-1} \left(\frac{\Delta_{\min}}{2} \right) \right) \right\rfloor \right\}$$

4 Experiments

After an excursion into the theoretical aspects of our framework, we now turn to experiments. Given a set $\mathcal{P}$ of CSP instances and a pool H of variable ordering heuristics, the learning objective is to identify for each instance $P \in \mathcal{P}$ an optimal heuristic $h \in H$, using the MAC algorithm for backtracking search and Luby's sequence for the restart scheme.

For the set $\mathcal{P}$, we have considered all CSP instances selected for the XCSP3 competitions from 2017 to 2019.[3] This amounts to 810 CSP instances classified into 83 families. For the pool H, we have considered 8 heuristics, namely:

$$H = \{\texttt{lex}, \texttt{dom}, \texttt{dom/ddeg}, \texttt{abs}, \texttt{ibs}, \texttt{dom/wdeg}, \texttt{chs}, \texttt{cacd}\}$$

[2] Interestingly, AST shares common features with the top-K rank aggregation algorithm proposed by [Mohajer *et al.*, 2017]. But their algorithm operates in the stochastic dueling bandit setting, which is different from the non-stochastic best arm identification setting.

[3] http://www.xcsp.org/competitions/

	#SOLV.	TIME (S)
VBS	574	52,175
$\texttt{AST}_8$	557	93,922
$\texttt{AST}_{16}$	556	94,844
$\texttt{AST}_1$	555	97,506
$\texttt{AST}_2$	555	99,044
$\texttt{AST}_4$	553	100,387
UCB	550	115,210
UNI	547	115,608
cacd	543	140,289

	#SOLV.	TIME (S)
chs	528	157,484
dom/wdeg	520	179,228
EXP3	480	269,192
abs	420	426,498
dom/ddeg	406	442,194
dom	404	467,824
ibs	383	495,997
lex	382	509,658

Table 1: Ranking of the branching strategies according to the number of solved instances and the cumulative runtime.

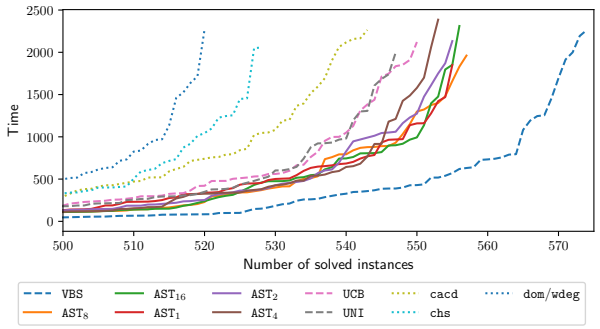


Figure 2: Cactus plots of the best branching strategies.

The best arm identification algorithm AST was compared with the state-of-the-art bandit algorithms UCB [Auer *et al.*, 2002a] and EXP3 [Auer *et al.*, 2002b], together with the baselines UNI and VBS. Here, UNI is the "uniform" strategy which draws uniformly at random a heuristic from H on each run of the restart sequence. VBS is the *Virtual Best Solver* that selects the best variable ordering heuristic in hindsight. This baseline can be derived by first playing each candidate heuristic $h \in H$ on all runs of the restart sequence, and then by selecting the heuristic for which P was solved with the least amount of time. For UCB and EXP3, we have considered the versions used in [Wattez *et al.*, 2020], and for AST, we have evaluated the algorithm using $m \in \{1, 2, 4, 8, 16\}$.

Our experiments have been performed with the constraint solver ACE[4], by keeping the default option for most parameters. Notably, the solver was allowed to record nogoods after each restart [Lecoutre *et al.*, 2007] in order to improve its performance during the sequence of runs. Actually, the unique exception to the default setting was to discard the last conflict policy [Lecoutre *et al.*, 2009], which would introduce a bias in the behavior of variable ordering heuristics. For all bandit algorithms UCB, EXP3, and AST, the reward of each selected heuristic was measured using the (normalized) pruned tree size, as specified in Section 3. Finally, each cutoff $\sigma_{\text{luby}}(t)$ of Luby's sequence was rescaled to $u \cdot \sigma_{\text{luby}}(t)$, where $u = 150$, corresponding to u wrong decisions (conflicts) per cutoff unit. All experiments have been conducted on a 3.3 GHz Intel XEON E5-2643 CPU, with 32 GB RAM, with a timeout set to $2,400$ seconds.

4.1 Main Results

In Table 1 are reported the results comparing the bandit algorithms UCB, EXP3 and AST (for different values of m), and the baselines UNI and VBS, together with the variable ordering heuristics in H. Here, we have discarded instances which could not be solved by any strategy. The competitors, referred to as branching strategies, are ranked according to the number of solved instances and, in case of tie, according to the cumulative runtime.

At the bottom of the ranking lie the heuristics with the worst performance: using lex, ibs, dom, dom/ddeg, or abs, the constraint solver ACE can solve at most 420 instances, taking more than 118 hours to handle them. The bandit algorithm EXP3 is just above, with a resolution of 480 instances in approximately 75 hours. We mention in passing that EXP3 operates in the *adversarial* bandit setting. Such a bad performance indicates that even if the solver can be non-stochastic, its behavior is not necessarily competitive with respect to the learner. For the next three heuristics dom/wdeg, chs and cacd, at least 520 instances could be solved within less than 50 hours. The simple uniform strategy UNI for selecting heuristics achieves a decent performance with 547 solved instances within 32 hours. To this point, we can observe that the bandit algorithm UCB, which operates in the *stochastic* regime is not much better: only 3 additional instances could be solved using almost the same amount of time. Unsurprisingly, VBS lies at the top of the ranking, with 574 instances solved in less than 15 hours. Interestingly, all configurations of AST are just below by solving, on average, 555 instances in less than 27 hours.

These results are corroborated by the cactus plots of the best strategies, as reported in Figure 2. Each plot indicates the number of solved instances (x-axis) at any time (y-axis). The dotted-line plots correspond to the performances of the three best heuristics in H, while the dashed-line plots capture the performances of the baselines UNI and VBS, and the bandit algorithm UCB.[5] Finally, the cactus plots of the different configurations of AST are indicated using solid lines. Again, we can observe that UCB and UNI have similar performances, when considering the number of solved instances per amount of time. We can also see that the behavior of ASH is remarkably stable when varying m.

4.2 Pairwise Comparisons

Although our bandit algorithm AST can solve more instances in $\mathcal{P}$ than any single heuristic in H, we would like to determine whether it is capable of solving any instance P that can be solved by *some* heuristic in H. More generally, the goal here is to compare bandit approaches with respect to each $h \in H$, on any CSP instance P for which we know that P can be solved by h before reaching the timeout.

To this end, Figure 3 provides a heatmap of the different branching strategies, by reporting cacd, UNI, UCB, AST (for $m \in \{1, 8\}$) and VBS on the rows, and all variable ordering heuristics in H on the columns. In this heatmap, each entry

[4] https://github.com/xcsp3team/ace

[5] As EXP3 could not solve at least 500 instances, its cactus plot is not reported here.

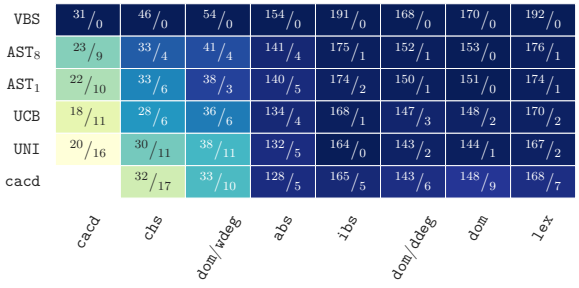


Figure 3: Heatmap of the branching strategies.

with row i and column j is a pair a/b, where a is the number of instances in $\mathcal{P}$ that could be solved by i but not by j, and conversely, b is the number of instances in $\mathcal{P}$ that could be solved by j but not by i. For example, the learning algorithm AST with $m = 8$ (second row) was able to solve 41 instances which could not be solved using the single heuristic dom/wdeg (third column), but the last one could solve 4 instances upon which AST failed. For the sake of readability, the colors of the entries are specified using the ratio $b/a+b$; a value close to 1 corresponds to a light color, which in turn indicates that the branching strategy at row i often fails in handling instances solved by the heuristic at column j.

Obviously, VBS never fails since it is selecting the best heuristic in hindsight. More interestingly, we can observe on the last row that cacd is not systematically better than other heuristics. The row comparisons between UNI, UCB and AST reveal important differences, when focusing on the first three columns of the heatmap, which correspond to the heuristics cacd, chs and dom/wdeg. Notably, the uniform strategy UNI is often dominated by these heuristics when they prove to be efficient on solving some CSP instances. For example, UNI was unable to deal with 16 instances which could be solved by cacd. On the other hand, UCB and AST ($m = 8$) are more robust by respectively solving 5 and 8 additional instances. Similar observations hold for chs and dom/wdeg. Finally, it turns out that UCB is always dominated by AST (for both $m = 1$ and $m = 8$), when they are compared with respect to the most efficient heuristics cacd, chs and dom/wdeg.

4.3 Finer-Grained Analysis

We conclude these experiments by comparing UCB and AST on some specific instances. Recall that UCB aims at maximizing the cumulative reward of selected heuristics, whatever being the sequence of cutoffs. Contrastingly, the goal of AST is to identify the best heuristic on the runs with large cutoffs. Figure 4 illustrates the resulting behaviors on two CSP instances (from families *Primes* and *Rlfap*). For both UCB and AST, each cutoff κ ranging from 1 to 1024 is associated with a bar-plot indicating the proportion of heuristics selected at the runs $\{t : \sigma_{\text{luby}}(t) = \kappa\}$. In light of these graphics, the divergence between UCB and AST is remarkable: while UCB fails at identifying a unique heuristic at large cutoffs, AST quickly converges through successive halving, which is easily recognizable from one bar-plot to the next.

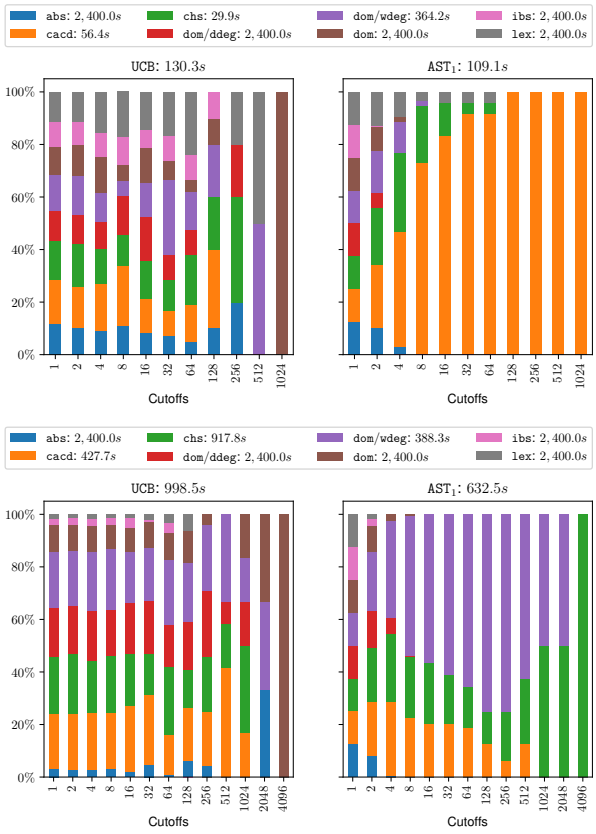


Figure 4: Proportions of heuristics selected by UCB (left) and AST (right) at each cutoff of Luby's sequence, for the CSP instances *Primes-15-60-3-5* and *Rlfap-scen-11-f01_c18*.

5 Conclusion

In this paper, we have focused on the best heuristic identification problem, which is to learn an optimal variable ordering heuristic for a CSP instance, given a set of candidate heuristics. By formulating this problem as a non-stochastic best-arm identification task, we have presented a bandit algorithm (AST) inspired from Successive Halving that takes into account the structure of Luby's universal sequence. For this algorithm, we have provided a convergence analysis, together with comparative results on various CSP instances.

A natural perspective of research that emerges from this study is to design best-heuristic identification algorithms for other universal restart schemes such as, for example, the geometric sequence examined in [Wu and van Beek, 2007]. An alternative and arguably more challenging perspective is to learn both branching heuristics *and* propagation techniques, for solving constraint satisfaction tasks. Thus, each arm is here a pair formed by a candidate heuristic and a candidate propagator. Yet, since the choice of the right propagation technique depends on the depth at which backtracking search is performed [Balafrej *et al.*, 2015], the corresponding best-arm identification problem should be examined in a *contextual bandit* setting, for which many questions remain open.

References

[Audibert *et al.*, 2010] Jean-Yves Audibert, Sébastien Bubeck, and Rémi Munos. Best arm identification in multi-armed bandits. In *Proc. of COLT'10*, pages 41–53, 2010.

[Auer *et al.*, 2002a] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2):235–256, May 2002.

[Auer *et al.*, 2002b] Peter Auer, Nicolò Cesa-Bianchi, Yoav Freund, and Robert E. Schapire. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.*, 32(1):48–77, 2002.

[Balafrej *et al.*, 2015] Amine Balafrej, Christian Bessiere, and Anastasia Paparrizou. Multi-armed bandits for adaptive constraint propagation. In *Proc. of IJCAI'15*, pages 290–296, 2015.

[Bessiere and Régin, 1996] Christian Bessiere and Jean-Charles Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proc. of CP'96*, pages 61–75, 1996.

[Bessiere *et al.*, 2004] Christian Bessiere, Bruno Zanuttini, and Cèsar Fernandez. Measuring search trees. In *Proc. of ECAI'04 workshop on Modelling and Solving Problems with Constraints*, pages 31–40, 2004.

[Boussemart *et al.*, 2004] Frédéric Boussemart, Fref Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proc. of ECAI'04*, pages 146–150, 2004.

[Bubeck and Cesa-Bianchi, 2012] Sébastien Bubeck and Nicolò Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *Found. Trends Mach. Learn.*, 5(1):1–122, 2012.

[Gent *et al.*, 1996] Ian P. Gent, Ewan MacIntyre, Patrick Presser, Barbara M. Smith, and Toby Walsh. An empirical study of dynamic variable ordering heuristics for the constraint satisfaction problem. In *Proc. of CP'96*, pages 179–193, 1996.

[Gomes *et al.*, 2000] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reason.*, 24(1):67–100, 2000.

[Habet and Terrioux, 2021] Djamal Habet and Cyril Terrioux. Conflict history based heuristic for constraint satisfaction problem solving. *J. Heuristics*, 27:951—-990, 2021.

[Jamieson and Talwalkar, 2016] Kevin Jamieson and Ameet Talwalkar. Non-stochastic best arm identification and hyperparameter optimization. In *Proc. of AISTATS'16*, pages 240–248, 2016.

[Karnin *et al.*, 2013] Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *Proc. of ICML'13*, pages 1238–1246, 2013.

[Kaufmann *et al.*, 2016] Emilie Kaufmann, Olivier Cappé, and Aurélien Garivier. On the complexity of best-arm identification in multi-armed bandit models. *J. Mach. Learn. Res.*, 17:1:1–1:42, 2016.

[Lecoutre *et al.*, 2007] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Recording and minimizing nogoods from restarts. *JSAT*, 1:147–167, 2007.

[Lecoutre *et al.*, 2009] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary, and Vincent Vidal. Reasonning from last conflict(s) in constraint programming. *Artif. Intell.*, 173(18):1592–1614, 2009.

[Li *et al.*, 2017] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.

[Liberatore, 2000] Paolo Liberatore. On the complexity of choosing the branching literal in DPLL. *Artif. Intell.*, 116(1):315–326, 2000.

[Luby *et al.*, 1993] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Inf. Process. Lett.*, 47(4):173–180, 1993.

[Mackworth, 1977] Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977.

[Michel and Hentenryck, 2012] Laurent Michel and Pascal Van Hentenryck. Activity-based search for black-box constraint programming solvers. In *Proc. of CPAIOR'12*, pages 228–243, 2012.

[Mohajer *et al.*, 2017] Soheil Mohajer, Changho Suh, and Adel Elmahdy. Active learning for top-k rank aggregation from noisy comparisons. In *Proc. of ICML'17*, pages 2488–2497, 2017.

[Refalo, 2004] Philippe Refalo. Impact-based search strategies for constraint programming. In *Proc. of CP'04*, pages 557–571, 2004.

[Sabin and Freuder, 1994] Daniel Sabin and Eugene C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proc. of CP'94*, pages 10–20, 1994.

[Smith and Grant, 1998] Barbara M. Smith and Stuart A. Grant. Trying harder to fail first. In *Proc. of ECAI'98*, pages 249–253, 1998.

[Wattez *et al.*, 2019] Hugues Wattez, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Refining constraint weighting. In *Proc. of ICTAI'19*, pages 71–77, 2019.

[Wattez *et al.*, 2020] Hugues Wattez, Frédéric Koriche, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. Learning variable ordering heuristics with multi-armed bandits and restarts. In *Proc. of ECAI'20*, pages 371–378, 2020.

[Wu and van Beek, 2007] Huayue Wu and Peter van Beek. On universal restart strategies for backtracking search. In *Proc. of CP'07*, pages 681–695, 2007.

[Xia and Yap, 2018] Wei Xia and Roland H. C. Yap. Learning robust search strategies using a bandit-based approach. In *Proc. of AAAI'18*, pages 6657–6665, 2018.

Accélération de l'algorithme de séparation et évaluation pour les diagrammes de décision grâce à la mémoïsation

Vianney Coppé*, Xavier Gillard, Pierre Schaus

UCLouvain

{vianney.coppe, xavier.gillard, pierre.schaus}@uclouvain.be

Résumé

L'optimisation discrète avec diagrammes de décision permet de résoudre des problèmes complexes sur base d'un modèle de programmation dynamique. L'approche consiste en un algorithme de séparation et évaluation dans l'espace des états du modèle, où les bornes dérivent de diagrammes de décision. Cet article explique comment accélérer l'algorithme en mémorisant certains résultats intermédiaires afin d'éviter au maximum l'exploration répétée de certaines parties de l'espace de recherche. Les résultats expérimentaux montrent que cette accélération est significative.

Mots-clés

Optimisation discrète, diagrammes de décision, programmation dynamique, séparation et évaluation.

Abstract

Discrete optimization with decision diagrams is a framework for solving complex problems based on a dynamic programming model. It consists in a branch-and-bound algorithm applied on the state space of the model, where the bounds are derived from decision diagrams. This paper explains how storing some intermediate results can help avoid the repeated exploration of some parts of the search space and speed up the algorithm. The experiments show that it results in a significant speed up.

Keywords

Discrete optimization, decision diagrams, dynamic programming, branch-and-bound.

1 Introduction

L'optimisation discrète avec *diagrammes de décision* (DDs) est une méthode générique d'optimisation introduite en 2016 par Bergman et al. [3]. Cette approche est basée sur les modèles de *programmation dynamique* (PD) pour résoudre des problèmes complexes. Lorsque leur résolution directe par PD est hors de portée en termes à la fois de temps d'exécution et de mémoire, un algorithme de *séparation et évaluation* (S&E) est proposé. Celui-ci repose sur des DDs *approximés – restreints* ou *relaxés* – pour explorer l'espace des états tout en élaguant certains sous-problèmes sur base des bornes fournies par les DDs. L'approche permet donc de profiter de la compacité des modèles PD tout en offrant la possibilité de résoudre des problèmes de grande taille. Cependant, l'algorithme tel que présenté originellement [3] ne possède aucun mécanisme qui permette d'éviter le traitement répété et inutile d'un même sous-problème. En effet, hormis la file constituée de l'ensemble des sous-problèmes restant à explorer, seules des bornes sont gardées en mémoire. Cela n'empêche un sous-problème ni d'être traité à différents stades de l'exécution de la S&E ni d'être traversé à de multiples reprises dans les DDs approximés. Or, le propre des modèles PD est de décomposer récursivement un problème difficile en un ensemble de sous-problèmes plus faciles à résoudre, avec un fort recoupement entre ceux-ci. Dans cet article, nous introduisons un mécanisme de mise en cache permettant de stocker un *seuil d'exploration* pour chaque sous-problème, indiquant à partir de quelle valeur il est nécessaire de traiter à nouveau le sous-problème en question, évitant ainsi de nombreux calculs redondants.

2 Notions préliminaires

Soit un problème d'optimisation discrète $\mathcal{P}$ défini comme $\max\{f(x) \mid x \in D, x \in C\}$ avec f la fonction objective à maximiser, $x = \langle x_0, \dots, x_{n-1} \rangle$ un vecteur de variables appartenant à $D = D_0 \times \dots D_{n-1}$ avec $x_i \in D_i$ et l'ensemble C représentant les contraintes imposées à x. Cette section présente comment modéliser et résoudre $\mathcal{P}$ grâce aux DDs.

2.1 Programmation dynamique

Un modèle PD pour $\mathcal{P}$ est constitué d'un *espace d'états* S partitionné en $n+1$ ensembles $S_0, \dots, S_n$ correspondant aux étapes successives du modèle. On distingue plusieurs états particuliers : l'état *racine* $\hat{r}$, *terminal* $\hat{t}$ et *non-admissible* $\hat{0}$. Les fonctions $t_i : S_i \times D_i \to S_{i+1}$ et $h_i : S_i \times D_i \to \mathbb{R}$ pour $i = 0, \dots, n-1$ déterminent respectivement les connexions entre les états d'étapes successives et la valeur associée à celles-ci. La *valeur racine* v_r permet de modéliser les constantes de la fonction objective. Sur base d'un tel modèle, la solution optimale est obtenue en résolvant :

$$\begin{aligned} \max\ & f(x) = v_r + \sum_{i=0}^{n-1} h_i(s^i, x_i) \\ \text{tel que } & s^{i+1} = t_i(s^i, x_i), i = 0, \dots, n-1 \\ & s^i \in S_i, i = 0, \dots, n \text{ et } x \in D \cap C. \end{aligned} \tag{1}$$

*L'auteur principal est doctorant.

2.2 Diagrammes de décision

Les DDs permettent de représenter un ensemble de solutions au modèle PD sous forme d'un graphe acyclique dirigé $\mathcal{B} = (U, A, \sigma, l, v)$ composé d'un ensemble de nœuds U organisé en couches $L_0, \dots, L_n$ et d'un ensemble d'arcs A. À chaque nœud $u \in U$, la fonction σ fait correspondre un état du modèle. De manière similaire, chaque arc $a = (u_i \xrightarrow{d} u_{i+1})$ reliant deux nœuds de couches successives $u_i \in L_i, u_{i+1} \in L_{i+1}$ matérialise une transition, associée à une décision $l(a) = d \in D_i$ assignée à la variable x_i et à une valeur $v(a)$. L'algorithme 1, dont les lignes grisées seront expliquées en Section 3, décrit la compilation d'un DD enraciné en un nœud u_r tel que $\sigma(u_r) \in S_i$, commençant par un première couche L_i contenant seulement u_r. Ensuite, à partir d'une couche L_j, on génère la couche L_{j+1} en appliquant toutes les transitions possibles du modèle PD aux nœuds de la couche L_j, jusqu'à atteindre L_n constituée du seul *nœud terminal* t. Si l'on dispose d'un chemin $p_1 : r \rightsquigarrow u_r$, sa combinaison avec chaque chemin $p_2 : u_r \rightsquigarrow t$ forme une solution du problème $\mathcal{P}$ dont la valeur est donnée par $v(p_1 \cup p_2) = v_r + \sum_{i=0}^{n-1} v(a_i)$ où a_i est l'arc en i-ième position du chemin.
Pour des problèmes complexes, compiler un DD exact est aussi coûteux que de résoudre directement le modèle PD. Deux variantes de DDs sont donc utilisées pour calculer des bornes, ensuite employées dans un algorithme de S&E.

DD restreints Afin de trouver des solutions admissibles et donc des bornes inférieures les DDs *restreints* [4] sont compilés à la manière d'une recherche en faisceau. Lorsqu'une couche L_i contient plus de W nœuds, on retire les $|L_i| - W$ nœuds les moins prometteurs, sur base d'une heuristique, avant de générer la couche suivante. On obtient donc un DD qui contient un sous-ensemble des solutions de $\mathcal{P}$, fournissant ainsi des bornes inférieures.

DD relaxés Dans la même veine, il est possible d'obtenir des bornes supérieures grâce aux DDs *relaxés* [1, 2] en fusionnant les nœuds excédentaires plutôt qu'en les supprimant. Pour ce faire, un opérateur de fusion d'états doit être spécifié pour le modèle PD étudié, dont le résultat conserve toutes les transitions comprises dans le DD exact correspondant. En général, la fusion peut introduire des solutions non-admissibles et fournit donc des bornes supérieures.

2.3 Séparation et évaluation

Sur base des DDs approximés, l'algorithme 2 explore l'espace des états du modèle PD par S&E [3]. Une file de sous-problèmes – c'est-à-dire des nœuds de DDs – à traiter est maintenue, et pour chacun de ceux-ci, un DD restreint est compilé afin d'obtenir des solutions admissibles et éventuellement améliorer la meilleure solution courante. Ensuite, la construction d'un DD relaxé joue un rôle double : celui d'identifier l'ensemble des sous-problèmes restant à explorer – appelé *ensemble-coupe exact* – mais également de calculer une borne supérieure pour chacun d'eux.

Définition 1 (Ensemble-coupe)**.** *Soit un DD relaxé $\overline{\mathcal{B}}$ enraciné en u_r, un ensemble-coupe $EC(\overline{\mathcal{B}})$ est un sous-ensemble des nœuds de $\overline{\mathcal{B}}$ tel que chaque chemin $u_r \rightsquigarrow t$ dans $\overline{\mathcal{B}}$ traverse au moins un nœud de $EC(\overline{\mathcal{B}})$.*

Algorithme 1 Compilation d'un DD, restreint ou relaxé, enraciné au nœud u_r et d'une largeur maximale W.

1: $i \leftarrow$ indice de la couche contenant u_r
2: $L_i \leftarrow \{u_r\}$
3: **for** $j = i$ **to** $n - 1$ **do**
4: $\quad \mathit{élagués} \leftarrow \emptyset$
5: $\quad$ **if** $j > i$ **then** // racines élaguées dans l'algorithme 2
6: $\quad\quad$ **for all** $u \in L_j$ **do**
7: $\quad\quad\quad$ **if** $Cache.contains(\sigma(u))$ **and** $v(u) \leq \theta(Cache.get(\sigma(u)))$ **then**
8: $\quad\quad\quad\quad \mathit{élagués} \leftarrow \mathit{élagués} \cup \{u\}$
9: $\quad L'_j \leftarrow L_j \setminus \mathit{élagués}$
10: $\quad$ **if** $|L'_j| > W$ **then**
11: $\quad\quad$ restriction ou relaxation de L'_j jusque W nœuds
12: $\quad L_{j+1} \leftarrow \emptyset$
13: $\quad$ **for all** $u \in L'_j$ **do**
14: $\quad\quad$ **if** $v(u) + \overline{v}_{gr.}(u) \leq \underline{v}$ **then**
15: $\quad\quad\quad$ **continue**
16: $\quad\quad$ **for all** $d \in D_j$ **do**
17: $\quad\quad\quad$ créer nœud u' avec $\sigma(u') = t_j(\sigma(u), d)$
18: $\quad\quad\quad$ créer arc $a = (u \xrightarrow{d} u')$ avec $v(a) = h_j(\sigma(u), d)$ et $l(a) = d$
19: $\quad\quad\quad$ $L_{j+1} \leftarrow L_{j+1} \cup \{u'\}$, $A \leftarrow A \cup \{a\}$

Par ailleurs, un nœud u de $\overline{\mathcal{B}}$ est dit *exact* si on obtient le même état PD en suivant tous les chemins $u_r \rightsquigarrow u$ dans $\overline{\mathcal{B}}$. Un ensemble-coupe *exact* (ECE) est un ensemble-coupe contenant uniquement des nœuds exacts. Il existe plusieurs manières de choisir l'ensemble-coupe exact, la plus populaire étant de sélectionner tous les nœuds appartenant à la couche la plus profonde dont tous les nœuds sont exacts.

2.4 Bornes supérieures

Étant donné un DD relaxé $\overline{\mathcal{B}}$, la *borne locale* [5] peut être calculée pour chaque nœud de $ECE(\overline{\mathcal{B}})$.

Définition 2 (Borne locale)**.** *Soit un DD relaxé $\overline{\mathcal{B}}$ et un nœud $u \in \overline{\mathcal{B}}$, la borne locale de u dans $\overline{\mathcal{B}}$ est définie par :*

$$\overline{v}_{loc.}(u \mid \overline{\mathcal{B}}) = \begin{cases} v^*(u \rightsquigarrow t \mid \overline{\mathcal{B}}), & \textit{si } (u \rightsquigarrow t) \in \overline{\mathcal{B}}, \\ -\infty, & \textit{sinon.} \end{cases} \quad (2)$$

Avec $v^*(u \rightsquigarrow t \mid \overline{\mathcal{B}})$ la valeur du meilleur chemin $u \rightsquigarrow t$ dans $\overline{\mathcal{B}}$.

D'autre part, une *borne supérieure grossière* $\overline{v}_{gr.}(u) \geq v^*(u \rightsquigarrow t \mid \mathcal{B})$ est calculée pour chaque nœud u avant de considérer les transitions qui en découlent [5], voir ligne 14 de l'algorithme 1. Soit $v^*(u \mid \mathcal{B})$ la plus haute valeur obtenue par un chemin terminant en u dans $\mathcal{B}$ et $\underline{v}$ une borne inférieure, si $v^*(u \mid \mathcal{B}) + \overline{v}_{gr.}(u) \leq \underline{v}$ alors le nœud u peut être élagué, c'est-à-dire que toutes les transitions qui en sont originaires peuvent être ignorées. Cette technique s'applique aux DDs restreints comme relaxés.
Pour décider si un nœud de $ECE(\overline{\mathcal{B}})$ doit être ajouté à la file de l'algorithme 2, la borne la plus stricte est utilisée à la ligne 20 : $\overline{v}(u' \mid \overline{\mathcal{B}}) = \min\left\{\overline{v}_{gr.}(u'), \overline{v}_{loc.}(u' \mid \overline{\mathcal{B}})\right\}$.

Algorithme 2 Algorithme de S&E avec DDs.

```
1: File ← {r}
2: Cache ← ∅
3: v̲ ← −∞
4: while File n'est pas vide do
5:    u ← meilleur nœud de File, retiré de File
6:    if v(u) + v̄(u) ≤ v̲ then
7:       continue
8:    if Cache.contains(σ(u)) then
9:       ⟨θ, exploré⟩ ← Cache.read(σ(u))
10:      if v(u) < θ or (v(u) = θ ∧ exploré) then
11:         continue
12:   B̲ ← Restreint(u)
13:   if v*(B̲) > v̲ then
14:      v̲ ← v*(B̲)
15:      x̲ ← x*(B̲)
16:   if B̲ n'est pas exact then
17:      B̄ ← Relaxé(u)
18:      mise à jour du Cache avec l'algorithme 3
19:      for all u' ∈ ECE(B̄) do
20:         if v*(u' | B̄) + v̄(u' | B̄) > v̲ then
21:            v(u') ← v*(u' | B̄), v̄(u') ← v̄(u' | B̄)
22:            ajouter u' à File
23: return (x̲, v̲)
```

3 Mémoïsation

Comme mentionné plus tôt, l'algorithme de S&E pour les DDs [3] n'exploite pas tout le potentiel des modèles PD puisque certains sous-problèmes sont explorés plusieurs fois sans pour autant qu'un meilleur chemin y menant ait été trouvé. L'idée générale des techniques introduites dans cet article est donc de mémoriser un *seuil d'exploration* pour chacun des sous-problèmes traités ou ajoutés à la file de S&E. Avant d'appliquer toutes les transitions possibles à chacun des nœuds lors de la compilation des DDs approximés, une comparaison est d'abord effectuée avec le potentiel seuil d'exploration en mémoire. L'expansion d'un nœud est seulement poursuivie si le meilleur chemin y menant dans le DD courant a une valeur supérieure au seuil d'exploration. Celui-ci combine deux seuils distincts, le premier relatif aux relations de dominance découvertes dans les DDs et le second à l'élagage effectué dans ceux-ci.

3.1 Seuils de dominance

Avec la compilation de DDs relaxés, on obtient une représentation partiellement exacte d'une partie de l'espace de recherche. Au sein de celle-ci, il est possible de déduire des relations de *dominance* entre solutions partielles.

Définition 3 (Dominance)**.** *Soient deux chemins $p_1 : r \rightsquigarrow u$ et $p_2 : r \rightsquigarrow u$ appartenant à un DD $\mathcal{B}$. On dit que p_1 domine p_2 – formellement noté $p_1 \succ p_2$ – si et seulement si $v(p_1) > v(p_2)$. Si $v(p_1) \geq v(p_2)$, on parle de dominance faible, notée $p_1 \succeq p_2$.*

Dans un DD relaxé $\overline{\mathcal{B}}$, le sous-problème correspondant à chaque nœud exact est décomposé de manière unique en plusieurs sous-problèmes appartenant à l'ECE de $\overline{\mathcal{B}}$, ceux-ci étant ajoutés dans la file de la S&E. Le premier scénario lors duquel un nœud u_1 doit être ré-exploré se produit lorsqu'un nouveau chemin terminant en u_1 permet de dominer le meilleur chemin $p^*(u_2 \mid \overline{\mathcal{B}})$ menant à un nœud $u_2 \in ECE(\overline{\mathcal{B}})$. Pour détecter ce scénario, on définit un *seuil de dominance individuel* de u_1 par rapport à u_2. Afin de simplifier les définitions suivantes, on pose $Succ.(u \mid \overline{\mathcal{B}}) = \{u\} \cup \{u' \mid (u \rightsquigarrow u') \in \overline{\mathcal{B}}\}$ l'ensemble des successeurs de u, y compris u lui-même.

Définition 4 (Seuil de dominance individuel)**.** *Soient deux nœuds exacts $u_1 \in \overline{\mathcal{B}}$ et $u_2 \in ECE(\overline{\mathcal{B}})$. Si $u_2 \in Succ.(u_1 \mid \overline{\mathcal{B}})$ alors le seuil de dominance individuel de u_1 par rapport à u_2 est défini par :*

$$\theta_{dom.i.}(u_1 \mid u_2, \overline{\mathcal{B}}) = v^*(u_2 \mid \overline{\mathcal{B}}) - v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) \quad (3)$$

et donne la valeur qu'un nouveau chemin $p_1 : r \rightsquigarrow u_1$ doit dépasser pour que $p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) \succ p^*(u_2 \mid \overline{\mathcal{B}})$.

Démonstration. Un nouveau chemin $p_1 : r \rightsquigarrow u_1$ permet de dominer $p^*(u_2 \mid \overline{\mathcal{B}})$ si $\overline{\mathcal{B}}$ contient un chemin $p_2 : u_1 \rightsquigarrow u_2$ tel que $p_1 \cdot p_2 \succ p^*(u_2 \mid \overline{\mathcal{B}})$, autrement dit $v(p_1) + v(p_2) > v^*(u_2 \mid \overline{\mathcal{B}})$. Le chemin $p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$ étant le meilleur chemin entre u_1 et u_2 dans $\overline{\mathcal{B}}$, la condition devient $v(p_1) + v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) > v^*(u_2 \mid \overline{\mathcal{B}})$ et de manière équivalente : $v(p_1) > v^*(u_2 \mid \overline{\mathcal{B}}) - v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$. □

De manière plus générale, on définit le *seuil de dominance* d'un nœud $u_1 \in \overline{\mathcal{B}}$ comme le seuil de dominance individuel minimal parmi tous les nœuds successeurs $u_2 \in ECE(\overline{\mathcal{B}})$.

Définition 5 (Seuil de dominance)**.** *Soit un nœud exact $u_1 \in \overline{\mathcal{B}}$, le seuil de dominance de u_1 dans $\overline{\mathcal{B}}$ est donné par :*

$$\theta_{dom.}(u_1 \mid \overline{\mathcal{B}}) = \min_{\substack{u_2 \in ECE(\overline{\mathcal{B}}) \\ u_2 \in Succ.(u_1 \mid \overline{\mathcal{B}})}} \theta_{dom.i.}(u_1 \mid u_2, \overline{\mathcal{B}}). \quad (4)$$

3.2 Seuils d'élagage

Le second cas de figure qui nécessite la ré-exploration d'un nœud se présente lorsqu'un nouveau chemin terminant en ce nœud permet d'éviter son élagage ou celui de l'un de ses successeurs.

Définition 6 (Seuil d'élagage individuel)**.** *Soient une borne inférieure $\underline{v}$, un nœud exact $u_1 \in \overline{\mathcal{B}}$ et un nœud élagué $u_2 \in \overline{\mathcal{B}}$ avec $\overline{v}(u_2 \mid \overline{\mathcal{B}})$ la borne grossière ou locale ayant permis de l'élaguer : $v^*(u_2 \mid \overline{\mathcal{B}}) + \overline{v}(u_2) \leq \underline{v}$. Si $u_2 \in Succ.(u_1 \mid \overline{\mathcal{B}})$, alors on définit le seuil d'élagage individuel de u_1 par rapport à u_2 :*

$$\theta_{él.i.}(u_1 \mid u_2, \overline{\mathcal{B}}) = \underline{v} - \overline{v}(u_2 \mid \overline{\mathcal{B}}) - v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) \quad (5)$$

et donne la valeur qu'un nouveau chemin $p_1 : r \rightsquigarrow u_1$ doit dépasser pour que $p_1 \cdot p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$ ne soit pas élagué.

Démonstration. Un nouveau chemin $p_1 : r \rightsquigarrow u_1$ combiné à un chemin $p_2 : u_1 \rightsquigarrow u_2$ dans $\overline{\mathcal{B}}$ évite l'élagage en u_2 si

Algorithme 3 Calcul du seuil d'exploration $\theta(u)$ de chaque nœud exact u d'un DD relaxé $\overline{\mathcal{B}}$ et mise à jour du cache.

1: $i \leftarrow$ indice de la couche du nœud racine de $\overline{\mathcal{B}}$
2: $(L_i, \ldots, L_n) \leftarrow Couches(\overline{\mathcal{B}})$
3: $\theta(u) \leftarrow \infty$ **for all** $u \in \overline{\mathcal{B}}$
4: **for** $j = n$ **down to** i **do**
5: **for all** $u \in L_j$ **do**
6: **if** $Cache.contains(\sigma(u))$ **and** $v^*(u \mid \overline{\mathcal{B}}) \leq \theta(Cache.read(\sigma(u)))$ **then**
7: $\theta(u) \leftarrow \theta(Cache.read(\sigma(u)))$
8: **else**
9: **if** $v^*(u \mid \overline{\mathcal{B}}) + \overline{v}_{gr.}(u) \leq \underline{v}$ **then**
10: $\theta(u) \leftarrow \underline{v} - \overline{v}_{gr.}(u)$
11: **else if** $u \in ECE(\overline{\mathcal{B}})$ **then**
12: **if** $v^*(u \mid \overline{\mathcal{B}}) + \overline{v}_{loc.}(u \mid \overline{\mathcal{B}}) \leq \underline{v}$ **then**
13: $\theta(u) \leftarrow \min\{\theta(u), \underline{v} - \overline{v}_{loc.}(u \mid \overline{\mathcal{B}})\}$
14: **else**
15: $\theta(u) \leftarrow v^*(u \mid \overline{\mathcal{B}})$
16: **if** u est exact **and** au-dessus de $ECE(\overline{\mathcal{B}})$ **then**
17: $explor\acute{e} \leftarrow$ false **if** $u \in ECE(\overline{\mathcal{B}})$ **else** true
18: $Cache.write(\sigma(u), \langle \theta(u), explor\acute{e} \rangle)$
19: **for all** arc $a = (u' \rightarrow u)$ **do**
20: $\theta(u') \leftarrow \min\{\theta(u'), \theta(u) - v(a)\}$

$v(p_1)+v(p_2)+\overline{v}(u_2 \mid \overline{\mathcal{B}}) > \underline{v}$. Le chemin $p^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$ étant le meilleur chemin entre u_1 et u_2 dans $\overline{\mathcal{B}}$, la condition devient $v(p_1) + v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}}) + \overline{v}(u_2 \mid \overline{\mathcal{B}}) > \underline{v}$, ou encore : $v(p_1) > \underline{v} - \overline{v}(u_2 \mid \overline{\mathcal{B}}) - v^*(u_1 \rightsquigarrow u_2 \mid \overline{\mathcal{B}})$. □

Puisque dépasser un seul seuil d'élagage individuel suffit à requérir la ré-exploration d'un nœud, on définit le seuil d'élagage de la manière suivante.

Définition 7 (Seuil d'élagage)**.** *Soient une borne inférieure $\underline{v}$ et un nœud exact $u \in \overline{\mathcal{B}}$, le seuil d'élagage de u dans $\overline{\mathcal{B}}$ est défini comme :*

$$\theta_{él.}(u_1 \mid \overline{\mathcal{B}}) = \min_{\substack{u_2 \in Succ.(u) \\ v^*(u_2|\overline{\mathcal{B}})+\overline{v}(u_2|\overline{\mathcal{B}}) \leq \underline{v}}} \theta_{él.i.}(u_1 \mid u_2, \overline{\mathcal{B}}). \quad (6)$$

3.3 Calcul et utilisation des seuils

Les deux seuils présentés dans cette section sont combinés en un seul *seuil d'exploration* indiquant à partir de quelle valeur un nœud doit être ré-exploré.

Définition 8 (Seuil d'exploration)**.** *Soit un DD relaxé $\overline{\mathcal{B}}$ et un nœud exact $u \in \overline{\mathcal{B}}$, le seuil d'exploration de u dans $\overline{\mathcal{B}}$ est donné par : $\theta(u \mid \overline{\mathcal{B}}) = \min\{\theta_{dom.}(u \mid \overline{\mathcal{B}}), \theta_{él.}(u \mid \overline{\mathcal{B}})\}$.*

Ces seuils d'exploration peuvent être calculés grâce à simple traversée de bas en haut du DD relaxé, comme formalisé par l'algorithme 3. Les seuils de dominance sont initialisés à la ligne 15 tandis que les seuils d'élagage le sont aux lignes 10 et 13. La ligne 20 se charge ensuite de propager les seuils vers la couche supérieure. Pour chaque nœud exact u appartenant à l'ECE ou se situant au-dessus de celui-ci, la ligne 18 met le seuil d'exploration associé $\theta(u)$ en cache. Si par ailleurs le cache contient déjà un seuil

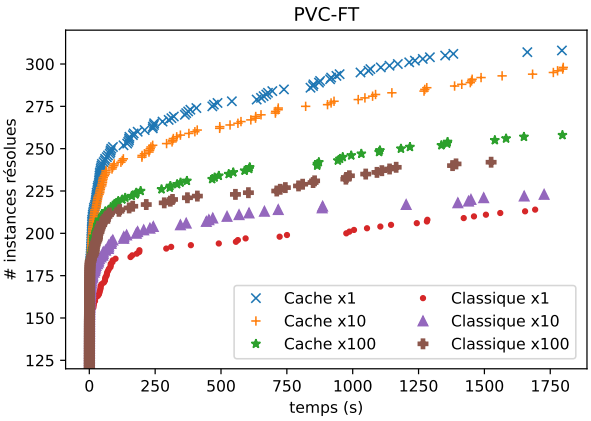


FIGURE 1 – Nombre d'instances résolues en fonction du temps pour différentes configurations de l'algorithme.

$\theta(u)$ supérieur à $v^*(u \mid \overline{\mathcal{B}})$, celui-ci peut directement être utilisé. Finalement, le cache est mis à profit dans les algorithmes 1 et 2 pour élaguer les nœuds dont l'exploration peut être évitée. Dans l'algorithme 2, un booléen $explor\acute{e}$ intervient pour décider si un nœud dont la valeur est égale au seuil d'exploration doit être traité ou non, ce qui est vrai pour les nœuds de l'ECE.

4 Validation expérimentale

Des expériences ont été conduites sur une suite de 467 instances du problème du voyageur de commerce avec fenêtres de temps, disponible à l'adresse suivante : `https://lopez-ibanez.eu/tsptw-instances`. Le modèle utilisé est détaillé dans [5]. L'algorithme *Classique* et avec *Cache* disposent de 30 minutes pour résoudre chaque instance. Une largeur maximale dynamique est imposée aux DDs suivant la formule $n \times (i+1) \times \alpha$ avec n le nombre de variables, i l'indice de la couche considérée et $\alpha \in \{1, 10, 100\}$. Comme présenté par la figure 1, la version *Cache* est supérieure à *Classique*, quelle que soit la largeur maximale utilisée, ce qui confirme l'utilité de mémoriser certains résultats intermédiaires. De plus, alors que les performances de *Classique* s'améliorent lorsque la largeur des DDs augmente, la tendance inverse est observée pour *Cache*. Il semble donc que des DDs assez étroits suffisent à identifier des parties de l'espace de recherche qui ne doivent plus être explorées. L'élagage ainsi obtenu permet aux DDs plus étroits de fournir de meilleures bornes, tout en étant très peu coûteux à compiler.

5 Conclusion

Dans cet article, un mécanisme visant à éviter au maximum l'exploration répétée de certains sous-problèmes, dans le cadre de l'optimisation discrète avec DDs, a été introduit. Pour chaque sous-problème rencontré, un seuil d'exploration est mémorisé. Celui-ci, sur base de relations de dominance et d'élagage, conditionne la nécessité de ré-explorer un sous-problème. Un algorithme permettant de calculer ef-

ficacement ces seuils a par ailleurs été présenté. Les résultats des expériences conduites sur le problème du voyageur de commerce avec fenêtres de temps démontrent l'impact significatif du mécanisme présenté.

Références

[1] Henrik Reif Andersen, Tarik Hadzic, John N Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In *International Conference on Principles and Practice of Constraint Programming*, pages 118–132. Springer, 2007.

[2] David Bergman, Andre A Cire, Willem-Jan van Hoeve, and John N Hooker. Optimization bounds from binary decision diagrams. *INFORMS Journal on Computing*, 26(2) :253–268, 2014.

[3] David Bergman, Andre A Cire, Willem-Jan van Hoeve, and John N Hooker. Discrete optimization with decision diagrams. *INFORMS Journal on Computing*, 28(1) :47–66, 2016.

[4] David Bergman, Andre A Cire, Willem-Jan van Hoeve, and Tallys Yunes. Bdd-based heuristics for binary optimization. *Journal of Heuristics*, 20(2) :211–234, 2014.

[5] Xavier Gillard, Vianney Coppé, Pierre Schaus, and André Augusto Cire. Improving the filtering of branch-and-bound mdd solver. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 231–247. Springer, 2021.

GrailSolver, à la dernière itération de l'obtention du graal de la programmation

Vinasetan Ratheil Houndji[1], Généreux Akotenou[1], Afis Malick Kousse[1], Klaus Bonou Selegbe[1]

[1] Institut de Formation et de Recherche en Informatique (IFRI), Université d'Abomey-Calavi, Bénin

Résumé

L'une des promesses de la programmation déclarative est d'atteindre le graal de la programmation. Aujourd'hui, grâce aux avancées des modèles de langage, nous sommes à la dernière étape pour converger vers cet objectif. Dans ce papier, nous présentons les travaux préliminaires de la première version de solveur `GrailSolver` *qui prend en entrée une description en texte du problème et qui le résout en présentant la formulation mathématique.* `GrailSolver` *est actuellement basé sur GPT-3 pour le traitement automatique du langage et ORTools pour la résolution du problème. Les tests montrent que le prototype est opérationnel dans plusieurs langues sur des problèmes simples et qu'il peut se généraliser assez rapidement.*

Mots-clés

solveur, graal de la programmation, problème de satisfaction de contraintes, traitement du langage naturel, optimisation.

Abstract

One of the promises of declarative programming is to achieve the "holy grail" of programming. Today, thanks to advances in language models, we are at the final stage of converging toward this goal. In this paper, we present the preliminary work of the first version of the `GrailSolver` *solver, which takes a text description of the problem as input and solves it by presenting the mathematical formulation.* `GrailSolver` *is currently based on GPT-3 for Natural Language Processing and ORTools for problem-solving. The tests show that the prototype is operational in several languages on simple problems and can be generalized quickly.*

Keywords

solver, grail of programming, constraint satisfaction problem, natural language processing, optimization

1 Introduction

Depuis les années 90, la programmation par contraintes était déjà considérée comme l'une des directions de recherche en informatique les plus prometteuses pour l'obtention du "saint graal" de la programmation, l'idée étant de donner une description du problème en langage naturel et de laisser la machine proposer les solutions. En effet, la programmation par contraintes nous rapproche de plus en plus de la "vraie" programmation déclarative : l'utilisateur énonce le problème, l'ordinateur le résout [1, 2, 3, 4]. Il faut noter que les solveurs de problèmes de satisfaction de contraintes et d'optimisation sont très efficaces actuellement, même pour la résolution de problèmes relativement grands et complexes. Malheureusement, l'impact de la programmation par contraintes est toujours limité en raison d'un manque de convivialité pour un utilisateur non expert [3]. Aujourd'hui, grâce aux progrès très important réalisés en Traitement Automatique de Langage (TAL), il est désormais possible d'automatiser de bout en bout le processus d'analyse, de modélisation et de résolution à partir d'une description en langage naturel du problème. Pour en démontrer la faisabilité, nous avons mis en place un premier prototype du solveur open-source `GrailSolver` accessible via **https://github.com/grail-solver**. La version actuelle de `GrailSolver` utilise GPT-3 [5] comme modèle de TAL et le framework d'optimisation OR-Tools [6] pour la résolution du problème généré à partir du texte.

La suite de ce papier est constituée de deux parties essentielles. Tout d'abord, nous décrivons notre approche en détail, en mettant l'accent sur les principales étapes de traitement de `GrailSolver`. Ensuite, nous présentons un exemple pratique pour illustrer le fonctionnement du solveur.

2 Notre approche

`GrailSolver` prend en entrée une description en texte d'un problème à résoudre, génère en sortie une formulation mathématique et résout le problème. L'organigramme de la figure 1 donne un aperçu de la structuration en couche du solveur. Nous décrivons ci-dessous les principales étapes du processus, de l'acquisition du texte descriptif du problème jusqu'à la résolution du problème, en passant par l'analyse et l'extraction des caractéristiques décisionnelles utiles telles que les contraintes, les domaines et les variables.

Le module d'acquisition du texte. L'objectif de `GrailSolver` est de permettre la résolution de problèmes d'optimisation à partir d'une description textuelle du problème. Ce premier module a donc pour but de permettre à un utilisateur de donner une description textuelle du problème à résoudre. `GrailSolver` offre à cet effet une interface d'acquisition de texte toute simple et très intuitive.

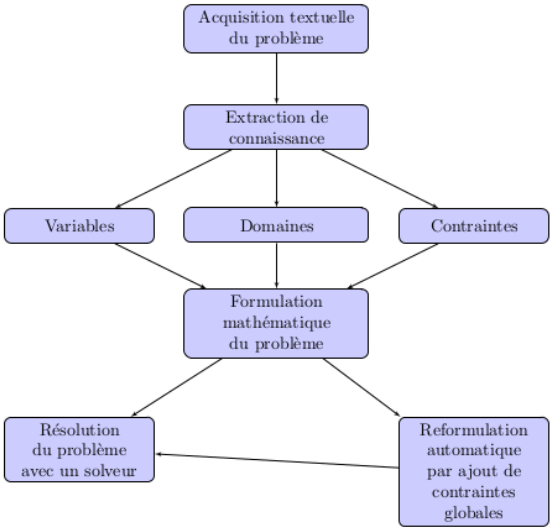


FIGURE 1 – Principales couches pour la résolution d'un problème avec GrailSolver

Le module d'extraction de connaissance. Il s'agit d'une partie essentielle du système. Ce module est déclenché par le précédent et prend en entrée la description du problème. Le module d'extraction de connaissance a alors pour but de procéder au traitement dudit texte afin d'y extraire les informations nécessaires à une modélisation mathématique du problème. Il s'agit notamment de :

- l'identification des variables impliquées dans le problème c'est à dire les éléments du problème qui peuvent prendre des valeurs différentes ;
- l'identification du domaine de chacune des variables ;
- l'identification des contraintes appliquées à ces variables représentant ainsi les relations entre les variables.

L'identification de ces trois éléments indispensables définissant un problème de satisfaction de contraintes requièrent des algorithmes de TAL. Ainsi, sur cette couche, nous avons incorporé l'un des plus récents LLM (Large Language Model) : le modèle GPT-3. Ce modèle est considéré comme l'un des meilleurs en la matière actuellement. Nous avons développé à cet effet `grail_xtract`, une bibliothèque Python conçue pour tirer parti de l'API GPT-3 accessible via : **https://github.com/grail-solver/GrailXtract**. Cette bibliothèque nous permet de spécifier les critères d'extraction de données et de choisir le format de retour des données extraites. Ainsi, pour chaque problème soumis, notre solveur envoie une requête à GPT-3 et récupère les données nécessaires à la résolution du problème. Un protocole a été mis en place pour permettre à ce module de communiquer avec le module suivant, celui de la formulation mathématique.

Le module de formulation mathématique du problème. Sur cette avant dernière couche du solveur, nous procédons à la modélisation mathématique du problème en fonction des informations extraites de la couche précédente. En effet, un protocole est établi pour définir le format des données en sortie du module précédent. Ce format se présente comme suit :

```
{
    "llm_status": "success",
    "variables": [
        {
            "id": "Integer",
            "var_name": "String",
            "type": "VariableType",
            "domain_type": DomainType,
            "domain_values": [values]
        }
    ],
    "constraints":[
        {
            "left_part": Equation,
            "relation": Enum.Relation,
            "right_part": Equation
        }
    ]
}
```

Un dictionnaire d'objets est attendu. Le premier objet est une liste de toutes les variables identifiées dans le module d'extraction de connaissance. Sur chaque variable sont renseignées les informations suivantes :

- id : un identifiant ;
- var_name : un nom de variable ;
- $domain\_type$: le type du domaine de valeurs de la variable. domain_type est une énumération avec les valeurs ci-après : $INTERVAL$ pour des domaines de valeurs qui correspondent à des intervalles ($[a, b]$, $[a, +\infty[$, $]-\infty, b]$, $]-\infty, +\infty[$) ;
- $domain\_values$: les valeurs que peuvent prendre la variable. C'est une liste de valeurs. Elle comporte deux nombres $(a, b \ ou \infty)$ lorsque le type de domaine est $INTERVAL$.

Le deuxième objet est une liste de contraintes devant s'appliquer aux variables identifiées. Pour chaque contrainte, on renseigne :

- $left\_part$: C'est une liste d'expressions littérales et d'opérateurs. Les expressions littérales sont renseignées sous forme de liste dont le deuxième élément est le nom de la variable formatée avec son id sous la forme : Var_id ; et le premier élément représente le coefficient de la variable dans l'expression. On pourrait avoir comme exemple : $[[5, Var\_1], " + ", [50, Var\_2], " + ", [100, Var\_3]]$ qui représente sous forme mathématique : $5A + 50B + 100C$ avec $A, B \ et \ C$ trois variables ayant respectivement pour id 1, 2 et 3.
- $relation$: Cette propriété peut prendre les valeurs : $<, >, <=, >=, =, !=$. Elle permet de définir le type de contrainte à appliquer à la/aux variable(s) concernée(s).

— $right\_part$: Ce champ représente également une liste d'expressions littérales tout comme $left\_part$. Celle-ci se situe à droite de la métrique.

Le module de résolution du problème. Le module de résolution de problème propose un protocole qui se veut universel pour les solveurs existants. Un format du problème à résoudre est donc proposé au niveau de ce module. Le but étant de maintenir l'outil le plus flexible possible. Plusieurs solveurs de problèmes de satisfaction de contraintes ou d'optimisation peuvent donc exploiter le format proposer pour résoudre les problèmes soumis. Cette première version de `GrailSolver` se base sur ORTools [6].

Reformulation par ajout de contraintes globales. La modélisation des problèmes est faite avec les contraintes mathématiques de base. Ces contraintes agissent indépendamment sur de petits ensembles de variables, et ne sont pas aussi efficace que les contraintes globales. L'architecture flexible de `GrailSolver` facilite l'ajout des contraintes globales basées sur celles existantes.

3 Le solveur GrailSolver

Nous présentons dans cette section les interfaces de `GrailSolver` puis nous déroulons un cas pratique.

Description de l'interface. La figure 2 présente l'interface de `GrailSolver`. Il s'agit principalement de trois blocs : *Editor*, *Formulation* et *Solution*. Le premier bloc *Editor* est une zone de texte qui permet d'entrer le problème de CP à résoudre. Le second bloc *Formulation* affiche la formulation mathématique du problème de CP entré une fois le bouton «submit» cliqué. C'est la sortie du module de formulation mathématique du problème. Le dernier bloc *Solution* affiche la solution du problème après sa résolution. Chaque bloc est extensible. Des images complémentaires des interfaces de GrailSolver sont disponibles au niveau du répertoire github : **https://github.com/grail-solver/grail_solver_ui/tree/main/src/Assets/ui_interfaces**

FIGURE 2 – Page d'accueil `GrailSolver`

Cas pratique. Dans ce cas pratique, nous essayons de résoudre un problème tout simple de satisfaction de contraintes. La description du problème est la suivante : Un fermier a 1000$. Avec les 1000$ il souhaite acheter 100 animaux de différentes espèces : des poussins, des porcs et des bœufs. On suppose qu'un poussin coûte 5$, un cochon 50$ et un bœuf 100$. Il souhaite avoir au moins 1 animal de chaque espèce et veut dépenser tous les 1000$ dont il dispose. Combien de poussins, de cochons et de bœufs devrait-il acheter ?

Pour un tel problème, nous avons respectivement les sorties suivantes :
- Formulation du problème suivant le protocole défini.

```
{
  "constraints":[
    {
      "left_part":[[1,"Var_1"],"+",[1,"
         Var_2"],"+",[1,"Var_3"]],
      "relation":"=",
      "right_part":[100]
    },
    {
      "left_part":[[5,"Var_1"],"+",[50,
         "Var_2"],"+",[100,"Var_3"]],
      "relation":"=",
      "right_part":[1000]
    },
    {
      "left_part":[[1,"Var_1"]],
      "relation":">=",
      "right_part":[1]
    },
    {
      "left_part":[[1,"Var_2"]],
      "relation":">=",
      "right_part":[1]
    },
    {
      "left_part":[[1,"Var_3"]],
      "relation":">=",
      "right_part":[1]
    }],
  "variables":[{
      "id":1,
      "name":"poussins",
      "type":"Integer",
      "domaine_type":"INTERVAL",
      "domaine_values":"[0,1000]"
    },
    {
      "id":2,
      "name":"cochons",
      "type":"Integer",
      "domaine_type":"INTERVAL",
      "domaine_values":"[0,1000]"
    },
    {
      "id":3,
      "name":"boeufs",
      "type":"Integer",
      "domaine_type":"INTERVAL",
      "domaine_values":"[0,1000]"
```

```
    }],
  "llm_status":"success"
}
```


- Présentation de la solution du modèle de CP


```
[
  [
      {
        "var_name":"poussins",
        "value":90
      },
      {
        "var_name":"cochons",
        "value":9
      },
      {
        "var_name":"boeufs",
        "value":1
      }
  ],
  [
    {
      "var_name":"poussins",
      "var_domain":[0,1000]
    },
    {
      "var_name":"cochons",
      "var_domain":[0,1000]
    },
    {
      "var_name":"boeufs",
      "var_domain":[0,1000]
    }
  ],
  [
    "1*poussins + 1*cochons + 1*boeufs
       = 100 ",
    "5*poussins + 50*cochons + 100*
       boeufs  = 1000 ",
    "1*poussins  >= 1 ",
    "1*cochons  >= 1 ",
    "1*boeufs  >= 1 "
  ]
]
```


La solution du problème avec GrailSolver se présente sur la figure 3.
Une cinquantaine de cas de test a été exécutée et sont accessibles via : **https://github.com/grail-solver/solver_test_case**. Les résultats montrent qu'en moyenne pour 60% des cas de test, GrailSolver fournit une formulation mathématique et résout les problèmes associés.

4 Conclusion et perspectives

Dans ce papier, nous avons présenté les premiers travaux liés à la mise en place de `GrailSolver`, un solveur de problème de satisfaction de contraintes et d'optimisation

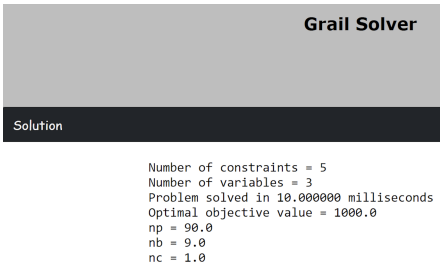


FIGURE 3 – `GrailSolver` : solution du problème du fermier

qui peut prendre en entrée la description en texte dans la plupart des langues internationales et résout le problème après avoir présenté une formulation mathématique. Actuellement, il est basé sur le modèle de TAL de GPT-3 et le solveur ORTools. Grâce à sa flexibilité, `GrailSolver` peut être utilisé avec d'autres modèles de TAL et d'autres solveurs.
Il faut noter que la précision des résultats fournis par `GrailSolver` dépend de la qualité de la description du problème et surtout de la détermination des variables et des contraintes par le modèle de TAL utilisé. En effet, bien que très efficaces, les modèles de langage ne sont pas encore parfaits. Ce qui peut entraîner des erreurs dans la formulation des contraintes et ainsi conduire à des résultats incorrects. Malgré cette limite, `GrailSolver` reste une solution intéressante et prometteuse. Les défis futurs consisteront à mettre en place des algorithmes efficaces qui permettront d'ajouter de façon automatique des contraintes globales et travailler sur des modèles de TAL plus spécifiques aux problèmes de satisfaction de contraintes. La librairie est open source, accessible via GitHub, https://github.com/grail-solver et ouverte à toute contribution.

Références

[1] E.C, Freuder, In Pursuit of the Holy Grail, *Constraints*, Vol. 2, pp. 57-61, 1997.

[2] E.C, Freuder, Progress towards the Holy Grail, *Constraints*, Vol. 23, pp. 158-171, 2018.
Freuder, E.C. Progress towards the Holy Grail. Constraints 23, 158–171 (2018). https ://doi.org/10.1007/s10601-017-9275-0

[3] B. O'Sullivan, Automated Modelling and Solving in Constraint Programming, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10)*. July 2010, pp. 1493–1497.

[4] R.A. Niemeijer and B. de Vries and J. Beetz, Freedom through constraints : User-oriented architectural design, *Advanced Engineering Informatics*, Vol. 28, pp. 28-36, 2014.

[5] OpenAI, GPT-3, *https ://openai.com/product*, 2022.

[6] Laurent Perron and Vincent Furnon, OR-Tools, Google, *https ://developers.google.com/optimization/*, 2019.

JFPC 4 - 4 juillet 2023, 16h30 - 18h00

Analyse de Code Automatique: Revisiter l'Inférence de Préconditions via l'Acquisition de Contraintes *

G. Menguy[1], S. Bardin[1], N. Lazaar[2], A. Gotlieb[3]

[1] Université Paris-Saclay, CEA, List, Palaiseau, France
[2] LIRMM, University of Montpellier, CNRS, Montpellier, France
[3] Simula Research Laboratory, Oslo, Norway

Résumé

Les annotations de programme, sous forme de pré/post-conditions de fonctions, sont cruciales pour accomplir différentes tâches, de l'ingénierie logicielle à la vérification de code. Malheureusement, ces annotations sont rarement fournies et doivent donc être rétro-ingéniées manuellement. Dans notre article, nous étudions comment l'acquisition de contraintes peut être utilisée pour inférer des préconditions. Cela a conduit à PRECA*, un outil qui infère des préconditions à partir d'observations d'exécution du code uniquement, et assurant des garanties claires de correction.*

Mots-clés

Acquisition de contraintes, analyse de code, préconditions

Abstract

Program annotations under the form of function pre/post-conditions are crucial for many software engineering and program verification applications. Unfortunately, these are rarely available and must be retrofit by hand. This paper explores how Constraint Acquisition (CA) can be leveraged to automatically infer program preconditions. This leads to PRECA*, which infers preconditions from input-output observations only, and presents clear correctness guarantees.*

Keywords

Constraint acquisition, code analysis, preconditions

1 Introduction

Les annotations de code sous la forme de pré- et post-conditions [8, 5, 3] sont cruciales en ingénierie logicielle et en vérification formelle de code. Elles permettent d'améliorer la compréhension du code pour les utilisateurs et pour les outils d'analyse automatique de code [9, 7]. Malheureusement, ces annotations sont rarement fournies par les développeurs et doivent donc être rétro-ingéniées à la main, ce qui limite leur intérêt, en particulier pour les composants tiers dont le code source n'est pas disponible.

Problème. De nombreuses méthodes ont été proposées pour inférer automatiquement des préconditions à partir du code. Cependant, l'état de l'art n'est toujours pas satisfaisant. En effet, même si les approches en *boîte blanche* basées sur l'analyse statique du code [8, 5, 3, 2] peuvent être utiles, elles présentent de nombreuses limites en termes de précision et ont du mal à être scalables. De plus, la gestion de structures de code complexes comme les boucles, la récursion et la mémoire dynamique représente un véritable défi. Les approches basées sur les exemples d'exécution du code pour inférer les annotations, appelées "boîte noire", ont été proposées pour dépasser les limites des approches basées sur l'analyse statique du code, appelées "boîte blanche" [4, 11, 6]. Cependant, la qualité des annotations inférées dépend fortement de la qualité des cas de test utilisés. Les cas de test peuvent être générés aléatoirement, fournis par l'utilisateur ou générés pendant l'inférence. Malheureusement, l'état de l'art ne fournit pas de garanties claires de correction.

Acquisition de contraintes. La programmation par contraintes (CP) [12] a connu de considérables avancées ces quarante dernières années. Cependant, modéliser un problème comme un réseau de contraintes reste une tâche difficile. Des méthodes d'acquisition de contraintes (CA) ont donc été proposées pour aider les utilisateurs non-experts. Par exemple, CONACQ infère un réseau de contraintes représentant le concept utilisateur à partir de solutions et de non-solutions classifiées par l'utilisateur. Le domaine de la recherche en CA est actif et a proposé de nombreuses extensions, telles que l'utilisation de requêtes partielles [1]. Bien que CONACQ offre des garanties théoriques fortes, ce type de système est difficile à utiliser en pratique car il nécessite de soumettre un grand nombre d'exemples (appelés requêtes) à l'utilisateur. Toutefois, en analyse de code, le nombre de requêtes n'est pas limitant car elles peuvent être classifiées automatiquement.

Objectifs et contributions. Dans ce papier, nous proposons une nouvelle approche pour l'inférence de préconditions en boîte noire basée sur l'acquisition de contraintes active. À notre connaissance, il s'agit de la première application de cette méthode en analyse de code. Notre approche, PRECA, bénéficie de meilleures propriétés théoriques que l'état de l'art. En effet, si notre langage est suffisamment expressif pour représenter la weakest-precondition (WP) [8] de la fonction (i.e., la plus générale et donc la meilleure), alors PRECA est sûr de l'inférer.

Nous décrivons également une spécialisation de PRECA

*Cet article se base sur des résultats publiés à IJCAI-ECAI 2022 [10].

pour l'inférence de préconditions sur la mémoire. Pour cela, nous avons développé un langage de contraintes dédiées gérant la validité, l'aliasing et le déréférencement des pointeurs. Par exemple, PRECA sur la fonction void find_first (int∗ a, int m, int∗ b, int n) infère la WP $(m > 0 \Rightarrow valid(a)) \wedge (m > 0 \wedge n > 0 \Rightarrow valid(b))$. De plus, nous avons proposé une stratégie pour accélérer l'inférence. Cette stratégie repose sur l'observation que les requêtes positives suppriment une plus grande partie de l'espace de recherche que les requêtes négatives. Nous générons donc en priorité des requêtes avec peu de pointeurs invalides et qui aliasent, ayant peu de chance de mener à un bug et qui seront donc probablement classées positives. Combinée avec un "background knowledge", cela permet d'accélérer significativement PRECA (cf. Table 1). Enfin, nous avons évalué expérimentalement notre approche sur un benchmark de 50 fonctions provenant de librairies standards (comme string.h) ou d'exemples présentés dans l'état de l'art. Nous avons comparé PRECA à trois autres méthodes d'inférence de préconditions en boîte noire, à savoir Daikon [4], PIE [11] et l'approche de Gehr et al. [6], que nous avons réimplémentée. Nous avons également comparé notre approche à l'approche d'inférence de préconditions en boîte blanche P-Gen [13]. Notre étude montre que PRECA est capable d'inférer plus de préconditions que ses concurrents (cf. Table 1). En particulier, nous avons observé que PRECA est capable d'inférer plus de weakest-preconditions en 5s que les concurrents en 1h. Cela reste vrai même face au concurrent en boîte blanche P-Gen qui a accès au code et est donc avantagé.

	1s		5s		5 mins		1h	
	$\#WP_\top$	$\#WP_Q$	$\#WP_\top$	$\#WP_Q$	$\#WP_\top$	$\#WP_Q$	$\#WP_\top$	$\#WP_Q$
Daikon	1.4/50	0.4/44	1.6/50	0.4/44	1.6/50	0.4/44	1.6/50	0.4/44
↳ PRECA	2/50	1/44	2/50	1/44	2/50	1/44	2/50	1/44
↳ Both	3.3/50	0/44	5.7/50	0/44	5.7/50	0/44	5.7/50	0/44
PIE	16.4/50	4.7/44	16.4/50	4.7/44	17.7/50	4.7/44	17.7/50	5.3/44
↳ PRECA	5/50	3/44	5/50	3/44	5/50	3/44	5/50	3/44
↳ Both	25.3/50	11.3/44	25.4/50	11.3/44	26.4/50	11.3/44	28.4/50	11.3/44
Gehr et al.	8.0/50	5.0/44	16.8/50	8.1/44	26.1/50	10.1/44	26.1/50	10.3/44
↳ PRECA	37/50	15/44	43/50	17/44	46/50	18/44	46/50	18/44
PRECA	29/50	11/44	**38/50**	**16/44**	**46/50**	**18/44**	**46/50**	**18/44**
↳ BK	15/50	8/44	38/50	16/44	45/50	18/44	46/50	18/44
↳ Preproc.	19/50	9/44	36/50	16/44	45/50	18/44	46/50	18/44
↳ ∅	13/50	7/44	35/50	15/44	45/50	18/44	46/50	18/44
↳ Random	29.9/50	12.1/44	29.9/50	12.1/44	30.0/50	12.1/44	30.0/50	12.1/44
P-Gen	**34/50**	**13/44**	37/50	15/44	37/50	15/44	37/50	15/44

Le nombre de Weakest Precondition inférées sans (resp. avec) une postcondition est représenté par $\#WP_\top$ (resp. $\#WP_Q$). Nous étudions trois variations de Daikon et PIE : (i) l'original (surligné) sur 100 exemples aléatoires ; (ii) sur les exemples générés par PRECA ; (iii) sur des exemples aléatoires et de PRECA. Nous examinons également la méthode actif originale de Gehr et al. (surlignée) et nous lui donnons les exemples de PRECA. Enfin, nous étudions PRECA avec son "background knowledge" et son prétraitement (surligné), avec seulement son "background knowledge" (BK), avec seulement le prétraitement (Preproc.), sans aucun des deux (∅) et en mode passif avec 100 requêtes aléatoires (Random). Étant donné que P-Gen est une méthode statique, nous ne considérons que sa forme originale.

TABLE 1 – Results depending on the time budget

2 Conclusion

Nous présentons la première application de l'acquisition de contraintes pour l'inférence de préconditions, qui est un problème majeur en analyse de code et en méthodes formelles. Cette approche représente la première méthode d'inférence de préconditions totalement boîte noire avec des garanties de correction solides. De plus, nos expériences sur l'inférence de préconditions orientées vers la mémoire ont montré que PRECA améliore significativement l'état de l'art, ce qui démontre l'intérêt de l'acquisition de contraintes dans ce domaine.

3 Remerciements

Ce travail a bénéficié du soutien de l'Institut de Cyber sécurité d'Occitanie (ICO), financé par la Région Occitanie en France, et du programme de recherche et d'innovation Horizon 2020 de l'Union européenne (projet TAILOR).

Références

[1] Christian Bessiere, Remi Coletta, Emmanuel Hebrard, George Katsirelos, Nadjib Lazaar, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Constraint acquisition via partial queries. In *IJCAI*, 2013.

[2] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *VMCAI'13*. Springer, 2013.

[3] Edsger W Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 1968.

[4] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *TSE*, 2001.

[5] Robert W Floyd. Assigning meanings to programs. In *Program Verification*. Springer, 1993.

[6] Timon Gehr, Dimitar Dimitrov, and Martin Vechev. Learning commutativity specifications. In *CAV'15*, 2015.

[7] Patrice Godefroid, Shuvendu K Lahiri, and Cindy Rubio-González. Statically validating must summaries for incremental compositional dynamic test generation. In *SAS*. Springer, 2011.

[8] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *CACM*, 1969.

[9] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c : A software analysis perspective. *Formal Aspects of Computing*, 2015.

[10] Grégoire Menguy, Sébastien Bardin, Nadjib Lazaar, and Arnaud Gotlieb. Automated program analysis : Revisiting precondition inference through constraint acquisition. In Lud De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 1873–1879. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Main Track.

[11] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 2016.

[12] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[13] Mohamed Nassim Seghir and Daniel Kroening. Counterexample-guided precondition inference. In *ESOP*. Springer, 2013.

Placement optimal de moniteurs dans un réseau pour la tomographie booléenne

A. Burlats[1], P. Schaus[1], C. Pelsser[1]

[1] UCLouvain, ICTEAM

Résumé

La tomographie booléenne permet de détecter les pannes dans un réseau et de les situer à partir seulement d'un sous-ensemble de noeuds moniteurs. On dit qu'un réseau est k-identifiable si cet ensemble de moniteurs permet de situer jusqu'à k pannes simultanées sans ambiguïté. Les moniteurs impliquant un coût, il faut faire un compromis entre identifiabilité et nombre de moniteurs. Nous présentons et comparons ici un modèle PLNE et un modèle PPC pour trouver le plus petit ensemble de moniteur de manière à garantir la couverture ou la 1-identifiabilité du réseau.

Mots-clés

Programmation par contraintes, tomographie booléenne, k-identifiabilité, supervision de réseaux, programmation linéaire en nombres entier

Abstract

With boolean tomography it is possible to detect and locate failures in a network with only a subset of nodes as monitors. A network is k-identifiable if the set of monitors is able to detect up to k simultaneous failures without ambiguity. Monitors are costly, thus a trade between identifiability and monitors quantity is necessary. We present and compare a MILP model and a CP model to find the smallest set of monitors such that all failures in a network are covered or identifiable.

Keywords

Constraint programming, boolean tomography, networks monitoring, k-identifiability, mixed-integer linear programming

1 Introduction

Pour assurer le bon fonctionnement d'un réseau, il est important d'être en capacité de le superviser. Il existe plusieurs approches pour cela, parmi elles, la tomographie consiste à combiner des mesures bout-en-bout entre des moniteurs à des méthodes d'inférences afin d'estimer l'état du réseau. L'intérêt de cette méthode est qu'elle ne nécessite qu'un sous-ensemble de moniteurs parmi les noeuds du réseau, permettant ainsi de superviser certaines zones où il est compliqué de placer des moniteurs internes.

La tomographie booléenne se concentre sur la détection et localisation d'échecs dans le réseau (pannes, congestions, ...).

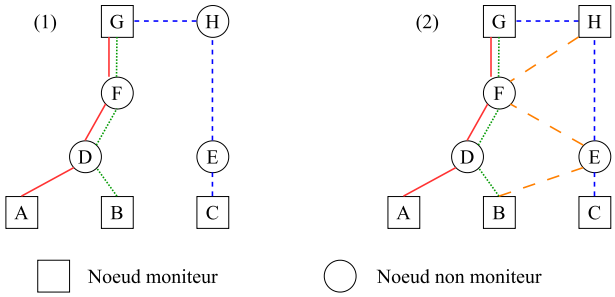


FIGURE 1 – Illustration de la notion de 1-identifiabilité (chaque couleur et style de ligne représente une route) : (A) Tous les noeuds sont couverts mais non 1-identifiables (B) Tous les noeuds sont 1-identifiables.

Dans ce contexte, cette méthode présente un avantage supplémentaire car elle est capable de détecter des pannes plus complexes telles que les *échecs silencieux* [11]. Dans ces situations, bien que tous les équipements soient fonctionnels, la circulation sur le réseau peut être bloquée en raison de conflits entre différents protocoles ou d'erreurs dans leur implémentation.

Concrètement, avec cette approche, les moniteurs s'envoient des messages à travers des *routes de mesures*. Lorsqu'une panne se situe au niveau d'un élément (noeud ou lien), toutes les routes le traversant échouent. La panne va donc être détectée en observant si des routes de mesures ne fonctionnent pas.

Le problème de la couverture consiste à pouvoir détecter la panne de n'importe quel noeud du réseau.

Le problème d'identifiabilité, consiste à pouvoir localiser une panne, et demande donc plus d'informations. Il faut être capable d'identifier la ressource impactée à partir du jeu de routes qui sont tombées et de celles qui restent actives. Suite à la panne d'un noeud, l'ensemble des routes traversant l'élément vont tomber en panne. Ces routes constituent le *symptôme* du noeud. S'il est unique parmi les éléments du réseau, alors on peut conclure que c'est cet élément qui est à l'origine de la panne. On dit d'un réseau qu'il est *1-identifiable* si chaque élément dispose d'un symptôme unique. La figure 1 donne un exemple pour illustrer la 1-identifiabilité. Dans le cas (A) tous les noeuds sont bien couverts car tous sont traversés par une route liant deux moniteurs. Mais si D ou F tombe en panne, dans les deux cas les routes liant A et G ainsi que B et G vont toutes

deux tomber en panne. La panne sera donc détectée, mais il sera impossible de savoir lequel des deux est à l'origine de la panne. Le problème est similaire avec les noeuds E et H. Dans le cas (B) H est un moniteur, offrant une nouvelle route de mesure entre B et H. Ainsi chaque noeud non moniteur dispose d'un symptôme unique : Si D tombe en panne les routes (A,G) et (B,G) vont tomber en panne ; si E tombe en panne ce seront les routes (C,G) et (B,H) ; si F ne fonctionne plus, les routes (A,G), (B,G) et (B,H) ne fonctionneront plus. En notant les routes hors service, il est possible de déterminer sans ambiguïté quel noeud est à l'origine de la panne.

Mais de l'ambiguïté peut apparaître si plusieurs pannes se produisent simultanément car plusieurs symptômes sont alors impliqués. Pour la localisation d'un *ensemble d'échecs*, les nouveaux symptômes sont constitués de l'union des symptômes des noeuds en panne. On dit qu'un réseau est *k-identifiable* si, lorsque que l'on considère tous les ensembles d'au plus k éléments possibles, chaque symptôme est unique.

Une forte identifiabilité est logiquement préférable, mais cela peut nécessiter un grand nombre de moniteurs et donc un coût de supervision plus élevé. Dans cet article, nous nous concentrons sur la sélection du plus petit ensemble de moniteurs pour couvrir le réseau ou y garantir la 1-identifiabilité. Une hypothèse importante des réseaux considérés dans cette étude est que les routes entre toutes les paires de nœuds sont considérées comme fixes et imposées par le protocole de routage. Une paire de moniteurs ne peut vérifier que l'état des routes entre eux. En pratique, les opérateurs de réseau configurent généralement les pondérations des liens (IGP) pour influencer le flux de trafic dans le réseau en supposant qu'ils suivent les chemins les plus courts (voir par exemple [4] pour l'optimisation des pondérations IGP). Alternativement, d'autres protocoles tels que le routage de segment ou MPLS [8, 9, 13] permettent d'introduire des déviations ou des configurations de route explicites entre paires de nœuds. Pour tous ces protocoles, les moniteurs sont en mesure de déterminer quels chemins de données entre eux sont affectés par une défaillance.

Dans cet article, nous proposons une formalisation du problème de choix de moniteurs pour assurer la couverture et la 1-identifiabilité. Nous proposons aussi deux modèles pour résoudre ce problème, un modèle de Programmation Linéaire en Nombres Entiers (PLNE) et un modèle de Programmation Par Contraintes (PPC). Enfin nous comparons ces deux modèles entre eux et avec l'état de l'art.

2 Travaux antérieurs

Le problème du placement de moniteur pour la détection de pannes de noeuds a déjà été exploré via d'autres approches que la PPC ou la PLNE. MNMP (Maximum Node-identifiability Monitor Placement) [14] est un algorithme glouton qui ajoute itérativement des moniteurs jusqu'à ce que la k-identifiabilité désirée soit atteinte pour ensuite retirer les moniteurs redondants. Malheureusement, il ne garantit pas l'optimalité de la solution. Bezerra et al. [2] proposent différentes améliorations de cet algorithme pour réduire fortement le temps de calcul et le rendre utilisable dans le cas de réseaux sans fils, plus susceptibles de changer régulièrement alors que Bartolini et al. [1] suggèrent une borne supérieure sur le nombre maximal de noeuds identifiables pour un budget de routes de mesures donné. Enfin, Stanic et al. [18] proposent un modèle PLNE et un algorithme glouton pour un problème de choix de moniteurs. Néanmoins le problème considéré y est légèrement différent, ce sont les pannes des liens et non des noeuds qui sont considérées et les routes disponibles sont des allers-retours symétriques, ne nécessitant donc qu'un seul moniteur pour devenir des routes de mesures. Cette approche perd donc en généralité car en réalité les routes dans un réseau ne sont pas forcément symétriques.

3 Formalisation du problème

Dans cet article nous proposons deux modes de résolutions optimaux aux problèmes de couverture et de 1-identifiabilité. Nous présentons ici ces deux problèmes.

3.1 Problème de couverture

Nous supposons que la topologie du réseau est connue, peut être modélisée par un graphe connexe (c.à.d. qu'il existe une route permettant de relier chaque paire de noeuds). Une route ne devient une route de mesure, utilisable pour la tomographie, que si ses noeuds de départ et d'arrivée sont tous les deux des moniteurs. On appelle symptôme d'un noeud l'ensemble des routes de mesure qui le traverse. Le but de ce problème est de choisir parmi les noeuds du réseau lesquels seront moniteurs de manière à :

— s'assurer que chaque noeud soit traversé par au moins une route de mesure.
— minimiser le nombre de moniteurs afin de minimiser les coûts de supervision.

Le problème peut être formalisé ainsi :

Considérons un graphe orienté $G = (V, A)$ où chaque paire de noeuds $(i, j) \in V^2$ dispose d'une route acyclique les reliants. $P(i, j) = \{i, ..., j\}$ est l'ensemble des noeuds traversés par la route allant de i à j. Les routes ne sont pas forcément symétriques, ainsi $P(i, j)$ et $P(j, i)$ ne contiennent pas nécessairement les mêmes noeuds. Le but est de trouver le plus petit ensemble de noeuds moniteurs M tel que $\cup_{(i,j)\in M^2} P(i, j) = V$.

3.2 Problème de 1-identifiabilité

Pour le problème de 1-identifiabilité, nous nous plaçons dans les mêmes conditions que pour la couverture. Ici, notre but est un peu différent car il faut s'assurer que chaque noeud soit identifiable, c'est à dire que si l'un d'entre eux tombe en panne, cette panne doit non seulement être détectable mais aussi localisable sans ambiguïté. Cela implique que chaque noeud soit couvert, ce problème peut donc être vu comme une variante plus contrainte du problème de couverture. Mais pour que la panne soit localisable, il faut que le symptôme lié à la source de cette panne soit unique. Ici, nous nous concentrons sur la 1-identifiabilité des noeuds et ne considérons donc que les symptômes des noeuds.

Le problème peut être formalisé ainsi :
Considérons un graphe orienté $G = (V, A)$ où il existe une route acyclique entre chaque paire de noeuds $(i, j) \in V^2$. $P(i, j) = \{i, ..., j\}$ est l'ensemble des noeuds traversés par la route allant de i à j. $P(i, j)$ et $P(j, i)$ ne contiennent pas nécessairement les mêmes noeuds étant donné que les routes ne sont pas symétriques. On note S_l le symptôme du noeud l, c'est à dire l'ensemble de routes de mesures qui le traversent. Le but est de trouver le plus petit ensemble de noeuds moniteurs M de sorte à ce que $\cup_{(i,j)\in M^2} P(i, j) = V$ (couverture) et que $\forall i \neq j \in V^2, S_i \neq S_j$ (1-identifiabilité).

4 Placement de moniteurs

Ici nous présentons nos deux modèles pour la résolution des problèmes de couverture et de 1-identifiabilité. Nous présentons d'abord un modèle de programmation linéaire en nombres entiers (PLNE), puis un modèle de programmation par contraintes (PPC).

4.1 Modèle PLNE

Notre problème est modélisé à l'aide de deux vecteurs booléens : x, un vecteur de taille $|V|$ modélisant l'ensemble de moniteur (x_i est vrai ssi le noeud i est moniteur) et y, un vecteur de taille $|P|$ modélisant l'ensemble de routes de mesures ($y_{P(i,j)}$ est vrai ssi la route $P(i, j)$ est une route de mesure.
Le but de ce modèle est de limiter le nombre de moniteur, l'objectif est donc :

$$\text{Minimiser} \sum_{i \in V} x_i \tag{1}$$

Une route $P(i, j)$ est une route de mesure ssi son point de départ i et d'arrivée j sont tous deux des moniteurs. Cette relation est modélisée par les 3 contraintes suivantes :

$$y_{P(i,j)} \leq x_i \tag{2}$$

$$y_{P(i,j)} \leq x_j \tag{3}$$

$$y_{P(i,j)} \geq x_i + x_j - 1 \tag{4}$$

Ces 3 contraintes sont appliquées pour chaque route $P(i, j) \in P$. Exiger que tous les noeuds soient couverts signifie que le symptôme de chaque noeud doit être non vide. On note S_i le sous ensemble de P contenant toutes les routes traversant le noeud i, pour qu'un noeud soit couvert il faut donc qu'au moins une route parmi S_i soit une route de mesure. Pour chaque noeud $i \in V$ nous avons donc la contrainte suivante :

$$\sum_{P(i',j') \in S_i} y_{P(i',j')} \geq 1 \tag{5}$$

En s'arrêtant ici, le modèle permet de résoudre le problème de couverture. Mais l'ajout d'un type de contrainte supplémentaire permet de résoudre le problème de 1-identifiabilité. Un noeud est 1-identifiable ssi il est distinguable de chaque autre noeud. Contraindre chaque noeud à être 1-identifiable revient à contraindre chaque paire de noeud à être distinguables. Pour que deux noeuds soient distinguables il suffit qu'au moins une route de mesure traverse l'un des noeuds sans traverser l'autre. On note $D_{i,j}$ l'ensemble des routes traversant le noeud i sans passer par j et inversement. Pour chaque $i, j \in V^2$ tels que $i < j$, les deux noeuds sont assurés d'être distinguables par la contrainte :

$$\sum_{P(i',j') \in D_{i,j}} y_{P(i',j')} \geq 1 \tag{6}$$

Pour cette contrainte nous avons définit i et j comme étant des noeuds, mais il est possible de considérer des ensembles de noeuds et de résoudre donc le problème de k-identifiabilité pour des valeurs de k plus grandes. Mais le nombre de contraintes augmente alors de manière exponentielle avec k : il faut considérer toutes les combinaisons de noeuds de taille inférieure ou égale à k. On a $n = \sum_{i=1}^{k} \binom{|V|}{i}$ ensembles à considérer. Et il y a une contrainte (6) par paire d'ensembles possible, soit $\binom{n}{2}$ contraintes. Le nombre de contraintes explose donc rapidement avec k.

4.2 Modèle PPC

Le modèle PPC suit la même logique que le modèle PLNE. Un vecteur de variables booléennes x de taille $|V|$ indique quels noeuds sont moniteurs, un autre vecteur y de taille $|P|$ indique quelles routes sont des routes de mesures. L'objectif est le même que celui exprimé par l'équation (1), c'est-à-dire minimiser la somme des x.
Une route n'est une route de mesure que si ses noeuds de départ et d'arrivée sont des moniteurs. Cette condition est modélisée par une réification de la contrainte ET entre $y_{P(i,j)}$, x_i et x_j, contraignant $y_{P(i,j)} \equiv x_i \wedge x_j$. Cette contrainte force la même relation que les contraintes (2, 3, 4) du modèle PLNE.
Notons S_i l'ensemble des routes qui traversent le noeud i. La contrainte OU force une liste de variables booléennes à contenir au moins une fois la valeur 1. Ainsi pour chaque noeud i, la contrainte $\bigvee_{P(i',j') \in S_i} y_{P(i',j')}$ permet de s'assurer que le noeud soit traversé par au moins une route de mesure. Ceci correspond à la contrainte (5) du modèle PLNE.
Enfin pour la 1-identifiabilité, si l'on note $D_{i,j}$ l'ensemble des routes permettant de distinguer les noeuds i et j, càd les routes qui passent par l'un des deux noeuds seulement, la contrainte $\bigvee_{P(i',j') \in D_{i,j}} y_{P(i',j')}$ permet de s'assurer qu'au moins une route de mesure permette de différencier i et j. L'appliquer pour chaque paire de noeud $i, j \in V^2$ permet donc d'assurer la 1-identifiabilité. Cette contrainte correspond à l'inégalité (6) du modèle PLNE.
Un noeud traversé par de nombreuses routes est plus susceptible d'être traversé par des routes de mesures et donc moins susceptible d'être un moniteur dans la solution optimale. Ce concept est similaire à la métrique de la centralité d'un noeud, courante dans la théorie des graphes [16]. Nous proposons donc une stratégie de branchement basée sur cette métrique. L'heuristique *centralité-max*, sélectionne le noeud non fixé i qui présente la plus forte cen-

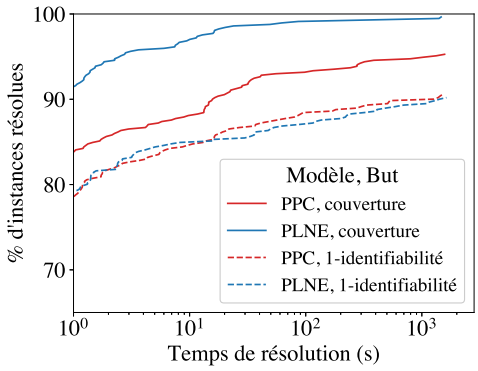


FIGURE 2 – Proportion d'instances résolues en fonction du temps de calcul de la résolution

Modèle	But	Meilleure solution	Optimalité
PLNE	Couv.	571 (100.0%)	571 (100.0%)
	1-id.	570 (99.82%)	569 (99.65%)
PPC	Couv.	565 (98.95%)	544 (95.27%)
	1-id.	563 (98.60%)	525 (91.94%)
MNMP	Couv.	546 (95.62%)	None
	1-id.	454 (79.51%)	None

TABLE 1 – Nombre d'instances où la meilleure solution est obtenue et où l'optimalité de la solution est prouvée pour chaque modèle

tralité et le marque comme non moniteur dans la branche la plus à gauche $x_i = 0$. Cette stratégie peut être combinée avec une recherche basée sur le conflit [5, 12].

5 Résultats expérimentaux

Dans cette section nous cherchons à évaluer nos deux modèles. Nous utilisons Gurobi [7] pour résoudre le modèle PLNE et le solveur MiniCP [15] pour le modèle PPC. Nous exécutons nos modèles sur des topologies venant de *Rocketfuel* [17], du *Internet Topology Zoo* [10], du *Internet Topology Data Kit* de CAIDA [3] datant de février 2022 et de *Repetita* [6]. Pour les topologies non connexes nous ne gardons que la plus grosse composante. Nous avons au total 571 topologies avec un nombre de noeuds allant de 4 à 631. Dans notre étude, nous supposons que la route entre deux noeuds correspond au plus court chemin. En cas d'égalité, une seule route est choisie arbitrairement parmi les plus courts chemins.

La figure 2 compare les temps de calculs nécessaires à la résolution de nos modèles selon le but recherché (assurer la couverture ou la 1-identifiabilité). Le temps de calcul limite a été fixé à 30 minutes. Les courbes montrent la proportion d'instances résolues dans un temps inférieur ou égal à la durée en abscisse. Les temps de calculs inférieurs à 1 seconde ont été tronqués. On observe sur cette figure que pour le problème de couverture le modèle PLNE est plus rapide, mais que pour le problème de 1-identifiabilité les deux modèles donnent des performances similaires.

Afin de constater la plus-value de nos modèles par rapport à l'état de l'art, nous comparons nos modèles avec notre implémentation de la méthode MNMP [14].

La table 1 recense pour chacun des modèles le nombre d'instances où leur solution correspond à la meilleure trouvée, ainsi que le nombre d'instances pour lesquelles ils prouvent l'optimalité de leur solution. MNMP étant un algorithme glouton, il n'est pas capable de prouver que ses solutions sont optimales mais présente un temps de calcul plus faible (au plus 2 secondes pour le problème de couverture et 365 secondes pour la 1-identifiabilité).

L'algorithme retourne une solution optimale dans 95,62% des cas pour le problème de couverture, et dans 79,51% des cas pour l'objectif de 1-identifiabilité. Bien qu'il soit fiable pour le problème de couverture, nous constatons que, pour plus de 100 topologies, les modèles PPC et PLNE peuvent fournir de meilleures solutions pour l'objectif de 1-identifiabilité. L'amélioration apportée par nos modèles est généralement de l'ordre de 1 à 3 moniteurs. Toutefois, l'ajout d'un moniteur entraîne la nécessité de sonder régulièrement $2 * |M| - 1$ nouvelles routes de mesure, M étant l'ensemble des moniteurs. Par conséquent, lorsqu'un grand nombre en est requis, ajouter quelques moniteurs supplémentaires peut avoir un impact significatif sur le trafic et entraîner une congestion. Il est donc crucial d'économiser leur nombre autant que possible. L'usage des modèles PPC et PLNE est donc pertinent pour ce problème malgré le temps de calcul plus élevé.

6 Conclusion

Nous avons présentés un modèle PLNE et un modèle PPC pour résoudre le problème du placement de moniteur dans un réseau. Leur évaluation sur 571 graphes tirés de topologies réelles montre un intérêt de ces approches de par leur capacité à trouver la solution optimale dans des temps de calculs raisonnables pour la plupart des topologies. Pour la suite, nos deux modèles peuvent être améliorés afin d'en réduire le temps de calcul.

Par exemple, notre modèle PPC n'exploite pas tout le potentiel de la PPC car il n'utilise pas de contraintes globales. Trouver de nouvelles contraintes globales pour ce problème permettrait une réduction plus forte de l'arbre de recherche et améliorerait donc le temps de calcul. Nos modèles pourraient alors être applicables à de plus grandes topologies.

Références

[1] Novella Bartolini, Ting He, Viviana Arrigoni, Annalisa Massini, Federico Trombetti, and Hana Khamfroush. On Fundamental Bounds on Failure Identifiability by Boolean Network Tomography. *IEEE/ACM Transactions on Networking*, 28(2) :588–601, April 2020. Conference Name : IEEE/ACM Transactions on Networking.

[2] Pamela Bezerra, Po-Yu Chen, Julie A. McCann, and Weiren Yu. Adaptive Monitor Placement for Near Real-time Node Failure Localisation in Wireless Sensor Networks. *ACM Transactions on Sensor Networks*, 18(1) :2 :1–2 :41, October 2021.

[3] CAIDA. The caida macroscopic internet topology data kit, February 2022.

[4] Bernard Fortz and Mikkel Thorup. Internet traffic engineering by optimizing ospf weights. In *Proceedings IEEE INFOCOM 2000. conference on computer communications. Nineteenth annual joint conference of the IEEE computer and communications societies (Cat. No. 00CH37064)*, volume 2, pages 519–528. IEEE, 2000.

[5] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. Conflict ordering search for scheduling problems. In *Principles and Practice of Constraint Programming : 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*, pages 140–148. Springer, 2015.

[6] Steven Gay, Pierre Schaus, and Stefano Vissicchio. Repetita : Repeatable experiments for performance evaluation of traffic-engineering algorithms, 2017.

[7] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[8] Renaud Hartert, Pierre Schaus, Stefano Vissicchio, and Olivier Bonaventure. Solving segment routing problems with hybrid constraint programming techniques. In *Principles and Practice of Constraint Programming : 21st International Conference, CP 2015, Cork, Ireland, August 31–September 4, 2015, Proceedings 21*, pages 592–608. Springer, 2015.

[9] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. *ACM SIGCOMM computer communication review*, 45(4) :15–28, 2015.

[10] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9) :1765 –1775, october 2011.

[11] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. Detection and Localization of Network Black Holes. In *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, pages 2180–2188, May 2007. ISSN : 0743-166X.

[12] Christophe Lecoutre, Lakhdar Saïs, Sébastien Tabary, and Vincent Vidal. Reasoning from last conflict (s) in constraint programming. *Artificial Intelligence*, 173(18) :1592–1614, 2009.

[13] Youngseok Lee, Yongho Seok, Yanghee Choi, and Changhoon Kim. A constrained multipath traffic engineering scheme for mpls networks. In *2002 IEEE International Conference on Communications. Conference Proceedings. ICC 2002 (Cat. No. 02CH37333)*, volume 4, pages 2431–2436. IEEE, 2002.

[14] Liang Ma, Ting He, Ananthram Swami, Don Towsley, and Kin K. Leung. On optimal monitor placement for localizing node failures via network tomography. *Performance Evaluation*, 91 :16–37, September 2015.

[15] L. Michel, P. Schaus, and P. Van Hentenryck. Minicp : a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1) :133–184, 2021.

[16] Mark Newman. *Networks*. Oxford university press, 2018.

[17] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP topologies with rocketfuel. *ACM SIGCOMM Computer Communication Review*, 32(4) :133–145, August 2002.

[18] Sava Stanic, Suresh Subramaniam, Gokhan Sahin, Hongsik Choi, and Hyeong-Ah Choi. Active monitoring and alarm management for fault localization in transparent all-optical networks. *IEEE Transactions on Network and Service Management*, 7(2) :118–131, June 2010. Conference Name : IEEE Transactions on Network and Service Management.

Variables de séquence pour le problème du voyageur de commerce avec fenêtres de temps

A. Delecluse[1,2], C. Thomas[1], P. Schaus[1]

[1] UCLouvain, Belgique
[2] TRAIL, Belgique

4 mai 2023

Résumé

Les variables de séquence sont une nouvelle approche en programmation par contraintes pour résoudre les problèmes de tournées de véhicules. Elles sont souvent couplées avec une recherche à voisinage large, nécessitant toutefois de démarrer d'une première solution. Nous proposons une utilisation de ces variables pour générer une solution initiale sur le problème du voyageur de commerce avec fenêtres de temps. En résolvant une relaxation du problème de départ, nous trouvons des solutions initiales plus rapidement que l'état de l'art.

Mots-clés

Variables de séquence, PVC, PVCFT.

1 Introduction

Le problème du voyageur de commerce avec fenêtres de temps (PVCFT) est un problème récurrent étudié en optimisation combinatoire. Dans ce problème, un voyageur visite un ensemble de nœuds, chacun ayant une fenêtre de temps durant laquelle la visite doit avoir lieu, de sorte à ce que tous les nœuds soient visités une et une seule fois. La plupart des approches se focalisent sur l'optimisation du trajet, à savoir, trouver un itinéraire minimisant la distance totale parcourue par le voyageur. Certaines de ces approches nécessitent une solution initiale pour pouvoir démarrer. Cependant, trouver un itinéraire satisfaisant les contraintes du problème est NP-difficile [23].

La programmation par contraintes (PPC) est une des approches les plus flexibles pour résoudre les problèmes de tournées de véhicules. Les variables de séquence [26, 4] sont un développement récent qui cible ce type de problème. Contrairement à l'approche standard, consistant en un modèle de successeurs, elles permettent de représenter facilement des visites optionnelles ainsi que des détours à un endroit arbitraire dans un itinéraire partiellement formé. Ces variables sont souvent employées en combinaison avec une recherche à voisinage large. Lors de celle-ci, une première solution valide, respectant toutes les contraintes du problème, est donnée en entrée. Sur base de cette solution, plusieurs nœuds sont relâchés et exclus de l'itinéraire. Une exploration d'arbre de recherche se charge ensuite de trouver une solution permettant de visiter les nœuds relâchés, de sorte à obtenir une meilleure valeur d'objectif que la solution de départ. Une limitation de cette approche est la nécessité de démarrer d'un premier itinéraire valide pour pouvoir fonctionner.

Notre contribution est une procédure de recherche utilisant les variables de séquence pour trouver des solutions initiales au PVCFT. Nous optimisons une relaxation du problème, où toutes les visites doivent toujours respecter les contraintes temporelles, mais où certains nœuds peuvent ne pas être visités. L'objectif de ce problème est de maximiser le nombre de nœuds effectivement visités par le voyageur. Une solution optimale du problème relaxé respecte toutes les contraintes du problème d'origine, pouvant ensuite être employée pour optimiser ce dernier.

Le papier est organisé comme suit. La section 2 reprend les travaux antérieurs sur le PVCFT ainsi que sur les variables de séquence. Les notations utiles à ces dernières sont décrites dans la section 3. La section 4 présente le modèle employé pour résoudre le problème. Ses performances sont analysées dans la section 5. Finalement, la section 6 conclut notre approche.

2 Travaux antérieurs

Plusieurs méthodes existent pour trouver des solutions initiales au PVCFT. Certaines se basent sur une répétition d'insertion et de ré-optimisation de route [6]. D'autres sur la résolution d'un problème d'assignation [2]. Ces approches ne permettent toutefois pas toujours de construire un itinéraire faisable [3].

L'algorithme le plus employé dans la littérature est la recherche à voisinage variable [15, 8, 18]. En particulier, la version présentée pour le PVCFT a prouvé son efficacité [3]. Plusieurs méthodes optimisant la distance parcourue du PVCFT s'en servent, nécessitant de démarrer d'une solution initiale [16, 3, 7].

Les variables de séquence ont été introduites dans les solveurs IBM CP Optimizer [12, 10, 11] et Google OR-Tools [19, 20]. Elles permettent de modéliser efficacement une séquence d'éléments où les éléments sélectionnés et l'ordre dans lequel ils sont arrangés est une décision du problème. La version utilisée dans ces deux solveurs est surtout prévue pour résoudre des problèmes de planification, étant donné que ces variables modélisent des séquences de variables in-

tervalle de temps.
Les variables employées dans notre approche correspondent à celles proposées dans [4, 26] qui sont prévues pour une utilisation plus générale et permettent plus de flexibilité au cours de la recherche. En effet, ces variables sont basées sur une séquence interne, initialement vide, à laquelle des éléments sont ajoutés durant la recherche grâce à des insertions.

3 Notations

La variable de séquence utilisée dans notre approche correspond à celle de [4], elle-même basée sur [26]. Certaines des notations introduites dans ces travaux sont repris dans cette section.
Étant donné un ensemble de nœuds $\mathcal{X}$, une variable de séquence, notée Sq, est une variable représentant l'ensemble des permutations possibles sur $\mathcal{X}$. Plus précisément, elle partitionne $\mathcal{X}$ en trois ensembles disjoints : les nœuds visités par la séquence (S), les nœuds possiblement visités par la séquence (P) et les nœuds exclus de la séquence (E). La séquence interne qui définit un ordre sur les éléments de S est notée $\overrightarrow{S}$. La variable représente aussi pour chaque nœud $x \in P$ un ensemble d'*insertions* possibles I^x. Chaque élément de I^x correspond à un nœud $i \in P \cup S$ qui est un prédécesseur valide de x. En d'autres termes, l'ensemble I^x définit pour un nœud x l'ensemble des nœuds après lesquels x pourrait être inséré durant la recherche.
Le domaine d'une variable de séquence est défini comme $Sq = \langle \overrightarrow{S}, I, P, E \rangle$. Une variable de séquence peut changer son état en déplaçant un nœud $x \in P$ vers soit E (ce qui correspond à exclure le nœud), soit vers S (ce qui correspond à l'ajouter à la séquence interne $\overrightarrow{S}$). En cas d'ajout, le nœud est alors inséré dans $\overrightarrow{S}$ après l'un de ses prédécesseurs $p \in I^x \cap S$. Cette opération d'insertion est notée $insérer(Sq, p, x)$.
Enfin, La notation $a \xrightarrow{\overrightarrow{S}} b$ indique qu'un élément $b \in S$ suit directement un élément $a \in S$ dans la séquence $\overrightarrow{S}$. L'opération $succ(Sq, a)$ pour un élément $a \in S$ permet de retourner $b \in S | a \xrightarrow{\overrightarrow{S}} b$, le successeur de a dans la séquence interne $\overrightarrow{S}$.

4 Problème relaxé

Dans le problème d'origine, une instance de PVCFT est composée d'un ensemble de nœuds $\mathcal{X}$ à visiter, avec $n = |\mathcal{X}|$. Une matrice de distance $dist$ définit la transition entre deux nœuds. Pour chaque nœud $i \in \mathcal{X}$, une fenêtre de temps est définie par deux valeurs : $début_i$ et fin_i, tel que $début_i \leq fin_i$. L'objectif du problème consiste à trouver l'itinéraire minimisant la distance parcourue tout en respectant les contraintes temporelles : chaque nœud doit être visité durant sa fenêtre de temps.
Dans la version relaxée du problème, nous considérons que certains nœuds peuvent ne pas être visités dans le parcours. Tous les nœuds visités respectent toutefois toujours les contraintes temporelles. Ce problème est formulé comme un problème de maximisation, où l'objectif consiste à visiter le plus de nœuds possibles. Une solution optimale de ce problème, visitant tous les nœuds, est une solution satisfaisant les contraintes du problème d'origine.

4.1 Modèle

Pour un PVCFT composé d'un ensemble de nœuds $\mathcal{X}$ à visiter, notre modèle comporte les variables suivantes :

- Une variable de séquence $Sq = \langle \overrightarrow{S}, I, P, E \rangle$ représentant le chemin parcouru par le voyageur ;
- Une variable entière $Temps_i$ par nœud, indiquant pour chaque nœud $i \in \mathcal{X}$ la fenêtre de temps de sa visite. Son domaine est initialisé à l'intervalle $\{début_i \dots fin_i\}$;
- Une variable entière $Membre$ représentant le nombre de nœuds visités par la séquence. Il s'agit de l'objectif à maximiser.

Deux contraintes sont appliquées au modèle :

TempsDeTransition permettant de capturer la dimension temporelle du problème [4]. Elle s'assure que chaque nœud visité dans la séquence Sq respectent sa fenêtre de temps $Temps$.

NMembres liant $Membre$ au nombre de nœuds visités dans la séquence Sq. Cette contrainte restreint la borne inférieure de $Membre$ au nombre de nœuds effectivement visités $|S|$ dans la séquence Sq et sa borne supérieure à la somme du nombre de nœuds non exclus $|S| + |P|$.

4.2 Procédure de recherche

La procédure de recherche est décomposée en deux étapes. Lors de la première étape, une solution est construite via un algorithme glouton, essayant de visiter le plus de nœuds possible. Lors de la seconde étape, si tous les nœuds du problème n'ont pas été visités, une recherche à voisinage large est employée afin de maximiser le nombre de nœuds visités $Membres$, jusqu'à former un parcours visitant tous les nœuds.
Les deux étapes utilisent des heuristiques de branchement sur base d'insertions. Ces dernières choisissent un nœud $x \in P$ pouvant encore être inséré dans la séquence Sq et tentent de le rajouter au trajet représenté par $\overrightarrow{S}$. Afin de réaliser les insertions à des endroits intéressant dans la séquence, les heuristiques de branchement se servent d'une fonction de coût. Ce dernier est calculé sur base du détour nécessaire pour inclure un nœud dans la séquence et prend également une mesure d'élasticité des fenêtres de temps. Il correspond au coût utilisé dans [9, 4]. Pour inclure un nœud $x \in P$ entre deux nœuds $p, q \in \overrightarrow{S}$ tels que $p \xrightarrow{\overrightarrow{S}} q$, le coût c de cette insertion est défini par :

$$c(p, x, q) = détour(p, x, q) - élasticité(p, x, q) \quad (1)$$

$$détour(p, x, q) = dist_{p,x} + dist_{x,q} - dist_{p,q} \quad (2)$$

$$\begin{aligned} élasticité(p, x, q) = &\lceil Temps_q \rceil - \lfloor Temps_p \rfloor \\ &- dist_{p,x} - dist_{x,q} \end{aligned} \quad (3)$$

4.2.1 Construction du premier parcours

L'heuristique de la première étape sélectionne le nœud ayant le plus grand regret. Étant donné un nœud $x \in P$ et ses insertions I^x dans la séquence, le regret de x correspond à la différence entre les deux plus faibles coûts c calculés sur base des insertions de x dans la séquence. Le calcul de cette valeur est détaillé dans l'Algorithme 1.

Algorithme 1 : Calcul du regret d'un nœud

Entrée : Séquence $Sq = \langle \overrightarrow{S}, I, P, E \rangle$, Nœud $x \in P$, fonction de coût c
Sortie : Regret de x

$coûtMin1 \leftarrow \infty$;
$coûtMin2 \leftarrow \infty$;
pour $p \in I^x \cap S$ **faire**
  $q \leftarrow succ(Sq, p)$;
  $coût \leftarrow c(p, x, q)$;
  si $coût < coûtMin2$ **alors**
    si $coût < coûtMin1$ **alors**
      $coûtMin2 \leftarrow coûtMin1$;
      $coûtMin1 \leftarrow coût$;
    sinon
      $coûtMin2 \leftarrow coût$;
retourner $coûtMin2 - coûtMin1$

Une fois un nœud $x \in P$ sélectionné, une branche est ensuite générée pour chaque possibilité d'insertion restante dans la séquence comme présenté dans l'Algorithme 2. Les branches sont explorées par ordre de coût croissant.

Algorithme 2 : Procédure de branchement

Entrée : Nœud $x \in P$, Séquence $Sq = \langle \overrightarrow{S}, I, P, E \rangle$, fonction de coût c
Sortie : Liste de branchement $branches$ insérant x dans Sq

$branches \leftarrow \{\}$;
pour $p \in I^x \cap S$ **faire**
  $branches \leftarrow branches + insérer(Sq, p, x)$;
tri de $branches$ par coût c croissant ;
retourner $branches$;

4.2.2 Parcours visitant tous les nœuds

Le parcours formé par la procédure de recherche de la section 4.2.1 peut ne pas visiter tous les nœuds. Dans ce cas, une seconde procédure, utilisant une recherche à voisinage large, est employée. Elle consiste en une succession de phases de relaxation et d'amélioration.

La relaxation débute sur base de la meilleure séquence (celle visitant le plus de nœuds) trouvée par la procédure de recherche. Un nœud non visité $x \in E$ est choisi aléatoirement. Une relaxation de Shaw [24] est ensuite employée afin de retirer les nœuds similaires au nœud choisi x du parcours. La similarité sim entre deux nœuds i, j est calculée sur base de la différence de distance entre les nœuds ainsi que la différence entre leurs fenêtres de temps. Elle est adaptée pour les problèmes de tournées de véhicules [24], et est présentée dans l'équation 4 :

$$sim(i,j) = \phi \frac{dist_{i,j}}{d_{max}} + \xi \left(\left| \frac{début_i}{t_{max}^{début}} - \frac{début_j}{t_{max}^{début}} \right| + \left| \frac{fin_i}{t_{max}^{fin}} - \frac{fin_j}{t_{max}^{fin}} \right| \right) \quad (4)$$

où ϕ et ξ sont des constantes. Notons que les valeurs relatives aux distances, aux temps de départ et de fin employées dans l'équation 4 sont divisées par la distance maximale d_{max} et le temps maximal de début $t_{max}^{début}$ et de fin t_{max}^{fin} de l'instance considérée, respectivement. Cela permet de conserver des valeurs inférieures à l'unité et évite de prendre trop en considération un aspect du problème, par exemple si les fenêtres de temps sont larges. Plus la similarité est faible, plus les nœuds sont semblables.

Après avoir relaxé les nœuds, une procédure de recherche vise à compléter la séquence partielle obtenue de manière à obtenir une meilleure solution. Le nœud $x \in P$ choisi pour l'insertion n'est cette fois-ci plus sélectionné sur base du regret maximal mais sur base d'une autre procédure, afin de favoriser la diversification. Celle-ci consiste à sélectionner le nœud ayant le moins d'insertions membres. Si plusieurs nœuds ont le même nombre minimum d'insertions membres, un nœud est choisi aléatoirement parmi ces derniers. Le nœud ainsi sélectionné est ensuite inséré à tous les endroits possibles suivant l'Algorithme 2.

L'entièreté de la procédure de recherche à voisinage large est présentée dans l'Algorithme 3. Elle s'arrête lorsque tous les nœuds du problème ont été visités, ou lorsqu'une limite de temps est atteinte. Plusieurs constantes sont utilisées dans cet algorithme : n_{min}, n_{max}, δ et n_{iter} permettent de contrôler la taille des voisinages considérés, ainsi que le nombre de fois que ces voisinages sont explorés.

5 Expériences

Les expériences ont été conduites sur un ordinateur composé de 2 Intel(R) Xeon(R) CPU E5-2687W et de 128GB de RAM. L'approche proposée a été implémentée dans le langage Java sur le solveur MiniCP [14]. Elle est comparée avec la recherche à voisinage variable (VNS) [3], implémentée dans le langage Rust. Les instances testées sont celles répertoriées sur [13]. Elles se déclinent sur six jeux de données et possèdent jusqu'à 233 nœuds à visiter :

AFG comporte 50 instances, introduites par [1].

Dumas contient 135 instances, présentées dans [5].

GendreauDumasExtended inclut 130 instances, présentées dans [6]. Elles ont été construites sur base des instances du jeux de données Dumas. [5].

OhlmannThomas est basé sur 25 instances, introduites dans [17].

SolomonPesant comporte 27 instances [21], dérivées d'un problème de tournées de véhicules [25].

SolomonPotvinBengio contient 30 instances [22], également dérivées de [25] mais néanmoins différentes de celles inclues dans SolomonPesant.

Algorithme 3 : Recherche à voisinage large

Entrée : Séquence Sq, nombre de nœuds visités $Membre$, premier parcours $initSol$
Sortie : Solution $meilleureSol$ visitant le plus de nœuds possibles

$meilleureSol \leftarrow initSol$;
pour $i \in \{n_{min} \dots (n_{max} - \delta)\}$ **faire**
  si $i = n_{max} - \delta$ **alors**
    $i \leftarrow n_{min}$;
  pour $j \in \{0 \dots \delta - 1\}$ **faire**
    pour $k \in \{1 \dots n_{iter}\}$ **faire**
      sélectionne un nœud x non visité dans $meilleureSol$, aléatoirement.;
      relâche $i + j$ nœuds similaires à x dans $meilleureSol$;
      $sol \leftarrow optimise(Membre)$;
      si *Membre* a été amélioré **alors**
        $meilleureSol \leftarrow sol$
      si limite de temps ou solution optimale atteinte **alors**
        retourner $meilleureSol$
retourner $branches$;

Les constantes ϕ et ξ de l'équation 4 ont été assignées à 9 et 3, respectivement. De plus, les constantes de l'Algorithme 3 valent $n_{min} = 5$, $\delta = 5$, $n_{iter} = 3$ et $n_{max} = \max(n/2 - \delta, n - \delta)$.

La figure 1 présente la moyenne de temps d'exécution entre les deux approches, sur 100 essais par instance. Nous pouvons y voir que notre approche est en général plus rapide. La moyenne de temps pour trouver une solution est en particulier toujours en dessous de la seconde. Le VNS se trouve être meilleure dans certains cas, mais ceux-ci se résument à des instances résolues en moins d'une seconde par les deux approches. Sur les autres instances, la démarche employant les variables de séquence peut trouver en un dixième de seconde des solutions nécessitant jusqu'à plusieurs minutes pour être générées par le VNS.

5.1 Discussion

Notre méthode se base sur de la recherche à voisinage large afin de trouver des solutions initiales. Notons toutefois que l'algorithme 3 pourrait démarrer d'une solution initiale non spécifiée. La procédure de recherche est dans ce cas plus longue que via l'initialisation proposée dans la section 4.2.1.

De plus, notre approche peut être facilement adaptée à d'autres problèmes de tournée de véhicules. En effet, l'algorithme 3 n'a pas connaissance des contraintes composant le problème. Nous n'avons eu qu'à spécifier une fonction de coût c et une similarité sim pour le faire fonctionner. Une solution initiale à un autre problème de tournées de véhicules pourrait être construite avec notre approche, et ne nécessiterait que de mettre à jour les contraintes du problème (section 4.1) sans devoir redéfinir l'entièreté du programme.

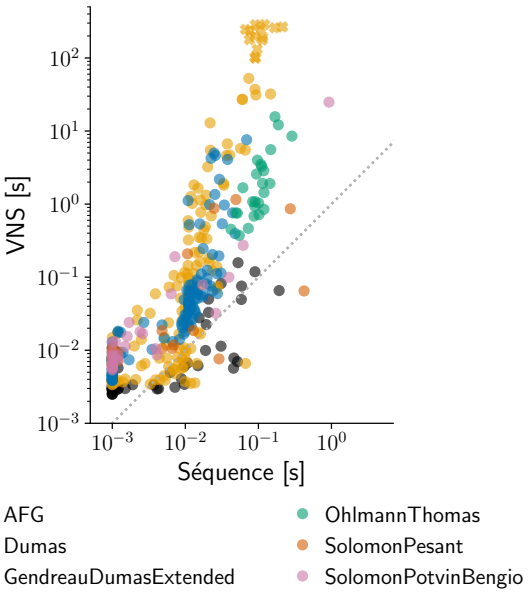


FIGURE 1 – Comparaison du temps d'exécution entre notre approche (Séquence) et la recherche à voisinage variable (VNS), en secondes. Chaque point correspond à la moyenne de temps sur 100 essais pour trouver une solution sur une instance donnée. Les couleurs indiquent l'appartenance à chaque jeu de données. Un point sur la diagonale en pointillés indique que les deux approches nécessitent le même temps d'exécution. Un point au-dessus de la diagonale indique que l'approche utilisant les variables de séquence est plus rapide. La durée maximale d'exécution a été fixée à 300 secondes par essai. Si un des 100 essais sur l'instance a excédé le délai maximal, la moyenne correspondante est indiquée avec une croix.

6 Conclusion

Le problème de voyageur de commerce avec fenêtres de temps est un problème couramment étudié en optimisation combinatoire. Plusieurs algorithmes d'optimisation dédiés à celui-ci nécessitent de démarrer d'une solution initiale afin de trouver un itinéraire minimisant la distance parcourue. Cependant, trouver un itinéraire faisable est NP-difficile. L'algorithme le plus employé dans ce cas se base sur de la recherche locale. Nous proposons une approche en programmation par contraintes, utilisant des variables de séquence, afin de construire des solutions faisables. Notre méthode construit des itinéraires excluant certains nœuds du problème, et maximise le nombre de visites jusqu'à former un parcours incluant effectivement tous les nœuds. Cette démarche se montre être plus rapide que l'état de l'art. Elle peut également être adaptée à d'autres problèmes de tournées de véhicules afin de trouver des solutions satisfaisant les contraintes.

Références

[1] Ascheuer, Norbert: *Hamiltonian path problems in the on-*

line optimization of flexible manufacturing systems. Thèse de doctorat, 1996.

[2] Calvo, Roberto Wolfler: *A new heuristic for the traveling salesman problem with time windows*. Transportation Science, 34(1) :113–124, 2000.

[3] da Silva, Rodrigo Ferreira et Sebastián Urrutia: *A General VNS heuristic for the traveling salesman problem with time windows*. Discrete Optimization, 7(4) :203–211, 2010, ISSN 1572-5286.

[4] Delecluse, Augustin, Pierre Schaus et Pascal Van Hentenryck: *Sequence Variables for Routing Problems*. Dans *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[5] Dumas, Yvan, Jacques Desrosiers, Eric Gelinas et Marius M Solomon: *An optimal algorithm for the traveling salesman problem with time windows*. Operations research, 43(2) :367–371, 1995.

[6] Gendreau, Michel, Alain Hertz, Gilbert Laporte et Mihnea Stan: *A generalized insertion heuristic for the traveling salesman problem with time windows*. Operations Research, 46(3) :330–335, 1998.

[7] Gillard, Xavier: *Discrete optimization with decision diagrams : design of a generic solver, improved bounding techniques, and discovery of good feasible solutions with large neighborhood search*. Thèse de doctorat, UCLouvain, 2022.

[8] Hansen, Pierre, Nenad Mladenović et Jose A Moreno Perez: *Variable neighbourhood search : methods and applications*. 4OR, 6 :319–360, 2008.

[9] Jain, Siddhartha et Pascal Van Hentenryck: *Large Neighborhood Search For Dial-a-Ride Problems*. Dans *International Conference on Principles and Practice of Constraint Programming*, pages 400–413. Springer, 2011.

[10] Laborie, Philippe et Jerome Rogerie: *Reasoning with Conditional Time-Intervals.* Dans *FLAIRS conference*, pages 555–560, 2008.

[11] Laborie, Philippe, Jerome Rogerie, Paul Shaw et Petr Vilím: *Reasoning with Conditional Time-Intervals. Part II : An Algebraical Model for Resources*. Dans *FLAIRS Conference*, 2009.

[12] Laborie, Philippe, Jérôme Rogerie, Paul Shaw et Petr Vilím: *IBM ILOG CP Optimizer for Scheduling*. Constraints, 23(2) :210–250, apr 2018, ISSN 1383-7133. `https://doi.org/10.1007/s10601-018-9281-x`.

[13] López-Ibáñez, Manuel: *Instances for the TSPTW*, Sep 2020. `https://lopez-ibanez.eu/tsptw-instances`, [Online ; accessed 15. Feb. 2022].

[14] Michel, L., P. Schaus et P. Van Hentenryck: *MiniCP : a lightweight solver for constraint programming*. Mathematical Programming Computation, 13(1) :133–184, 2021.

[15] Mladenović, Nenad et Pierre Hansen: *Variable neighborhood search*. Computers & operations research, 24(11) :1097–1100, 1997.

[16] Mladenovic, Nenad, Raca Todosijević et Dragan Urosevic: *An efficient General Variable Neighborhood Search for large Travelling Salesman Problem with Time Windows*. YUJOR. Yugoslav Journal of Operations Research, 23, janvier 2013.

[17] Ohlmann, Jeffrey W et Barrett W Thomas: *A compressed-annealing heuristic for the traveling salesman problem with time windows*. INFORMS Journal on Computing, 19(1) :80–90, 2007.

[18] Papalitsas, Christos, Konstantinos Giannakis, Theodore Andronikos, Dimitrios Theotokis et Angelo Sifaleras: *Initialization methods for the TSP with time windows using variable neighborhood search*. Dans *2015 6th International Conference on Information, Intelligence, Systems and Applications (IISA)*, pages 1–6. IEEE, 2015.

[19] Perron, Laurent et Vincent Furnon: *OR-Tools*. `https://developers.google.com/optimization/`.

[20] Perron, Laurent et Vincent Furnon: *OR-Tools Sequence Var*. `https://developers.google.com/optimization/reference/constraint_solver/constraint_solver/SequenceVar`.

[21] Pesant, Gilles, Michel Gendreau, Jean Yves Potvin et Jean Marc Rousseau: *An exact constraint logic programming algorithm for the traveling salesman problem with time windows*. Transportation Science, 32(1) :12–29, 1998.

[22] Potvin, Jean Yves et Samy Bengio: *The vehicle routing problem with time windows part II : genetic search*. INFORMS journal on Computing, 8(2) :165–172, 1996.

[23] Savelsbergh, Martin WP: *Local search in routing problems with time windows*. Annals of Operations research, 4(1) :285–305, 1985.

[24] Shaw, Paul: *Using constraint programming and local search methods to solve vehicle routing problems*. Dans *Principles and Practice of Constraint Programming—CP98 : 4th International Conference, CP98 Pisa, Italy, October 26–30, 1998 Proceedings 4*, pages 417–431. Springer, 1998.

[25] Solomon, Marius M: *Algorithms for the vehicle routing and scheduling problems with time window constraints*. Operations research, 35(2) :254–265, 1987.

[26] Thomas, Charles, Roger Kameugne et Pierre Schaus: *Insertion Sequence Variables for Hybrid Routing and Scheduling Problems*. Dans *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 457–474. Springer, 2020.

Jouer avec des Cryptarithmes en Programmation par Contraintes

Arnaud Malapert[1], Margaux Schmied[2], Davide Fissore[2], Marie Pelleau[1], Ambre Picard Marchetto[2]

[1] Université Côte d'Azur, CNRS, I3S, France
[2] Université Côte d'Azur, France

[1] firstname.lastname@univ-cotedazur.fr [2] firstname.lastname@etu.univ-cotedazur.fr

Résumé

Un cryptarithme est un casse-tête mathématique et logique dans lequel des mots forment une équation où les lettres représentent des chiffres à déterminer dans une base donnée. Ce problème est populaire en mathématiques récréatives, dans l'enseignement, et en programmation par contraintes. Nous proposons une approche générale, efficace et simple d'utilisation pour résoudre ce problème NP-Complet. La suite naturelle est une approche hiérarchique pour leur génération. Les évaluations expérimentales ont engendré une vaste collection, variée et remarquable.

Mots-clés

Cryptarithme ; puzzle ; résolution ; génération.

1 Introduction

Un cryptarithme est un casse-tête mathématique et logique dans lequel un ensemble de mots est écrit sous la forme d'une équation où les lettres représentent des chiffres à déterminer dans une base donnée. Il faut remplacer les lettres par des chiffres en respectant les règles suivantes :

- Règle i) Chaque lettre est associée à un seul chiffre.
- Règle ii) L'équation est vérifiée dans une base donnée en remplaçant les lettres par les chiffres.
- Règle iii) Les lettres représentent des chiffres distincts.
- Règle iv) Le chiffre de poids fort des mots est non nul.

Les deux dernières règles sont quelquefois relâchées pour obtenir une solution. Idéalement, il existe une solution unique.

Si l'invention des cryptarithmes remontent à la Chine antique, le cryptarithme le plus connu a été publié en juillet 1924 dans le *Strand Magazine* [8] par H.E. Dudeney :

$$\begin{array}{rr} & \texttt{SEND} \\ + & \texttt{MORE} \\ \hline & \texttt{MONEY} \end{array} \qquad \begin{array}{rr} & 9567 \\ + & 1085 \\ \hline & 10652 \end{array}$$

$$\texttt{S}=9;\texttt{E}=5;\texttt{N}=6;\texttt{D}=7;\texttt{M}=1;\texttt{O}=0;\texttt{R}=8;\texttt{Y}=2$$

Malgré son apparente simplicité, le problème est NP-complet [9] ce qui est prouvé par une réduction de 3-SAT.

Dans sa forme originelle, un cryptarithme est une addition avec un second membre réduit à un seul terme, et les nombres sont écrits en base 10. Mais, il existe d'autres variantes dignes d'intérêt présentées ultérieurement.

Ce puzzle est populaire en mathématiques récréatives comme en attestent les différentes publications et ressources glanées sur internet. Il existe plusieurs livres classiques [3, 10, 13] ou plus récents [7, 14] au contenu similaire avec un catalogue de puzzles classés par niveau avec leurs solutions accompagnés de conseils et méthodes de résolution « à la main ». Les cryptarithmes sont aussi présents dans des magazines, par exemple le journal *Sphinx* publié en Belgique en langue Française dans les années 1930 qui organisait aussi des compétitions. De nos jours, ils sont toujours utilisés dans les compétitions de la fédération Française des jeux mathématiques. Il existe aussi plusieurs sites web [6, 12, 19–21] consacrés à ce puzzle sur lesquels on trouvera des contenus similaires aux livres souvent avec des solveurs ou générateurs de cryptarithmes. Les solveurs disponibles utilisent le plus souvent des algorithmes de recherche exhaustive plus ou moins avancés et efficaces.

Ce puzzle est aussi populaire en enseignement, de l'école primaire à l'université. En primaire et au collège, il permet d'acquérir des compétences en résolution de problème, logique et calcul ce qui a fait l'objet de travaux en sciences de l'éducation [4, 22]. Au lycée et à l'université, c'est un problème simple et structuré qui est donc adapté pour l'apprentissage et la programmation de recherches exhaustives. La difficulté de programmer des recherches exhaustives générales et efficaces permet de présenter les avantages de la modélisation à l'université. C'est particulièrement vrai en programmation par contraintes [18] où $\texttt{SEND}+\texttt{MORE}=\texttt{MONEY}$ apparaît dans la majorité des tutoriels et manuels des solveurs de contraintes. Cependant, le problème n'a jamais fait l'objet d'une publication et n'apparaît pas non plus dans la CSPLib [1].

Les approches actuelles pour résoudre un cryptarithme montrent plusieurs limites. Résoudre un cryptarithme en décimal est un problème difficile pour les humains qui devient extrêmement difficiles dans les autres bases où nos capacités de calcul sont bien moindres.

Programmer une recherche exhaustive générale et efficace est une tâche complexe et la plupart des solveurs disponibles imposent des restrictions sur les instances du problème. Deux restrictions courantes sont de se limiter à une addition ou à la base décimale. Une autre restriction fréquente est de se limiter à l'arithmétique simple précision ce qui limite la taille des mots de l'équation.

Écrire un modèle spécifique est une tâche plus aisée, mais

cela ne résout pas le problème, seulement une instance ! Écrire un modèle général est une tâche plus difficile qui n'a jamais été réalisée à notre connaissance.
Finalement, il est temps de mettre à l'épreuve la réputation de facilité du problème et les performances de la programmation par contraintes. Les deux objectifs principaux sont de participer à la diffusion scientifique de la programmation par contraintes et d'être utilisé pour la création de contenu pédagogique dans l'enseignement.
La première contribution est de proposer un solveur de cryptarithme basé sur la programmation par contraintes qui soit général, efficace, et facile d'utilisation. Les solveurs disponibles satisfont au mieux deux de ces trois critères, abandonnant le plus souvent la généralité.
Assez naturellement, la seconde contribution est de proposer des générateurs de cryptarithmes, et de variantes, à partir d'une liste de mots. En effet, la découverte « à la main » d'un nouveau cryptarithme reste réservée aux experts et les générateurs disponibles sont plus limités que les solveurs. La génération de cryptarithmes constitue un défi plus difficile que leur simple résolution.
Ces contributions prennent la forme d'une bibliothèque Java CRYPTATOR [1] sous licence libre basée sur le solveur Choco [5] proposant deux applications en ligne de commandes.
La suite de ce document est structurée de la manière suivante. La section 2 introduit une forme générale de cryptarithme définie par une grammaire et présente plusieurs variantes du puzzle. La section 3 décrit un modèle pour la résolution de cryptarithme en arithmétique simple précision, et un modèle restreint aux additions en arithmétique multiprécision. La section 4 décrit plusieurs modèles avec un socle commun pour la génération de différentes variantes à partir d'une liste de mots. La section 5 présente l'évaluation expérimentale des performances des solveurs. La section 6 présente les expériences pour créer une vaste collection de cryptarithmes aux propriétés variées et remarquables. La section 7 décrit les choix technologiques et des éléments d'architecture de la bibliothèque, et présente quelques cas d'usage.

2 Définition d'un cryptarithme

Un cryptarithme est défini par une équation mathématique et une base arithmétique. Une équation est définie par une grammaire présentée en section 2.1 pour la facilité d'utilisation. Par défaut, on suppose que les règles i, ii, iii, et iv de la section 1 doivent être satisfaites par une solution. Cependant, nous verrons en section 3 que les règles iii et iv peuvent être relâchées.
À partir de ces entrées, on peut poser plusieurs questions : existence d'une solution ; existence d'une solution unique ; énumération des solutions. Historiquement, par un souci d'élégance, l'existence d'une solution unique est la question fondamentale.
La section 2.2 introduit plusieurs variantes de cryptarithmes considérées dans ce travail.

1. https://github.com/arnaud-m/cryptator

2.1 Grammaire d'une équation

La grammaire sert à simplifier la définition d'une équation. Elle est exprimée en *Extended Backus-Naur Form* (EBNF), afin de ne pas allourdir cet article nous ne donnons pas ici la grammaire.
La grammaire permet la reconnaissance d'une forme infixe, la forme classique que les humains manipulent, mais elle n'est pas nécessaire pour la reconnaissance des formes préfixe ou suffixe. Elle permet aussi de détecter et d'expliquer des erreurs lors de la saisie d'une équation.
La représentation d'un cryptarithme se base sur la construction d'un arbre syntaxique réalisé à partir d'un parseur. Ce parseur est capable de reconnaître les cryptarithmes classiques, mais il est plus général pour capturer ou imaginer des variantes. La grammaire est une grammaire hors contexte dont les règles de dérivation suivent la structure suivante $X \rightarrow \alpha$ où X est un symbole non terminal et α est une suite de symboles terminaux ou non terminaux.
La vérification de la structure d'une grammaire est subdivisée en deux parties essentielles : un *lexeur*, ayant le rôle de renvoyer sous forme de *tokens* les mots reconnus à partir du texte entré et un *parseur* qui vérifie la bonne structure syntactico-sémantique de la suite des *tokens* reçus par le *lexeur*.
En particulier, le lexeur distingue les *termes* (des suites alphanumériques composées des symboles *UTF-8*), les *comparateurs* tels que l'égalité et l'inégalité stricte ou large, et les *opérateurs* tels que l'addition, la multiplication, la soustraction, la division, l'élévation à la puissance et le modulo. De plus, l'opérateur de conjonction de cryptarithmes && ou ; permet de séparer une équation en plusieurs cryptarithmes. Tous ces opérateurs suivent les règles de priorités usuelles. Le parseur accepte les parenthèses pour permettre la manipulation des priorités d'opération. Tous les espaces, tabulations ou sauts de ligne sont autorisés et ignorés pour procurer un confort d'utilisation.
Le lexeur distingue deux types de terme. Un *mot* est une séquence de lettres à remplacer par des chiffres. Un *nombre* est une séquence de chiffres décimaux dont la valeur décimale est utilisée pour vérifier l'équation. Dans la grammaire, un nombre est délimité par des guillemets simples ou doubles. Ainsi, le cryptarithmes `"11" + 89 = "40"` contient deux constantes, 11 et 40, et un mot `89`. Dans la solution, les lettres (`8` et `9`) peuvent prendre des valeurs différentes de leur propre sémantique (2 et 9).
L'opérateur de conjonction et les nombres ont été introduits pour capturer certaines variantes introduites ci-dessous.

2.2 Variantes

Au cours de son histoire, de nombreuses variantes de cryptarithmes ont été introduites. Ici, nous nous limiterons à celles présentes dans la collection de la section 6.

2.3 Doublement vrai

Un cryptarithme est doublement vrai si chaque mot est un nombre écrit en toute lettre et que l'équation textuelle est elle aussi vérifiée. Ci-dessous, l'équation textuelle est à gauche, le cryptarithme au milieu, et sa solution droite.

```
   0        CERO       8027
 + 6      + SEIS     + 3013
 + 7      + SIETE    + 31040
----     -------    --------
  13       TRECE      42080
```

2.4 Mots croisés

Un mots croisés est une variante dans laquelle plusieurs équations sont écrites dans une grille et une seule affectation des chiffres aux lettres doit permettre de vérifier toutes les équations.

```
 AN + TA | DOL      87 + 38 | 125
 +       +          +       +
ODE + TEL | LAD     216 + 365 | 581
----------+---     ----------+---
TUT + SUT | NUE     303 + 403 | 706
```

Donc, chaque ligne et chaque colonne représente une équation et il faut résoudre la conjonction de ces cryptarithmes donnée dans la grammaire ci-dessous.

$$\texttt{AN} + \texttt{TA} = \texttt{DOL}; \texttt{ODE} + \texttt{TEL} = \texttt{LAD}; \texttt{SUT} + \texttt{TUT} = \texttt{NUE};$$
$$\texttt{AN} + \texttt{ODE} = \texttt{TUT}; \texttt{TA} + \texttt{TEL} = \texttt{SUT}; \texttt{DOL} + \texttt{LAD} = \texttt{NUE}$$

2.5 Multiplications

On va distinguer plusieurs types de multiplications. D'abord, la forme classique de la multiplication courte a un seul terme dans le second membre.

$$\texttt{GREY} \times \texttt{BLUE} = \texttt{DARKBLUE}$$
$$8601 \times 3450 = 29673450$$

Il y a aussi les multiplications courtes doublement vrai.

$$\texttt{CINQ} \times \texttt{SIX} = \texttt{TRENTE}$$
$$5409 \times 142 = 768078$$

Et, il y a des multiplications courtes avec plusieurs termes dans le second membre comme dans $\texttt{ORC} \times \texttt{FREAK} = \texttt{ELF} \times \texttt{FAIRY}$.
Une dernière variante amusante est la multiplication longue, une méthode de calcul classique enseignée à l'école.

```
   MU        16
 × MU      × 16
-----     -----
   NU        96
 + MU•     + 160
-----     -----
  TAU       256
```

Une multiplication longue décompose le produit d'une multiplicande et d'un multiplicateur en une somme des produits du multiplicande avec la valeur de chaque chiffre du multiplicateur. Une multiplication longue est divisée en trois parties : le produit du multiplicande et du multiplicateur ; la somme des produits partiels, et le résultat du produit.
Pour définir un tel cryptarithme, la grammaire a besoin de l'opérateur de conjonction comme pour les mots croisés, mais aussi de la multiplication par un nombre, ici une puissance de la base.

$$\texttt{MU} * \texttt{MU} = \texttt{TAU}; \texttt{MU} * \texttt{M} = \texttt{MU}; \texttt{MU} * \texttt{U} = \texttt{NU};$$
$$\texttt{NU} * \texttt{'1'} + \texttt{MU} * \texttt{'10'} = \texttt{TAU};$$

3 Résolution d'un cryptarithme

Nous expliquons ici la compilation d'un cryptarithme vers un modèle en programmation par contraintes. Cette explication est divisée en trois parties. La section 3.1 modélise l'affectation des chiffres aux lettres, soit l'application des règles i, iii, et iv. La section 3.2 modélise l'équation en arithmétique simple précision, soit l'application de la règle ii. Et, la section 3.3 modélise l'équation en arithmétique multiprécision, mais est restreint aux additions. En base décimale, la valeur d'un mot de dix caractères n'est plus représentable en simple précision.

3.1 Affectation des chiffres aux lettres

Un mot w de longueur $|w|$ est représenté par une suite finie de lettres $w = l_{|w|-1} l_{|w|-2} \dots l_1 l_0$. L'entrée d'un cryptarithme définit un ensemble W de mots et un alphabet A, l'union des lettres des mots.

$$A = \bigcup_{w \in W} \bigcup_{i=0}^{|w|-1} \{l_i\}$$

Soit b la base arithmétique du cryptarithme, la variable entière $x_l \in [0, b-1]$ représente le chiffre associé à la lettre $l \in A$ imposant ainsi la règle i.
Le nombre d'occurrences de chaque chiffre parmi les lettres est imposé par la contrainte *global cardinality* [17]. On généralise la règle iii pour traiter le cas où il y a plus de lettres que de chiffres dans la base.

$$\left\lfloor \frac{|A|}{b} \right\rfloor \leq |\{l \in A | x_l = v\}| \leq \left\lceil \frac{|A|}{b} \right\rceil \tag{1}$$

La règle iv est imposée par une contrainte arithmétique.

$$x_{l_{|w|-1}} > 0 \qquad \forall w \in W \tag{2}$$

3.2 Arithmétique simple précision

L'équation est construite en créant des variables auxiliaires associées aux valeurs des mots et en utilisant l'arithmétique du solveur pour vérifier l'équation (règle ii).
La variable auxiliaire V_w représente la valeur du mot $w \in W$ dans la base b. Elle peut être définie par la méthode d'exponentiation.

$$V_w = \sum_{i=0}^{|w|-1} b^i \times x_{l_i} \qquad \forall w \in W \tag{3}$$

Une alternative est la méthode de Ruffini-Horner.

$$V_w = ((\dots((b x_{l_{|w|-1}} + x_{l_{|w|-2}})b + x_{l_{|w|-3}})b + \dots)b + x_{l_1})b + x_{l_0} \qquad \forall w \in W \tag{4}$$

3.3 Arithmétique multiprécision

En multiprécision, il n'est plus possible d'utiliser une variable auxiliaire représentant la valeur d'un mot pour éviter un dépassement de capacité. Il est quelquefois possible de changer l'arithmétique d'un programme, mais c'est plus

compliqué pour un solveur. Il n'existe pas non plus de solveurs multiprécision à notre connaissance. Donc, la seule solution est de proposer un modèle pour le calcul multiprécision. La difficulté est que le résultat d'une opération devient le résultat d'un algorithme et non plus d'une opération gérée par le processeur. Pour le moment, le modèle est restreint à l'addition qui est l'opération la plus facile, mais aussi la plus importante pour un cryptarithme.
Le modèle calcule la somme des chiffres des opérandes (mots ou nombres) à chaque position, puis propage les retenues pour calculer le résultat final de l'addition. Soit W^1 (resp. W^2) la suite des termes du membre gauche (resp. droit) de l'addition. Chaque mot ou nombre peut être répétés dans plusieurs termes. Soit $m = \max_W |w| - 1$ la position maximale dans un mot, la variable $S_i^j \in \mathbb{N}^+$ représente la somme des chiffres en position i des opérandes du membre j.

$$\sum_{\substack{w \in W^j \\ |w| > i}} x_{l_i} = S_i^j \qquad \forall i \in [0, m], \forall j \in [1, 2] \qquad (5)$$

Ensuite, soit la variable auxiliaire $D_i^j \in [0, b-1]$ représentant le chiffre à la position i du résultat de l'addition du membre j, et la variable auxiliaire $C_i^j \in \mathbb{N}^+$ représentant la retenue à la position i. Les chiffres du résultat sont calculés en propageant les retenues de l'addition sauf la dernière.

$$b \times C_i^j + D_i^j = S_i^j + C_{i-1}^j \quad \forall i \in [1, m], \forall j \in [1, 2] \quad (6)$$
$$b \times C_0^j + D_0^j = S_0^j \quad \forall j \in [1, 2] \quad (7)$$

La dernière retenue n'est pas propagée et la valeur de C_m^j peut donc être supérieure à la base b.
Pour vérifier l'égalité des résultats, tous les chiffres des résultats et la dernière retenue des membres droit et gauche doivent être égaux.

$$D_i^1 = D_i^2 \qquad \forall i \in [1, m] \qquad (8)$$
$$C_m^1 = C_m^2 \qquad (9)$$

4 Génération de cryptarithmes

La génération de cryptarithmes est une tâche bien plus ardue que leur résolution, car l'explosion combinatoire du choix des mots et de l'équation est considérable. Une méthode naturelle pour la contrôler est de restreindre le dictionnaire et la forme de l'équation.
Les méthodes de génération prennent en entrée une liste de mots et énumèrent sous contraintes des cryptarithmes candidats. Chaque candidat est résolu et la décision est prise de le conserver ou non.
Certaines restrictions sont assez naturelles. Un cryptarithme n'est réellement amusant qu'avec des mots d'une langue naturelle. De plus, la difficulté pour le construire est bien moindre avec des mots quelconques, car le calcul devient facile. Résoudre un cryptarithme dans une base non décimale est plus difficile et moins amusant. L'élégance demande à ce que la solution soit unique. Sauf mention du contraire, les restrictions suivantes sont appliquées :

1. Les mots sont en langue naturelle.
2. Il n'y a pas de répétition de mots.
3. Les lettres représentent des chiffres distincts.
4. La base est décimale.
5. La solution est unique.

Chaque modèle est spécifique au type de cryptarithme, mais ils partagent un socle commun pour la sélection des mots. Chaque modèle spécifique impose les restrictions communes et des contraintes supplémentaires sur les longueurs des mots du cryptarithme. En effet, chaque modèle intègre des raisonnements sur la longueur du résultat d'une opération en fonction de la longueur de ses opérandes. Ces raisonnements ne seront pas présentés en détail, car les explications seraient trop longues et ils manquent de généralité. Remarquez qu'aucun modèle de génération ne considère l'affectation, partielle ou totale, des chiffres aux lettres du candidat qui est laissée entièrement à la charge de la résolution. En d'autres termes, lors de la génération rien ne certifie qu'il puisse exister une solution et que celle-ci est unique.
Prouver qu'un cryptarithme ne provoque pas de dépassement d'entier n'est pas trivial. À titre exemple, le cryptarithme suivant admet une solution unique.

$$\mathtt{T}^{\mathtt{E}} \times \mathtt{S}^{\mathtt{T}} = \mathtt{TEST}$$
$$2^5 \times 9^2 = 2592$$

Le membre de droite étant sur quatre caractères cela nous assure de ne pas avoir de dépassement d'entier. Cependant si on ne considère que le membre de gauche, celui-ci peut provoquer un dépassement d'entier. L'ordre dans lequel les contraintes associées à cette équation vont être construites dépend du solveur sous-jacent. C'est pourquoi nous n'intégrons pas de vérification au moment de la génération et nous nous reposons uniquement sur le solveur.

4.1 Sélection d'un ensemble de mots

Le modèle de sélection des mots joue un rôle central. Il définit des variables booléennes de décision y_w qui indiquent la présence d'un mot w de l'ensemble W. Les variables auxiliaires N et L représentent le nombre d'opérandes et la longueur du plus long mot.

$$N = \sum_W y_w \qquad (10)$$
$$L = \max_W |w| \times y_w \qquad (11)$$

Le modèle déclare des contraintes pour que les lettres représentent des chiffres distincts. Il est nécessaire que le nombre de lettres distinctes soit inférieur ou égal à la base b. La variable booléenne X_l indique la présence de la lettre $l \in A$. Soit $W_l \subseteq W$ l'ensemble des mots avec la lettre l.

$$W_l = \{w \in W | \exists i \in [0, |w| - 1], l_i = l\} \qquad \forall l \in A$$

La contrainte (12) impose que la variable X_l soit vraie si et seulement si un mot de W_l est présent. Soit b la base arithmétique du cryptarithme, la contrainte (13) impose que

les lettres représentent des chiffres distincts.

$$X_l = \bigvee_{W_l} y_w \qquad \forall l \in A \tag{12}$$

$$\sum_{A} X_l \leq b \tag{13}$$

4.2 Addition et multiplication

La variable de décision y_w^1 (resp. y_w^2) indique la présence d'un mot $w \in W$ dans le membre gauche (resp. droit). Les variables auxiliaires associées sont aussi définies par les contraintes (10) et (11). Par contre, les contraintes (12) et (13) pour les lettres distinctes ne sont pas définies, car elles seraient redondantes.
La contrainte (14) interdit la répétition des mots à gauche et à droite. La contrainte (15) brise partiellement les symétries.

$$y_w = y_w^1 + y_w^2 \tag{14}$$

$$L^1 \leq L^2 \tag{15}$$

Des contraintes supplémentaires sur la longueur du résultat en fonction de la longueur des opérandes sont aussi déclarées. Elles ne sont pas présentées ici, car les explications seraient trop longues et elles manquent encore de généralité.

4.3 Doublement vrai

On va simplement étendre les modèles pour l'addition et la multiplication. Soit v_w la valeur du nombre écrit dans le mot $w \in W$, on ajoute la contrainte (16) pour l'addition.

$$\sum_{W} v_w \times y_w^1 = \sum_{W} v_w \times y_w^2 \tag{16}$$

Pour la multiplication, poser la contrainte de produit équivalente est difficile en programmation par contraintes à cause de l'arithmétique simple précision. Passer au logarithme nécessite la gestion des domaines réels ou flottants ce qui n'est pas fréquent. En pratique, on calcule une approximation naïve de la contrainte (17) en normalisant et arrondissant au plus près le logarithme.

$$\left| \sum_{W} \log(v_w) \times (y_w^1 - y_w^2) \right| \leq \epsilon \tag{17}$$

4.4 Mots croisés et multiplication longue

Nous ne détaillerons pas les modèles de mots croisés et de multiplication longue. Le modèle de mots croisés inclut un modèle d'affectation des mots dans la grille couplé à un modèle d'addition pour chaque ligne et chaque colonne. Le modèle de multiplication longue est complètement spécifique, car les contraintes sur la longueur des opérandes sont très fortes.

5 Évaluation expérimentale

Nous présentons ici les expériences menées pour évaluer nos approches de résolution et de génération de cryptarithmes et les comparer avec le solveur specialisé CRYPT [21]. Le solveur CRYPT écrit en C utilise un algorithme de retour arrière. Il se restreint aux cryptarithmes en simple précision, avec additions décimales et des lettres minuscules et majuscules de l'alphabet latin. Seules les additions seront évaluées, car c'est la forme la plus classique et pour comparaison avec CRYPT qui n'accepte que des additions.

La section 5.1 compare les performances de résolution d'additions de nos solveurs PPC et du solveur CRYPT. La section 5.2 étudie le passage à l'échelle de notre solveur lorsque la base arithmétique augmente. La section 5.3 évalue et compare les performances pour la génération d'additions de notre approche et du solveur CRYPT.

Les expériences ont eu lieu sur une machine Dell avec 256 GB de RAM et 4 Intel E7-4870 2.40 GHz processeurs sous CentOS Linux 7.9 (chaque processeur a 10 cœurs).

5.1 Performances de la résolution

Nous analysons les performances de la résolution sur les additions de la collection de la section 6. Toutes les instances ont donc une solution unique. Notre jeu d'instances est composé de 35800 additions dont les mots ont 8 lettres ou moins. Pour résoudre un cryptarithme, un solveur trouve une solution, puis prouve qu'aucune autre existe. Ce choix est guidé par la difficulté et le moindre intérêt d'une collection de cryptarithmes sans ou avec plusieurs solutions.

Nous évaluons quatre solveurs :

SCALAR le modèle simple précision de la section 3.2 avec la contrainte (3) ;
HORNER le même modèle avec la contrainte (4) ;
BIGNUM le modèle multiprécision de la section 3.3 ;
CRYPT un solveur spécialisé [21] écrit en C.

Les solveurs PPC résolvent les 35800 instances. Le solveur CRYPT résout seulement les 35104 en alphabet latin.

La figure 1 donne le nombre d'instances résolues en fonction du temps (en secondes) pour chaque solveur. Le premier constat est sévère puisque le solveur spécialisé CRYPT (à gauche) est extrêmement rapide en résolvant chaque instance en moins d'une demi-seconde. De manière symétrique, la contrainte HORNER (à droite) est de loin la plus inefficace et ne permet même pas de résoudre toutes les instances dans le temps imparti (la ligne verticale à l'extrémité droite). Au milieu, le modèle simple précision SCALAR et multiprécision BIGNUM sont presque 100 fois moins rapides que CRYPT. Après un démarrage plus lent, le modèle SCALAR prend un léger avantage sur BIGNUM et résout plus de 90% des instances en moins d'une seconde. Le temps de résolution moyen de SCALAR est trois fois celui de BIGNUM, mais les temps médians sont quasiment égaux.
Finalement, la complexité d'un solveur de contraintes induit un surcoût non négligeable par rapport à un algorithme très spécialisé sur des additions décimales. Cependant, les performances sont acceptables pour une utilisation interactive du solveur et sont compensées par la généralité de l'approche PPC.

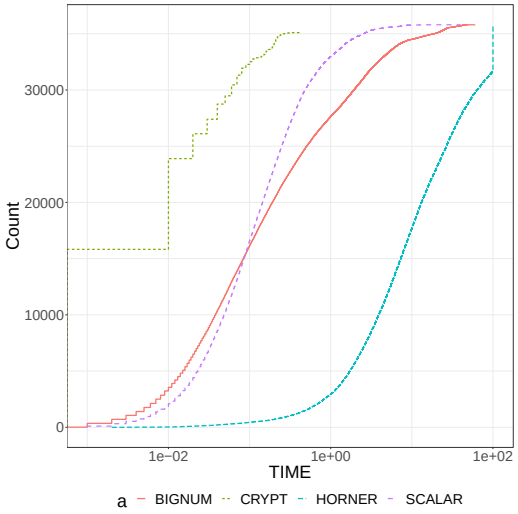


FIGURE 1 – Instances résolues en fonction du temps.

5.2 Passage à l'échelle de la résolution

Nous n'avons trouvé que trois instances non décimales dans notre bibliographie, deux en base 15 et une en base 16. Seul le solveur BIGNUM est capable de les résoudre.
Le tableau 1 donne le nombre d'opérandes, le nombre total de lettres, et la longueur maximale des mots de chaque cryptarithme accompagnés du temps de résolution et du nombre de nœud du solveur. Les métriques du solveur indiquent que la difficulté des instances a considérablement augmenté. Chacune de ces additions en base 15 ou 16 est plus difficile que n'importe quel cryptarithme décimal de la collection.

b	$\|W\|$	$\sum \|w\|$	$\max \|w\|$	Time (s)	Nodes
15	47	215	9	2185.619	549106
15	54	249	9	11716.522	1561266
16	33	154	10	2968.624	1573488

TABLE 1 – Résolution d'additions en base 15 et 16.

Les investigations sur le passage à l'échelle n'ont pas été poussées plus loin. D'abord, générer une collection en base non décimale est une tâche difficile. Puis, l'intérêt d'une telle collection est faible dans les cas d'utilisation pédagogiques, car le calcul devient compliqué ce qui n'est pas très amusant.
En conclusion, la réputation de facilité du problème reflète l'attention quasi-exclusive à la base 10.

5.3 Performances de la génération

Une méthode de génération examine des candidats générés à partir d'une liste de mots, et détermine s'ils ont une solution unique ou non. Une méthode doit filtrer les candidats pour être efficace, car la combinatoire des additions même sans répétition est considérable. Remarquez que la résolution des candidats est séquentielle, et non parallèle. Ici, nous comparons les performances de notre approche CRYPTATOR et du solveur CRYPT pour générer toutes les additions avec un nombre d'opérandes fixé et un terme unique à droite. CRYPTATOR utilise le modèle SCALAR si aucun mot n'a plus de 8 lettres et BIGNUM sinon.
Pour ces expérimentations, nous utilisons deux listes de mots. La première est composée de 24 mots correspondant aux noms des lettres de l'alphabet grec (alpha, beta, ...). La seconde liste est composée de 143 mots correspondant à des couleurs. La table 2 donne quelques caractéristiques des listes de mots utilisées pour la génération de cryptarithmes.

	$\|W\|$	$\min \|w\|$	$\text{med}\|w\|$	$\text{mean}\,\|w\|$	$\max \|w\|$
alphabet	24	2	4	4.167	7
couleurs	143	3	9	9.168	20

TABLE 2 – Caractéristiques des listes de mots.

Le tableau 3 récapitule les performances des deux méthodes avec les indicateurs suivants. Le nombre n indique le nombre de termes à gauche de l'addition. Le nombre s donne le nombre de cryptarithmes avec une solution unique. La partie haute donne les résultats pour l'alphabet grec, et la basse pour les couleurs. Pour chaque méthode, le tableau donne le temps t de génération avec résolution des candidats en secondes, et le nombre c de candidats. Pour CRYPTATOR, le tableau indique en plus le temps t_c de génération sans résolution. À droite, le nombre $\overline{c}$ d'additions sans répétition est une borne supérieure sur le nombre de candidats.

		CRYPTATOR			CRYPT		
n	s	t_c	t	c	t	c	$\overline{c}$
2	4	0.8	13.0	1.4K	**0.1**	2.9K	6.1K
3	38	1.5	165.7	5.9K	**1.9**	17.0K	42.6K
4	128	1.7	631.1	11.4K	**10.9**	75.2K	215.2K
5	207	1.5	652.4	10.6K	**26.1**	261.8K	807.6K
6	184	1.0	540.4	4.7K	**28.6**	735.8K	2.4M
7	30	0.7	120.5	817	**21.5**	1.7M	5.9M
8	2	0.6	**4.1**	13	11.4	3.3M	11.8M
9	0	0.4	**0.4**	0	13.7	5.4M	19.6M
2	66	5.6	67.7	7.7K	**1.8**	558.9K	1.4M
3	315	6.7	307.0	17.2K	**47.0**	20.2M	66.8M
4	357	7.3	**390.4**	24.3K	1232.5	574.8M	2.3G
5	163	6.8	**447.2**	25.8K	31148.9	557.8M	64.1G
6	46	6.2	**537.5**	20.4K	–	–	1.5T

TABLE 3 – Génération avec l'alphabet grec (24 mots) en haut ou des couleurs (143 mots) en bas.

Premièrement, la croissance du nombre $\overline{c}$ d'additions sans répétition est telle qu'une méthode de génération doit absolument en éliminer une large majorité pour passer à l'échelle. On observe aussi que le nombre s de cryptarithmes avec une solution unique est très petit en comparaison. Deuxièmement, la génération des candidats avec CRYPTATOR prend peu de temps t_c pour l'alphabet grec et un temps court par rapport au temps t de résolution des

candidats pour les couleurs. L'intérêt est évident puisque le nombre c de candidats de CRYPTATOR est inférieur d'un ou plusieurs de grandeurs à celui de CRYPT pour les couleurs et dès que n atteint 5 pour l'alphabet. Troisièmement, la comparaison des temps t de génération avec résolution est plus contrastée. Les meilleurs temps sont indiqués en gras. Pour l'alphabet, la réduction du nombre c de candidats par CRYPTATOR par rapport à CRYPT n'est pas suffisante pour compenser sa moindre rapidité de résolution pour les plus petites valeurs de n. Par contre, la comparaison s'inverse pour les plus grandes valeurs de n pour lesquelles le nombre de candidats devient très faible. Pour les couleurs, la liste est plus grande et l'explosion combinatoire bien plus rapide. La plus grande rapidité de CRYPT ne compense plus du tout le plus grand nombre c de candidats. Le temps de CRYPT à l'avant dernière ligne dépasse largement le temps cumulé de CRYPTATOR. Pour la dernière ligne, la résolution a été interrompue après 4 jours de calcul en ayant découvert seulement 2 cryptarithmes sur 46.

Ainsi, si la programmation par contraintes induit un surcoût pour la résolution par rapport à un solveur spécialisé comme CRYPT, elle reprend largement l'avantage pour la génération. La collection de la section 6 n'aurait pas pu être créés avec CRYPT.

Une perspective intéressante est d'hybrider la méthode de génération par la programmation par contraintes avec le solveur CRYPT pour la résolution.

6 Une collection remarquable

Nous présentons les résultats des expériences menées pour créer une collection de cryptarithmes complexes et remarquables. Le temps consacré à cette tâche est important, mais ne fait pas l'objet d'une analyse. Les résultats sont plutôt analysés à travers les caractéristiques et nouveautés de la collection.

6.1 Additions

Les additions sont la forme la plus fréquente de cryptarithmes. Par conséquent, elles constituent la majorité de la collection et se répartissent en trois groupes :

— des collections thématiques les plus complètes possibles ;
— des collections obtenues par échantillonage du dictionnaire français présentant des caractéristiques variées et remarquables en termes de nombres d'opérandes ou de longueurs des mots ;
— des collections de cryptarithmes doublement vrai dans une quinzaine de langues différentes pour les sommes de 1 à 500.

Le tableau 4 présente quelques statistiques de ces additions, de gauche à droite, leur nombre d'opérandes, leur nombre total de lettres, et les longueurs moyenne, minimale, et maximale des mots. On remarque d'abord que la collection est assez variée en termes de nombre d'opérandes et de nombre total de lettres. Une majorité d'additions ont peu d'opérandes, mais un petit quart a plus de vingt opérandes. Même si le plus long cryptarithme connu est un ovni de 200 mots (avec répétitions), la plus longue addition atteint quand même 49 opérandes. Les trois quarts des cryptarithmes sont composés de mots courts entre trois et six lettres.

Indicateur	$\lvert W\rvert$	$\sum \lvert w\rvert$	mean $\lvert w\rvert$	min $\lvert w\rvert$	max $\lvert w\rvert$
Minimum	3.0	12.0	3.0	2.0	4.0
1er Quartile	5.0	27.0	4.5	3.0	6.0
Médiane	7.0	38.0	4.9	3.0	6.0
Moyenne	13.5	64.1	5.0	3.3	6.3
3ème Quartile	18.0	88.0	5.2	4.0	6.0
Maximum	49.0	216.0	12.8	12.0	20.0

TABLE 4 – Statistiques sur les 36821 additions.

La figure 2 présente une carte thermique dont l'abscisse représente le nombre d'opérandes et l'ordonnée la longueur du mot le plus court. Le gradient de couleur indique le logarithme décimal du nombre de cryptarithmes dans la collection. Le nombre de cryptarithmes par catégorie ($\lvert W\rvert$, $\min \lvert w\rvert$, $\max \lvert w\rvert$) est limité à mille pour la méthode par échantillonage du dictionnaire. La zone claire, donc dense, en bas à gauche correspond aux collections thématiques. La zone foncé, donc peu dense, en haut à gauche correspond aux cryptarithmes avec de longs mots. La zone moins foncé, en bas à droite correspond aux cryptarithmes avec beaucoup d'opérandes.

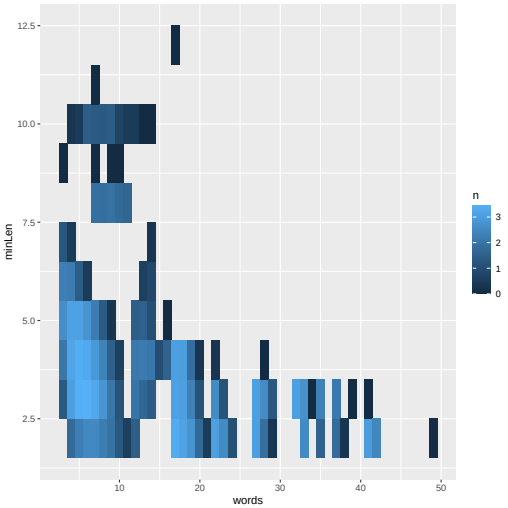


FIGURE 2 – Carte thermique des additions.

Certains des plus longs cryptarithmes, et le plus long en particulier, ne sont composés que de palindromes !

AA + ALLA + ANA + ANONA + ARA + ASA + AXA + ELLE + ERE + ERRE +ESSE + ETE + ETETE + EUE + NANAN + NON + OXO + REER + ROTOR +SALAS + SANAS + SAS + SASSAS + SELLES + SENES + SENNES+ SERES + SERRES + SES + SEXES + SOLOS + SONOS + SOS + STATS+ STOTS + STUUTS + SUS + TALAT + TALLAT + TANNAT + TARAT+ TASSAT + TATAT + TAXAT + TET + TNT + TOT + TUT = NAURUAN

Finalement, la découverte de cryptarithmes doublement vrai s'est avérée très fructueuse avec presque mille cryptarithmes dans dix langues différentes, dont plus de la moitié en Hindi !

SEIS + SETENTA + TRESCIENTOSCATORCE = TRESCIENTOSNOVENTA
ZERO + TRES + SEIS + CENTOEDOIS = CENTOEONZE
BES + ON + ONIKI + ELLIUC = SEKSEN

6.2 Multiplications

Cette fonctionnalité est plus récente et limitée par l'arithmétique simple précision, mais elle est également plus rare. Il a été assez rapide de générer quelques milliers de multiplications thématiques, ainsi que des doublement vrai. l'échantillonnage du dictionnaire est inutile, car l'arithmétique simple précision limite le nombre d'opérandes et la longueur des mots.

Nous n'avons réussi à générer pour le moment qu'une dizaine de multiplications longues dont les résultats ont au plus cinq lettres. Cependant, nous retrouvons rapidement celles-ci lorsque la liste de mots en contient.

6.3 Mots croisés

La collection contient une centaine de mots croisés composés de mots de trois lettres ou moins. À notre connaissance, c'est la première fois qu'ils sont créés dans une langue naturelle.

7 Diffusion scientifique

La bibliothèque CRYPTATOR est développée en Java et utilise Maven pour la compilation et le déploiement automatique du projet. Ses dépendances principales sont le solveur Choco [5], le générateur de parseur antl4 [16], le parseur de ligne de commandes args4j [11], et le framework de tests unitaire junit 5 [2].

Plus précisément, Choco est utilisé pour la modélisation, la résolution, et la génération de cryptarithmes sous la forme de modèles en programmation par contraintes. Antlr4 est utilisé pour définir la grammaire, lire l'équation du problème, et la génération de l'arbre syntaxique correspondant à cette équation. Args4j est utilisé pour définir les options et lire les arguments de la ligne de commandes. Ces options permettent de changer les niveaux de verbosité dans la console, de configurer les modèles, de choisir les variantes, de relâcher les contraintes. La libraire junit 5 sert à tester profondément CRYPTATOR. Avec une couverture du code dépassant les 85%, les tests vérifient le bon fonctionnement de la résolution et de la génération. Entre outre, de nombreux résultats ont été doublement validés avec le solveur CRYPT ou les collections de cryptarithmes en ligne.

Une application mobile [2] est à un stade précoce de développement. Ses fonctionnalités basées sur la bibliothèque Java intégrerons la résolution et la génération de cryptarithmes, mais aussi un jeu pédagogique autour de leur résolution. Le support pour la création de l'application est le framework *React Native* [15]. Ce framework permet la création d'applications utilisables autant sur les systèmes Android que sur les systèmes iOS ou sur le web. Un objectif principal est de faire connaître au grand public la programmation par contraintes par le vecteur des cryptarithmes. L'application

2. https://github.com/FissoreD/CryptatorApp

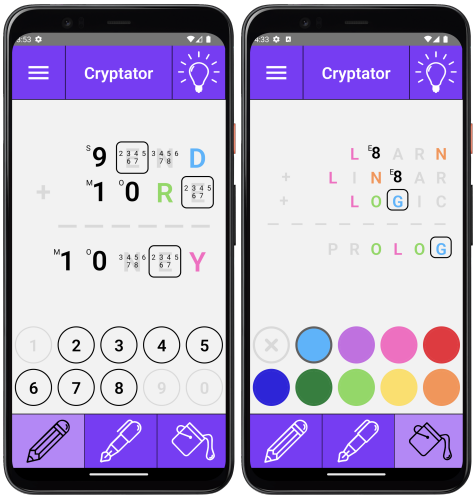


FIGURE 3 – L'application mobile du CRYPTATOR

pourra s'utiliser en autonomie ou dans un cadre pédagogique.

L'application doit permettre de saisir et visualiser un cryptarithme, une solution, ou une liste de mots. Le jeu doit aussi permettre d'annoter des cryptarithmes, de construire une solution même partielle, ou d'afficher des indices. Le jeu devra aussi proposer différents niveaux de difficulté pour s'adapter à tous les publics.

Le mode de résolution des cryptarithmes permet d'affecter à chaque lettre une valeur numérique. Lorsqu'une lettre est affectée à une valeur, cette valeur est supprimée des domaines des autres lettres. Lorsque toutes les lettres sont affectées à une valeur, le cryptarithme est résolu. Lorsque l'utilisateur affecte une valeur à une occurrence d'une lettre, cette valeur est affectée à toutes les autres occurrences de cette lettre. En sélectionnant une lettre, on peut ainsi voir les valeurs possibles pour cette lettre. Pour aider à la résolution, il est possible d'annoter les lettres par des valeurs encore disponibles dans leurs domaines. Il est donc possible d'attribuer plusieurs valeurs provisoires à une lettre.

La figure 3 présente une capture d'application d'un prototype du jeu. Le concept pédagogique est de représenter visuellement le domaine des lettres et des chiffres. L'utilisateur résout le problème en réduisant les domaines par des décisions prises graphiquement avec l'outil stylo. Il peut aussi annoter ces domaines avec l'outil pinceau et colorier les lettres et chiffres avec l'outil seau de peinture. Le moteur de jeu peut indiquer quand une décision invalide est prise ou afficher des indices avec l'outil ampoule lumineuse.

8 Conclusion

Nous proposons une approche générale, efficace, et facile d'utilisation pour la résolution et la génération de cryptarithmes en programmation par contraintes. Ce travail n'a jamais été réalisé à notre connaissance malgré la popularité du problème. L'approche est très générale en simple précision, c'est-à-dire quand les mots sont courts, mais restreinte aux additions en multiprécision. La résolution est ef-

ficace, mais reste largement dominé par un algorithme spécialisé moins général. La génération est sans conteste un point fort comme en atteste la variété de la collection de cryptarithmes. L'approche est disponible actuellement sous la forme d'une bibliothèque Java et d'une application en ligne de commande.
On peut dégager trois perspectives principales. L'ajout de la multiplication pour généraliser le modèle multiprécision. L'hybridation avec l'algorithme spécialisé améliorerait l'efficacité de la génération. La finalisation de l'application mobile avec les améliorations et les ajouts nécessaires et son adaptation à une page web pour faciliter l'utilisation et la diffusion.

Références

[1] CSPLib : A problem library for constraints. http://www.csplib.org, 1999.

[2] junit 5. https://junit.org/junit5/, 2023.

[3] Maxey Brooke. *150 Puzzles in Crypt-Arithmetic*. Dover Publications, Inc., 1963. URL http://cryptarithms.awardspace.us/150-puzzles-in-crypt-arithmetic.pdf.

[4] V. Chandra Prakash, V. Kantharao, JKR Sastry, and V. Bala Chandrika. Expert system for building cognitive model of a student using crypt arithmetic game and for career assessment. *International Journal of Recent Technology and Engineering*, 7 :684–689, 03 2019.

[5] Choco Team. Choco : an Open-Source Java Constraint Programming Library. www.choco-solver.org, 2023.

[6] Truman Collins. Alphametic puzzles. http://www.tkcs-collins.com/truman/alphamet/index.shtml, 2023.

[7] F. Rea Cyrus. *KLOOTO Games Cryptodigits*. CreateSpace Independent Publishing Platform, 2015. ISBN 151681973X.

[8] Henry Dudeney. The strand magazine vol. 68, juillet 1924, p. 97 et 214. 1924.

[9] D Epstein. On the NP-completeness of cryptarithms. *SIGACT News*, 18(3) :38–40, apr 1987. ISSN 0163-5700. doi : 10.1145/24658.24662. URL https://doi.org/10.1145/24658.24662.

[10] Steven Kahan. *At last ! ! Encoded totals second addition*. Baywood Publishing Co., 1994.

[11] Kohsuke Kawaguchi. args4j : Java command line arguments parser. https://args4j.kohsuke.org/, 2023.

[12] Mike Keith. An alphametic page. http://www.cadaeic.net/alphas.htm, 2023.

[13] Joseph S. Madachy. *Madachy's Mathematical Recreations*. Dover Pubns, 1979.

[14] Mahmoha. *Alphametic Puzle - Numbers Behind Letters . Special Edition : Names of Countries with Colorful Flags*. 2022. ISBN 979-8360917267.

[15] Meta Open Source. React native. https://reactnative.dev/, 2022.

[16] Terence Parr. antlr4. https://www.antlr.org/, 2023.

[17] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*, AAAI'96, page 209–215. AAAI Press, 1996. ISBN 026251091X.

[18] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, USA, 2006. ISBN 0444527265.

[19] Torsten Sillke. Alphametics. https://www.math.uni-bielefeld.de/~sillke/PUZZLES/ALPHAMETIC/, 2023.

[20] Jorge A. C. B. Soares. Cryptarithms online. http://cryptarithms.awardspace.us/index.html, 2002.

[21] Naoyuki Tamura. Cryptarithmetic puzzle solver. https://tamura70.gitlab.io/web-puzzle/cryptarithm/, 2023.

[22] S Widodo, U Najati, and P Rahayu. A study of cryptarithmetic problem-solving in elementary school. *Journal of Physics : Conference Series*, 1318(1) : 012120, oct 2019. doi : 10.1088/1742-6596/1318/1/012120. URL https://dx.doi.org/10.1088/1742-6596/1318/1/012120.

JFPC 5 - 5 juillet 2023, 10h30 - 12h00

Conception des lignes d'un réseau de transport maritime à l'aide de la PPC

Yousra El Ghazi[1], Djamal Habet[1], Cyril Terrioux[1]

[1] Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France

prénom.nom@univ-amu.fr

Résumé

Le problème de conception des lignes d'un réseau de transport maritime consiste, pour un armateur, à déterminer, d'une part, quelles sont les lignes maritimes (sous forme de rotations permettant de desservir un ensemble de ports) à ouvrir, et, d'autre part, l'affectation des navires (porte-conteneurs) avec les tailles adaptées pour les différentes lignes permettant d'acheminer tous les flux de conteneurs.
Dans cet article, nous proposons une modélisation de ce problème à l'aide de la programmation par contraintes. Puis, nous présentons une étude préliminaire de sa résolution à l'aide de solveurs de l'état de l'art.

Mots-clés

Modèle, résolution, application industrielle

Abstract

The liner shipping network design problem consists, for a shipowner, in determining, on the one hand, which maritime lines (in the form of rotations allowing to serve a set of ports) to open, and, on the other hand, the assignment of ships (container ships) with the adapted sizes for the different lines allowing to carry all the container flows.
In this paper, we propose a modeling of this problem using constraint programming. Then, we present a preliminary study of its solving using state-of-the-art solvers.

Keywords

Model, solving, industrial application

1 Introduction

Le transport maritime joue, à l'heure actuelle, un rôle prépondérant dans le commerce mondial. D'après l'organisation maritime internationale (OMI), plus de 80 % des échanges internationaux sont réalisés par voie maritime. Le transport de marchandises conteneurisées constitue la majeure partie de ces échanges. Il repose sur plus de 5 000 porte-conteneurs qui desservent plus de 500 ports à travers le monde. Dans ce contexte, de nombreux problèmes d'optimisation combinatoire [4, 6, 25] peuvent se poser avec des retombées économiques et écologiques non négligeables compte tenu de l'échelle.
Dans cet article, nous nous intéressons à l'un de ces problèmes, à savoir le problème de conception des lignes d'un réseau de transport maritime (appelé LSNDP pour Liner Shipping Network Design Problem [6]). Une *ligne maritime*, appelée aussi *service*, est définie par une route cyclique (appelée *rotation*) qui visite un ensemble donné de ports dans un ordre déterminé et à horaires réguliers (voir, par exemple, la figure 1). Chaque port est ainsi visité par un navire de la ligne à une fréquence hebdomadaire ou bihebdomadaire. L'ensemble des navires d'une ligne sont supposés homogènes du point de vue de leurs principales caractéristiques (capacité de chargement, vitesse, consommation de carburant, ...). Opérer une ligne à fréquence hebdomadaire dont la rotation dure k semaines requiert donc k navires du même type. Étant donné un ensemble de ports, une flotte de porte-conteneurs et un flux de conteneurs (défini par un ensemble de triplets constitués du port d'origine des marchandises, de leur port de destination et du nombre de conteneurs qu'elles représentent), le problème LSNDP consiste, pour un armateur, à déterminer, d'une part, quelles lignes maritimes ouvrir, et, d'autre part, quels navires opérer sur chaque ligne afin d'acheminer tous les flux de conteneurs tout en assurant une fréquence hebdomadaire de visite de chaque port. Il est classé comme étant NP-difficile [5]. Pour donner un ordre d'idée de sa difficulté, nous pouvons noter que, pris séparément, chacun de ses deux sous-problèmes constitue déjà un problème NP-difficile [5, 9]. De plus, d'un point de vue pratique, sa résolution par des méthodes exactes se limite, à l'heure actuelle, à des instances possédant une douzaine de ports au maximum. Bien que récent, ce problème a fait l'objet de nombreux travaux, notamment ces dix dernières années. La plupart sont issus de la recherche opérationnelle. Notons que, dans la littérature, différentes variantes du problème LSNDP sont étudiées selon les hypothèses et propriétés prises en compte (temps de transit, transbordement, vitesse constante ou variable d'une rotation à l'autre ou d'un trajet à l'autre, le type de service, la possibilité de refuser des marchandises, ...).

Différentes approches ont été considérées en s'appuyant souvent sur une modélisation en programmation linéaire (mixte) en nombres entiers (par exemple [16, 17, 18, 19, 21, 24, 26]). Elles reposent principalement sur deux types de formulations. Le premier type de formulation est axé sur les services. L'ensemble des services possibles est calculé en amont et fourni en entrée au modèle. Ce dernier se limite alors à sélectionner les services à conserver parmi les candidats [2]. Le principal inconvénient de ce type de for-

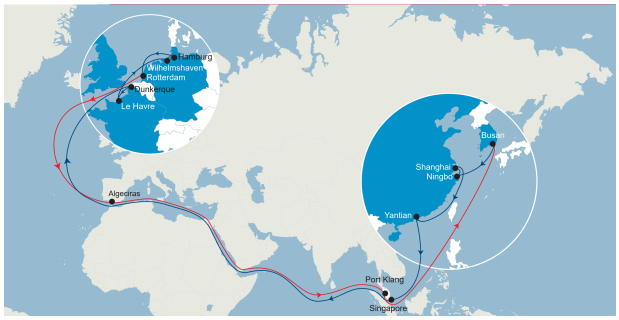


FIGURE 1 – Exemple de ligne reliant l'Asie et l'Europe.

mulation est que le nombre de services possibles croît exponentiellement avec le nombre de ports, ce qui en limite l'intérêt pratique dans le cadre d'une résolution effectuée avec des méthodes complètes. Par contre, il peut avoir son intérêt, dans le cadre de méthodes incomplètes, car on peut alors se contenter de considérer un sous-ensemble des services possibles. En pratique, la solution proposée dans [2] et reposant sur une méthode taboue couplée à la génération de colonnes a permis de traiter des instances allant jusqu'à 120 ports. Le second type de formulation repose sur la sélection des arcs du graphe représentant les liaisons possibles entre chaque paire de ports. Un service est alors défini par les arcs qui le composent, un même arc pouvant être utilisé pour la définition de plusieurs services. D'un point de vue pratique, de telles modélisations couplées à des méthodes complètes [16, 21, 19] permettent de traiter des instances ayant jusqu'à une douzaine de ports [6].
D'autres approches (par exemple [1, 2]) reposent sur des résolutions en deux temps. Le problème LSNDP étant constitué de deux sous-problèmes, elles traitent chaque sous-problème séparément. Par exemple, les approches présentées dans [2, 3, 9, 8] résolvent, dans une première phase, le problème de création des services et, dans une seconde, considèrent l'affectation des navires et la gestion du flux de marchandises en fonction des services trouvés par la première phase. Dans [13], la première phase est consacrée à la gestion du flux tandis que la seconde définit les services. Généralement, la résolution s'effectue en plusieurs passes, la première phase bénéficiant alors d'un retour d'expérience de la part de la seconde phase de la passe précédente. Bien sûr, ce type d'approches relève des méthodes incomplètes. En pratique, ces approches permettent d'obtenir des résultats satisfaisants pour des instances comptant jusqu'à une cinquantaine de ports [6]. Elles s'appuient pour cela sur des matheuristiques (par exemple, [3, 9, 8]) ou sur l'algorithme Variable Neighborhood Search (VNS) comme [13].
Au-delà, il existe de nombreux problèmes connexes au problème LNSDP. Par exemple, le problème de tournées de véhicules (appelé VRP pour Vehicle Routing Problem [11]) et ses variantes présentent de fortes similitudes avec le problème LSNDP. En particulier, les tournées sont des circuits et, pour certaines variantes, la charge du véhicule ou des temps de transit peuvent être pris en compte. Dans le transport maritime, le problème de repositionnement de navires (appelé LSFRP pour Liner Shipping Fleet Repositioning Problem [25]) consiste à déplacer les porte-conteneurs d'un service à un autre tout en tenant compte des marchandises à acheminer, des conteneurs vides à relocaliser et en maximisant la différence entre les revenus et les coûts engendrés. Parmi les approches étudiées pour résoudre ce problème, on peut souligner l'intérêt d'utiliser la programmation par contraintes mis en avant dans [10].
Si les problèmes VRP et LSNDP (et d'autres problématiques liées au transport maritime comme [12, 23]) ont fait l'objet de travaux du point de vue de la programmation par contraintes, ce ne semble pas être le cas pour le problème de conception de réseaux maritimes. Dans cet article, nous proposons un modèle permettant de traiter une version relativement générale du problème LSNDP. Notre modèle considère des vitesses variables d'un trajet à l'autre et prend en compte les transbordements et les temps de transit. Un des objectifs de ce travail est notamment d'étudier l'intérêt que peut avoir la programmation par contraintes pour la modélisation et la résolution de tels problèmes.
Cet article est organisé comme suit. La section 2 introduit les notions nécessaires à la compréhension du papier. Puis, dans la section 3, nous décrivons la modélisation que nous proposons pour le problème LSNDP. Enfin, nous présentons des résultats expérimentaux préliminaires, dans la section 4, avant de conclure dans la section 5.

2 Rappels

2.1 Programmation par contraintes

Une instance *COP (Constraint Optimization Problem [22])* P se définit par la donnée d'un quadruplet (X, D, C, f). $X = \{x_1, \ldots, x_n\}$ est un ensemble de n variables. À chaque variable x_i, on associe un domaine D_{x_i}, issu de $D = \{D_{x_1}, \ldots, D_{x_n}\}$, qui contient les valeurs possibles pour x_i. $C = \{c_1, ..., c_e\}$ désigne un ensemble de e *contraintes* qui traduisent les interactions entre les variables et définissent les combinaisons de valeurs autorisées. Enfin, f spécifie le critère à optimiser. Résoudre une instance COP $P = (X, D, C, f)$ revient à essayer de trouver une affectation de toutes les variables de X satisfaisant toutes les contraintes de C et optimisant le critère donné par f. Il s'agit d'un problème NP-difficile.
Un des atouts de la programmation par contraintes réside dans l'existence de contraintes spécialisées (les *contraintes globales*) qui vont faciliter la modélisation des problèmes, mais aussi, leur résolution grâce à des algorithmes de filtrage dédiés. Par la suite, nous exploiterons les contraintes globales suivantes :

- `Alldiff-except`(Y, v) qui assure que les valeurs des variables de Y sont deux à deux distinctes, sauf dans le cas où elles sont égales à v,
- `Circuit`(Y) qui impose que les valeurs des variables de Y forment un circuit (au sens graphe du terme) composé d'au moins deux sommets, chaque variable y_i ayant pour valeur i si elle ne participe pas au circuit, et j (avec $i \neq j$) si j est le successeur de i dans le circuit,
- `Count`$(Y, V) \odot k$ qui assure que le nombre de variables de Y dont la valeur appartient à V satisfait la

condition imposée par la relation $\odot$ vis-à-vis de k,
— $\texttt{Elt}(Y, i) = k$ qui assure que la $i$ème valeur de Y est égale à k (Y pouvant être ici un ensemble de variables ou de valeurs),
— $\texttt{Elt}_{\texttt{m}}(Y, i, j) = k$ qui assure la même propriété que $\texttt{Elt}$, mais, pour un ensemble Y de variables ou de valeurs organisé sous la forme d'une matrice à deux dimensions,
— $\texttt{Maximum}(Y) = k$ qui assure que la plus grande valeur de Y est égale à k (Y pouvant être ici un ensemble de variables, de valeurs ou d'expressions),
— $\texttt{Sum}(Y, \Lambda) \odot k$ qui impose que la somme des valeurs de Y pondérées par les coefficients de Λ satisfait la condition imposée par la relation $\odot$ vis-à-vis de k. Par la suite, cette contrainte sera représentée sous la forme plus explicite $\sum_i \lambda_i.y_i \odot k$.

2.2 Problèmes de conception des lignes

Le transport maritime de ligne consiste à utiliser des navires standardisés qui vont convoyer des marchandises, de manière fiable, entre des ports, selon un itinéraire et un horaire préétablis. Il est souvent comparé à un service de transport de passagers régulier, tel qu'un service de train ou de bus, car il fonctionne selon un horaire fixe et assure un service régulier et prévisible pour les expéditeurs et les destinataires des marchandises. Une *ligne maritime*, appelée aussi *service*, est définie par une route cyclique (appelée *rotation*) qui dessert un ensemble de ports donné dans un ordre déterminé et à horaires réguliers. La figure 1 décrit l'exemple d'une ligne reliant l'Asie et l'Europe.

Dans cet article, nous considérons uniquement le transport de marchandises sous leur forme conteneurisée, ce mode de transport constituant l'essentiel du transport de marchandises en termes de quantité et de valeurs. Ainsi, un client qui souhaite transporter des marchandises d'un *port d'origine* (POL) à un *port de destination* (POD) devra les placer à l'intérieur d'un ou plusieurs conteneurs. Les conteneurs présentent l'avantage, pour l'armateur, que leurs dimensions sont standardisées, facilitant ainsi leur manipulation et leur placement à bord des navires spécialisés que sont les porte-conteneurs. Il existe principalement deux tailles de conteneurs : les conteneurs de 20 pieds de long (soit environ 6,1 m) et ceux de 40 pieds (12,2 m) pour une hauteur de 8,6 pieds (2.6 m) et une largeur de 8 pieds (2,4 m). La majorité des conteneurs transportés sont d'une de ces deux tailles. Aussi, l'espace de stockage des navires est divisé en espace unitaire de 40 pieds sur lesquels il est possible d'empiler des unités de 40 pieds comme de 20 pieds. L'*équivalent vingt pieds* ou *EVP* (*Twenty-foot Equivalent Unit* ou *TEU* en anglais) est l'unité généralement employée pour compter une quantité de conteneurs. Un conteneur de 40 pieds compte alors pour 2 EVP.

Du point de vue de l'armateur, chaque marchandise k est vue comme une quantité $q(k)$ conteneurs (exprimée en EVP) à acheminer du port d'origine $pol(k)$ au port de destination $pod(k)$ en échange d'un revenu $rev(k)$ par EVP (exprimé en dollars). Ce revenu peut être nul dans le cas des conteneurs vides. Certaines marchandises peuvent avoir *un temps de transit* $tt_{max}(k)$ à respecter. Ce temps correspond à la durée maximale autorisée pour leur transport. Généralement, de telles marchandises sont transportées dans le cadre d'offres premium proposées par les armateurs. Précisons qu'un lot de conteneurs envoyé par un client d'un port à un autre ne peut être divisé en plusieurs sous-lots. Enfin, une marchandise peut être acheminée de son port d'origine à son port de destination via l'utilisation successive de différentes lignes. L'opération consistant à décharger une marchandise d'une ligne pour la charger sur une autre ligne est appelée *transbordement*. Elle peut nécessiter de stocker la marchandise plusieurs jours dans le port de transbordement, le temps que le navire de la ligne suivante arrive et la charge à son bord. Elle peut donc engendrer des frais (voir les coûts de transbordement ci-dessous) et allonger les temps de trajet.

Concernant les navires, les porte-conteneurs sont regroupés par type de navires ayant des caractéristiques identiques ou similaires. Ainsi, chaque classe v est caractérisée par sa *capacité* $\kappa(v)$ (c'est-à-dire le nombre maximal de conteneurs (en EVP) pouvant être transportés), son *taux d'affrètement journalier* $tc(v)$ (correspondant au coût journalier d'utilisation du navire), son *intervalle de vitesses possibles* $[\nu_{min}(v), \nu_{max}(v)]$ (en nœuds), sa *consommation horaire* $cons(v, \nu)$ *de carburant* pour le moteur principal (en tonnes par heure), pour chaque *type de carburant* $fuel(v)$ et pour chaque vitesse ν possibles. Concernant la consommation, d'autres paramètres qui pourraient avoir un impact comme la force du vent, les courants marins, le tirant d'eau ou la charge à bord du navire sont ignorés. Ces paramètres peuvent être variables dans le temps et difficilement anticipables, les lignes étant définies généralement à l'échelle d'une année.

Chaque port p possède également ses propres caractéristiques, à savoir sa *productivité* $prod(p, v)$ (c'est-à-dire le nombre de conteneurs chargés ou déchargés par heure pour des navires de type v), son temps d'attente $wt(p, v)$ (temps de mouillage avant de pouvoir rentrer dans le port), ses *temps de manœuvre* $man^{in}(p, v)$ et $man^{out}(p, v)$ respectivement pour entrer dans le port et pour en sortir, ses *frais d'escale* $pc(p, v)$ (en dollars), son *coût de transbordement* $ts(p)$ (en dollars). Les temps sont exprimés en heures et dépendent du type v de navires, tout comme les frais d'escale. Les canaux c (comme ceux de Suez ou de Panama) se caractérisent par un temps d'attente $wt(c, v)$, par une durée de traversée $trav(c)$ (en heures) et par un coût d'utilisation $pc(c, v)$ (en dollars).

Le nombre de navires opérant sur une ligne est déterminé en fonction de la durée de la rotation du service et de la fréquence de départs. En effet, une rotation doit garantir une fréquence régulière de visite de chaque port qu'elle dessert. Cette *fréquence de rotation* est généralement hebdomadaire ou bihebdomadaire. Pour une fréquence hebdomadaire, la durée de la rotation doit être égale à un multiple de sept jours. Le nombre de navires déployés par rotation doit alors être égal au nombre de semaines que dure cette rotation. Par exemple, la ligne présentée dans la figure 1 a une durée de 91 jours, soit 13 semaines. Elle est donc opérée avec 13

navires.
Le problème de conception de réseau de transport maritime de ligne (LSNDP) peut être défini comme suit : Étant donné un ensemble de *ports*, un ensemble de *navires* répartis par type (chaque type v disposant de $nb(v)$ navires) et un ensemble de *marchandises* à acheminer, définir un ensemble de rotations ayant une fréquence hebdomadaire et déterminer les navires les opérant afin de transporter les marchandises tout en respectant, le cas échéant, les temps de transit et en maximisant le bénéfice. Le bénéfice est défini comme la différence entre les revenus apportés par les marchandises transportées et l'ensemble des coûts engendrés par ce transport (frais de carburant, frais d'utilisation des navires, frais d'escales et d'utilisation des canaux, frais de transbordements, ...). Pour calculer les frais de carburant, pour chaque type f de carburant, on dispose du prix $fp(f)$ de la tonne de carburant (exprimé en dollars).
Si la finalité première de ce problème est de concevoir des réseaux de transport maritime, il peut être également employé pour aider à la prise de décision. Par exemple, il permet de simuler des situations comme des embouteillages pour entrer dans certains ports et de déterminer s'il est pertinent ou non d'adapter les rotations existantes. Il permet aussi d'envisager des évolutions du flux de conteneurs à transporter, d'évaluer l'intérêt de la prise de parts de marché sur certains flux de marchandises ou encore d'anticiper la construction de nouveaux navires.

3 Modèle

3.1 Choix de modélisation

Dans notre modèle, nous reprenons les hypothèses usuelles de la littérature. Notamment, nous considérons que tous les porte-conteneurs d'un type de bateaux donné possèdent des caractéristiques identiques et que la fréquence des services est hebdomadaire. Par ailleurs, nous faisons le choix de traiter les canaux (comme ceux de Suez ou de Panama) de la même manière que les ports. Le temps de traversée d'un canal remplace alors le temps de chargement/déchargement du bateau dans un port. Il en résulte que la notion de rotation prend désormais aussi en compte les canaux. Une rotation pouvant passer plusieurs fois par le même canal, mais pas par le même port, nous considérons, dans notre modèle, deux instances de chaque canal de sorte qu'un canal puisse être emprunté aussi bien à « l'aller » qu'au « retour ». Par exemple, on peut voir que la ligne représentée dans la figure 1 passe deux fois par le canal de Suez, une fois à « l'aller » (trajet bleu) et une fois au « retour » (trajet rouge). Notons que créer plus de deux instances d'un même canal n'a que peu d'intérêt, car une solution passant plus de deux fois par le même canal a peu de chance d'être optimale.
Notre modèle prend, en entrées, toutes les informations relatives aux infrastructures (ports et canaux), au flux de marchandises, aux types de bateaux et aux distances entre paires d'infrastructure. Il repose également sur le nombre maximal r_{max} de rotations à définir, la durée maximale h_{max} (en heures) pour accomplir une rotation et le nombre maximum ts_{max} de transbordements autorisés. Une des particularités de ce modèle est que les principales opérations seront horodatées avec, pour objectif, de calculer des temps de rotation ou de transit des marchandises aussi précis que possible. Par ailleurs, notre modèle prend en compte des vitesses variables d'un trajet à un autre. Enfin, il tient compte de la possibilité de refuser une marchandise dans le réseau si son transport n'est pas rentable ou est impossible. Par la suite, compte tenu du nombre important de variables, nous définissons les variables au fur et à mesure des besoins. L'ensemble des ports est noté $\mathcal{P} = \{0, 1, \ldots, |\mathcal{P}| - 1\}$, celui des canaux $\mathcal{C} = \{c, c + |\mathcal{C}_0| \ t.q. \ c \in \mathcal{C}_0\}$ (avec $\mathcal{C}_0 = \{|\mathcal{P}|, \ldots, |\mathcal{P}| + |\mathcal{C}_0| - 1\}$ l'ensemble des canaux avant duplication), celui des types de bateaux disponibles $\mathcal{V} = \{1, 2, \ldots, |\mathcal{V}|\}$ et celui des marchandises $\mathcal{K} = \{1, 2, \ldots, |\mathcal{K}|\}$. Nous notons respectivement $\mathcal{I}$ et $\mathcal{I}^+$ les ensembles d'indices $[0, ts_{max}]$ et $[0, ts_{max} + 1]$.

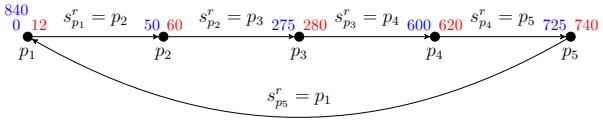


FIGURE 2 – Exemple de rotation entre cinq ports.

3.2 Définition des rotations et des routes

Notre modèle n'utilise pas nécessairement toutes les r_{max} rotations possibles. Aussi, nous considérons une variable v_r par rotation r. Elle aura pour valeur un entier compris entre 1 et $|\mathcal{V}|$ représentant le type de bateaux exploité si la rotation est utilisée, 0 sinon. Chaque rotation r doit correspondre à un circuit. Pour définir de tels circuits, nous introduisons une variable s^r_p par infrastructure p et par rotation r. Elle a pour valeur p si le port/canal p n'intervient pas dans le circuit, le port/canal successeur de p sinon. La figure 2 illustre ceci pour une rotation entre cinq ports. Pour prendre en compte le cas où la rotation r n'est pas utilisée, nous introduisons deux ports factices f_1 et f_2 afin de satisfaire la contrainte `Circuit`. Les domaines des variables s^r_p (pour $p \in \mathcal{P} \cup \mathcal{C} \cup \{f_1, f_2\}$) sont définies de sorte que les ports factices ne peuvent être successeurs des ports de $\mathcal{P}$ ou des canaux de $\mathcal{C}$ et réciproquement. Le domaine pour les ports factices est donc réduit à $\{f_1, f_2\}$. Ainsi, pour chaque rotation r, l'existence d'un circuit peut être garantie grâce à la contrainte :

$$\texttt{Circuit}(\{s^r_p | p \in \mathcal{P} \cup \mathcal{C} \cup \{f_1, f_2\}\}) \qquad \text{(R.1)}$$

Notons que cette contrainte permet d'éviter l'existence de sous-tour. Ensuite, nous devons garantir que, pour une rotation utilisée, le circuit ne porte que sur des ports et des canaux et que, pour une rotation non utilisée, il ne repose que sur les deux ports factices :

$$v_r = 0 \iff s^r_{f_1} = f_2 \wedge s^r_{f_2} = f_1 \qquad \text{(R.2)}$$

$$\forall p \in \mathcal{P} \cup \mathcal{C}, v_r = 0 \Rightarrow s^r_p = p \qquad \text{(R.3)}$$

Une rotation ne peut pas reposer que sur des canaux :

$$\sum_{p \in \mathcal{P} \cup \{f_1, f_2\}} (s^r_p \neq p) \geq 2 \qquad \text{(R.4)}$$

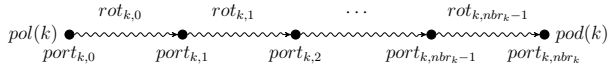


FIGURE 3 – Transport d'une marchandise k de son port d'origine $pol(k)$ à son port de destination port $pod(k)$.

3.3 Flux de marchandises

Notre modèle prévoit la possibilité de ne pas transporter une marchandise k si cela ne s'avère pas possible ou pas rentable. Pour cela, nous définissons une variable booléenne α_k qui sera vraie si la marchandise k est prise en charge, fausse sinon. Prendre en charge la marchandise k signifie qu'elle va être chargée au port d'origine $pol(k)$ et déchargée au port de destination $pod(k)$ en transitant éventuellement par des ports intermédiaires. Dans notre modélisation, nous ne considérons que les ports intermédiaires où la marchandise va être transbordée comme illustré dans la figure 3. Aussi, nous introduisons une variable $rot_{k,i}$ par marchandise k et par étape i pour représenter la $i$ème rotation utilisée pour le transport de la marchandise k. La variable $rot_{k,i}$ a pour valeur le numéro de la rotation utilisée (entre 0 et $r_{max} - 1$) lors de la $i$ème étape, -1 si une telle étape n'est pas nécessaire. Nous considérons également des variables $port_{k,i}$. La variable $port_{k,i}$ représente le port où la marchandise k intègre la $i$ème rotation utilisée tandis que $port_{k,i+1}$ correspond au port où elle quitte sa $i$ème rotation. Ces variables ont pour valeur le port p correspondant ($p \in \mathcal{P}$) si la $i$ème rotation est employée, -1 sinon. Pour chaque marchandise k, une variable nbr_k spécifie le nombre de rotations employées (entre 0 et $ts_{max}+1$). Nous pouvons maintenant définir les contraintes associées. Une marchandise k est acceptée dans le réseau si et seulement s'il existe au moins une rotation qui la transporte :

$$\alpha_k = 1 \iff nbr_k > 0 \tag{F.1}$$

Le port de départ d'une marchandise k transporté est nécessairement son port d'origine $pol(k)$:

$$\alpha_k = 1 \iff port_{k,1} = pol(k) \tag{F.2}$$

En revanche, si elle n'est pas acceptée, il n'y a pas de premier port :

$$\alpha_k = 0 \iff port_{k,1} = -1 \tag{F.3}$$

Le dernier port utilisé est forcément le port de destination $pod(k)$.

$$nbr_k > 0 \Rightarrow \texttt{Elt}(\{port_{k,i+1} | i \in \mathcal{I}]\}, nbr_k) = pod(k) \tag{F.4}$$

Aucun port ou rotation n'est utilisé au-delà du port de destination :

$$i \geq nbr_k \iff rot_{k,i} = -1 \tag{F.5}$$

$$i > nbr_k \Rightarrow port_{k,i} = -1 \tag{F.6}$$

$$port_{k,i} = -1 \Rightarrow i \geq nbr_k \tag{F.7}$$

Par ailleurs, une marchandise ne peut pas transiter plusieurs fois par le même port ou par la même rotation :

$$\texttt{Alldiff-except}(\{port_{k,i} | i \in \mathcal{I}^+\}, -1) \tag{F.8}$$

$$\texttt{Alldiff-except}(\{rot_{k,i} | i \in \mathcal{I}\}, -1) \tag{F.9}$$

Afin de faciliter l'expression de certaines contraintes relatives au trajet suivi par les marchandises, nous introduisons des variables booléennes $from^r_{k,p}$ (resp. $to^r_{k,p}$) qui sont vraies si la marchandise k entre dans (resp. sort de) la rotation r au port p, fausses sinon. Ces variables sont liées aux précédentes ainsi :

$$from^r_{k,p} = \sum_{i \in \mathcal{I}} (port_{k,i} = p).(rot_{k,i} = r) \tag{F.10}$$

$$to^r_{k,p} = \sum_{i \in \mathcal{I}} (port_{k,i+1} = p).(rot_{k,i} = r) \tag{F.11}$$

Ces variables sont également employées pour lier le flux de marchandises à la définition des routes. En effet, si une marchandise k est (dé)chargée au niveau d'un port p pour une rotation r, cela implique que le port p est utilisé dans cette rotation :

$$from^r_{k,p} = 1 \Rightarrow s^r_p \neq p \tag{F.12}$$

$$to^r_{k,p} = 1 \Rightarrow s^r_p \neq p \tag{F.13}$$

Réciproquement, si un port p est utilisé dans une rotation r, alors il existe au moins une marchandise qui est (dé)chargée dans ce port pour cette rotation :

$$s^r_p \neq p \Rightarrow \texttt{Count}(\{port_{k,i} | k \in \mathcal{K}, i \in \mathcal{I}^+\}, \{p\}) \geq 1 \tag{F.14}$$

$$s^r_p \neq p \Rightarrow \texttt{Count}(\{rot_{k,i} | k \in \mathcal{K}, i \in \mathcal{I}\}, \{r\}) \geq 1 \tag{F.15}$$

Enfin, si une rotation n'est pas utilisée, aucune marchandise ne peut transiter par son biais, et réciproquement :

$$v_r = 0 \iff \texttt{Count}(\{rot_{k,i} | k \in \mathcal{K}, i \in \mathcal{I}^+\}, \{r\}) = 0 \tag{F.16}$$

3.4 Propriétés des rotations et des navires

Dans certains modèles MIP de la littérature (par exemple [6]), chaque service est associé à un type de bateaux prédéfini. Si ce choix facilite la prise en compte des spécificités de chaque type de navires, il conduit à considérer de nombreuses rotations dont peu seront utilisées au final. Dans notre modélisation, nous avons fait le choix de laisser le solveur décider du type de navires associé à chaque rotation. Aussi, il faut s'assurer de la concordance du type de navires retenu pour une rotation et des caractéristiques de la rotation. Cela nécessite d'introduire un certain nombre de variables dont les valeurs seront fixées ensuite à l'aide de contraintes `Elt`. La variable κ_r représente la capacité maximale (exprimée en EVP) de marchandises transportables via la rotation r. Elle a pour valeur 0 si la rotation n'est pas utilisée, la capacité du type de navires utilisé sinon. Les variables ν^r_{min} et ν^r_{max} spécifient les vitesses minimale et maximale de la rotation r (0 si la rotation n'est

pas exploitée). La variable fp_r exprime le prix de la tonne de carburant pour la rotation r (0 si la rotation n'est pas exploitée). Pour chaque rotation r, nous posons alors les contraintes suivantes :

$$\texttt{Elt}(\{0\} \cup \{\kappa(v)|v \in \mathcal{V}\}, v_r) = \kappa_r \quad \text{(P.1)}$$

$$\texttt{Elt}(\{0\} \cup \{\nu_{min}(v)|v \in \mathcal{V}\}, v_r) = \nu^r_{min} \quad \text{(P.2)}$$

$$\texttt{Elt}(\{0\} \cup \{\nu_{max}(v)|v \in \mathcal{V}\}, v_r) = \nu^r_{max} \quad \text{(P.3)}$$

$$\texttt{Elt}(\{0\} \cup \{fp(fuel(v))|v \in \mathcal{V}\}, v_r) = fp_r \quad \text{(P.4)}$$

De même, certaines informations (frais d'escale, temps d'attente, ...) relatives aux ports ou aux canaux dépendent également du type de navires associé à la rotation r. Pour chaque rotation r, nous introduisons les variables $wt_{p,r}$, $man^{in}_{p,r}$ et $man^{out}_{p,r}$ qui spécifient respectivement le temps d'attente du port ou du canal p et le temps de manœuvre pour entrer et sortir du port p. Le coût d'escale au port p pour la rotation r est représentée par la variable $pc_{p,r}$ tandis que la productivité pour le port p et la rotation r est exprimée par la variable $prod_{p,r}$.

$$\texttt{Elt}(\{0\} \cup \{wt(p,v)|v \in \mathcal{V}\}, v_r) = wt_{p,r} \quad \text{(P.5)}$$

$$\texttt{Elt}(\{0\} \cup \{man^{in}(p,v)|v \in \mathcal{V}\}, v_r) = man^{in}_{p,r} \quad \text{(P.6)}$$

$$\texttt{Elt}(\{0\} \cup \{man^{out}(p,v)|v \in \mathcal{V}\}, v_r) = man^{out}_{p,r} \quad \text{(P.7)}$$

$$\texttt{Elt}(\{0\} \cup \{pc(p,v)|v \in \mathcal{V}\}, v_r) = pc_{p,r} \quad \text{(P.8)}$$

$$\texttt{Elt}(\{-1\} \cup \{prod(p,v)|v \in \mathcal{V}\}, v_r) = prod_{p,r} \quad \text{(P.9)}$$

Enfin, nous considérons la variable tc_r qui, pour chaque rotation r, spécifie le coût journalier d'utilisation des navires associés à la rotation.

$$\texttt{Elt}(\{0\} \cup \{tc(v)|v \in \mathcal{V}\}, v_r) = tc_r \quad \text{(P.10)}$$

3.5 Charge des navires

Nous devons garantir qu'à la sortie de chaque port, les navires ne sont pas chargés au-delà de leur capacité maximale. Cela nécessite de savoir, pour chaque rotation, quelles sont les marchandises qu'elle transporte à la sortie de chaque port. Pour cela, nous utilisons une variable booléenne $leave^r_{k,p}$ par marchandise k, par port p et par rotation r. Cette variable est vraie si la marchandise k quitte le port p à bord de la rotation r, fausse sinon. Les contraintes (C.1) et (C.2) traitent les cas où la marchandise est respectivement chargée et déchargée de la rotation r tandis que la contrainte (C.3) garantit la transitivité tout le long du trajet. Enfin, la contrainte (C.4) correspond au cas où un port p ne figure pas dans la rotation r.

$$from^r_{k,p} = 1 \Rightarrow leave^r_{k,p} = 1 \quad \text{(C.1)}$$

$$to^r_{k,p} = 1 \Rightarrow leave^r_{k,p} = 0 \quad \text{(C.2)}$$

$$(s^r_p = p' \wedge from^r_{k,p'} = 0 \wedge to^r_{k,p'} = 0) \Rightarrow leave^r_{k,p} = leave^r_{k,p'} \quad \text{(C.3)}$$

$$s^r_p = p \Rightarrow leave^r_{k,p} = 0 \quad \text{(C.4)}$$

La contrainte suivante nous permet maintenant d'assurer que, pour chaque port p, la charge, à la sortie du port, ne dépasse pas la capacité maximale κ_r de la rotation r :

$$\sum_{k \in \mathcal{K}} q(k).leave^r_{k,p} \leq \kappa_r \quad \text{(C.5)}$$

3.6 Horodatage et temps de transit

3.6.1 Durée des opérations au port et de traversée des canaux

Pour exprimer la durée des opérations de chargement/déchargement dans un port ou la durée de traversée d'un canal, nous introduisons une variable t^r_p par port ou canal p et par rotation r. Dans le cas d'un canal, la valeur de cette variable est définie comme égale à la durée de la traversée si le canal est utilisé, 0 sinon :

$$t^r_p = trav(p).(s^r_p \neq p) \quad \text{(T.1)}$$

Pour un port p, deux cas de figure sont envisageables. Si le port n'est pas utilisé dans la rotation r, la variable t^r_p vaut 0 (voir la contrainte (T.2)). Sinon, sa valeur dépend de la quantité de marchandises chargées et déchargées dans le port p pour la rotation r. Pour calculer cette quantité, nous considérons la variable $teu_{p,r}$ qui indique la quantité en EVP de marchandises chargées et déchargées dans le port p pour la rotation r, et la contrainte (T.3). Nous pouvons alors calculer la durée des opérations grâce à la contrainte (T.4). Un mouvement de grue permet de déplacer un conteneur quelle que soit sa taille. Afin de prendre en compte l'existence de conteneurs de 20 pieds et de 40 pieds parmi les marchandises à traiter, le paramètre μ permet de calculer le nombre de conteneurs à manipuler et donc le nombre de mouvements de grues nécessaires à partir du nombre de conteneurs exprimé en EVP.

$$s^r_p = p \iff t^r_p = 0 \quad \text{(T.2)}$$

$$teu_{p,r} = \sum_{k \in \mathcal{K}} (from^r_{k,p} + to^r_{k,p}).q(k) \quad \text{(T.3)}$$

$$s^r_p \neq p \Rightarrow t^r_p = \left\lceil \frac{\mu.teu_{p,r}}{prod_{p,r}} \right\rceil \quad \text{(T.4)}$$

3.6.2 Horaires des escales

Pour établir les horaires de chaque escale, il est nécessaire de désigner un port comme port de départ dans chaque rotation. Pour cela, nous considérons une variable dep_r par rotation r qui aura, pour valeur, un port p si la rotation est utilisée, -1 sinon. Le choix du port de départ étant purement arbitraire, nous choisissons celui de plus grand indice.

$$v_r = 0 \iff dep_r = -1 \quad \text{(T.5)}$$

$$v_r > 0 \Rightarrow \texttt{Maximum}(\{p.(s^r_p \neq p)|p \in \mathcal{P}\}) = dep_r \quad \text{(T.6)}$$

Dans notre modélisation, nous considérons deux moments clés : le moment où le navire arrive à quai (resp. entre dans le canal) et le moment où il le quitte (resp. sort du canal). Pour chaque rotation r et chaque port ou canal p, ces deux moments sont représentés respectivement par les variables $time^{in}_{p,r}$ et $time^{out}_{p,r}$ qui prennent leurs valeurs dans $[0, h_{max}]$. Pour chaque rotation, nous considérons que l'instant 0 coïncide avec le moment d'arrivée à quai dans le port de départ grâce à la contrainte suivante :

$$dep_r = p \Rightarrow time^{in}_{p,r} = 0 \quad \text{(T.7)}$$

Le moment de sortie ne dépend que du moment d'arrivée et de la durée des opérations au port ou de traversée du canal :

$$time^{out}_{p,r} = time^{in}_{p,r} + t^r_p \quad \text{(T.8)}$$

Ensuite, pour déterminer le temps d'arrivée à un port ou un canal en fonction du temps de sortie du port ou canal qui le précède dans la rotation, nous devons calculer le temps de trajet en fonction de la vitesse du navire. Les variables st^r_p et ν^r_p représentent respectivement le temps de trajet pour aller du port p à son successeur (s'il en existe un) dans la rotation r et la vitesse (exprimée en nœuds) utilisée sur ce trajet. Les deux variables sont corrélées par la contrainte suivante :

$$s^r_p = p' \Rightarrow st^r_p = \left\lceil \frac{\delta(p,p')}{\nu^r_p} \right\rceil \quad \text{(T.9)}$$

Les vitesses employées doivent être conformes aux possibilités des navires opérant la rotation :

$$(v_r > 0 \wedge s^r_p \neq p) \Rightarrow \nu^r_p \geq \nu^r_{min} \quad \text{(T.10)}$$

$$v_r > 0 \Rightarrow \nu^r_p \leq \nu^r_{max} \quad \text{(T.11)}$$

Nous pouvons, à présent, définir le moment d'arrivée au port ou canal p' en fonction de son prédécesseur p dans la rotation r :

$$(s^r_p = p' \wedge p' \neq dep_r) \Rightarrow$$
$$time^{in}_{p',r} = time^{out}_{p,r} + man^{out}_{p,r} + st^r_p + wt_{p',r} + man^{in}_{p',r} \quad \text{(T.12)}$$

À noter que pour les canaux, nous considérons que les variables $man^{in}_{p,r}$ et $man^{in}_{p,r}$ valent 0. Cela nous permet de ne pas avoir à décliner la contrainte précédente selon les différentes possibilités de successions de ports et de canaux.
Dans le cas où un port p n'est pas exploité dans une rotation r, nous fixons les valeurs des variables st^r_p, ν^r_p et $time^{in}_{p,r}$ à la valeur 0 :

$$s^r_p = p \iff st^r_p = 0 \quad \text{(T.13)}$$

$$s^r_p = p \Rightarrow time^{in}_{p,r} = 0 \quad \text{(T.14)}$$

$$s^r_p = p \iff \nu^r_p = 0 \quad \text{(T.15)}$$

Enfin, si une marchandise k est chargée dans la rotation r au port p et déchargée au port p', cela impose que l'arrivée au port p a lieu avant l'arrivée au port p' si le trajet entre p et p' ne passe pas par le port de départ de la rotation r. Si ce trajet passe par le port de départ, alors, l'arrivée au port p' aura lieu lors de la rotation suivante, et l'arrivée au port p se situe entre les deux visites du port p'. Par exemple, si on considère la rotation de la figure 2 (qui dure 840 heures) et une marchandise envoyée du port p_2 au port p_4, un navire effectuant cette rotation entre dans le port p_2 à l'heure 50 (en bleu) et arrive dans le port p_4 à l'heure 600. Dans ce cas, on a bien $time^{in}_{p_2,r} < time^{in}_{p_4,r}$. Par contre, si on considère une marchandise allant de p_4 à p_3, le bateau entre le port p_3 à l'heure 275 avant de visiter p_4. Cette marchandise ne sera donc délivrée qu'au prochain passage du bateau à l'heure 1 115. On a alors $time^{in}_{p_3,r} < time^{in}_{p_4,r} < time^{in}_{p_3,r} + T_r$. Cela se traduit par la contrainte suivante :

$$(from^r_{k,p} = 1 \wedge to^r_{k,p'} = 1) \Rightarrow$$
$$(time^{in}_{p,r} < time^{in}_{p',r} \vee time^{in}_{p',r} < time^{in}_{p,r} < time^{in}_{p',r} + T_r) \quad \text{(T.16)}$$

3.6.3 Temps de transit des marchandises

Pour calculer précisément le temps de transit des marchandises, nous avons besoin de connaître les moments clés de leur transport, à savoir quand elles sont chargées à bord d'une rotation ou déchargées. Pour simplifier la modélisation, nous considérons qu'une marchandise est chargée à bord d'une rotation au moment où la rotation quitte le port et qu'elle est déchargée au moment où la rotation arrive au port. Ces deux moments sont représentés respectivement par les variables $ctime^{in}_{i,k}$ et $ctime^{out}_{i,k}$. Les deux contraintes suivantes assurent la correspondance entre les moments clés des rotations et des marchandises :

$$\texttt{Elt}_{\mathbb{m}}(\{time^{out}_{p,r} | p \in \mathcal{P}, r \in R\}, port_{k,i}, rot_{k,i}) = ctime^{in}_{i,k} \quad \text{(T.17)}$$

$$\texttt{Elt}_{\mathbb{m}}(\{time^{in}_{p,r} | p \in \mathcal{P}, r \in R\}, port_{k,i+1}, rot_{k,i}) = ctime^{out}_{i,k} \quad \text{(T.18)}$$

Le temps passé par la marchandise k dans sa $i$ème rotation est représenté par la variable $\delta_{i,k}$. Il correspond naturellement à la différence entre le moment de sortie et le moment d'entrée. Toutefois, il faut tenir compte du cas particulier où le trajet passe par le port de départ. Dans ce cas, la marchandise sera déchargée lors de la prochaine rotation. Par exemple, si on considère l'exemple précédent, une marchandise envoyée du port p_2 au port p_4 quitte le port p_2 à l'heure 60 (en rouge) et arrive dans le port p_4 à l'heure 600 (en bleu). Cela donne un temps de trajet de 540 heures. Par contre, une marchandisée expédiée du port p_4 au port p_3 quitte le port p_4 à l'heure 620 et arrive au port p_3 à l'heure 1 115 et met donc 495 heures pour arriver à destination. La contrainte (T.19) traite le premier cas et la contrainte (T.20) le second.

$$ctime^{in}_{i,k} \leq ctime^{out}_{i,k} \Rightarrow \delta_{i,k} = ctime^{out}_{i,k} - ctime^{in}_{i,k} \quad \text{(T.19)}$$

$$(rot_{k,i} = r \wedge ctime^{in}_{i,k} > ctime^{out}_{i,k}) \Rightarrow$$
$$\delta_{i,k} = ctime^{out}_{i,k} - ctime^{in}_{i,k} + T_r \quad \text{(T.20)}$$

Dans le cas de figure où un transbordement a lieu, il faut considérer le temps que la marchandise passe à quai entre les deux rotations. Compte tenu de la fréquence hebdomadaire des rotations, ce temps peut être de l'ordre d'une semaine au maximum. Pour le calculer plus finement, nous introduisons une variable $\Delta_{i,k}$ par marchandise k et par $i$ème rotation utilisée. Calculer la valeur de cette variable nécessite de prendre en compte la fréquence hebdomadaire des rotations. Par exemple, considérons une marchandise k qui arrive à un port p à l'heure 200 (d'après sa valeur $ctime^{out}_{i,k}$) sur une rotation r et le quitte à l'heure 2 000

(d'après sa valeur $ctime^{in}_{i+1,k}$) via une rotation r'. La fréquence hebdomadaire des rotations r et r' implique qu'en pratique la marchandise k quittera le port à l'heure 320. La contrainte (T.21) prend en compte ce cas de figure tandis que la contrainte (T.22) traite le cas où l'arrivée dans le port est postérieur au départ du port (d'après les valeurs $ctime^{out}_{i,k}$ et $ctime^{in}_{i+1,k}$).

$$(i+1 < nbr_k \wedge ctime^{in}_{i+1,k} \geq ctime^{out}_{i,k}) \Rightarrow \Delta_{i,k} = (ctime^{in}_{i+1,k} - ctime^{out}_{i,k})\%(7 \times 24) \quad \text{(T.21)}$$

$$(i+1 < nbr_k \wedge ctime^{in}_{i+1,k} < ctime^{out}_{i,k}) \Rightarrow \Delta_{i,k} = 7 \times 24 + (ctime^{out}_{i,k} - ctime^{in}_{i+1,k})\%(7 \times 24)) \quad \text{(T.22)}$$

Les $i$èmes rotations qui ne sont pas utilisées pour la marchandise k ont une variable $\Delta_{i,k}$ qui est nulle :

$$i+1 \geq nbr_k \Rightarrow \Delta_{i,k} = 0 \quad \text{(T.23)}$$

Le temps de transit de la marchandise k peut alors être garanti grâce à la contrainte suivante :

$$\sum_{i \in \mathcal{I}^+]} \delta_{i,k} + \sum_{i \in \mathcal{I}} \Delta_{i,k} \leq tt_{max}(k) \quad \text{(T.24)}$$

À noter que toutes les marchandises n'ont pas nécessairement une contrainte de temps de transit à respecter. Aussi, si une marchandise k ne se voit pas imposer un temps de transit maximum, les variables et contraintes présentées dans cette partie ne sont pas considérées pour la marchandise k.

3.7 Disponibilité des navires

La fréquence des rotations étant hebdomadaire, chaque port est visité par un bateau de la rotation chaque semaine. Le nombre de bateaux nécessaires correspond donc au temps total de la rotation divisé par 7. Si les variables n_r et T_r représentent respectivement le nombre de porte-conteneurs utilisés par la rotation r et le temps total de la rotation r, nous pouvons poser la contrainte suivante :

$$n_r = \left\lceil \frac{T_r}{7 \times 24} \right\rceil \quad \text{(D.1)}$$

Le temps de la rotation est, bien entendu, nul si la rotation n'est pas utilisée :

$$v_r = 0 \iff T_r = 0 \quad \text{(D.2)}$$

Sinon, chaque rotation démarrant au temps 0, le temps total est calculé à partir du moment d'arrivée au port de départ en provenance du dernier port de la rotation :

$$dep_r = p' \wedge s^r_p = p' \Rightarrow T_r = time^{out}_{p,r} + man^{out}_{p,r} + st^r_p + wt_{p',r} + man^{in}_{p',r} \quad \text{(D.3)}$$

Enfin, nous devons garantir que, pour chaque type de bateaux, le nombre de bateaux utilisés ne dépasse pas le nombre de navires disponibles :

$$\sum_{r \in R} n_r.(v_r = v) \leq nb(v) \quad \text{(D.4)}$$

3.8 Fonction objectif

Sommairement, la fonction objectif consiste à faire la différence entre les revenus apportés par l'acceptation des marchandises dans le réseau et l'ensemble des coûts qu'engendrent leur transport (carburant, exploitation des navires, escales, ...).
Pour calculer le coût lié au carburant, nous devons, au préalable, déterminer la consommation de carburant de chaque trajet réalisé. Pour cela, nous considérons une variable $cons^r_p$ par port et par rotation qui spécifie la quantité de carburant consommée par heure par la rotation r pour le trajet réalisé entre le port p et son successeur. En l'absence de successeurs, la variable $cons^r_p$ a, bien sûr, une valeur nulle. La quantité consommée dépend ici uniquement du type de bateaux utilisés et de la vitesse.

$$cons^r_p = \texttt{Elt}_{\texttt{m}}(\{cons(v,\nu) | v \in \{0\} \cup \mathcal{V}, \nu \in \{0\} \cup [\nu_{min}(v), \nu_{max}(v)]\}, v_r, \nu^r_p) \quad \text{(O.1)}$$

Nous supposons ici que $cons(v,\nu)$ vaut 0 si v vaut 0 ou si ν vaut 0.
Pour calculer les coûts liés au transbordement, nous devons connaître la quantité de marchandises transbordées dans chaque port. Pour cela, nous introduisons une variable teu^{ts}_p par port. Les marchandises k transbordées au port p sont celles qui sont déchargées au port p (c'est-à-dire celles pour lesquelles $to^r_{k,p}$ vaut 1) et pour lesquelles le port p n'est pas leur port de destination. Nous pouvons donc poser la contrainte suivante :

$$teu^{ts}_p = \sum_{k \in \mathcal{K} | p \neq pod(k)} to^r_{k,p}.q(k) \quad \text{(O.2)}$$

Nous pouvons maintenant exprimer notre fonction objectif dans laquelle figurent les revenus (R), les coûts liés au carburant (C), les frais d'escales dans les ports et de passage par les canaux (E), les frais d'exploitations des navires (X) et les frais de transbordement (T) :

$$\begin{aligned} \max \quad & \sum_{k \in \mathcal{K}} rev(k).q(k).\alpha_k && (R) \\ - \quad & \sum_{r \in R} \sum_{p \in \mathcal{P} \cup \mathcal{C}} fp_r \times cons^r_p \times st^r_p && (C) \\ - \quad & \sum_{r \in R} \sum_{p \in \mathcal{P} \cup \mathcal{C}} pc_{pr}.(s^r_p \neq p) && (E) \\ - \quad & 7 \sum_{r \in R} tc_r.n_r && (X) \\ - \quad & \sum_{p \in \mathcal{P}} ts(p).teu^{ts}_p && (T) \end{aligned}$$

3.9 Contraintes additionnelles

Compte tenu de l'ampleur de l'espace de recherche, il peut être souhaitable d'éviter autant que possible certaines symétries. Commencer chaque rotation au temps 0 (voir la contrainte (T.7)) permet d'éviter toute translation sur l'axe du temps. Toutefois, d'autres symétries peuvent exister. Par exemple, les variables v_r sont interchangeables. Pour éviter cela, nous pouvons faire en sorte que les premières rotations soient utilisées en priorité grâce à la contrainte suivante :

$$v_r = 0 \Rightarrow v_{r+1} = 0 \quad \text{(S.1)}$$

Il est également possible d'imposer une condition plus forte sur les rotations en posant la contrainte suivante :

$$T_1 \geq T_2 \geq \ldots \geq T_r \quad \text{(S.2)}$$

Certains ports ne peuvent pas accueillir certains types de bateaux. Par exemple, le port de Dutch Harbor en Alaska n'est pas suffisamment profond. Il ne peut donc accueillir que de petits porte-conteneurs. Aussi, si les bateaux de type v ne peuvent accoster au port p, nous posons, pour chaque rotation r, la contrainte suivante :

$$v_r = v \Rightarrow s_p^r = p \quad \text{(S.3)}$$

De même, certains ports ne disposent pas d'une place suffisante pour stocker des conteneurs. Il est donc impossible d'y effectuer des transbordements. Pour de tels ports, nous pouvons alors poser la contrainte suivante pour interdire tout transbordement :

$$teu_p^{ts} = 0 \quad \text{(S.4)}$$

4 Premières expérimentations

Le jeu d'instances LINER-LIB [5] constitue la référence pour les expérimentations concernant le problème LSNDP. Il est constitué de sept instances ayant de 12 à 197 ports, permettant ainsi d'évaluer aussi bien des méthodes complètes qu'incomplètes. Afin d'avoir des instances de taille raisonnable et variée, nous avons produit des sous-instances à partir d'instances du jeu LINER-LIB. Pour cela, à partir d'une instance, nous sélectionnons n ports de la manière suivante. Le premier port sélectionné est celui qui échange le plus de marchandises. Les $n-1$ ports suivants sont ceux qui échangent le plus de marchandises avec les ports déjà sélectionnés. Pour notre jeu de tests, nous avons ainsi considéré la plus petite instance (`Baltic`) du jeu LINER-LIB et 21 instances produites à partir des instances `Baltic`, `WAF` et `Mediterranean` avec un nombre de ports à allant de 3 à 10 pour les deux premières et de 3 à 6 pour la dernière. Le nombre de rotation r_{max} est fixé à 4 et celui des transbordements à 2 (une valeur supérieure n'étant pas souhaitée par les experts) tandis que la durée maximale h_{max} est de 1 344 heures (soit 8 semaines). Nous prenons la valeur 0,54 pour μ.

Pour pouvoir évaluer plus facilement différents solveurs, nous avons implémenté notre modèle (incluant les contraintes (S.3) et (S.4)) dans l'API PyCSP3 [15]. Pour nos premières expérimentations, nous avons considéré les solveurs ACE (version 2.1 [14]) et Choco (version 4.10.11 [20]). ACE est utilisé avec son paramétrage par défaut tandis que Choco est exploité avec l'heuristique de choix de variables CHS [7] et l'heuristique de choix de valeur lexicographique, les choix par défaut de Choco ayant tendance à considérer prioritairement des solutions de coût négatif. Les expérimentations sont réalisées sur des serveurs DELL PowerEdge R440 dotés de processeurs Intel Xeon Silver 4112 cadencés à 2,6 GHz et dotés de 32 Go de mémoire avec un temps d'exécution d'une heure au maximum.

La table 1 présente les résultats obtenus. Pour les plus petites instances, les deux solveurs parviennent généralement à trouver l'optimum et, dans ce cas, ACE s'avère plus rapide que Choco. Malheureusement, ACE ne parvient pas à traiter les autres instances du fait du nombre de variables et de valeurs qui devient trop important vis-à-vis de la manière dont ACE représente les décisions prises durant la résolution. Choco, pour sa part, se révèle plus robuste, mais, ne parvient pas à traiter l'instance `Baltic`, l'amplitude de la fonction objectif devenant trop important (certains coûts pouvant être de l'ordre de plusieurs millions).

Instance	$\lvert\mathcal{P}\rvert$	ACE		Choco	
		Rés.	Tps	Rés.	Tps
`subinstance Baltic`	3	OPT	16	OPT	18
	4	OPT	23	OPT	174
	5	OPT	202	SAT	3 600
	6	-	-	SAT	3 600
	7	-	-	SAT	3 600
	8	-	-	SAT	3 600
	9	-	-	SAT	3 600
	10	-	-	SAT	3 600
`Baltic`	12	-	-	-	-
`subinstance WAF`	3	OPT	12	OPT	7
	4	OPT	45	OPT	154
	5	SAT	3 600	SAT	3 600
	6	-	-	SAT	3 600
	7	-	-	SAT	3 600
	8	-	-	SAT	3 600
	9	-	-	SAT	3 600
	10	-	-	SAT	3 600
`subinstance Mediterranean`	3	OPT	12	OPT	17
	4	OPT	21	OPT	58
	5	-	-	OPT	180
	6	-	-	OPT	271

TABLE 1 – Résultats et temps d'exécution (en secondes) des solveurs ACE et Choco.

Le passage par l'API PyCSP3 nous a permis de tester rapidement plusieurs solveurs. Toutefois, compte tenu des variables et contraintes auxiliaires générées par les solveurs lors de la lecture des instances, il est clair qu'une implémentation fine du modèle directement au sein des solveurs sera nécessaire pour obtenir une approche compétitive.

5 Conclusions et perspectives

Dans cet article, nous avons proposé un modèle basé sur la PPC permettant de traiter le problème LSNDP. Les résultats que nous avons obtenus sont très préliminaires et devront être étendus à d'autres solveurs. Ils permettent toutefois d'esquisser plusieurs pistes. Les premières visent à implémenter le modèle directement au sein des solveurs, à mieux maîtriser l'amplitude de la fonction objectif ou encore à casser certaines symétries. Par ailleurs, il faudra envisager d'autres approches pour sa résolution (résolution successive d'instances CSP, VNS, ...). Enfin, le modèle devra être étendu pour prendre en compte les contraintes imposées par l'OMI concernant les émissions de gaz des navires.

Remerciements

Ce travail est soutenu par Bpifrance dans le cadre du projet PIA Transformation Numérique du Transport Maritime (TNTM).

Références

[1] R. Agarwal and Ö. Ergun. Ship Scheduling and Network Design for Cargo Routing in Liner Shipping. *Transportation Science*, 42(2) :175–196, 2008.

[2] J. F. Álvarez. Joint routing and deployment of a fleet of container vessels. *Maritime Economics & Logistics*, 11(2) :186–208, 2009.

[3] B. D. Brouer, G. Desaulniers, C. V. Karsten, and D Pisinger. A Matheuristic for the Liner Shipping Network Design Problem with Transit Time Restrictions. In *Computational Logistics - 6th International Conference, ICCL*, volume 9335 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2015.

[4] B. D. Brouer, C. V. Karsten, and D. Pisinger. Optimization in liner shipping. *Annals of Operations Research*, 271(1) :205–236, 2018.

[5] B. D. Brouer, J. F. Álvarez, C. E. M. Plum, D. Pisinger, and M. M. Sigurd. A base integer programming model and benchmark suite for liner-shipping network design. *Transportation Science*, 48(2) :281–312, 2014.

[6] M. Christiansen, E. O. Hellsten, D. Pisinger, D. Sacramento, and C. Vilhelmsen. Liner shipping network design. *European Journal of Operational Research*, 286, 2020.

[7] D. Habet and C. Terrioux. Conflict history based heuristic for constraint satisfaction problem solving. *Journal of Heuristics*, 27(6) :951–990, 2021.

[8] C. V. Karsten, B. D. Brouer, G. Desaulniers, and D. Pisinger. Time constrained liner shipping network design. *Transportation Research. Part E : Logistics and Transportation Review*, 105 :152–162, 2017.

[9] C. V. Karsten, D. Pisinger, S. Røpke, and B. D. Brouer. The time constrained multi-commodity network flow problem and its application to liner shipping network design. *Transportation Research Part E : Logistics and Transportation Review*, 76 :122–138, 2015.

[10] E. Kelareva, K. Tierney, and P. Kilby. CP methods for scheduling and routing with time-dependent task costs. *EURO Journal on Computational Optimization*, 2(3) :147–194, 2014.

[11] P. Kilby and P. Shaw. Vehicle Routing. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 801–836. Elsevier, 2006.

[12] D. Kizilay, P. Van Hentenryck, and D. Türsel Eliiyi. Constraint programming models for integrated container terminal operations. *European Journal of Operational Research*, 286(3) :945–962, 2020.

[13] A. Krogsgaard, D. Pisinger, and J. Thorsen. A flow-first route-next heuristic for liner shipping network design. *Networks*, 72(3) :358–381, 2018.

[14] C. Lecoutre. Ace, a generic constraint solver. *CoRR*, abs/2302.05405, 2023.

[15] C. Lecoutre and N. Szczepanski. PYCSP3 : modeling combinatorial constrained problems in python. *CoRR*, abs/2009.00326, 2020.

[16] Q. Meng and S. Wang. Liner shipping service network design with empty container repositioning. *Transportation Research Part E : Logistics and Transportation Review*, 47(5) :695–708, 2011.

[17] Q. Meng and S. Wang. Optimal operating strategy for a long-haul liner service route. *European Journal of Operational Research*, 215(1) :105–114, 2011.

[18] C. E. M. Plum, D. Pisinger, J. J. Salazar-González, and M. M. Sigurd. Single liner shipping service design. *Computers & Operations Research*, 45 :1–6, 2014.

[19] C. E. M. Plum, D. Pisinger, and M. M. Sigurd. A service flow model for the liner shipping network design problem. *European Journal of Operational Research*, 235(2) :378–386, 2014.

[20] C. Prud'homme and J.-G. Fages. Choco-solver : A java library for constraint programming. *J. Open Source Softw.*, 7(78) :4708, 2022.

[21] L. B. Reinhardt and D. Pisinger. A branch and cut algorithm for the container shipping network design problem. *Flexible Services and Manufacturing Journal*, 24(3) :349–374, 2012.

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

[23] D. Sacramento, C. Solnon, and D. Pisinger. Constraint Programming and Local Search Heuristic : a Matheuristic Approach for Routing and Scheduling Feeder Vessels in Multi-terminal Ports. *Annals of Operations Research*, 1(4), 2020.

[24] K. Thun, H. Andersson, and M. Christiansen. Analyzing complex service structures in liner shipping network design. *Flexible Services and Manufacturing Journal*, 29(3-4) :535–552, 2017.

[25] K. Tierney. *Optimizing Liner Shipping Fleet Repositioning Plans*, volume 57 of *Operations research / computer science interfaces series*. Springer, 2015.

[26] S. Wang and Q. Meng. Liner shipping network design with deadlines. *Computers & Operations Research*, 41 :140–149, 2014.

Optimisation de la consommation d'énergie et de la collecte de données dans les réseaux de capteurs sans fil.

Khadidja.FELLAH[1], Soumaya.FELLAH[2], Bouabdellah.KECHAR[3]

[1]Ecole supérieure d'économie d'Oran.
[2]Ecole Nationale Polytechnique d'Oran, Maurice AUDIN.
[3]Université d'Oran1, laboratoire RIIR.

Résumé

Notre objectif dans ce travail (1) *consiste à proposer des solutions basées sur les techniques d'optimisation afin de minimiser la consommation d'énergie dans les réseaux de capteurs sans fil (RCSF) organisés en cluster. Nous avons conçu un ILP0-1 qui permet de déterminer la position optimale dans chaque Cluster pour le séjour du Sink lors de son passage dans ce Cluster et le niveau optimal de transmission pour chaque nœud capteur avec la préservation d'autres contraintes liées aux caractéristiques des RCSF. Nous avons par la suite utilisé la courbe de Hilbert comme trajectoire optimale pour la collecte de données.*

Mots-clés

Réseaux de capteurs sans fil, Economie de l'énergie, optimisation, mobilité, courbes de Hilbert.

Abstract

Our objective in this work (1) *is to propose solutions based on optimization techniques in order to minimize energy consumption in wireless sensor networks (WSN) organized in clusters. We designed an ILP0-1 which determine the optimal position in each Cluster for the Sink sejourning in this Cluster and the optimal transmission level for each sensor node with the preservation of other constraints related to the characteristics of WSN. We then used the Hilbert curve as the optimal trajectory for data collection.*

Keywords

Wireless Sensor Networks, energy saving, mobility, optimization, Hilbert curve.

1 Introduction

Nous supposons que le RCSF est subdivisé en Clusters (en utilisant par exemple le protocole LEACH (2)) et nous appliquons ensuite notre programme linéaire en variable 0-1 au niveau de chaque Cluster. Le résultat d'optimisation de notre PL 0-1 au niveau de chaque Cluster permet de trouver la position optimale pour le séjour du Sink (Nœud puit pour la collecte de données) dans ce Cluster considérée comme un point de rendez vous (RDV) et de déterminer également les niveaux de transmissions optimaux pour chaque nœud capteur membre du Cluster. La communication effective entre chaque membre du Cluster et le Sink pour la collecte de données aura lieu une fois que ce dernier atteint le point de séjour.

La collecte de données du réseau en entier se fera en calculant une trajectoire optimisée. Cette trajectoire doit passer par les points de séjours déterminés par le PL 0-1. Plusieurs solutions pour optimiser la trajectoire du Sink mobile pour la collecte de données ont été proposées dans la littérature (3), certaines se basent sur les méta-heuristiques afin de trouver une trajectoire proche de l'optimum. D'autres se basent sur la méthode exacte du problème de voyageur de commerce qui pose un problème de passage à l'échelle. Afin de surmonter ce dernier problème nous avons introduit la courbe de Hilbert qui permet un passage à l'échelle tout en ayant un parcours optimale.

2 Démarche d'optimisation

2.1. ILP

Nous proposons un programme linéaire en variable 0-1, dont l'objectif est l'économie d'énergie dans les RCSF (4), où certain nœuds (capteurs et/ou Sink) ont la possibilité de se déplacer dans le champ de déploiement. Le déplacement des nœuds est réalisé dans le but de minimiser la distance de communication. Les nœuds capteurs peuvent se déplacer pour assurer aussi la couverture et la connectivité dans le réseau. Cette première contribution est une extension de notre travail (5). Elle permet aussi la prise en considération de l'ajustement de la puissance de transmission, l'ajustement du rayon de capture et la considération de l'énergie résiduelle et la possibilité d'alimenter les nœuds par une énergie supplémentaire, par exemple à partir de l'environnement. Nous avons effectué plusieurs expériences avec les paramètres réalistes de l'environnement de simulation Castalia (6), c-à-d les caractéristiques physiques du module radio CC2420. Le modèle d'optimisation PL 0-1 a été résolu à l'aide du solveur CPLEX (7).

2.2. La courbe de Hilbert

Hilbert en 1891 (8) propose une manière alternative de faire correspondre les points d'un carré avec ceux d'un segment. Le principe de la courbe de Hilbert consiste à partager le carré $[0, 1]^2$ en 4 petits carrés égaux, numéroter chacun de ces carrés de sorte que deux carrés successifs se touchent par un côté, en commençant par le carré en bas à gauche, et en terminant par le carré en bas à droite. Ensuite, partager chacun de ces carrés en 4 micro carrés égaux; numéroter chacun de ces carrés de sorte que deux micro carrés successifs se touchent par un côté, en commençant par le micro-carré en bas à gauche, et terminant par le micro-carré en bas à droite, le premier micro-carré d'un petit carré devant avoir un côté en commun avec le dernier micro-carré du petit carré précédent et le dernier micro-carré devant toucher par un coté le petit carré suivant. Ce processus est recommencé à l'infini. La courbe de Hilbert d'ordre n est alors la ligne brisée joignant les centres successifs de ces carrés. Ceci est valable dans le cas d'une application à deux dimensions. La figure 1(a) montre le parcours du Sink suivant la courbe de Hilbert et la figure 1(b) montre le parcours du Sink suivant la courbe de Hilbert optimisée avec optimisation au niveau des clusters.

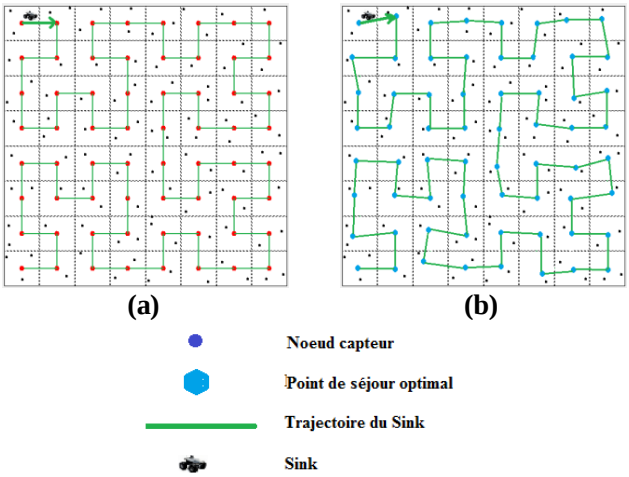


Figure 1 : Exemple de génération de trajectoire pour la collecte de données avec la nouvelle stratégie mixte d'optimisation : (a) sans optimisation, (b) avec optimisation.

3 Conclusion

Dans cet article, nous avons étudié le problème de l'économie d'énergie dans un réseau de capteurs sans fil en considérant la mobilité du Sink pour la collecte de données. Nous avons proposé une stratégie combinée d'économie d'énergie visant à prolonger la durée de vie du réseau. La première stratégie est basée sur l'optimisation de l'énergie dans chaque cluster en appliquant l'ILP 0-1 qui détermine le VRP optimal et les niveaux de transmission optimaux pour chaque nœud capteur membre de ce cluster. Après cette phase d'optimisation au niveau du Cluster, nous sommes passés à la phase d'optimisation de la collecte des données dans le réseau. Nous avons opté pour les courbes de Hilbert comme trajectoire optimisée passant par le VRP optimal. La simulation réalisée montre que notre nouvelle approche prolonge significativement la durée de vie du réseau.

4 References

[1]. *New approach based on Hilbert curve forenergy efficient data collection in WSN with mobile sink.* Khadidja, FELLAH and KECHAR, Bouabdellah. 2020, IET Wireless Sensor Systems.

[2]. *Energy-efficient communication protocol for wireless microsensor networks.* Heinzelman, W R, Chandrakasan, A and Balakrishn, H. Maui, HI, USA : IEEE, 2000. the 33rd Annual Hawaii International Conference on System Sciences. pp. 3005–3014.

[3]. *Static vs. mobile sink: The influence of basic parameters on energy efficiency in wireless sensor networks.* Khan, M I, Gansterer, W N and Haring, G. 9, 2013, Computer Communications, Vol. 36, pp. 965–978.

[4]. *An integer Linear Programming Approach for Optimizing Energy Consumption in Mobile Wireless Sensor Networks under realistic constraints.* Fellah, K and Kechar, B. 2, 2017, International Journal Mathematical Modelling and Numerical Optimisation,, Vol. 8, pp. 162-182.

[5]. Fellah, K. *Techniques d'optimisation pour l'économie d'énergie dans les réseaux de capteurs sans fil.* Mémoire de magister : université d'Oran1, 2008.

[6]. Castalia. *Castalia:Wireless Sensor Network Simulator.* [Online] [visiter: 12 06, 2020.] https://castalia.forge.nicta.com.au/index.php/en/.

[7]. Cplex. [Online] [Visiter: 12 01, 2023.] https://www.ibm.com/developerworks/downloads/ws/ilogcple

Une contrainte globale pour l'ordonnancement des transferts de données dans les missions spatiales

J. Rouzot[1,2], C. Artigues[1], P. Garnier[2], E. Hebrard[1], P. Lopez[1]

[1] LAAS-CNRS, Université de Toulouse, CNRS, Toulouse
[2] IRAP, Université de Toulouse, CNRS, UT3, CNES, Toulouse

Résumé

Dans le cadre des missions spatiales où les ressources à bord sont limitées, il est essentiel de transférer les données acquises de manière efficace afin de limiter le risque de saturation des mémoires. Nous nous intéressons ici en particulier à l'ordonnancement des transferts de données de la mission Rosetta de l'ESA (l'agence spatiale européenne). Ce problème a été résolu par des méthodes heuristiques qui ont donné de bons résultats, voire des résultats optimaux dans certains cas. Dans la lignée d'un travail précédent, nous proposons dans cet article un modèle de programmation par contraintes pour résoudre exactement ce problème, nous proposons une nouvelle contrainte globale pour le partage de bande passante en fonction de priorités, ainsi qu'une contrainte globale pour les classements de type Dense Ranking.

Des premiers résultats montrent la pertinence et l'efficacité d'une telle approche exacte pour la résolution de ce problème. Un des buts dans la conception de ces contraintes est leur généricité, afin qu'elles puissent être appliquées à toute mission spatiale nécessitant des transferts de données, ainsi qu'à tout problème nécessitant un classement Dense Ranking.

Mots-clés

Ordonnancement, Programmation par contraintes, Contrainte globale, Mission d'exploration spatiale, Dense Ranking

1 Introduction

La mission Rosetta a été lancée par l'Agence spatiale européenne (ESA) en 2004 dans le but d'analyser la comète 67P/Tchourioumov-Gerassimenko, et a abouti en 2014. Cette mission est représentative de nombreuses missions soutenues par le "*Deep Space Network*", et l'étudier permet donc de concevoir des systèmes de planification efficaces pour les missions futures.

Nous nous intéressons ici au problème du vidage des données de l'orbiteur Rosetta, défini par Chien, Rabideau et al. [2, 5], similaire au problème de vidage des données de l'atterrisseur Philae [7]. Dans les deux cas, un certain nombre d'activités, qui produisent des données temporairement stockées dans la mémoire de leur instrument, doivent être ordonnancées. De plus, un ordre de priorité doit être établi pour chacune des fenêtres de visibilité entre la sonde et la Terre. Cet ordre de priorité dicte le partage de bande passante entre les différents instruments lors du transfert de données pendant cette fenêtre. L'objectif est de minimiser le pic d'utilisation maximal des mémoires (i.e. maximiser la marge de sécurité, où la marge de sécurité est le ratio entre la capacité de la mémoire et son remplissage maximal au cours de la mission).

Dans le cas de Philae, l'ordre de priorité est considéré fixe, et les variables de décision correspondent à l'ordonnancement des activités. Dans le cas de Rosetta, à l'inverse, l'ordonnancement des activités est une entrée, et les variables correspondent à l'ordre de priorité pour le transfert des données.

Il s'agit d'un ordre partiel, plus précisément un *classement* : les mémoires de même priorité se partagent la bande passante grâce à un algorithme de type Round-Robin (file d'attente circulaire). Ainsi, les mémoires avec la plus forte priorité peuvent utiliser toute la bande passante initiale, puis la bande passante restante est redistribuée jusqu'à ce qu'il n'en reste plus, ou que les mémoires soient toutes vides. On considère ici que le remplissage des mémoires est complètement déterminé par la planification des expériences scientifiques et donc connu à l'avance. Les priorités ne peuvent être changées qu'en début de visibilité, mais sont déterminées en amont, même si elles peuvent être amenées à être recalculées au cours de la mission.

Comme la relation entre l'utilisation des mémoires et les priorités est complexe, utiliser une modélisation de programmation par contraintes classique est très peu efficace. Afin d'accélérer la résolution, nous avons donc développé les contraintes globales suivantes :

— La contrainte `PRIORITY TRANSFER` permet de déterminer le pic maximal d'utilisation des mémoires au cours d'une fenêtre de vidage et l'utilisation des mémoires à la fin de cette fenêtre en fonction d'une affectation de priorités. Cette contrainte permet également d'inférer de nouvelles bornes sur les variables de priorité en fonction de la borne supérieure du pic d'utilisation maximal courant. Cette contrainte utilise les algorithmes décrits par Hebrard et al. [3].

— La contrainte `DENSE RANKING` permet de filtrer efficacement les domaines des variables de priorité pour obtenir un classement sans symétries.

Nous décrivons dans cet article les contraintes globales précédentes, puis un algorithme de calcul rapide de bornes inférieures sur les mémoires et les pics d'utilisation pour chaque fenêtre de vidage. Nous concluons sur l'intérêt de ces contraintes globales ainsi que sur les perspectives pour améliorer le modèle.

2 Modélisation du problème

Dans le problème du vidage des données de Rosetta, ou *overlapping Memory Dumping Problem (oMDP)* [6], on considère m fenêtres de vidage aux dates de début et fin $[s_j, e_j[_{\{j=1..m\}}$ pendant lesquelles la bande passante à partager entre les mémoires est une constante δ_j. On a n mémoires concurrentes, avec une capacité limitée. On note $r_{i,j}$ l'usage maximum de la mémoire i par rapport à sa capacité au cours de la fenêtre j. Pour ce problème, nous considérons que la fonction de remplissage de chaque mémoire au cours du temps est une fonction constante par morceaux. Cela implique que l'évolution de l'utilisation des mémoires au cours du temps suit une fonction linéaire par morceaux, puisque le transfert des données est aussi constant par morceaux.

Nous définissons les priorités de la façon suivante :

- Des groupes de priorité sont définis, pouvant prendre une valeur de 1 à n. Le groupe 1 est le plus prioritaire, tandis que le groupe n est le moins prioritaire.
- Les mémoires sont affectées à un groupe de priorité pour chaque fenêtre de vidage ; plusieurs mémoires peuvent être affectées au même groupe.
- Les mémoires au sein d'un même groupe de priorité partagent la bande passante disponible grâce à un algorithme de type Round-Robin.

Nous proposons un modèle de programmation par contraintes pour résoudre les instances du oMDP. Dans notre modèle, nous séparons les instances par fenêtre de visibilité j et nous introduisons les variables suivantes :

- $p_{i,j}$ représente le groupe de priorité de la mémoire i au cours de la fenêtre j.
- $mem^s_{i,j}$ et $mem^e_{i,j}$ représentent respectivement l'utilisation de la mémoire i au début et à la fin de la fenêtre j.
- $r_{i,j}$ représente le pic d'utilisation maximum de la mémoire i au cours de la fenêtre j.
- $rmax$ représente le pic d'utilisation maximum des mémoire sur l'instance globale.

L'objectif est de minimiser l'usage maximal des mémoires sur l'instance : $\min\ rmax$.

La figure 1 montre l'évolution de l'utilisation des mémoires au cours du temps pour une solution particulière (affectation de priorités pour chaque fenêtre de vidage). Les zones vertes représentent les fenêtres de visibilité, pendant lesquelles les données des mémoires peuvent être transférées sur Terre à un taux δ_j, partagé entre les mémoires en fonction de leur priorité. Pendant ces fenêtres de visibilité, les mémoires ont une priorité fixe, représentée sur la figure 1 en rouge. Pour chaque mémoire, on représente le remplissage

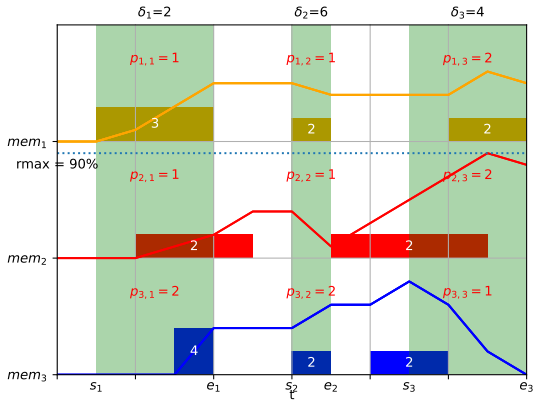


FIGURE 1 – Utilisation des mémoires au cours du temps et pic maximum pour une affectation de priorités.

au cours du temps par des rectangles de couleur où la hauteur représente le taux (constant) de remplissage. L'évolution de l'utilisation de chaque mémoire est représentée par une fonction linéaire par morceaux. Dans cet exemple, la capacité de chaque mémoire est de 10 unités ; le pic maximum est donc atteint pour la mémoire 2, pour une valeur de 9 unités, soit $rmax = r_{2,3} = \frac{9}{10} = 90\,\%$.

3 Contraintes globales

La principale difficulté pour oMDP est d'exprimer efficacement la relation entre une affectation de priorité et l'évolution de l'utilisation des mémoires résultant. Cette difficulté a motivé le développement de la contrainte globale `PRIORITY TRANSFER` qui, pour chaque fenêtre j, lie les variables $p_{i,j}, mem^e_{i,j}, mem^s_{i,j}$ et $r_{i,j}$.

De plus, la modélisation des variables de priorité a l'avantage d'être directe par rapport à des priorités relatives entre chaque mémoire, mais comporte de nombreuses symétries qui doivent être cassées pour accélérer la résolution. Nous présentons la contrainte globale `DENSE RANKING` qui permet de briser ces symétries efficacement.

3.1 Contrainte Priority Transfer

Une fois que les variables $p_{i,j}$ et $mem^s_{i,j}$ sont fixées, on peut calculer l'évolution des mémoires au cours du temps dans la fenêtre j. On utilise l'algorithme `SIMULATION` [3] pour déterminer les valeurs des variables $mem^e_{i,j}$ et $r_{i,j}$ au cours de la résolution. Cependant, cet algorithme ne permet pas de faire de propagation sur les priorités.

Comme l'objectif de notre modèle est de minimiser le pic d'utilisation maximum de la mémoire, il est possible d'éliminer certaines valeurs dans les domaines des variables de priorité, en fonction de la valeur de la borne supérieure de la variable $rmax$ au cours de la résolution. En effet, on peut montrer que certaines affectations de priorité mènent nécessairement à un dépassement du pic maximum courant, grâce à l'algorithme `SINGLE WINDOW` [3].

Plus précisément, cet algorithme construit une affectation

de priorités partielles pour un pic d'utilisation maximum donné. L'affectation est partielle car elle ne donne pas une valeur stricte pour les variables de priorité, mais un ordre de priorité relatif entre les mémoires.
Puisque cet ordre partiel doit être respecté pour satisfaire le pic maximal courant sur la fenêtre, on peut inférer de nouvelles bornes sur les variables de priorité sur la fenêtre. L'ordre partiel nous donne un ensemble de relations $p_{i,j} < p_{i',j}$ qui permettent de filtrer les bornes des variables de priorité.
La complexité de l'algorithme `SINGLE WINDOW` étant en $O(n^3 \log(n))$, la contrainte `PRIORITY TRANSFER` permet donc un filtrage des domaines des variables de priorité aussi en $O(n^3 \log(n))$ puisque la mise à jour des domaines, après avoir construit l'ordre partiel entre les mémoires, se fait en $O(n^2)$.

3.2 Contrainte Dense Ranking

Il existe plusieurs façon d'effectuer des classements (ex. classement standard "1224", classement modifié "1334", etc.) dont certains ont déjà été étudiés en programmation par contraintes, comme le classement standard [1].
Dans notre modèle, les priorités sont définies pour chaque mémoire par l'appartenance à un groupe de priorité et on autorise plusieurs mémoires à avoir une priorité égale. Afin d'éviter les symétries dans les solutions réalisables, nous avons décidé d'adopter un classement *Dense Ranking*.
Les règles suivantes doivent être respectées pour qu'une solution soit considérée valide à l'égard d'un *Dense Ranking* :

- Le groupe de priorité 1 doit être non vide.
- Si le groupe de priorité $i+1$ est non vide, alors le groupe de priorité i doit être non vide.

Pour éliminer les solutions qui ne respectent pas ces règles, nous présentons la contrainte globale `DENSE RANKING` qui permet un filtrage efficace de type cohérence de borne des priorités. L'algorithme de filtrage des domaines est découpé en deux parties :

- L'algorithme `PRIORITY SAT` vérifie si les domaines des variables de priorité permettent de construire une solution qui satisfait la contrainte `DENSE RANKING`.
- L'algorithme `SYMMETRY FILTERING` permet de réaliser la cohérence de bornes en utilisant l'algorithme précédent comme oracle.

Dans oMDP, on a n variables de priorité avec des domaines $[1, ..., n]$. Les domaines des variables de priorité sont valides si et seulement si l'on peut construire une solution en respectant les règles de *Dense Ranking*. L'algorithme `PRIORITY SAT` (algorithme 1) permet de vérifier l'existence d'une solution à partir des domaines des variables de priorité en $O(n \log(n))$. Les variables de priorité p sont d'abord triées par borne inférieure croissante (ligne 2) puis, pour chaque rang de priorité, on ajoute dans un tas h, ordonnées par borne supérieure croissante, toutes les variables de priorité dont la borne inférieure est égale au rang courant (lignes 5-8). S'il est possible d'affecter au moins une variable au rang courant à chaque étape (lignes 9-10), jusqu'à ce que toutes les variables soient affectées, il existe une solution réalisable pour les domaines courants. Si l'on ne peut pas traiter un des rangs (lignes 12-13), et qu'il reste des variables qui n'ont pas été affectées, alors il n'existe pas de solution réalisable pour les domaines courants. Pour chaque rang, il faut retirer du tas les variables qui n'ont pas été affectées, mais dont la borne supérieure est égale au rang courant, ce qui revient à affecter la même priorité à plusieurs variables (ligne 13-14).
En effectuant une recherche dichotomique sur les domaines des variables de priorité, on peut donc faire le filtrage des bornes en $O(n^2 \log(n)^2)$.
La contrainte globale Dense Ranking peut s'exprimer de manière équivalente avec la contrainte AtLeastNValues(N, X), basée sur SoftAllDiff(X), qui assure que les variables de l'ensemble X prennent au moins N valeurs distinctes [4]. En effet, Dense Ranking(X) est équivalent à AtLeastNValues(Max(X), X). Cependant, l'algorithme pour trouver un support pour la contrainte AtLeastNValues se base sur le couplage de cardinalité maximum dans un graphe biparti. Avec $n = |X|$ le nombre de variables et $m = \sum |D(X)|$, l'algorithme est en $O(m\, n^{\frac{1}{2}})$, soit en $O(n^{\frac{5}{2}})$ pour nos domaines. Le filtrage complet des domaines se fait aussi en $O(n^{\frac{5}{2}})$, ce qui est moins efficace qu'avec notre méthode. De plus, le test de satisfaisabilité est bien plus efficace avec la contrainte globale Dense Ranking.

Algorithm 1 PRIORITY SAT

```
1:  Input : p
2:  Trier p par borne inférieure
3:  i ← 0, current ← −1
4:  while i < n do
5:      current ← current + 1
6:      while p_i.Lb() = current do
7:          h.Add(p_i)
8:          i ← i + 1
9:      if not h.Empty() then
10:         h.RemoveMin()
11:     else
12:         return False
13:     while h.Min().Ub() = current do
14:         h.RemoveMin()
15: return True
```

4 Borne inférieure

Une borne inférieure sur les variables $mem^s_{i,j}$, $mem^e_{i,j}$ et $r_{i,j}$ est facile à déterminer en utilisant l'algorithme `SIMULATION` [3]. Du point de vue d'une mémoire i, le meilleur scénario possible est d'être plus prioritaire que les autres mémoires à chaque fenêtre de vidage. Ainsi, pour chaque mémoire i, en affectant la priorité $p_{i,j} = lb(p_{i,j})$ et $p_{i',j} = ub(p_{i',j})\ \forall\ i' \neq i$, et en supposant que $mem^s_{i,j}$ est égale à sa borne inférieure on peut déterminer un meilleur cas d'utilisation de mémoire i en simulant cette affectation particulière de priorités, et donc une borne inférieure sur $mem^e_{i,j}$ et $r_{i,j}$. L'algorithme `SIMULATION` est de complexité $O(n \log(n))$, ce qui est relativement peu coûteux.

Instances	Dense Ranking	Temps moyen	Temps médian	% solutions optimales
2 mémoires, 2 fenêtres (*2m/2f*)	Non	**351 μs**	**341 μs**	100%
4 mémoires, 4 fenêtres (*4m/4f*)	Non	**170 ms**	169 ms	40%
6 mémoires, 6 fenêtres (*6m/6f*)	Non	**6 s**	**448 ms**	30%
2 mémoires, 2 fenêtres (*2m/2f*)	Oui	650 μs	572 μs	100%
4 mémoires, 4 fenêtres (*4m/4f*)	Oui	2 s	**12 ms**	**90%**
6 mémoires, 6 fenêtres (*6m/6f*)	Oui	28 s	6 s	**50%**

TABLE 1 – Temps et % des instances résolues à l'optimal avec et sans la contrainte Dense Ranking.

On peut donc déterminer une borne inférieure sur chaque variable $mem^s_{i,j}$, $mem^e_{i,j}$ et $r_{i,j}$ en $O(n\log(n))$.

5 Premiers résultats expérimentaux

Pour étudier l'utilité de la contrainte `DENSE RANKING`, nous avons généré 30 instances aléatoires de taille variable (2 mémoires et 2 fenêtres *2m/2f*, 4 mémoires et 4 fenêtres *4m/4f*, 6 mémoires et 6 fenêtres *6m/6f*). Nous résolvons ces instances avec et sans la contrainte globale `DENSE RANKING` avec la configuration suivante :

- Solveur : OR-Tools Original CP solver [1].
- Temps limite : 100 secondes.
- Configuration matérielle : 12th Gen Intel Core i7-1265U 100 MHz, 32 GB de RAM.

Les résultats expérimentaux montrent une forte augmentation du taux d'instances résolues à l'optimum. On observe cependant une augmentation globale du temps de résolution avec la contrainte `DENSE RANKING`.

6 Conclusion

Nous avons présenté deux nouvelles contraintes globales pour l'ordonnancement des transferts de données par priorités, et nous montrons l'efficacité de la contrainte `DENSE RANKING` pour améliorer la résolution. La contrainte `PRIORITY TRANSFER` permet un filtrage efficace des variables de priorité, mais nécessite de trouver des bornes supérieures sur le pic d'utilisation maximum pour être vraiment utile. Trouver des solutions de qualité est donc un point crucial pour l'amélioration du modèle. Nous dirigeons aujourd'hui notre recherche vers l'intégration d'heuristiques efficaces dans notre modèle de programmation par contraintes, en particulier celles décrites dans la référence [3], ainsi que sur le calcul de bornes de meilleure qualité. Ces améliorations feront l'objet d'une étude des performances du modèle sur des instances plus grandes. La construction d'une procédure plus efficace que celle présentée dans cet article pour faire le filtrage de bornes des variables de priorité pour la contrainte `DENSE RANKING` est aussi en cours d'étude. La contrainte sera testée sur des problèmes variés et comparée avec d'autres méthodes pour démontrer son efficacité.

Références

[1] Christian Bessiere, Emmanuel Hebrard, George Katsirelos, Zeynep Kiziltan, and Toby Walsh. Ranking constraints. In Subbarao Kambhampati, editor, *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 705–711. IJCAI/AAAI Press, 2016.

[2] Steve Chien, Gregg Rabideau, Daniel Tran, Martina Troesch, Joshua Doubleday, Federico Nespoli, Miguel Perez Ayucar, Marc Costa Sitja, Claire Vallat, Bernhard Geiger, et al. Activity-based scheduling of science campaigns for the Rosetta orbiter. In *Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-15*, Buenos Aires, Argentina, July 2015.

[3] Emmanuel Hebrard, Christian Artigues, Pierre Lopez, Arnaud Lusson, Steve Chien, Adrien Maillard, and Gregg Rabideau. An efficient approach to data transfer scheduling for long range space exploration. In Luc De Raedt, editor, *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, pages 4635–4641. International Joint Conferences on Artificial Intelligence Organization, 7 2022. Main Track.

[4] Thierry Petit, Jean-Charles Régin, and Christian Bessiere. Specific filtering algorithms for over-constrained problems. In *Principles and Practice of Constraint Programming—CP 2001 : 7th International Conference, CP 2001 Paphos, Cyprus, November 26–December 1, 2001 Proceedings 7*, pages 451–463. Springer, 2001.

[5] Gregg Rabideau, Steve Chien, M. Galer, Federico Nespoli, and Manuel Costa. Managing spacecraft memory buffers with concurrent data collection and downlink. *Journal of Aerospace Information Systems*, 14(12) :637–651, 2017.

[6] Gregg Rabideau, Steve Chien, Federico Nespoli, and Manuel Costa. Managing spacecraft memory buffers with overlapping store and dump operations. In *Workshop on Scheduling and Planning Applications, International Conference on Automated Planning and Scheduling (SPARK, ICAPS 2016)*, London, UK, June 2016.

[7] Gilles Simonin, Christian Artigues, Emmanuel Hebrard, and Pierre Lopez. Scheduling scientific experiments for comet exploration. *Constraints An Int. J.*, 20(1) :77–99, 2015.

1. https ://developers.google.com/optimization/cp/original_cp_solver