Actes des seizièmes Journées Francophones de Programmation par Contraintes

JFPC 2021

22 - 24 juin 2021 – ~~Nice~~ « en ligne »

Comité de programme

Président

Philippe Vismara	Institut Agro Montpellier

Membres

David Allouche	INRAE Toulouse
Noureddine Aribi	Université d'Oran, Algérie
Gilles Audemard	CRIL, Lens
Vincent Barichard	LERIA, Université d'Angers
Mohamed-Bachir Belaid	Simula Research Laboratory, Norvège
Quentin Cappart	École Polytechnique de Montréal, Canada
Clément Carbonnel	LIRMM, Montpellier
Thi-Bich-Hanh Dao	LIFO, Université d'Orléans
Simon De Givry	INRAE Toulouse
Sophie Demassey	CMA, MINES ParisTech, Sophia Antipolis
Redouane Ezzahir	Université Ibn Zohr, Agadir, Maroc
Carmen Gervet	Espace Dev, Montpellier
Gaël Glorian	CRIL, Lens
Ratheil Houndji	Université d'Abomey-Calavi, Bénin
Frederic Lardeux	LERIA, Université d'Angers
Christophe Lecoutre	CRIL, Lens
Chu-Min Li	MIS, Université de Picardie Jules Verne, Amiens
Arnaud Malapert	Université Côte d'Azur
Margaux Nattaf	G-SCOP, INP Grenoble
Bertrand Neveu	Ecole des Ponts ParisTech
Abdelkader Ouali	GREYC, Université Caen
Anastasia Paparrizou	CRIL, Lens
Eric Piette	Maastricht University, Pays-Bas
Nicolas Prcovic	LIS, Marseille
Laurent Simon	LABRI, Bordeaux
Christine Solnon	INSA Lyon
Julien Vion	LAMIH, Valenciennes

Comité d'organisation

Présidence

Arnaud Malapert	Université Côte d'Azur
Marie Pelleau	Université Côte d'Azur

Membres

Elisabetta De Maria	Université Côte d'Azur
Arthur Finkelstein	Instant System, Université Côte d'Azur
Rémy Garcia	Université Côte d'Azur
Loïc Germerie	Université Côte d'Azur
Nicolas Isoart	Université Côte d'Azur
Victor Jung	Université Côte d'Azur
Laetitia Laversa	Université Côte d'Azur
Jean-Charles Régin	Université Côte d'Azur
Sara Riva	Université Côte d'Azur

Programme

Préface

Les JFPC (Journées Francophones de Programmation par Contraintes) permettent chaque année de réunir la communauté francophone en programmation par contraintes (CP) qui regroupe des chercheurs en Intelligence Artificielle et/ou Recherche Opérationnelle autour de nombreux thèmes comme la résolution de problèmes de satisfaction de contraintes (CSP), d'optimisation sous contraintes (COP), de satisfiabilité d'une formule logique propositionnelle (SAT) ou de programmation logique avec contraintes (CLP).

Nées en 2005 de la fusion des JFPLC (Journées Francophones de la Programmation Logique avec Contraintes) et des JNPC (Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets), les JFPC devaient fêter leurs 15 ans à Nice à l'occasion des JFPC 2020. Malheureusement, la crise de la Covid nous a contraints à annuler l'édition 2020 ou plutôt à la reporter d'un an, en conservant le même comité d'organisation et le même comité de programme. Cette année supplémentaire a été remplie d'espoirs et de beaucoup d'efforts pour essayer d'organiser une session en présentiel, les rencontres entre membres de notre communauté étant une des raisons d'être des JFPC. Une fois encore, il a finalement fallu renoncer à organiser les JFPC 2021 en présentiel. Pour ne pas pénaliser les doctorants une année de plus, nous nous sommes résolus à organiser cette session 2021 en distanciel, ce qui était une première pour nous tous.

Malgré le contexte, il y a eu 25 articles soumis et, comme les années précédentes, 96% ont été acceptés. Je remercie le comité de programme pour son travail de relecture de grande qualité qui a aidé les auteurs à améliorer leur article, ce qui est particulièrement important pour les doctorants qui représentent les 2/3 des présentations. La moitié des articles soumis ont déjà été publiés dans des conférences internationales (AAAI, CP, ECAI, ECML ICTAI, IJCAI) et sont présentés dans ces actes en version courte suivie de l'article original.
Deux éminents conférenciers invités présenteront leurs travaux : Thomas Schiex (INRAE Toulouse, France) abordera les réseaux de fonctions de coûts et Claude-Guy Quimper (Université de Laval, Québec) nous parlera d'ordonnancement sous contraintes.

Je tiens enfin à remercier chaleureusement le comité d'organisation – et tout particulièrement Arnaud Malapert et Marie Pelleau – qui ont successivement travaillé sur l'organisation de 3 conférences, avec persévérance, bonne humeur et beaucoup d'efficacité. Je remercie également les sponsors, l'AFPC et toutes les personnes avec qui j'ai pu échanger pour essayer d'imaginer une version en ligne des JFPC qui préserve un peu l'esprit de convivialité qui fait la force de ces journées. Au moment où je complète ces lignes, les JFPC 2021 sont terminées et notre communauté a largement participé aux sessions en ligne et, si j'en crois les nombreux témoignages reçus, des échanges passionnants ont même eu lieu dans les salons virtuels. J'espère néanmoins que cette première session « en ligne » des JFPC sera aussi la dernière et je remercie tous les participants des JFPC 2021 qui ont permis le relatif succès de cette quinzième édition.

Philippe Vismara

Président du Comité de programme des JFPC ~~2020~~ 2021

Actes JFPC 2021

Ddo, un cadre générique et performant pour l'optimisation à base de diagrammes de décision

Xavier Gillard* **Pierre Schaus** **Vianney Coppé**

UCLouvain, 1348 Louvain-la-Neuve, Belgique

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

Résumé

Cet article est un combine et résume les articles suivants : [6, 5]. Il présente *ddo*, une bibliothèque générique et performante pour résoudre des problèmes d'optimisation à l'aide de diagrammes de décision. Pour ce faire, notre bibliothèque implémente l'approche par "branchements et bornage" proposée par [3] afin de trouver la solution optimale de programmes dynamiques. Grâce à notre bibliothèque nous avons non seulement été en mesure d'implémenter des solveurs pour les problèmes MISP, MCP, MAX2SAT et TSPTW. Aussi, *ddo* tire parti des capacités de calcul en parallèle du matériel sur lequel il s'exécute sans que le développeur n'ait d'effort particulier à fournir. Il en résulte que les solveurs implémentés avec *ddo* sont hautement performants. En particulier, nos expériences montrent que *ddo* surclasse Gurobi pour MCP. Ddo et nos solveurs d'exemples sont publiés sous une licence libre et leur code source est accessible en ligne [1].

1 Introduction

Les diagrammes de décision multivalués (MDD) sont une généralisation des diagrammes de décision binaires (BDD), lesquels sont utilisés depuis longtemps, e.a. parce qu'ils permettent de réaliser le model checking de systèmes complexes[4]. Plus récemment, ces modèles graphiques ont attiré l'attention de chercheurs dans les communautés PC et RO. La popularité des diagrammes de décision (DD) vient de leur capacité à encoder de larges espaces d'états de façon très compacte. C'est pourquoi ils sont entre autre utilisés dans le cas de la contrainte `Table`[7, 8]. Cet intérêt accru pour les DD a donné naissance à l'*optimisation à base de DD* (DDO) [2]. L'objectif de cette technique est de résoudre efficacement des problèmes d'optimisation combinatoires en exploitant la structure de ceux-ci au travers de DDs. Cet article s'inscrit dans cette ligne et poursuit un double objectif : d'une part, faire connaitre la technique au plus grand nombre. Et d'autre part, faciliter l'intégration de celle-ci avec d'autres solveurs et outils grâce à une bibliothèque générique et performante.

2 Les fondements

Un problème d'optimisation discrète est avant tout un problème de satisfaction de contraintes auquel une fonction d'objectif est associée. Parmi ces problèmes, certains ont une structure de *sous-problème* optimal qui leur permet d'être formulés comme des programmes dynamiques (DP). Bien que les modèles DP soient typiquement envisagés sous l'angle de la récursion, il est aussi naturel de les considérer comme des systèmes à transitions d'états. Auquel cas, un modèle DP consiste de : (a) un espace de solutions défini par les variables du problème et leurs domaines ; (b) un état initial, (c) une valeur initiale ; (d) une fonction de transition et (e) une fonction de coût de transition.

Au coeur de DDO, on trouve l'idée selon laquelle un système de transition DP se matérialise facilement sous la forme d'un diagramme de décision (réduit). Toutefois, bien qu'ils soient compacts, la construction de ces DD peut requérir une quantité de mémoire (et de temps) exponentielle. C'est pourquoi il est impossible d'encoder exactement l'espace d'états pour des instances de problèmes réelles. Pour palier à cela, DDO utilise des DDs ayant une taille maximum bornée. Ceux-ci permettent de fournir deux types d'approximations pour le problème à résoudre. En supposant une fonction d'objectif à maximiser, les DD *restreints* et *relâchés* [1] permettent respectivement de dériver une borne inférieure et supérieure pour le problème à résoudre.

Compiler un DD restreint à partir d'une formulation DP est assez simple : il suffit de s'assurer que la lar-

*Papier doctorant : Xavier Gillard est auteur principal.

1. https ://github.com/xgillard/ddo

geur des niveaux du DD soit limitée en supprimant les nœuds les moins prometteurs de chaque niveau. Ceci supprime un certain nombre de solutions du DD, mais n'introduit jamais de non-solutions dans celui-ci. La compilation d'un DD relâché est un peu différente car elle nécessite qu'on lui fournisse une relaxation permettant la *fusionner* des noeuds surnuméraires. C'est pourquoi, lorsqu'on souhaite résoudre un problème avec DDO, il est nécessaire de donner à la fois un modèle DP et une relaxation du problème.

3 La bibliothèque *ddo*

C'est là tout ce dont notre bibliothèque *ddo* a besoin pour résoudre un problème automatiquement et efficacement : la définition de ce problème et une relaxation. Naturellement, *ddo* permet en outre de guider la résolution via des heuristiques propres au problème. Mais leur emploi n'est pas *requis*.

Nous illustrons notre propos via un exemple minimaliste mais complet. Celui-ci montre comment modéliser et résoudre un problème de sac à dos avec *ddo*. Le Listing 1, montre bien à quel point le modèle *ddo* ressemble aux abstractions mathématiques évoquées dans la section précédente. En particulier, l'implémentation du trait `Problem<usize>` par la structure `Sac` décrit la formulation DP du problème de sac à dos dont l'état consiste en un entier non signé (`usize`). L'espace de solution du problème (a) est caractérisé par les méthodes `nb_vars()` et `domain_of()` (lignes 8–15). De même les quatre autres éléments constitutifs d'un modèle DP (état init. (b), valeur init. (c), fonction de transition (d) et fonction de coût (e)) sont tous impléméntés par leurs fonctions éponymes (lignes 16–29). Aussi, la structure `SacRelax` qui implémente le trait `Relaxation<usize>` montre un exemple de fusion d'états (lines 36–45). Dans notre exemple, le nouvel état relâché est obtenu en gardant la capacité maximum des états à fusionner et il n'est pas nécessaire de modifier le poids des arcs entrants du noeud ainsi relâché.

```
1  #[derive(Clone, Debug)]
2  struct Sac {
3      capacite: usize,
4      profit: Vec<usize>,
5      poids: Vec<usize>
6  }
7  impl Problem<usize> for Sac {
8      fn nb_vars(&self) -> usize {
9          self.profit.len()
10     }
11     fn domain_of<'a>(&self,
12         state: &'a usize,
13         var: Variable) -> Domain<'a> {
14         vec![0, 1].into()
15     }
16     fn initial_state(&self) -> usize {
17         self.capacite
18     }
19     fn initial_value(&self) -> isize {
20         0
21     }
22     fn transition(&self, state: &usize,
23         vars: &VarSet, d: Decision) -> usize {
24         let var = d.variable.id();
25         state - self.poids[var] * d.value as usize
26     }
27     fn transition_cost(&self, state: &usize,
28         vars: &VarSet, d: Decision) -> isize {
29         let var = d.variable.id();
30         self.profit[var] as isize * d.value
31     }
32 }
33 #[derive(Clone, Copy)]
34 struct SacRelax;
35 impl Relaxation<usize> for SacRelax {
36     fn merge_states(&self,
37         states: &mut dyn Iterator<Item=&usize>)
38      -> usize {
39         states.copied().max().unwrap_or(0)
40     }
41     fn relax_edge(&self, src: &usize, dst: &usize,
42         relaxed: &usize, d: Decision, cost: isize)
43      -> isize {
44         cost
45     }
46 }
47 fn main() {
48     let problem = Sac {/*omis*/};
49     let config = config_builder(&problem, SacRelax)
50       .build();
51     let mdd = DeepMDD::from(config);
52     let mut solver = ParallelSolver::new(mdd);
53     solver.maximize();
54 }
```

Listing 1 – Detailed example

Enfin, les lignes 48–53 du Listing 1 montrent comment instancier le solveur et l'utiliser pour résoudre une instance du problème de sac à dos binaire en utilisant tous les fils d'exécution matériels de la machine.

4 Résultats expérimentaux et conclusion

Nous voudrions clôturer notre présentation de *ddo* en pointant certains résultats expérimentaux très encourageants. En effet, celui-ci montre que malgré que *ddo* soit entièrement générique il parvient à être extrêmement performant. En l'occurrence, la Figure 1 montre que lors de nos expériences, notre solveur a su résoudre les 265 instances de MCP testées en un peu moins de 800 secondes sur 24 fils d'exécution alors que dans les mêmes conditions, Gurobi 9.0.2 n'est pas parvenu à toutes les résoudre en 30 minutes.

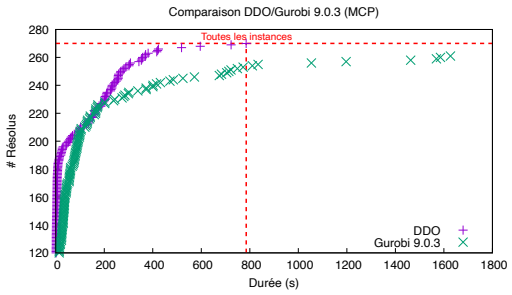


Figure 1 – Comparaison *ddo*/Gurobi

Références

[1] H. R. Andersen, T. Hadzic, J. N. Hooker et P. Tiedemann : A constraint store based on multivalued decision diagrams. *In* Christian Bessière, éditeur : *Principles and Practice of Constraint Programming*, volume 4741 de *LNCS*, pages 118–132. Springer, 2007.

[2] David Bergman et Andre A. Cire : Theoretical insights and algorithmic tools for decision diagram-based optimization. *Constraints*, 21(4):533–556, 2016.

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

[4] J.R. Burch, Clarke E.M., McMillan K.L., Dill D.L. et Hwang H.L. : Symbolic model checking : 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[5] Xavier Gillard, Pierre Schaus et André Ciré Coppé, Vianney : Improving the filtering of branch-and-bound mdd solver. 2021.

[6] Xavier Gillard, Pierre Schaus et Vianney Coppé : Ddo, a generic and efficient framework for mdd-based optimization. 2020.

[7] Guillaume Perez et Jean-Charles Régin : Efficient operations on mdds for building constraint programming models. *In Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-15)*, pages 374–380, 2015.

[8] Hélène Verhaeghe, Christophe Lecoutre et Pierre Schaus : Compact-mdd : Efficiently filtering (s) mdd constraints with reversible sparse bit-sets. *In Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 1383–1389, 2018.

Ddo, a Generic and Efficient Framework for MDD-Based Optimization

Xavier Gillard [*], **Pierre Schaus** and **Vianney Coppé**
UCLouvain
{xavier.gillard, pierre.schaus, vianney.coppe}@uclouvain.be

Abstract

This paper presents *ddo*, a generic and efficient library to solve constraint optimization problems with decision diagrams. To that end, our framework implements the branch-and-bound approach which has recently been introduced by [Bergman *et al.*, 2016b] to solve dynamic programs to optimality. Our library allowed us to successfully reproduce the results of Bergman et al. for MISP, MCP and MAX2SAT while using a single generic library. As an additional benefit, *ddo* is able to exploit parallel computing for its purpose without imposing any constraint on the user (apart from memory safety). *Ddo* is released as an open source[1] rust library (crate) alongside with its companion example programs to solve the aforementioned problems. To the best of our knowledge, this is the first public implementation of a generic library to solve combinatorial optimization problems with branch-and-bound MDD.

1 Introduction

Multivaluated Decision Diagrams (MDD) are a generalization of *Binary Decision Diagrams* (BDD) which have long been used in the verification community, e.g. for model checking purposes [Burch *et al.*, 1992]. More recently, these graphical models have drawn the attention of researchers from the CP and OR communities. The popularity of these decision diagrams (DD) stems from their ability to provide a compact representation of large solution spaces as in the case of the table constraint [Perez and Régin, 2015; Verhaeghe *et al.*, 2018]. One of the research streams which emerged from this increased interest about MDDs is *decision-diagram-based optimization* (DDO) [Bergman and Cire, 2016]. Its purpose is to efficiently solve combinatorial optimization problems by exploiting the structure of the problem being solved through the use of DDs. So far, the techniques developed in this context have largely been successful and outperforms state-of-the-art IP solvers for the problems where they are applicable. This paper belongs to the DDO subfield and intends to broaden the DDO-awareness and facilitate its integration with other solvers and techniques through the release of a generic and efficient open-source rust library implementing these algorithms and data structures.

[*]Contact Author

[1]https://github.com/xgillard/ddo

2 Background

A discrete optimization problem is first and foremost a constraint *satisfaction* problem with an associated objective function to be maximized. Among these problems, some exhibit an *optimal subproblem structure* making them suitable for a dynamic programming (DP) formulation. Even though DP models are typically thought of in terms of recursion, it is also natural to consider them as transition systems. In that case, a DP model consists of: (a) a solution space defined by the problem variables and their domains; (b) an initial state, (c) an initial value; (d) a transition function and (e) a transition cost function.

At the heart of DDO, is the idea that DP transition systems naturally lend themselves to materialization in the form of a (reduced) decision diagram. However, despite their compactness, the construction of DD suffers from a potentially exponential time and memory requirements. Using DDs to exactly encode the solution space of a problem is thus out of reach for any practical problem instance. This is why, DDO relies on the use of bounded-size DDs to approximate a solution of the actual problem. Two types of approximate bounded-size DDs have been devised for that purpose: *relaxed* and *restricted* DDs. These respectively encode an over- and under-approximation of the solution-space. Assuming a maximization problem, relaxed DDs [Andersen *et al.*, 2007] are thus capable of providing an upper bound on the optimal solution. Conversely, restricted DDs yield good lower bounds, as they contain a subset of the feasible solutions of the problem.

Deriving a restricted MDD from the DP formulation of a problem is quite simple. For that purpose, it suffices to limit the width of the MDD layers by simply dropping the less promising nodes of that layer. This process only removes solutions from the set of solutions represented by the MDD but it does not create any infeasible solution. Deriving a relaxed MDD from the same DP formulation is a different matter though. For that purpose, one needs to provide a relaxation to *merge nodes* that exceed the maximum width bound. For that reason, anyone willing to use DDO to solve a new kind of problem must provide both a DP formulation and a suit-

able relaxation for the problem; and in an ideal world, these would be the only two required inputs.

3 The *ddo* Library

This is exactly what our *ddo* framework aims to do: it starts from the definition of a problem and its relaxation to automatically and efficiently solve the problem to optimality. Furthermore, it allows a user to specify and use custom heuristics. But these are not mandatory, and the framework readily provides default heuristics.

We illustrate our point going through a minimalistic yet extensive example. Which one shows how to model and solve the binary knapsack problem with *ddo*. From Listing 1, one can observe how closely the *ddo* model matches with the mathematical abstractions outlined in the previous section. In particular, the implementation of the `Problem<usize>` trait by `Knapsack` describes the DP formulation of a binary knapsack problem whose state consists of a single unsigned integer (`usize`). The solution space (a) of the problem is characterized by the methods `nb_vars()` and `domain_of()` (lines 9–16). Similarly, the other four elements constitutive of a DP model (initial state (b), initial value (c), transition function (d) and transition cost function (e)) are all implemented by their eponymous method (lines 16–32). Also, `KPRelax` implementing the trait `Relaxation<usize>` shows what it takes to merge several nodes so as to derive a new relaxed node standing for them all (lines 38–49). In our example, the relaxed state of the new node is obtained by taking the maximum remaining capacity available in any of the merged nodes. The arcs towards the new relaxed node are obtained by (approximately) considering that the longest path to any of the merged nodes yields the relaxed node.

```
1  /// Lines 1-33 describe the problem DP formulation
2  #[derive(Debug, Clone)]
3  struct Knapsack {
4    capacity: usize,
5    profit  : Vec<usize>,
6    weight  : Vec<usize>
7  }
8  impl Problem<usize> for Knapsack {
9    fn nb_vars(&self) -> usize {
10     self.profit.len()
11   }
12   fn domain_of<'a>(&self,state: &'a usize,
13                    var  : Variable)
14   ->Domain<'a> {
15     vec![0, 1].into()
16   }
17   fn initial_state(&self) -> usize {
18     self.capacity
19   }
20   fn initial_value(&self) -> i32 {
21     0
22   }
23   fn transition(&self,state:&usize,
24                 vars :&VarSet,
25                 dec  :Decision) -> usize {
26     state - self.weight[dec.variable.id()]
27   }
28   fn transition_cost(&self,state:&usize,
29                      vars :&VarSet,
30                      dec  :Decision) -> i32 {
31     self.profit[dec.variable.id()] as i32 * dec.value
32   }
33 }
34 /// Lines 34-50 implement the problem relaxation
35 #[derive(Debug, Clone)]
36 struct KPRelax;
37 impl Relaxation<usize> for KPRelax {
38   fn merge_nodes(&self, nodes: &[Node<usize>])
39   -> Node<usize> {
40     let lp_info = nodes.iter()
41                        .map(|n| &n.info)
42                        .max_by_key(|i| i.lp_len);
43     let max_capa= nodes.iter()
44                        .map(|n| n.state)
45                        .max();
46     Node::merged(max_capa,
47                  lp_info.lp_len,
48                  lp_info.lp_arc.clone())
49   }
50 }
51 fn main() {
52   let problem = Knapsack {/* elided */};
53   let mdd = mdd_builder(&problem, KPRelax).build();
54   let mut solver = ParallelSolver::new(mdd);
55   let (optimal, solution) = solver.maximize();
56 }
```

Listing 1: Detailed example

Finally, the last fragment (lines 51–56) of Listing 1 show what it takes to instantiate the solver and use it to solve a knapsack problem instance with *ddo* using all the hardware threads available on the machine.

4 Experimental Results

To conclude our brief presentation of *ddo*, we would like to showcase some experimental results (Table 1). These figures measure the time it took (in seconds) to solve a subset of the well known MISP/Max-Clique instances from the DIMACS challenge. These measurements have been taken on a machine equipped with two Intel E5-2640v3 CPU (2.60GHz, 8 cores, 2 threads/core for a total of 32 available hardware threads) and 128G of RAM. The timeout for each run was set to 600 seconds and we set a maximum width of 100 nodes per layer of our restricted and relaxed MDDs.

These results are very promising as they indicate that even though our library is truly generic, it delivers an overall performance on par with that of DDX10[Bergman *et al.*, 2014; Bergman *et al.*, 2016a]. The latter having been favorably compared by its authors to IBM ILOG CPLEX 12.5.1.

Instance	1 thread	16 threads	32 threads
hamming8-4.clq	25.45	2.58	2.17
brock200_4.clq	18.65	1.78	1.56
san400_0.7_1.clq	48.67	4.98	4.35
p_hat300-2.clq	14.98	1.88	1.64
san1000.clq	124.46	23.18	21.78
p_hat1000-1.clq	73.98	20.07	19.58
sanr400_0.5.clq	74.07	6.80	6.21
san200_0.9_2.clq	64.94	3.13	2.62
sanr200_0.7.clq	69.67	5.81	4.91
san400_0.7_2.clq	250.07	19.74	15.94
p_hat1500-1.clq	timeout	89.28	88.40
brock200_1.clq	316.30	25.64	21.01

Table 1: Runtime (seconds) to solve MISP/Max-Clique instances from the DIMACS challenge. Timeout 600 seconds.

5 Demonstration

This demonstration will focus on how a practician can use our library to solve combinatorial optimization problems. Starting from the above knapsack example, we will show how one can tune the behavior of the solver to make the most of the available resources and problem knowledge. In particular, we will show how to opt for a static vs dynamic maximum layer width; how to opt for a single vs multi-threaded resolution and how to specify a custom variable selection heuristic in replacement of the default (natural-order) one.

References

[Andersen *et al.*, 2007] Henrik Reif Andersen, Tarik Hadzic, John Hooker, and Peter Tiedemann. A constraint store based on multivalued decision diagrams. In Christian Bessière, editor, *Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 118–132. Springer, 2007.

[Bergman and Cire, 2016] David Bergman and Andre Cire. Theoretical insights and algorithmic tools for decision diagram-based optimization. *Constraints*, 21(4):533–556, 2016.

[Bergman *et al.*, 2014] David Bergman, Andre Cire, Ashish Sabharwal, Samulowitz Horst, Saraswat Vijay, and Willem-Jan and van Hoeve. Parallel combinatorial optimization with decision diagrams. In Helmut Simonis, editor, *Integration of AI and OR Techniques in Constraint Programming*, volume 8451, pages 351–367. Springer, 2014.

[Bergman *et al.*, 2016a] David Bergman, Andre Cire, Willem-Jan van Hoeve, and John Hooker. *Decision Diagrams for Optimization*. Springer, 2016.

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

[Burch *et al.*, 1992] Jerry Burch, Clarke Edmund, McMillan Kenneth, Dill David, and Hwang H.L. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[Perez and Régin, 2015] Guillaume Perez and Jean-Charles Régin. Efficient operations on mdds for building constraint programming models. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-15)*, pages 374–380, 2015.

[Verhaeghe *et al.*, 2018] Hélène Verhaeghe, Christophe Lecoutre, and Pierre Schaus. Compact-mdd: Efficiently filtering (s) mdd constraints with reversible sparse bit-sets. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)*, pages 1383–1389, 2018.

Improving the filtering of Branch-And-Bound MDD solver

Xavier Gillard[1 [0000−0002−4493−6041]], Vianney Coppé[1 [0000−0001−5050−0001]], Pierre Schaus[1 [0000−0002−3153−8941]], and André Augusto Cire[2 [0000−0001−5993−4295]]

Université Catholique de Louvain, BELGIUM
University of Toronto Scarborough and Rotman School of Management, CANADA
{xavier.gillard, pierre.schaus, vianney.coppe}@uclouvain.be,
andre.cire@rotman.utoronto.ca

Abstract. This paper presents and evaluates two pruning techniques to reinforce the efficiency of constraint optimization solvers based on multi-valued decision-diagrams (MDD). It adopts the branch-and-bound framework proposed by Bergman et al. in 2016 to solve dynamic programs to optimality. In particular, our paper presents and evaluates the effectiveness of the local-bound (LocB) and rough upper-bound pruning (RUB). LocB is a new and effective rule that leverages the approximate MDD structure to avoid the exploration of non-interesting nodes. RUB is a rule to reduce the search space during the development of bounded-width-MDDs. The experimental study we conducted on the Maximum Independent Set Problem (MISP), Maximum Cut Problem (MCP), Maximum 2 Satisfiability (MAX2SAT) and the Traveling Salesman Problem with Time Windows (TSPTW) shows evidence indicating that rough-upper-bound and local-bound pruning have a high impact on optimization solvers based on branch-and-bound with MDDs. In particular, it shows that RUB delivers excellent results but requires some effort when defining the model. Also, it shows that LocB provides a significant improvement automatically; without necessitating any user-supplied information. Finally, it also shows that rough-upper-bound and local-bound pruning are not mutually exclusive, and their combined benefit supersedes the individual benefit of using each technique.

Introduction

Multi-valued Decision Diagrams (MDD) are a generalization of *Binary Decision Diagrams* (BDD) which have long been used in the verification, e.g., for model checking purposes [10]. Recently, these graphical models have drawn the attention of researchers from the CP and OR communities. One of the research streams which emerged from this increased interest about MDDs is *decision-diagram-based optimization* (DDO) [5]. Its purpose is to efficiently solve combinatorial optimization problems by exploiting problem structure through DDs. This paper belongs to the DDO sub-field and intends to further improve the

efficiency of DDO solvers through the introduction of two bounding techniques: local-bounds pruning (LocB) and rough-upper-bound pruning (RUB).

This paper starts by covering the necessary background on DDO. Then, it presents the local-bound and rough-upper-bound pruning techniques in Sections 2.1 and 2.2. After that, it presents an experimental study which we conducted using 'ddo' [17][1], our open source fast and generic MDD-based optimization library. This experimental study investigates the relevance of RUB and LocB through four disinct NP-hard problems: the Weighted Maximum Independent Set Problem (MISP), Maximum Cut Problem (MCP), Maximum 2 Satisfiability Problem (MAX2SAT) and the Traveling Salesman Problem with Time Windows (TSPTW). Finally, section 4 discusses previous related work before drawing conclusions.

1 Background

The coming paragraphs give an overview of discrete optimization with decision diagrams. Most of the formalism presented here originates from [8]. Still, we reproduce it here for the sake of self-containedness.

Discrete optimization. A discrete optimization problem is a constraint *satisfaction* problem with an associated objective function to be maximized. The discrete optimization problem $\mathcal{P}$ is defined as $\max\{f(x) \mid x \in D \wedge C(x)\}$ where C is a set of constraints, $x = \langle x_0, \ldots, x_{n-1}\rangle$ is an assignment of values to variables, each of which has an associated finite domain D_i s.t. $D = D_0 \times \cdots \times D_{n-1}$ from where the values are drawn. In that setup, the function $f : D \to \mathbb{R}$ is the objective to be maximized.

Among the set of feasible solutions $Sol(\mathcal{P}) \subseteq D$ (i.e. satisfying all constraints in C), we denote the optimal solution by x^*. That is, $x^* \in Sol(\mathcal{P})$ and $\forall x \in Sol(\mathcal{P}) : f(x^*) \geq f(x)$.

Dynamic programming. Dynamic programming (DP) was introduced in the mid 50's by Bellman [3]. This strategy is significantly popular and is at the heart of many classical algorithms (e.g., Dijkstra's algorithm [12, p.658] or Bellman-Ford's [12, p.651]).

Even though a dynamic program is often thought of in terms of recursion, it is also natural to consider it as a labeled transition system. In that case, the *DP model* of a given discrete optimization problem $\mathcal{P}$ consists of:

- a set of state-spaces $S_0, \ldots, S_n$ among which one distinguishes the *initial state* r, the *terminal state* t and the *infeasible state* $\perp$.
- a set of transition functions $t_i : S_i \times D_i \to S_{i+1}$ for $i = 0, \ldots, n-1$ taking the system from one state s^i to the next state s^{i+1} based on the value d assigned to variable x_i (or to $\perp$ if assigning $x_i = d$ is infeasible). These functions should never allow one to recover from infeasibility ($t_i(\perp, d) = \perp$ for any $d \in D_i$).

[1] https://github.com/xgillard/ddo

– a set of transition cost functions $h_i : S_i \times D_i \to \mathbb{R}$ representing the immediate reward of assigning some value $d \in D_i$ to the variable x_i for $i = 0, \ldots, n-1$.
– an initial value v_r.

On that basis, the objective function $f(x)$ of $\mathcal{P}$ can be formulated as follows:

$$\text{maximize } f(x) = v_r + \sum_{i=0}^{n-1} h_i(s^i, x_i)$$
$$\text{subject to}$$
$$s^{i+1} = t_i(s^i, x_i) \text{ for } i = 0, \ldots, n-1; x_i \in D_i \wedge C(x_i)$$
$$s^i \in S_i \text{ for } i = 0, \ldots, n$$

where $C(x_i)$ is a predicate that evaluates to *true* when the partial assignment $\langle x_0, \ldots, x_i \rangle$ does not violate any constraint in C.

The appeal of such a formulation stems from its simplicity and its expressiveness which allows it to effectively capture the problem structure. Moreover, this formulation naturally lends itself to a DD representation; in which case it represents an exact DD encoding the complete set $Sol(\mathcal{P})$.

1.1 Decision diagrams

Because DDO aims at solving constraint *optimization* problems and not just constraint *satisfaction* problems, it uses a particular DD flavor known as reduced weighted DD – DD as of now. As initially posed by Hooker[21], DDs can be perceived as a compact representation of the search trees. This is achieved, in this context, by superimposing isomorphic subtrees.

To define our DD more formally, we will slightly adapt the notation from [5]. A DD $\mathcal{B}$ is a layered directed acyclic graph $\mathcal{B} = \langle n, U, A, l, d, v, \sigma \rangle$ where n is the number of variables from the encoded problem, U is a set of nodes; each of which is associated to some state $\sigma(u)$. The mapping $l : U \to \{0 \ldots n\}$ partitions the nodes from U in disjoint layers $L_0 \ldots L_n$ s.t. $L_i = \{u \in U : l(u) = i\}$ and the states of all the nodes belonging to the same layer pertain to the same DP-state-space ($\forall u \in L_i : \sigma(u) \in S_i$ for $i = 0, \ldots, n$). Also, it should be the case that no two distinct nodes of one same layer have the same state ($\forall u_1, u_2 \in L_i : u_1 \neq u_2 \implies \sigma(u_1) \neq \sigma(u_2)$, for $i = 0, \ldots, n$).

The set $A \subseteq U \times U$ from our formal model is a set of directed arcs connecting the nodes from U. Each such arc $a = (u_1, u_2)$ connects nodes from subsequent layers ($l(u_1) = l(u_2) - 1$) and should be regarded as the materialization of a branching decision about variable $x_{l(u_1)}$. This is why all arcs are annotated via the mappings $d : A \to D$ and $v : A \to \mathbb{R}$ which respectively associate a decision and value (weight) with the given arc.

Example 1. An arc a connecting nodes $u_1 \in L_3$ to $u_2 \in L_4$, annotated with $d(a) = 6$ and $v(a) = 42$ should be understood as the assignment $x_3 = 6$ performed from state $\sigma(u_1)$. It should also be understood that $t_3(\sigma(u_1), 6) = \sigma(u_2)$ and the benefit of that assignment is $v(a) = h_3(\sigma(u_1), 6) = 42$.

Because each r-t path describes an assignment that satisfies $\mathcal{P}$, we will use $Sol(\mathcal{B})$ to denote the set of all the solutions encoded in the r-t paths of DD $\mathcal{B}$. Also, because unsatisfiability is irrecoverable, r-$\bot$ paths are typically omitted from DDs. It follows that a nice property from using a DD representation $\mathcal{B}$ for the DP formulation of a problem $\mathcal{P}$, is that finding x^* is as simple as finding the longest r-t path in $\mathcal{B}$ (according to the relation v on arcs).

Exact-MDD. For a given problem $\mathcal{P}$, an exact MDD $\mathcal{B}$ is an MDD that exactly encodes the solution set $Sol(\mathcal{B}) = Sol(\mathcal{P})$ of the problem $\mathcal{P}$. In other words, not only do all r-t paths encode valid solutions of $\mathcal{P}$, but no feasible solution is present in $Sol(\mathcal{P})$ and not in $\mathcal{B}$. An exact MDD for $\mathcal{P}$ can be compiled in a top-down fashion[2]. This naturally follows from the above definition. To that end, one simply proceeds by a repeated unrolling of the transition relations until all variables are assigned.

1.2 Bounded-Size Approximations

In spite of the compactness of their encoding, the construction of DD suffers from a potentially exponential memory requirement in the worst case[3]. Thus, using DDs to exactly encode the solution space of a problem is often intractable. Therefore, one must resort to the use of *bounded-size* approximation of the exact MDD. These are compiled generically by inserting a call to a width-bounding procedure to ensure that the width (the number $|L_i|$ of distinct nodes belonging to the L_i) of the current layer L_i does not exceed a given bound W. Depending on the behavior of that procedure, one can either compile a restricted-MDD (= an under-approximation) or a relaxed-MDD (= an over-approximation).

Restricted-MDD: Under-approximation. A restricted-MDD provides an under-approximation of some exact-MDD. As such, all paths of a restricted-MDD encode valid solutions, but some solutions might be missing from the MDD. This is formally expressed as follows: given the DP formulation of a problem $\mathcal{P}$, $\mathcal{B}$ is a restricted-MDD iff $Sol(\mathcal{B}) \subseteq Sol(\mathcal{P})$.

To compile a restricted-MDD, it is sufficient to simply delete certain nodes from the current layer until its width fits within the specified bound W. To that end, the width-bounding procedure simply selects a subset of the nodes from L_i which are heuristically assumed to have the less impact on the tightness of the bound. Various heuristics have been studied in the literature [7], and *minLP* was shown to be the heuristic that works best in practice. This heuristic decides to select (hence remove) the nodes having the shortest longest path from the root.

[2] An incremental refinement *a.k.a. construction by separation* procedure is detailed in [11, pp. 51–52] but we will not cover it here for the sake of conciseness.

[3] Consequently, it also suffers from a potentially exponential time requirement in the worst case. Indeed, time is constant in the final number of nodes (unless the transition functions themselves are exponential in the input).

Relaxed-MDD: Over-approximation. A relaxed-MDD $\mathcal{B}$ provides a bounded-width over-approximation of some exact-MDD. As such, it may hold paths that are no solution to $\mathcal{P}$, the problem being solved. We have thus formally that $Sol(\mathcal{B}) \supseteq Sol(\mathcal{P})$.

Compiling a relaxed-MDD requires one to be able to *merge* several nodes into an inexact one. To that end, we use two operators:

- $\oplus$ which yields a new node combining the states of a selection of nodes so as to over-approximate the states reachable in the selection.
- Γ which is used to possibly relax the weight of arcs incident to the selected nodes.

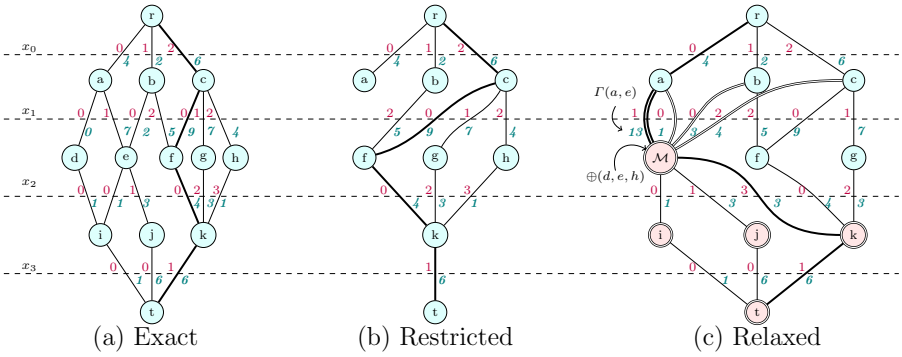
These operators are used as follows. Similar to the restricted-MDDs case, the width-bounding procedure starts by heuristically selecting the least promising nodes and removing them from layer L_i. Then the states of these selected nodes are combined with one another so as to create a merged node $\mathcal{M} = \oplus(selection)$. After that, the inbound arcs incident to all selected nodes are Γ-relaxed and redirected towards $\mathcal{M}$. Finally, the result of the merger ($\mathcal{M}$) is added to the layer in place of the initial selection of nodes.

Summary. Fig. 1 summarizes the information from sections 1.1 and 1.2. It displays the three MDDs corresponding to one same example problem having four variables. The exact MDD (a) encodes the complete solution set and, equivalently, the state space of the underlying DP encoding. One easily notices that the restricted DD (b) is an under approximation of (a) since it achieves its width boundedness by removing nodes d and e and their children (i, j). Among others, it follows that the solution $[x_0 = 0, x_1 = 0, x_2 = 0, x_3 = 0]$ is not represented in (b) even though it exists in (a). Conversely, the relaxed diagram (c) achieves a maximum layer with of 3 by merging nodes d, e and h into a new inexact node $\mathcal{M}$ and by relaxing all arcs entering one of the merged nodes. Because of this, (c) introduces solutions that do not exist in (a) as is for instance the case of the assignment $[x_0 = 0, x_1 = 0, x_2 = 3, x_3 = 1]$. Moreover, because the operators $\oplus$ and Γ are correct[4], the length of the longest path in (c) is an upper bound on the optimal value of the objective function. Indeed, one can see that the length of the longest path in (a) (= the exact optimal solution) has a value of 25 while it amounts to 26 in (c).

1.3 The Dynamics of Branch-and-Bound with DDs

Being able to derive good lower and upper bounds for some optimization problem $\mathcal{P}$ is useful when the goal is to use these bounds to strengthen algorithms [13, 31, 32]. But it is not the only way these approximations can be used. A complete and efficient branch-and-bound algorithm relying on those approximations was proposed in [8] which we hereby reproduce (Alg. 1).

[4] The very definition of these operators is problem-specific. However, [22] formally defines the conditions that are necessary to correctness.



(a) Exact (b) Restricted (c) Relaxed

Fig. 1. The exact (a), restricted (b) and relaxed (c) versions of an MDD with four variables. The width of MDDs (b) and (c) have been bounded to a maximum layer width of three. The decision labels of the arcs are shown above the layers separation lines (dashed). The arc weights are shown below the layer separation lines. The longest path of each MDD is boldfaced. In (c), the node $\mathcal{M}$ is the result of merging nodes d, e and h with the $\oplus$ operator. Arcs that have been relaxed with the Γ operator are pictured with a double stroke. Note, because these arcs have been Γ-relaxed, their value might be greater than that of corresponding arcs in (a), (b). Similarly, all "inexact" nodes feature a double border.

This algorithm works as follows: at start, the node r is created for the initial state of the problem and placed onto the *fringe* – a global priority queue that tracks all nodes remaining to explore and orders them from the most to least promising. Then, a loop consumes the nodes from that fringe (line 1), one at a time and explores it until the complete state space has been exhausted. The *exploration* of a node u inside that loop proceeds as follows: first, one compiles a restricted DD $\underline{\mathcal{B}}$ for the sub-problem rooted in u (line 5). Because all paths in a restricted DD are feasible solutions, when the lower bound $v^*(\underline{\mathcal{B}})$ derived from the restricted DD $\underline{\mathcal{B}}$ improves over the current best known solution $\underline{v}$; then the longest path of $\underline{\mathcal{B}}$ (best sol. found in $\underline{\mathcal{B}}$) and its length $v^*(\underline{\mathcal{B}})$ are memorized (lines 7-9).

In the event where $\underline{\mathcal{B}}$ is exact (no restriction occurred during the compilation of $\underline{\mathcal{B}}$), it covers the complete state space of the sub-problem rooted in u. Which means the processing of u is complete and we may safely move to the next node. When this condition is not met, however, some additional effort is required.In that case, a *relaxed* DD $\overline{\mathcal{B}}$ is compiled from u(line 11). That relaxed DD serves two purposes: first, it is used to derive an upper bound $v^*(\overline{\mathcal{B}})$ which is compared to the current best known solution (line 12). This gives us a chance to prune the unexplored state space under u when $v^*(\overline{\mathcal{B}})$ guarantees it does not contain any better solution than the current best. The second use of $\overline{\mathcal{B}}$ happens when $v^*(\overline{\mathcal{B}})$ cannot provide such a guarantee. In that case, the exact cutset of $\overline{\mathcal{B}}$ is used to enumerate residual sub-problems which are enqueued onto the fringe (lines 13-14).

A cutset for some relaxed DD $\overline{\mathcal{B}}$ is a subset $\mathcal{C}$ of the nodes from $\overline{\mathcal{B}}$ such that any $r-t$ path of $\overline{\mathcal{B}}$ goes through at least one node $\in \mathcal{C}$. Also, a node u is said to be exact iff all its incoming paths lead to the same state $\sigma(u)$. From there, an exact cutset of $\overline{\mathcal{B}}$ is simply a cutset whose nodes are all exact. Based on this definition, it is easy to convince oneself that an exact cutset constitutes a frontier up to which the relaxed DD $\overline{\mathcal{B}}$ and its exact counterpart $\mathcal{B}$ have not diverged. And, because it is a cutset, the nodes composing that frontier cover all paths from both $\mathcal{B}$ and $\overline{\mathcal{B}}$; which guarantees the completeness of Alg. 1 [8].

Any relaxed-MDD admits at least one exact cutset – e.g. the trivial $\{r\}$ case. Often though, it is not unique and different options exist as to what cutset to use. It was experimentally shown by [8] that most of the time, the Last Exact Layer (LEL) is superior to all other exact cutsets in practice. LEL consists of the *deepest* layer of the relaxed-MDD having all its nodes exact.

Example 2. In Fig.-1 (c), the first inexact node $\mathcal{M}$ occurs in layer L_2. Hence, the LEL cutset comprises all nodes (a, b, c) from the layer L_1. Because $\mathcal{M}$ is inexact, and because it is a parent of nodes i, j and k, these three nodes are considered inexact too.

Algorithm 1 Branch-And-Bound with DD

```
1: Create node r and add it to Fringe
2: x ← ⊥
3: v ← −∞
4: while Fringe is not empty do
5:    u ← Fringe.pop()
6:    B ← Restricted(u)
7:    if v*(B) > v then
8:       v ← v*(B)
9:       x ← x*(B)
10:   if B is not exact then
11:      B̄ ← Relaxed(u)
12:      if v*(B̄) > v then
13:         for all u' ∈ B̄.exact_cutset() do
14:            Fringe.add(u')
15: return (x, v)
```

Algorithm 2 Local bound pruning

```
1: Create node r and add it to Fringe
2: x ← ⊥
3: v ← −∞
4: while Fringe is not empty do
5:    u ← Fringe.pop()
6:    if v|*_u ≤ v then
7:       continue
8:    B ← Restricted(u)
9:    if v*(B) > v then
10:      v ← v*(B)
11:      x ← x*(B)
12:   if B is not exact then
13:      B̄ ← Relaxed(u)
14:      if v*(B̄) > v then
15:         for all u' ∈ B̄.exact_cutset() do
16:            if v|*_u' > v then
17:               Fringe.add(u')
18: return (x, v)
```

2 Improving the filtering of branch-and-bound MDD

In the forthcoming paragraphs, we introduce the local bound and present the rough upper bound: two reasoning techniques to reinforce the pruning strength of Alg. 1.

2.1 Local bounds (LocB)

Conceptually, pruning with local bounds is rather simple: a relaxed MDD $\overline{\mathcal{B}}$ provides us with *one* upper bound $v^*(\overline{\mathcal{B}})$ on the optimal value of the objective function for some given sub-problem. However, in the event where $v^*(\overline{\mathcal{B}})$

is greater than the best known lower bound $\underline{v}$ (best current solution) nothing guarantees that all nodes from the exact cutset of $\overline{\mathcal{B}}$ admit a longest path to t with a length of $v^*(\overline{\mathcal{B}})$. Actually, this is quite unlikely. This is why we propose to attach a *"local" upper bound* to each node of the cutset. This local upper bound – denoted $v|^*_u$ for some cutset node u – simply records the length of the longest r-t path passing through u in the relaxed MDD $\overline{\mathcal{B}}$.

In other words, LocB allows us to refine the information provided by a relaxed DD $\overline{\mathcal{B}}$. On one hand, $\overline{\mathcal{B}}$ provides us with $v^*(\overline{\mathcal{B}})$ which is the length of the longest r-t path in $\overline{\mathcal{B}}$. As such, it provides an upper bound on the optimal value that can be reached from the root node of $\overline{\mathcal{B}}$. With the addition of LocB, the relaxed DD provides us with an additional piece of information. For each individual node u in the exact cutset of $\overline{\mathcal{B}}$, it defines the value $v|^*_u$ which is an upper bound on the value attainable from that node.

As shown in Alg. 2, the value $v|^*_u$ can prove useful at two different moments. First, in the event where $v|^*_u \leq \underline{v}$, this value can serve as a justification to not enqueue the subproblem u (line 16) since exhausting this subproblem will yield no better solution than $\underline{v}$. More formally, by definition of a cutset and of LocB, it must be the case that the longest r-t path of $\overline{\mathcal{B}}$ traverses one of the cutset nodes u and thus that $v^*(\overline{\mathcal{B}}) = v|^*_u$ (where $v|^*_u$ is the local bound of u). Hence we have: $\exists u \in$ cutset of $\overline{\mathcal{B}} : v^*(\overline{\mathcal{B}}) = v|^*_u$. However, because $v^*(\overline{\mathcal{B}})$ is the length of the *longest* r-t path of $\overline{\mathcal{B}}$, there may exist cutset nodes that only belong to r-t paths shorter than $v^*(\overline{\mathcal{B}})$. That is: $\forall u' \in$ cutset of $\overline{\mathcal{B}} : v^*(\overline{\mathcal{B}}) \geq v|^*_{u'}$. Which is why $v|^*_{u'}$ can be stricter than $v^*(\overline{\mathcal{B}})$ and hence let LocB be stronger at pruning nodes from the frontier.

The second time when $v|^*_u$ might come in handy occurs when the node u is popped out of the fringe (line 6). Indeed, because the fringe is a global priority queue, any node that has been pushed on the fringe can remain there for a long period of time. Thus, chances are that the value $\underline{v}$ has increased between the moment when the node was pushed onto the fringe (line 17) and the moment when it is popped out of it. Hence, this gives us an additional chance to completely skip the exploration of the sub-problem rooted in u.

Let us illustrate that with the relaxed MDD shown on Fig.2, for which the exact cutset comprises the highlighted nodes a and b. Please note that because this scenario may occur at any time during the problem resolution, we will assume that the fringe is not empty when it starts. Assuming that the current best solution $\underline{v}$ is 20 when one explores the pictured subproblem, we are certain that exploring the subproblem rooted in a is a waste of time, because the local bound $v|^*_a$ is only 16. Also, because the fringe was not empty, it might be the case that b was left on the fringe for a long period of time. And because of this, it might be the case that the best known value $\underline{v}$ was improved between the moment when b was pushed on the fringe and the moment when it was popped out of it. Assuming that $\underline{v}$ has improved to 110 when b is popped out of the fringe, it may safely be skipped because $v|^*_b$ guarantees that an exploration of b will not yield a better solution than 102.

Alg. 3 describes the procedure to compute the local bound $v|^*_u$ of each node u belonging to the exact cutset of a relaxed MDD $\overline{\mathcal{B}}$. Intuitively, this is achieved by doing a bottom-up traversal of $\overline{\mathcal{B}}$, starting at t and stopping when the traversal crosses the last exact layer (line 5). During that bottom-up traversal, the algorithm marks the nodes that are reachable from t. This way, it can avoid the traversal of dead-end nodes. Also, Alg. 3 maintains a value $v^*_{\uparrow_t}(u)$ for each node u it encounters. This value represents the length of the longest u-t path. Afterwards (line 13), it is summed with the length of the longest r-u path v^*_{r-u} to derive the exact value of the local bound $v|^*_u$.

Algorithm 3 Computing the local bounds

```
1: lel ← Index of the last exact layer
2: v*↑t(u) ← −∞ for each node u ∈ B̄                    // init. longest u-t path
3: mark(t) ←true
4: v*↑t(t) ← 0                                          // longest t-t path
5: for all i = n to lel do
6:    for all node u ∈ L_i do
7:       if mark(u) then
8:          for all arc a = (u', u) incident to u do
9:             mark(u') ← true
10:            v*↑t(u') ← max(v*↑t(u'), v*↑t(u) + v(a))      // longest u'-t path
11: for all node u ∈ B̄.exact_cutset() do
12:    if mark(u) then
13:       v|*u ← v*r−u + v*↑t(u)          // longest r-u path + longest u-t path
14:    else
15:       v|*u ← −∞
```

2.2 Rough upper bound (RUB)

Rough upper bound pruning departs from the following observation: assuming the knowledge of a lower bound $\underline{v}$ on the value of v^*, and assuming that one is able to swiftly compute a rough upper bound $\overline{v_s}$ on the optimal value v^*_s of the subproblem rooted in state s; any node u of a MDD having a rough upper bound $\overline{v_{\sigma(u)}} \leq \underline{v}$ may be discarded as it is guaranteed not to improve the best known solution. This is pretty much the same reasoning that underlies the whole branch-and-bound idea. But here, it is used to prune portions of the search space explored *while compiling* approximate MDDs.

To implement RUB, it suffices to adapt the MDD compilation procedure (top-down , iterative refinement, ...) and introduce a check that avoids creating a node u' with state $next$ when $\overline{v_{next}} \leq \underline{v}$.

The key to RUB effectiveness is that RUB is used while compiling the restricted and relaxed DDs. As such, its computation does not directly appear in Alg. 1, but rather is accounted within the compilations of $Restricted(u)$ and

Relaxed(u) from Alg. 1. Thus, it really is not used as yet-an-other-bound competing with that of line 12, but instead to speed up the computation of restricted and relaxed DDs. More precisely, this speedup occurs because the compilation of the DDs discards some nodes that would otherwise be added to the next layer of the DD and then further expanded, which are ruled out by RUB. A second benefit of using RUBs is that it helps tightening the bound derived from a relaxed DD (Alg.1 line 12). Because the layers that are generated in a relaxed DD are narrower when applying RUB, there are fewer nodes exceeding the maximum layer width. The operator $\oplus$ hence needs to merge a smaller set of nodes in order to produce the relaxation.

The dynamics of RUB is graphically illustrated by Fig.-3 where the set of highlighted nodes can be safely elided since the (rough) upper bound computed in node s is lesser than the best lower bound.

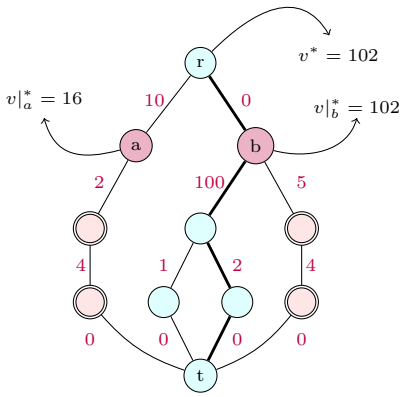


Fig. 2. An example relaxed-MDD having an exact cutset $\{a, b\}$ with local bounds $v|_a^*$ and $v|_b^*$. The nodes with a simple border represent exact nodes and those with a double border represent "inexact" nodes. The edges along the longest path are displayed in bold.

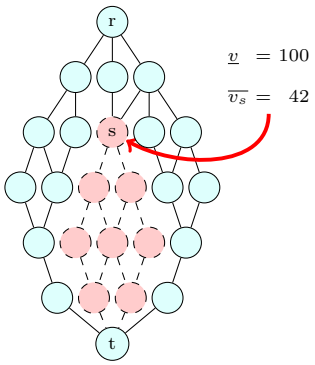


Fig. 3. Assuming a lower bound $\underline{v}$ of 100 and a rough upper bound $\overline{v_s}$ of 42 for the node s, all the highlighted nodes (in red, with a dashed border) may be pruned from the MDD.

Important Note It is important to understand that because the RUB is computed at each node of each restricted and relaxed MDD compiled during the instance resolution, it must be extremely inexpensive to compute. This is why RUB is best obtained from a fast and simple problem specific procedure.

3 Experimental Study

In order to evaluate the impact of the pruning techniques proposed above, we conducted a series of experiments on four problems. In particular, we conducted experiments on the Maximum Independent Set Problem (MISP), the Maximum Cut Problem (MCP), the Maximum Weighted 2-Satisfiablility Problem (MAX2SAT) and the Traveling Salesman Problem with Time Windows (TSPTW). For the first three problems, we generated sets of random instances which we attempted to solve with different configurations of our own open source solver written in Rust [17][5]. For TSPTW, we reused openly available sets of benchmarks which are usually used to assess the efficiency of new solvers for TSPTW[27]. Thanks to the generic nature of our framework, the model and all heuristics used to solve the instances were the same for all experiments. This allowed us to isolate the impact of RUB and LocB on the solving performance and neutralize unrelated factors such as variable ordering. Indeed, the only variations between the different solver flavors relate to the presence (or absence) of RUB and LocB. All experiments were run on the same physical machine equipped with an AMD6176 processor and 48GB of RAM. A maximum time limit of 1800 seconds was allotted to each configuration to solve each instance.

The details of the DP models and RUBs we formulated for all four problems are given in the appendices to the extended version of this paper [6].

MISP. To assess the impact of RUB and LocB on MISP, we generated random graphs based on the Erdos-Renyi model G(n, p) [15] with the number of vertices n = 250, 500, 750, 1000, 1250, 1500, 1750 and the probability of having an edge connecting any two vertices p = 0.1, 0.2, ... , 0.9. The weight of the edges in the generated graphs were drawn uniformly from the set $\{-5, -4, -3, -2, -1, 1, 2, 3, 4, 5\}$. We generated 10 instances for each combination of size and density (n, p).

MCP. In line with the strategy used for MISP, we generated random MCP instances as random graphs based on the Erdos-Renyi model G(n, p). These graphs were generated with the number of vertices n = 30, 40, 50 and the probability p of connecting any two vertices = 0.1, 0.2, 0.3, .., 0.9. The weights of the edges in the generated graphs were drawn uniformly among $\{-1, 1\}$. Again, we generated 10 instances per combination n, p.

MAX2SAT. Similar to the above, we used random graphs based the Erdos-Renyi model G(n, p) to derive MAX2SAT instances. To this end, we produced graphs with n = 60, 80, 100, 200, 400, 1000 (hence instances with 30, 40, 50, 100, 200 and 500 variables) and p = 0.1, 0.2, 0.3, .. , 0.9. For each combination of size (n) and density (p), we generated 10 instances. The weights of the clauses in the generated instances were drawn uniformly from the set $\{1, 2, 3, 5, 6, 7, 8, 9, 10\}$.

[5] https://github.com/xgillard/ddo
[6] Available online at: http://hdl.handle.net/2078.1/245322

TSPTW. To evaluate the effectiveness of our rules on TSPTW, we used the 467 instances from the following suites of benchmarks, which are usually used to assess the efficiency of new TSPTW solvers. AFG [2], Dumas [14], Gendreau-Dumas [16], Langevin [26], Ohlmann-Thomas [28], Solomon-Pesant [29] and Solomon-Potvin-Bengio [30].

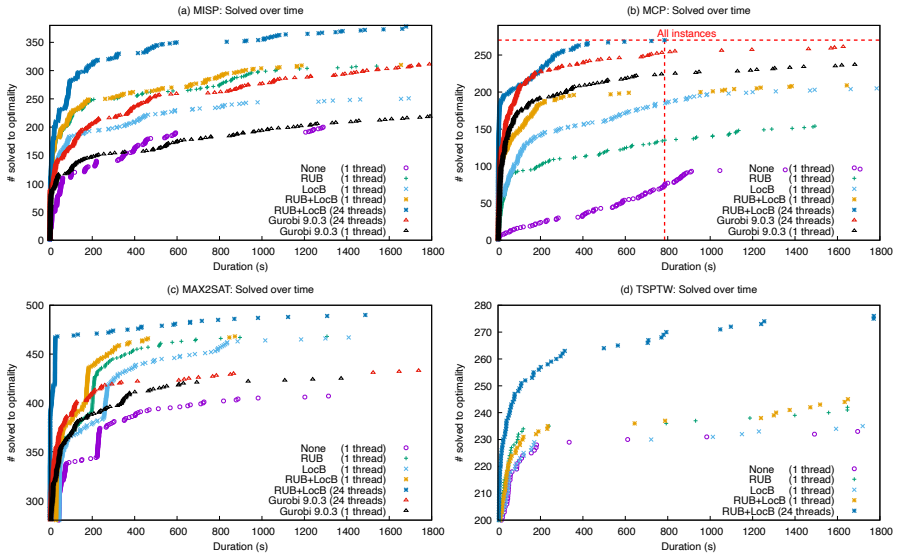


Fig. 4. Number of solved instances over time for each considered problem

Figure 4 gives an overview of the results from our experimental study. It respectively depicts the evolution over time of the number of instances solved by each technique for MISP (a), MCP (b) and MAX2SAT (c) and TSPTW (d).

As a first step, our observation of the graphs will focus on the differences that arise between the single threaded configurations of our *ddo* solvers. Then, in a second phase, we will incorporate an existing state-of-the-art ILP solver (Gurobi 9.0.3) in the comparison. Also, because both Gurobi and our *ddo* library come with built-in parallel computation capabilities, we will consider both the single threaded and parallel (24 threads) cases. This second phase, however, only bears on MISP, MCP and MAX2SAT by lack of a Gurobi TSPTW model.

DDO configurations The first observation to be made about the four graphs in Fig.4, is that for all considered problems, both RUB and LocB outperformed the 'do-nothing' strategy; thereby showing the relevance of the rules we propose. It is not clear however which of the two rules brings the most improvement to the problem resolution. Indeed, RUB seems to be the driving improvement factor for MISP (a) and TSPTW (d) and the impact of LocB appears to be moderate or

weak on these problems. However, it has a much higher impact for MCP (b) and MAX2SAT (c). In particular, LocB appears to be the driving improvement factor for MCP (b). This is quite remarkable given that LocB operates in a purely black box fashion, without any problem-specific knowledge. Finally, it should also be noted that the use of RUB and LocB are not mutually exclusive. Moreover, it turns out that for all considered problems, the combination RUB+LocB improved the situation over the use of any single rule.

Furthermore, Fig.5 confirms the benefit of using both RUB *and* LocB together rather than using any single technique. For each problem, it measures the "performance" of using RUB+LocB vs the best single technique through the end gap. The end gap is defined as $\left(100 * \frac{|UB|-|LB|}{|UB|}\right)$. This metric allows us to account for all instances, including the ones that could not be solved to optimality. Basically, a small end gap means that the solver was able to confirm a tight confidence interval of the optimum. Hence, a smaller gap is better. On each subgraphs of Fig.5, the distance along the x-axis represents the end gap for reach instance when using both RUB and LocB whereas the distance along y-axis represents the end gap when using the best single technique for the problem at hand. Any mark above the diagonal shows an instance for which using both RUB and LocB helped reduce the end gap and any mark below that line indicates an instance where it was detrimental.

From graphs 5-a, 5-c and 5-d it appears that the combination RUB+LocB supersedes the use of RUB only. Indeed the vast majority of the marks sit above the diagonal and the rest on it. This indicates a beneficial impact of using both techniques even for the hardest (unsolved) instances. The case of MCP (graph 5-b) is less clear as most of the marks sit on the diagonal. Still, we can only observe three marks below the diagonal and a bit more above it. Which means that even though the use of RUB in addition to LocB is of little help in the case of MCP, its use does not degrade the performance for that considered problem.

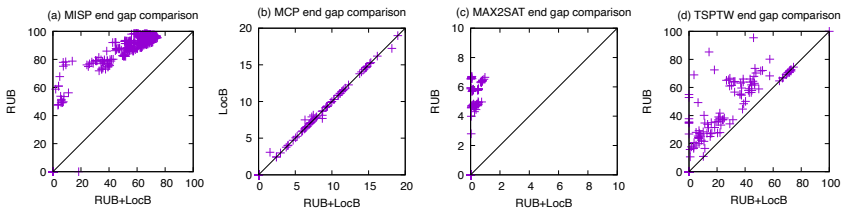


Fig. 5. End gap: The benefit of using both techniques vs the best single one

Comparison with Gurobi 9.0.3 The first observation to be made when comparing the performance of Gurobi vs the DDO configurations, is that when running on a single thread, ILP outperforms the basic DDO approach (without RUB and LocB). Furthermore, Gurobi turns out to be the best single threaded solver for

MCP by a fair margin. However, in the MISP and MAX2SAT cases, Fig. 4 shows that the DDO solvers benefitting from RUB and LocB were able to solve more instances and to solve them faster than Gurobi. Which underlines the importance of RUB and LocB.

When lifting the one thread limit, one can see that the DD-based approach outperform ILP on each of the considered problems. In particular, in the case of MCP for which Gurobi is the best single threaded option; our DDO solver was able to find and prove the optimality of all tested instances in a little less than 800 seconds. The ILP solver, on the other end, was not able to prove the optimality of the 9 hardest instances within 30 minutes. Additionally, we also observe that in spite of the performance gains of MIP when running in parallel, Gurobi fails to solve as many MISP and MAX2SAT instances and to solve them as fast as the single threaded DDO solvers with RUB and LocB. This emphasizes once more the relevance of our techniques. It also shows that the observation from [9] still hold today: despite the many advances of MIP the DDO approach still scales better than MIP on the considered problems when invoked in parallel.

4 Previous work

DDO emerged in the mid' 2000's when [24] proposed to use decision diagrams as a way to solve discrete optimization problems to optimality. More or less concomitantly, [1] devised relaxed-MDD even though the authors envisioned its use as a CP constraint store rather than a means to derive tight upper bounds for optimization problems. Then, the relationship between decision diagrams and dynamic programming was clarified by [21].

Recently, Bergman, Ciré and van Hoeve investigated the various ways to compile decision diagrams for optimization (top-down, construction by separation) [11]. They also investigated the heuristics used to parameterize these DD compilations. In particular, they analyzed the impact of variable ordering in [11, 7] and node selection heuristics (for merge and deletion) in [7]. Doing so, they empirically demonstrated the crucial impact of variable ordering on the tightness of the derived bounds and highlighted the efficiency of minLP as a node selection heuristic. Later on, the same authors proposed a complete branch-and-bound algorithm based on DDs [8]. This is the algorithm which we propose to adapt with extra reasoning mechanisms and for which we provide a generic open-source implementation in Rust [17]. The impressive performance of DDO triggered some theoretical research to analyze the quality of approximate MDDs [5] and the correctness of the relaxation operators [22].

This gave rise to new lines of work. The first one focuses on the resolution of a larger class of optimization problems; chief of which multi-objective problems [4] and problems with a non-linear objective function. These are either solved by decomposition [4] or by using DDO to strengthen other IP techniques [13]. A second trend aims at hybridizing DDO with other IP techniques. For instance, by using Lagrangian relaxation [23] or by solving a MIP [6] to derive with very tight bounds. But the other direction is also under active investigation: for example,

[31, 32] use DD to derive tight bounds which are used to replace LP relaxation in a cutting planes solver. Very recently, a third hybridization approach has been proposed by Gonzàlez et al.[18]. It adopts the branch-and-bound MDD perspective, but whenever an upper bound is to be derived, it uses a trained classifier to decide whether the upper bound is to be computed with ILP or by developing a fixed-width relaxed MDD.

The techniques (ILP-cutoff pruning and ILP-cutoff heuristic) proposed by Gonzalez et al.[18] are related to RUB and LocB in the sense that all techniques aim at reducing the search space of the problem. However, they fundamentally differ as ILP-cutoff pruning acts as a replacement for the compilation of a relaxed MDD whereas the goal of RUB is to speed up the development of that relaxed MDD by removing nodes *while* the MDD is being generated. The difference is even bigger in the case of ILP-cutoff heuristic vs LocB: the former is used as a primal heuristic while LocB is used to filter out sub-problems that can bear no better solution. In that sense, LocB belongs more to the line of work started by [1, 19, 20]: it enforces the constraint $lb \leq f(x) \leq ub$ and therefore provokes the deletion of nodes and arcs that cannot lead to the optimal solution.

More recently, Horn et al explored an idea in [25] which closely relates to RUB. They use "fast-to-compute dual bounds" as an admissible heuristic to guide the compilation of MDDs in an A* fashion for the prize-collecting TSP. It prunes portions of the state space during the MDD construction, similarly to when RUB is applied. Our approach differs from that of [25] in that we attempt to incorporate problem specific knowledge in a framework that is otherwise fully generic. More precisely, it is perceived here as a problem-specific pruning that exploits the combinatorial structure implied by the state variables. It is independent of other MDD compilation techniques, e.g., our techniques are compatible with node merge ($\oplus$) operators and other methodologies defined in the DDO literature. We also emphasize that, as opposed to more complex LP-based heuristics that are now typical in A* search, we investigate quick methodologies that are also easy to incorporate in a MDD branch and bound.

Conclusion and future work

This paper presented and evaluated the impact of the local bound and rough upper bound techniques to strengthen the pruning of the branch-and-bound MDD algorithm. Our experimental study on MISP, MCP, MAX2SAT and TSPTW confirmed the relevance of these techniques. In particular, our experiments have shown that devising a fast and simple rough upper bound is worth the effort as it can significantly boost the efficiency of a solver. Similarly, our experiments showed that the use of local bound can significantly improve the efficiency of DDO solver despite its problem agnosticism. Furthermore, it revealed that a combination of RUB and LocB supersedes the benefit of any single reasoning technique. These results are very promising and we believe that the public availability of an open source DDO framework implementing RUB and LocB might serve as a basis for novel DP formulation for classic problems.

References

1. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A constraint store based on multivalued decision diagrams. In: Bessière, C. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 4741, pp. 118–132. Springer (2007)
2. Ascheuer, N.: Hamiltonian path problems in the on-line optimization of flexible manufacturing systems (1996)
3. Bellman, R.: The theory of dynamic programming. Bulletin of the American Mathematical Society **60**(6), 503–515 (11 1954), https://projecteuclid.org:443/euclid.bams/1183519147
4. Bergman, D., Cire, A.A.: Multiobjective optimization by decision diagrams. In: Rueher, M. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 9892, pp. 86–95. Springer (2016)
5. Bergman, D., Cire, A.A.: Theoretical insights and algorithmic tools for decision diagram-based optimization. Constraints **21**(4), 533–556 (2016). https://doi.org/10.1007/s10601-016-9239-9, https://doi.org/10.1007/s10601-016-9239-9
6. Bergman, D., Cire, A.A.: On finding the optimal bdd relaxation. In: Salvagnin, D., Lombardi, M. (eds.) Integration of AI and OR Techniques in Constraint Programming. LNCS, vol. 10335, pp. 41–50. Springer (2017)
7. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Optimization bounds from binary decision diagrams. INFORMS Journal on Computing **26**(2), 253–268 (2014). https://doi.org/10.1287/ijoc.2013.0561, https://doi.org/10.1287/ijoc.2013.0561
8. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.N.: Discrete optimization with decision diagrams. INFORMS Journal on Computing **28**(1), 47–66 (2016). https://doi.org/10.1287/ijoc.2015.0648, https://doi.org/10.1287/ijoc.2015.0648
9. Bergman, D., Cire, A.A., Sabharwal, A., Samulowitz, H., Saraswat, V., van Hoeve, W.J.: Parallel combinatorial optimization with decision diagrams. International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems pp. 351–367 (2014)
10. Burch, J., E.M., C., K.L., M., D.L., D., H.L., H.: Symbolic model checking: 10^{20} states and beyond. Information and Computation **98**(2), 142–170 (1992). https://doi.org/10.1016/0890-5401(92)90017-A, https://doi.org/10.1016/0890-5401(92)90017-A
11. Cire, A.A.: Decision Diagrams for Optimization. Ph.D. thesis, Carnegie Mellon University Tepper School of Business (2014)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT press (2009)
13. Davarnia, D., van Hoeve, W.J.: Outer approximation for integer nonlinear programs via decision diagrams (2018)
14. Dumas, Y., Desrosiers, J., Gelinas, E., Solomon, M.M.: An optimal algorithm for the traveling salesman problem with time windows. Operations research **43**(2), 367–371 (1995)
15. Erdös, P., Rényi, A.: On random graphs i. Publicationes Mathematicae Debrecen **6**, 290 (1959)
16. Gendreau, M., Hertz, A., Laporte, G., Stan, M.: A generalized insertion heuristic for the traveling salesman problem with time windows. Operations Research **46**(3), 330–335 (1998)

17. Gillard, X., Schaus, P., Coppé, V.: Ddo, a generic and efficient framework for mdd-based optimization. Accepted at the International Joint Conference on Artificial Intelligence (IJCAI-20); DEMO track (2020)
18. Gonzalez, J.E., Cire, A.A., Lodi, A., Rousseau, L.M.: Integrated integer programming and decision diagram search tree with an application to the maximum independent set problem. Constraints pp. 1–24 (2020)
19. Hadžić, T., Hooker, J., Tiedemann, P.: Propagating separable equalities in an mdd store. In: CPAIOR. pp. 318–322 (2008)
20. Hoda, S., Van Hoeve, W.J., Hooker, J.N.: A systematic approach to mdd-based constraint programming. In: International Conference on Principles and Practice of Constraint Programming. pp. 266–280. Springer (2010)
21. Hooker, J.N.: Decision diagrams and dynamic programming. In: Gomes, C., Sellmann, M. (eds.) Integration of AI and OR Techniques in Constraint Programming. LNCS, vol. 7874, pp. 94–110. Springer (2013)
22. Hooker, J.N.: Job sequencing bounds from decision diagrams. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming. LNCS, vol. 10416, pp. 565–578. Springer (2017)
23. Hooker, J.N.: Improved job sequencing bounds from decision diagrams. In: Schiex, T., de Givry, S. (eds.) Principles and Practice of Constraint Programming. LNCS, vol. 11802, pp. 268–283. Springer (2019)
24. Hooker, J.: Discrete global optimization with binary decision diagrams. GICOLAG 2006 (2006)
25. Horn, M., M̃aschler, J., R̃aidl, G.R., R̃önnberg, E.: A*-based construction of decision diagrams for a prize-collecting scheduling problem. Computers & Operations Research **126**, 105125 (2021). https://doi.org/https://doi.org/10.1016/j.cor.2020.105125, http://www.sciencedirect.com/science/article/pii/S0305054820302422
26. Langevin, A., Desrochers, M., Desrosiers, J., Gélinas, S., Soumis, F.: A two-commodity flow formulation for the traveling salesman and the makespan problems with time windows. Networks **23**(7), 631–640 (1993)
27. López-Ibáñez, M., Blum, C.: Benchmark instances for the travelling salesman problem with time windows. Online (2020), http://lopez-ibanez.eu/tsptw-instances
28. Ohlmann, J.W., Thomas, B.W.: A compressed-annealing heuristic for the traveling salesman problem with time windows. INFORMS Journal on Computing **19**(1), 80–90 (2007)
29. Pesant, G., Gendreau, M., Potvin, J.Y., Rousseau, J.M.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. Transportation Science **32**(1), 12–29 (1998)
30. Potvin, J.Y., Bengio, S.: The vehicle routing problem with time windows part ii: genetic search. INFORMS journal on Computing **8**(2), 165–172 (1996)
31. Tjandraatmadja, C.: Decision Diagram Relaxations for Integer Programming. Ph.D. thesis, Carnegie Mellon University Tepper School of Business (2018)
32. Tjandraatmadja, C., van Hoeve, W.J.: Target cuts from relaxed decision diagrams. INFORMS Journal on Computing **31**(2), 285–301 (2019). https://doi.org/10.1287/ijoc.2018.0830, https://doi.org/10.1287/ijoc.2018.0830

Actes JFPC 2021

Descente Agressive de Borne en Optimisation sous Contraintes

Thibault Falque[1] *__Christophe Lecoutre__[2] **Bertrand Mazure**[2] **Hugues Wattez**[2]

[1] Exakis Nelite, Paris, France
[2] CRIL, Univ Artois & CNRS
thibault.falque@exakis-nelite.com {lecoutre,mazure,wattez}@cril.fr

Résumé

La recherche avec retour en arrière est une approche complète classique pour explorer l'espace de recherche d'un problème d'optimisation sous contraintes. Chaque fois qu'une nouvelle solution est trouvée pendant la recherche, la borne qui lui est associée est utilisée pour contraindre davantage le problème, et donc la recherche restante. Un scénario extrême (mauvais) est celui où les solutions sont trouvées en séquence avec de très petites différences entre les bornes successives. Dans cet article, nous proposons une approche ABD (*Aggressive Bound Descent*) pour remédier à ce problème : les nouvelles bornes sont modifiées de manière exponentielle tant que l'algorithme de recherche est efficace. Nous montrons que cette approche peut rendre le solveur plus robuste, surtout au début de la recherche. Nos expérimentations confirment ce comportement pour l'optimisation sous contraintes et les problèmes Pseudo-Booléens.

Abstract

Backtrack search is a classical complete approach for exploring the search space of a constraint optimization problem. Each time a new solution is found during search, its associated bound is used to constrain more the problem, and so the remaining search. An extreme (bad) scenario is when solutions are found in sequence with very small differences between successive bounds. In this paper, we propose an aggressive bound descent (ABD) approach to remedy this problem: new bounds are modified exponentially as long as the searching algorithm is successful. We show that this approach can render the solver more robust, especially at the beginning of search. Our experiments confirm this behavior for both constraint optimization and Pseudo-Boolean problems.

*Papier doctorant : Thibault Falque[1] est auteur principal.

1 Introduction

En programmation par contraintes (CP), même si de nombreuses extensions ont été proposées depuis les années 70, il est habituel de traiter soit des problèmes de satisfaction de contraintes (CSP), soit des problèmes d'optimisation sous contraintes (COP). En plus d'un ensemble de variables (entières) à assigner tout en satisfaisant un ensemble de contraintes, une instance COP implique une fonction objective à optimiser (c'est-à-dire une fonction de coût à minimiser ou une fonction de récompense à maximiser). La résolution d'une instance COP nécessite non seulement de prouver la satisfaisabilité (c'est-à-dire de trouver au moins une solution), mais aussi, idéalement, de prouver l'optimalité ou au moins de trouver des solutions d'assez bonne qualité (c'est-à-dire proches de l'optimalité).

La recherche avec retour en arrière est une approche complète classique pour explorer l'espace de recherche d'une instance COP. Cela revient à résoudre une série d'instances CSP. En supposant une fonction objective f à minimiser où initialement f est traitée sous la forme d'une contrainte $f < \infty$, et chaque fois qu'une nouvelle solution de coût B est trouvée, B est utilisé comme nouvelle borne pour la contrainte objective, de façon à devenir $f < B$ (formant ainsi une instance CSP plus contrainte). De cette façon, toute nouvelle solution trouvée est garantie d'être de meilleure qualité (coût inférieur) que la précédente, et l'optimalité peut être prouvée lorsque l'instance du problème devient insatisfaisable.

Il n'est pas rare que les problèmes d'optimisation issus de l'industrie soient faiblement contraints, ce qui signifie qu'un grand nombre de solutions existent avec des qualités diverses dans différentes parties de l'espace de recherche. Dans de telles situations, l'application

de la recherche avec retour en arrière peut être pénalisée parce que la descente des bornes (c'est-à-dire la séquence décroissante des bornes successivement trouvées) peut être très lente : la distance entre deux bornes successives peut être plutôt faible. Dans cet article, nous proposons une approche pour tenter de réduire ce phénomène en modifiant les bornes d'une manière plus agressive : tant qu'elle est réussie, de nouvelles bornes pour la contrainte objective sont calculées en suivant une croissance exponentielle. Bien sûr, dans le cas où la borne rend le problème insatisfaisable ou très difficile à résoudre, un mécanisme de redémarrage nous permet de reprendre la recherche sur des pistes plus certaines.

Même si d'autres techniques de recherche existent dans la littérature, comme par exemple la métaheuristique largement utilisée *Large Neighborhood Search* (LNS) [12], dans cet article, nous nous concentrons sur la recherche en profondeur avec retour en arrière, équipée du mécanisme de *solution(-based phase) saving* qui s'est avérée être très efficace [14, 5].

Cet article est organisé comme suit. La Section 2 présente le contexte technique de la programmation par contraintes. Ensuite, dans la Section 3, le principe de Descente Agressive de Borne (ABD pour *Aggressive Bound Descent*) est présenté, et avant de conclure, les résultats expérimentaux sont donnés dans la Section 4.

2 Contexte Technique

Un *réseau de contraintes* (CN pour *Constraint Network*) est constitué d'un ensemble fini de variables et d'un ensemble fini de contraintes. Chaque variable x peut prendre une valeur dans un ensemble fini appelé le *domaine* de x, noté $\texttt{dom}(x)$. Chaque contrainte c est spécifiée par une relation sur un ensemble de variables. Une *solution* d'un CN est l'affectation d'une valeur à toutes les variables de façon à satisfaire toutes les contraintes. Un CN est *satisfaisable* si il admet au moins une solution, et le *problème de satisfaction de contraintes* (CSP) correspondant consiste à déterminer si un CN donné est satisfaisable ou non.

Un *réseau de contraintes sous optimisation* (CNO pour *Constraint Network under Optimization*) est un réseau de contraintes associé à une fonction objectif $\texttt{obj}$ faisant correspondre toute solution à une valeur [1] dans $\mathbb{D}$. Sans aucune perte de généralité, nous considérerons que $\texttt{obj}$ doit être minimisée. Une solution S d'un CNO est une solution du CN sous-jacent ; S est *optimal* s'il n'existe aucune autre solution S' telle que $\texttt{obj}(S') < \texttt{obj}(S)$. La tâche habituelle du *problème d'optimisation sous contraintes* (COP) est de trouver une solution optimale pour un CNO donné. Notez que les CN et les CNO sont également appelés instances de CSP/COP.

La recherche avec retours en arrière est une procédure complète classique pour résoudre les instances de CSP/COP. Elle alterne les affectations de variables (et les réfutations) et un mécanisme appelé propagation de contraintes afin de filtrer l'espace de recherche. Typiquement, comme pour MAC (*Maintaining Arc Consistency*) [11] qui propage les contraintes en maintenant la propriété de cohérence d'arc, un arbre de recherche binaire $\mathcal{T}$ est construit : à chaque nœud interne de $\mathcal{T}$, (i) on sélectionne une paire (x, v) où x est une variable non-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 $x \neq v$.

L'ordre dans lequel les variables sont choisies pendant le parcours en profondeur de l'espace de recherche est décidé par une *heuristique de choix de variables* ; une heuristique générique classique étant $\texttt{dom/wdeg}$ [3]. L'ordre dans lequel les valeurs sont choisies lors de l'affectation des variables est déterminé par une *heuristique de choix de valeurs* ; pour les COPs, il est fortement recommandé d'utiliser d'abord la valeur présente dans la dernière solution trouvée, ce qui est une technique connue sous le nom de *solution(-based phase) saving* [14, 5].

La recherche avec retours en arrière pour le COP repose sur la résolution de CSP : le principe consiste à ajouter au réseau de contraintes une *contrainte objectif* spéciale $\texttt{obj} < \infty$ (bien qu'elle soit initialement satisfaite), et de mettre à jour la borne de cette contrainte chaque fois qu'une nouvelle solution est trouvée. Cela signifie qu'à chaque nouvelle solution S trouvée avec un coût $B = \texttt{obj}(S)$, la contrainte objectif devient $\texttt{obj} < B$. Par conséquent, une séquence de solutions, dont la qualité est croissante, est générée (la SATisfiabilité est systématiquement prouvée par rapport à la borne actuelle de la contrainte objectif) jusqu'à ce qu'il n'en existe plus aucune (l'insatisfaisabilité (*UNSATisfiability*) est finalement prouvée par rapport à la borne imposée par la dernière solution trouvée), ce qui garantit que la dernière solution trouvée est optimale.

Les politiques de redémarrage jouent un rôle important dans les solveurs de contraintes modernes, car elles permettent d'aborder le phénomène du *heavy-tailed* des instances SAT (*Satisfiability Testing*) et CSP/COP [6]. En substance, une politique de redémarrage correspond à une fonction $\texttt{restart} : \mathbb{N}^+ \to \mathbb{N}^+$, qui indique le nombre maximal d'*étapes* autorisés pour l'algorithme de recherche lors d'une exécution, appelée *run*, $j \geq 1$. Cela signifie que la recherche avec retour en arrière piloté par un mécanisme de redémarrageconstruit une séquence d'arbres de recherche binaires $\langle \mathcal{T}_1, \mathcal{T}_2, \ldots \rangle$, où $\mathcal{T}_j$ est l'arbre de recherche exploré au run j.

1. Pour simplifier la présentation, nous considérons que les coûts sont donnés par des valeurs entières.

Notez que le *cutoff*, qui est le nombre maximum d'étapes autorisées pendant une exécution, peut correspondre au nombre de retours en arrière, au nombre de mauvaises décisions [1], ou toute autre mesure pertinente. Dans une stratégie de redémarrage au cutoff *fixé*, `restart`(j) est constant quelle que soit le run j. Dans une stratégie de redémarrage à cutoff *dynamique*, `restart` augmente géométriquement le cutoff [15], ce qui garantit que tout l'espace des solutions partielles sera exploré.

3 Descente Agressive de Borne

Lors de la résolution d'une instance COP, la *descente de borne* est définie comme la séquence $D = \langle B_1, B_2, \ldots \rangle$ de bornes successives identifiées par l'algorithme de recherche. À un extrême, cette séquence ne contient qu'une seule valeur, la borne optimale. A un autre extrême, elle contient une grande séquence de valeurs, chacune étant proche de la précédente : la descente de borne est dite *lente*. C'est le cas lorsque la valeur moyenne de la séquence dérivée des gains (ou écarts) de bornes $G = \langle B_1 - B_2, B_2 - B_3, \ldots \rangle$ est petite (proche de 1).

Il est certain qu'une descente lente de borne indique qu'il y a une certaine marge d'amélioration dans la façon dont la recherche avec retour arrière est menée. En effet, l'énumération d'un grand nombre de solutions proches les unes des autres avant d'atteindre l'optimalité implique la résolution d'un grand nombre d'instances de problèmes de satisfaction dérivés et cela peut être pénalisant. C'est pourquoi nous proposons une politique *agressive* de descente de bornes : la politique ABD. Au lieu de fixer la borne stricte de la contrainte objective à B lorsqu'une nouvelle solution de coût B est trouvée, nous proposons de la fixer à une valeur éventuellement inférieure B'.

Une première politique ABD simple pourrait consister à utiliser une différence statique entre B et B' : $B' = B - \Delta$ où Δ est une valeur entière positive fixe. Cependant, cette politique *statique* souffre clairement d'un manque d'adaptabilité, et de plus, fixer la bonne valeur pour Δ peut dépendre du problème et ne pas être très facile à réaliser.

C'est pourquoi nous proposons des politiques *dynamiques* pour ABD, inspirées des études concernant les suites utilisées par les politiques de redémarrage.

Pour définir des politiques ABD dynamiques, nous introduisons d'abord quelques suites classiques d'entiers strictement positifs, c'est-à-dire des fonctions `abd` : $\mathbb{N}^+ \to \mathbb{N}^+$. Bien que détaillé plus loin, le paramètre $i \geq 1$ de ces séquences correspond au nombre de mises à jour successives réussies de la borne, c'est-à-dire de mises à jour successives agressives de la borne de la contrainte objective tout en conservant la satisfaisabilité.

Plus précisément, quatre suites entières sont utilisées dans notre étude :

$$\texttt{exp}(i) = 2^{i-1} \tag{1}$$

$$\texttt{rexp}(i) = \begin{cases} 2^{k-1}, & \text{si } i = \frac{k(k+1)}{2} \\ 2^{i-\frac{k(k+1)}{2}-1}, & \text{si } \frac{k(k+1)}{2} < i < \frac{(k+1)(k+2)}{2} \end{cases} \tag{2}$$

$$\texttt{luby}(i) = \begin{cases} 2^{k-1}, & \text{si } i = 2^k - 1 \\ \texttt{luby}(i - 2^{k-1} + 1), & \text{si } 2^{k-1} \leq i < 2^k - 1 \end{cases} \tag{3}$$

$$\texttt{prev}(i) = \begin{cases} 1, & \text{si } i = 1 \\ G_{i-1} \times 2, & \text{sinon} \end{cases} \tag{4}$$

où, dans l'Équation 4, G_i est la ième valeur de la suite de gains, telle que définie précédemment.

L'Équation 1 correspond à la fonction exponentielle classique `exp` (en base 2). Dérivée de cette classique progression exponentielle, `rexp` dans l'Équation 2 correspond à une suite de `exp` régulièrement réinitialisée. Les premières valeurs de cette suite sont : $1, 1, 2, 1, 2, 4, \ldots$. En ne considérant que les nombres les plus élevés produits par le premier terme (condition) de l'équation, on obtient une progression légèrement plus lente que la précédente : $O(2^{\sqrt{i}})$. Une autre suite, couramment utilisée dans les politiques de redémarrage, est la suite de Luby [10], donnée par l'Équation 3. Les premières valeurs de la suite de Luby sont : $1, 1, 2, 1, 1, 2, 4, \ldots$. En considérant à nouveau les plus grands nombres produits par le premier terme, on constate que la progression est en $O(i)$. Enfin, la dernière suite, donnée par l'Équation 4, est basée sur la suite des gains, et suit également une progression exponentielle.

Chaque suite dans $\Psi = \{\texttt{exp}, \texttt{rexp}, \texttt{luby}, \texttt{prev}\}$ nous permet de définir un éponyme de la politique ABD.

3.1 Politique ABD

Soit $\mathcal{T}$ l'arbre de recherche actuel construit par l'algorithme de recherche (c'est-à-dire pendant l'exécution actuelle). Soit D la descente de bornes produite depuis le début du run courant, et soit `abd` $\in \Psi$. L'exécution courante peut rencontrer trois situations distinctes :

1. le run en cours est arrêté car la valeur limite est atteinte,
2. le run en cours est arrêté parce que les algorithmes de recherche indiquent qu'il n'existe plus de solution,

3. une nouvelle solution S est trouvée.

Tout d'abord, nous discutons du cas le plus intéressant : le troisième. La politique ABD stipule que lorsqu'une nouvelle solution S de coût B est trouvée, B est ajouté à D, et la borne de la contrainte objective est fixée à $B + 1 - \texttt{abd}(i)$, où $i = |D|$. En d'autres termes, la contrainte objectif devient : $\texttt{obj} < B+1-\texttt{abd}(i)$; notez que 1 est ajouté à B car les fonctions `abd` ne retournent que des valeurs supérieures ou égales à 1. Maintenant, nous donnons une description générale précise (traitant en particulier les deux premières situations ci-dessus) de la façon dont une politique ABD peut être implémentée dans une recherche avec retour en arrière.

3.2 Simple Implémentation d'ABD

La fonction `solve`, Algorithme 1, tente de résoudre le CNO P grâce à l'usage de la politique de descente agressive de bornes `abd` spécifiée.

Algorithm 1: `solve`$(P, \texttt{abd})$

Output: $\underline{B_P}..\overline{B_P}$, runStatus

1 $\underline{B_P}..\overline{B_P} \leftarrow -\infty..+\infty$
2 **do**
3 $\quad P, \text{runStatus} \leftarrow \texttt{run}(P, \texttt{abd})$
4 **while** runStatus = CONTINUE
5 **return** $(\underline{B_P}..\overline{B_P}, \text{runStatus})$

Tout d'abord, les bornes inférieure et supérieure, désignées par $\underline{B_P}$ et $\overline{B_P}$, de la fonction objectif de P sont respectivement initialisées à $-\infty$ et $+\infty$ (ou toute autre valeur pertinente pouvant être pré-calculée). Pendant la recherche, ces bornes seront mises à jour (mais pour des raisons de simplicité, cela ne sera pas explicitement montré dans le pseudo-code). À la ligne 2, la séquence de runs (redémarrages) est lancée. Chaque fois qu'une nouvelle exécution est terminée, elle renvoie le réseau de contraintes (éventuellement mis à jour avec certaines contraintes ou certains *nogoods* qui ont été appris ; ceci sera discuté plus en détail plus tard) ainsi qu'un statut. Le statut prend l'une des valeurs suivantes : CONTINUE si le solveur est autorisé à poursuivre avec un nouveau run ; COMPLETE si le dernier run a exploré exhaustivement l'espace des solutions ; INCOMPLETE si le solveur a atteint la limite de temps sans avoir exploré entièrement l'espace de recherche. Enfin, la fonction renvoie les meilleures bornes trouvées (dans le cas où l'optimalité a été prouvée, nous avons $\underline{B_P} = \overline{B_P}$) et l'état final de la recherche.

La fonction `run`, Algorithme 2, effectue une recherche, en suivant les politiques de redémarrage et de `abd`. Avant d'aller plus loin, nous devons introduire la notion de *safe/unsafe* : lorsqu'on demande au solveur de diminuer agressivement sa borne objective, nous pouvons entrer dans une partie de l'espace de recherche qui est UNSAT. Si l'insatisfaisabilité est prouvée pendant l'exécution actuelle, cela peut être dû à notre approche agressive, et par conséquent, nous devons traiter ce problème. Ce point est abordé ci-dessous.

La fonction commence par initialiser un compteur i à 1. Il correspond au nombre de fois où l'on a essayé de trouver une nouvelle solution pendant l'exécution courante. À la ligne 2, Σ_i est la séquence de décisions prises le long de la branche la plus à droite du run actuel, juste avant de commencer la prochaine tentative de trouver une nouvelle solution ; de cette façon, nous pouvons continuer à chercher à partir du même endroit (en pratique, nous reprenons simplement la recherche après l'avoir arrêtée). Initialement, aucune décision n'est prise (et donc, Σ_1 est l'ensemble vide). D'un point de vue pratique, comme nous le verrons, seules les deux dernières séquences Σ_i et Σ_{i-1} seront utiles (pour traiter les cas de résolution sûre et non sûre).

Algorithm 2: `run`$(P, \texttt{abd})$

Output: (P, status)

1 $i \leftarrow 1$
2 $\Sigma_i \leftarrow \emptyset$
3 **do**
4 $\quad \Delta \leftarrow \texttt{abd}(i)$
5 $\quad i \leftarrow i + 1$
6 $\quad \Sigma_i, \text{status} \leftarrow \texttt{search\_next\_sol}(P, \Sigma_{i-1}, \Delta)$
7 **while** status = SAT
8 safe $\leftarrow \Delta = 1$
The global timeout of the resolution
9 **if** status = TIMEOUT **then**
10 $\quad$ **return** $(P, \text{INCOMPLETE})$
An unsat status with $\Delta = 1$
11 **if** status = UNSAT & safe **then**
12 $\quad$ **return** $(P, \text{COMPLETE})$
An unsat status with $\Delta > 1$
13 **if** status = UNSAT & ¬safe **then**
14 $\quad$ **return** $(P \oplus nld(\Sigma_{i-1}), \text{CONTINUE})$
The run cutoff is reached with $\Delta > 1$
15 **if** status = CUTOFF_REACHED & ¬safe **then**
16 $\quad$ **return** $(P \oplus nld(\Sigma_{i-1}), \text{CONTINUE})$
The run cutoff is reached with $\Delta = 1$
17 **if** status = CUTOFF_REACHED & safe **then**
18 $\quad$ **return** $(P \oplus nld(\Sigma_i), \text{CONTINUE})$

Ensuite, l'algorithme effectue des parcours itératifs tant que de nouvelles solutions peuvent être trouvées. À chaque tour de boucle, la prochaine limite d'écart Δ est calculée en sollicitant la politique `abd` et i est incrémenté (lignes 4*et*5). Pour effectuer une partie de la recherche, la fonction `search_next_sol` est appelée, tout en considérant la séquence spécifiée de décisions de départ, et l'écart de borne spécifié. L'écart Δ est utilisé par `search_next_sol` pour calculer une borne supérieure temporaire B' qui remplace la borne supérieure courante $\overline{B_P}$: on a $B' = \overline{B_P} - \Delta + 1$, forçant alors la contrainte objectif à être $f < B'$ pendant cet appel à `search_next_sol`. Si une nouvelle solution de coût B (nécessairement, $B < B'$) est trouvée par `search_next_sol`, l'appel est arrêté, et la contrainte objectif est mise à jour pour devenir sans risque $f < B$. Sinon, l'appel est arrêté (parce que les limites de cutoff ou de temps sont atteintes), et la contrainte objectif est mise à jour pour devenir $f < \overline{B_P}$ (en récupérant la précédente borne supérieure). Pour résumer, cette fonction met implicitement à jour les bornes d'optimisation avant de retourner la nouvelle séquence de décisions (l'endroit exact où la recherche s'est arrêtée) et un statut (local). Le statut local est soit SAT, auquel cas l'exécution peut se poursuivre avec une nouvelle itération de la boucle, soit une valeur parmi UNSAT, CUTOFF_REACHED et TIMEOUT.

Le run actuel exécute nécessairement certaines vérifications à partir de la ligne 8. À cette ligne, un booléen est défini, nous informant si l'exécution actuelle était sûre ou non, en ce qui concerne la dernière valeur Δ calculée. Lorsque la limite de temps globale est atteinte, le réseau de contraintes et l'état INCOMPLETE sont renvoyés (lignes 9 et 10). Lorsque le statut local notifie UNSAT, nous avons deux cas à considérer. Si la recherche en cours a été effectuée de façon *safe*, COMPLETE peut être retourné car l'espace de recherche est garanti d'avoir été entièrement exploré. Sinon, CONTINUE est retourné avec le réseau de contraintes P intégrant éventuellement quelques nouvelles contraintes (nogoods). La notation $P \oplus nld(\Sigma_{i-1})$ indique que tous les nogoods qui peuvent être extraits de l'avant-dernière séquence de décisions (voir [8]) sont ajoutés à P ; ceci est valable car cette séquence était celle correspondant à la dernière solution trouvée. Lorsque l'état local notifie CUTOFF_REACHED, on peut aussi continuer, tout en considérant l'ajout de quelques nogoods de redémarrage, à partir de Σ_i ou Σ_{i-1}.

Nous concluons cette section par deux remarques. Premièrement, l'algorithme est introduit dans le contexte d'un schéma d'enregistrement de nogoods légers (seuls sont considérés les nogoods qui peuvent être extraits de la branche la plus à droite, lorsque la recherche est temporairement arrêtée). Cependant, il est possible de l'adapter à d'autres schémas d'apprentissage, en gardant la trace du moment exact où un nogood (clause) est inféré ; ceci est purement technique. Deuxièmement, il y a un cas spécifique concernant l'insatisfaisabilité : si jamais nous rencontrons une situation où $B' \leq \underline{B_P}$ en essayant de fixer une nouvelle borne supérieure temporaire B' pendant l'exécution actuelle, la suite est réinitialisée en forçant le retour de $i = 1$ et B' est recalculé.

3.3 Travail Connexe

Comme approche connexe, nous pouvons mentionner la division de domaine (par exemple [13]) dont le rôle est de partitionner le domaine de la variable sélectionnée par l'heuristique de choix de variables, et de brancher sur les sous-domaines résultants. Lorsque la fonction objectif est simplement représentée par une variable autonome (dont la valeur doit être minimisée ou maximisée), une approche de division de domaine peut être réglée pour simuler une descente de borne agressive. Cependant, aucun contrôle n'est possible car aucun mécanisme ne permet d'abandonner des choix trop optimistes, contrairement à ABD qui est bien intégrée dans une politique de redémarrage. De plus, lorsque la fonction objectif a une forme plus générale qu'une variable (comme par exemple une somme ou une valeur minimum/maximum à calculer), il n'existe pas de correspondance directe (et introduire systématiquement une variable auxiliaire pour représenter l'objectif peut être très intrusif, voire source d'inefficacité, pour le solveur).

Enfin, notons que la question d'éviter les longues séquences de solutions qui s'améliorent lentement a été abordée dans les MIP par l'introduction d'heuristiques primaires, qui visent à trouver et à améliorer les solutions réalisables au début du processus de résolution.

4 Résultats expérimentaux

Cette section présente quelques résultats expérimentaux concernant l'approche ABD sur un large éventail de problèmes d'optimisation. Pour réaliser les expériences, nous avons utilisé le solveur de contraintes `ACE`, qui est le nouvel avatar de AbsCon [2]. Les options par défaut du solveur ont été utilisées : $\texttt{wdeg}^{\texttt{ca.cd}}$ [17] comme heuristique de choix de variables, `lexico` comme heuristique de choix de valeurs, *last-conflict* (lc) [8] comme méthode paresseuse pour simuler un retour en arrière intelligent, `solution-saving` [14, 5] pour simuler une forme de recherche de voisinage, et une politique de redémarrage géométrique [16] avec

2. `https://www.cril.univ-artois.fr/~lecoutre/#/softwares`

un cutoff de base fixé à 10 mauvaises décisions et une raison fixée à 1.1. Nous étudions notamment l'impact de certains facteurs sur les suite les plus prometteuses parmi Ψ. Nous avons également réalisé des expériences similaires avec le solveur Pseudo-Booléen (PB) `Sat4j` [7]. Tous les solveurs ont été lancés sur un processeur Intel Xeon de 2.66 GHz, avec 32 Go de RAM, tandis que le timeout était fixé à $1,200$ secondes.

4.1 A propose des scores

Tout d'abord, nous présentons les méthodes de notation que nous avons utilisées pour évaluer les résultats expérimentaux. Pour des raisons de simplicité, nous continuons à considérer que nous n'avons que des problèmes de minimisation.

Étant donné un ensemble $\mathcal{I}$ d'instances et un ensemble $\mathcal{S}$ de solveurs, $b^t_{i,s}$ correspond à la meilleure borne (c'est-à-dire la plus basse) obtenue par le solveur $s \in \mathcal{S}$ sur l'instance $i \in \mathcal{I}$ au temps t, où $t \in [0, ..., T]$ et T est le timeout. Nous avons également un booléen $e^t_{i,\mathcal{S}}$ dont la valeur est *vrai* lorsqu'une solution a été trouvée au temps t par au moins un solveur de $\mathcal{S}$. Nous pouvons maintenant définir trois valeurs spécifiques :

$$\texttt{min}^t_i = \begin{cases} \min_{s \in S} b^t_{i,s}, & \text{si } e^t_{i,\mathcal{S}} \\ 0, & \text{sinon} \end{cases} \tag{5}$$

$$\texttt{max}^t_i = \begin{cases} \max_{s \in S} b^t_{i,s}, & \text{si } e^t_{i,\mathcal{S}} \\ 0, & \text{sinon} \end{cases} \tag{6}$$

$$\texttt{def}^t_i = b^t_{i,\texttt{def}} \tag{7}$$

où $b^t_{i,s}$ est égale à $\texttt{min}^t_i$ si $e^t_{i,\{s\}}$ est faux.

Les deux premières expressions correspondent respectivement à la plus petite (meilleure) et à la plus grande (pire) borne obtenue par un solveur sur une instance donnée i au temps t. La troisième expression est simplement la borne obtenue par le solveur par défaut sur i au temps t. Le *solveur par défaut* est le solveur original utilisant son comportement par défaut (et donc, n'utilisant aucune politique ABD).

Ensuite, nous pouvons calculer deux récompenses différentes pour une paire (i, s) :

$$r^t_{i,s} = \begin{cases} 0, & \text{si } \neg e^t_{i,\{s\}} \\ 1 - \frac{b^t_{i,s} - \texttt{min}^t_i}{\texttt{max}^t_i - \texttt{min}^t_i}, & \text{si } \texttt{max}^t_i \neq \texttt{min}^t_i \\ 1, & \text{sinon} \end{cases} \tag{8}$$

$$r'^t_{i,s} = \begin{cases} -\mathbb{1}_{e^t_{i,\{\texttt{def}\}}}, & \text{si } \neg e^t_{i,\{s\}} \\ -\frac{b^t_{i,s} - \texttt{def}^t_i}{\texttt{max}^t_i - \texttt{min}^t_i}, & \text{si } \texttt{max}^t_i \neq \texttt{min}^t_i \\ \mathbb{1}_{\neg e^t_{i,\{\texttt{def}\}}}, & \text{sinon} \end{cases} \tag{9}$$

où $\mathbb{1}_\alpha$ retourne 1 si α est vrai, sinon 0.

La première fonction de récompense (Équation 8) correspond à une normalisation classique *min-max* avec des valeurs manquantes possibles. Dans le cas où un solveur n'a trouvé aucune solution, sa récompense est de 0, et dans le cas où les bornes, la plus petite et la plus haute, sont égales (ce qui signifie que le solveur a trouvé l'unique solution connue au temps t), sa récompense est de 1. Sinon, on utilise *min-max* en prenant le complément à 1 du résultat (car la *borne inférieure* est la meilleure).

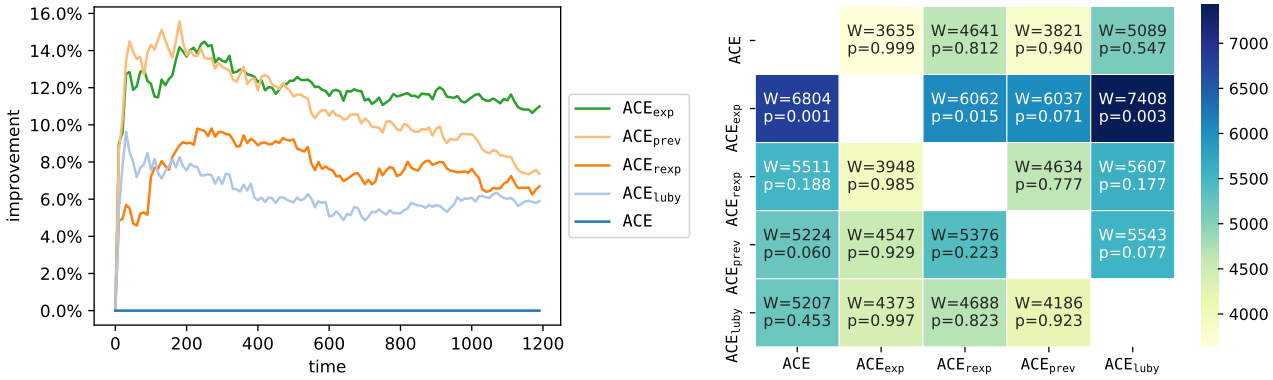
La deuxième fonction de récompense (Équation 9) est dérivée de la première. Au lieu de soustraire $\texttt{min}^t_i$ de $b^t_{i,s}$, nous soustrayons $\texttt{def}^t_i$: cela permet une comparaison plus fine et plus facile avec le solveur de base *default*. En effet, cette récompense montre la capacité de chaque solveur à être respectivement meilleur ou pire que le solveur par défaut avec une récompense positive ou négative. Le solveur par défaut reçoit toujours 0 (considéré comme une valeur neutre) comme récompense. Si un solveur n'a pas trouvé de solution, sa récompense est soit -1 si le solveur par défaut a trouvé une solution, soit 0. Dans le cas où la plus petite et la plus grande borne trouvée sont différentes, nous appliquons le calcul *min-max* où $\texttt{min}^t_i$ est remplacé par $\texttt{def}^t_i$ au numérateur : le résultat étant compris entre -1 et 1, la valeur opposée est considérée comme cohérente avec le but de minimiser la fonction objectif. La dernière condition dépend de la valeur de $e^t_{i,\{\texttt{def}\}}$: si le solveur par défaut a trouvé l'unique borne trouvée, il reçoit 0. Ce n'est pas un problème car nous savons que, dans notre implémentation, la même première solution est toujours trouvée (avec ou sans l'utilisation d'une politique ABD) ; et donc lorsqu'une solution unique a été trouvée, tous les solveurs reçoivent systématiquement 0.

Enfin, la récompense moyenne (basée sur r') de chaque solveur s au temps t est définie comme suit :

$$R^t_s = \frac{1}{|\mathcal{I}|} \times \sum_{i \in \mathcal{I}} r'^t_{i,s} \tag{10}$$

Cette dernière équation sera utile pour dessiner des graphiques (par exemple, Figure 1a) représentant la progression moyenne dans le temps de tout solveur par rapport au solveur par défaut. En raison de la manière dont il est défini, le solveur par défaut correspond à $y = 0$. Ainsi, les solveurs dont la courbe est située au-dessus de $y = 0$ peuvent être considérés comme meilleurs que le solveur par défaut, contrairement à ceux dont la courbe est sous à $y = 0$.

D'autre part, le test du rang signé de Wilcoxon [18] est une autre façon de comparer les solveurs. Le test de Wilcoxon est un test d'hypothèse statistique non paramétrique, ce qui signifie que nous n'avons pas besoin



(a) Impact des politiques ABD sur `ACE`

(b) Carte chaude de Wilcoxon

FIGURE 1 – Comparaison des politiques ABD Ψ

de comparer des échantillons respectant une distribution normale. Ce test est une métrique permettant de mesurer la différence, et la signification de la différence, entre n'importe quelle paire de solveurs. Il teste l'hypothèse nulle selon laquelle deux échantillons proviennent de la même distribution.

Pour appliquer le test de Wilcoxon sur une paire distincte de solveurs (s, s'), nous procédons comme suit :

1. pour chaque instance $i \in \mathcal{I}$, nous calculons la distance entre les récompenses des solveurs s et s' : $d_i = |r^t_{i,s} - r^t_{i,s'}|$
2. avec w_i désignant le rang de chaque instance ordonnée par les valeurs de d_i, nous appliquons le test statistique suivant :

$$\mathcal{W}_{s,s'} = \sum_{i \in \mathcal{I}} \Big[\texttt{sgn}(r^t_{i,s} - r^t_{i,s'}) \times w_i \Big] \qquad (11)$$

où t est le temps fixé à $1,200$ secondes (le timeout) et `sgn` est la fonction signée qui extrait le signe d'un nombre réel : 1 si le nombre est positif, sinon -1.

Nous avons utilisé la fonction de Wilcoxon de Python [3] incluant le calcul de la *p-value* (critère de signification).

La p-value doit être aussi proche que possible de 0 afin d'écarter l'hypothèse nulle. Dans notre cas, nous voulons vérifier l'hypothèse que s est meilleur que s', et donc, l'hypothèse nulle est : « s n'est pas meilleur que s' ».

Nous présentons les résultats du test de Wilcoxon pour chaque paire de solveurs au moyen d'une carte chaude (par exemple, la Figure 1b). Pour lire la carte chaude, nous devons considérer un solveur s le long de l'axe des ordonnées, un solveur s' le long de l'axe des abscisses. Si la valeur p ($p=$) dans la cellule pour (s, s') est proche de 0, nous pouvons rejeter l'hypothèse nulle, et donc admettre que s est significativement meilleur que s'. Plus le score $\mathcal{W}$ est élevé (Équation 11), plus la différence entre les deux solveurs est importante.

4.2 Optimisation sous contraintes

Notre approche a été évaluée sur un large éventail de problèmes d'optimisation provenant de la distribution XCSP [4, 9]. Nous avons utilisé deux benchmarks : un premier, appelé $\mathcal{I}_{\texttt{light}}$, correspondant à l'ensemble des instances COP de la compétition XCSP18, résultant en 22 problèmes et 362 instances, et un second, appelé $\mathcal{I}_{\texttt{full}}$, étendant $\mathcal{I}_{\texttt{light}}$ avec 27 problèmes supplémentaires et 331 instances provenant des compétitions XCSP17 et XCSP19 (et intégrant quelques nouveaux problèmes PyCSP[3]).

Notre campagne expérimentale est composée de deux parties concernant $\mathcal{I}_{\texttt{light}}$ et d'une troisième partie concernant $\mathcal{I}_{\texttt{full}}$. Nous montrons d'abord sur $\mathcal{I}_{\texttt{light}}$ les résultats expérimentaux concernant notre ensemble primaire de politiques ABD Ψ, ainsi que quelques variantes de Ψ (définies dans un sous-ensemble Ψ'). Le comportement des politiques les plus efficaces est ensuite présenté sur $\mathcal{I}_{\texttt{full}}$.

4.2.1 Étude de Ψ.

La Figure 1 nous permet de comparer le comportement respectif de `ACE` sans et avec les politiques ABD (de Ψ). En lien avec la Section 4.1, la Figure 1a montre, en fonction du temps, une comparaison directe des politiques ABD par rapport au solveur par défaut (rappelons que `ACE` par défaut est représentée par $y = 0$). Nous pouvons tout d'abord observer un

3. La fonction utilisée est `scipy.stats.wilcoxon` avec le paramètre `alternative = 'greater'` de `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html`

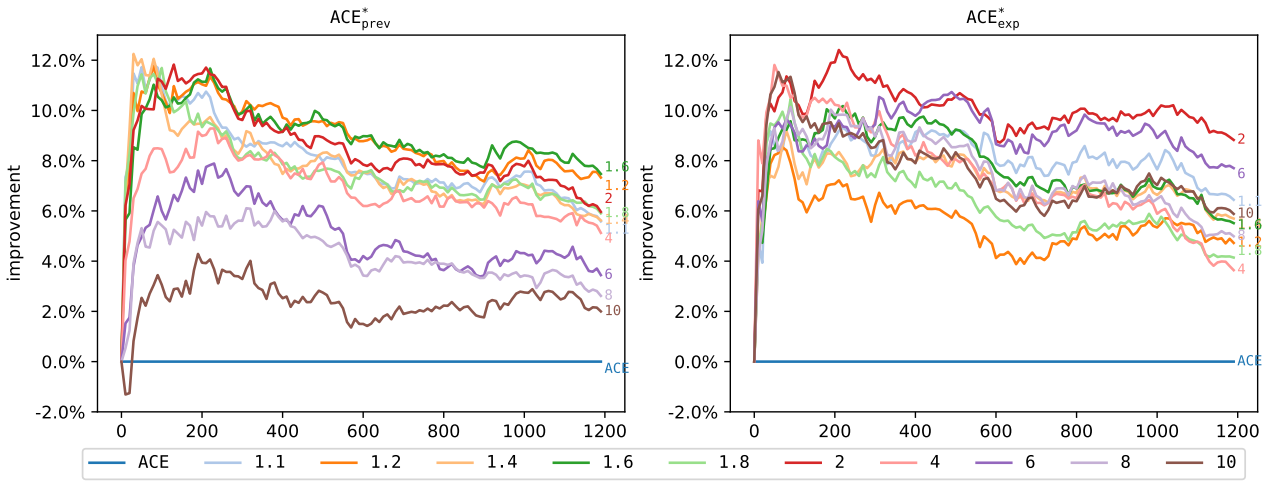


FIGURE 2 – Comparaison des politiques ABD sur Ψ'

intérêt net à utiliser les politiques ABD pendant les 200 premières secondes : une amélioration de 8% (pour les pires politiques) à 14 - 16% (pour $\texttt{ACE}_{\texttt{exp}}$ et $\texttt{ACE}_{\texttt{prev}}$). Nous pouvons également constater que $\texttt{ACE}_{\texttt{exp}}$ reste à (un niveau d'amélioration de) environ 12%, tandis que $\texttt{ACE}_{\texttt{prev}}$ diminue à 8%.

La Figure 1b montre la carte chaude construite à partir du test de Wilcoxon. Rappelons que chaque cellule correspond à un test de Wilcoxon entre une paire de solveurs. Prenons la cellule supérieure la plus à gauche avec $p = 0.1\%$: le test de Wilcoxon renvoie un score $\mathcal{W}_{\texttt{ACE}_{\texttt{exp}},\texttt{ACE}} = 6,804$ avec une p-value très proche de 0. En d'autres termes, le score actuel $\mathcal{W}$ est fortement soutenu par la réfutation de l'hypothèse nulle qui est « $\texttt{ACE}_{\texttt{exp}}$ n'est pas meilleur que ACE ». Notez que $\texttt{ACE}_{\texttt{exp}}$ surpasse les autres politiques lorsque nous les comparons directement par la p-value (voir la ligne $\texttt{ACE}_{\texttt{exp}}$). Comme deuxième choix, on pourrait envisager de sélectionner $\texttt{ACE}_{\texttt{rexp}}$ car son score est plutôt bon (par rapport à ACE), mais sa p-value est trop élevée pour rejeter l'hypothèse nulle. Par conséquent, comme le confirme la Figure 1a, $\texttt{ACE}_{\texttt{prev}}$ semble être un meilleur second choix car il présente une p-value raisonnable.

En résumé, les deux meilleures politiques ABD sont $\texttt{ACE}_{\texttt{exp}}$ et $\texttt{ACE}_{\texttt{prev}}$, dont l'agressivité durant les (200) premières secondes est payante (tout en restant robuste après ces trois premières minutes d'exécution).

4.2.2 Étude de Ψ'.

Certaines variantes de ces deux meilleures politiques, $\texttt{ACE}_{\texttt{exp}}$ et $\texttt{ACE}_{\texttt{prev}}$, sont maintenant considérées.

Commençons par généraliser, avec l'introduction d'un paramètre r, les suites originales sous-jacentes de ces deux politiques (jusqu'à présent, r a été fixé à la valeur 2). Nous pouvons alors définir un nouvel ensemble Ψ' de politiques ABD :

$$\texttt{exp}(i, r) = r^{i-1} \tag{12}$$

$$\texttt{prev}(i, r) = \begin{cases} 1, & \text{si } i = 1 \\ \lceil G_{i-1} \times r \rceil, & \text{sinon} \end{cases} \tag{13}$$

$$\Psi' = \left\{ \texttt{abd}_s^r \,\middle|\, (s, r) \in \{\texttt{exp}, \texttt{prev}\} \times \{1.1, 1.2, 1.4, 1.6, 1.8, 2, 4, 6, 8, 10\} \right\} \tag{14}$$

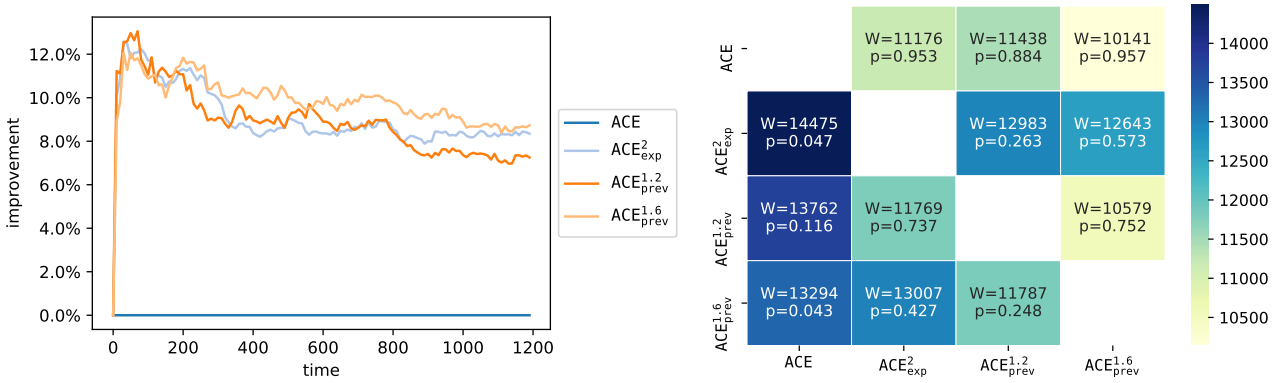
$\texttt{ACE}_s^r$ est le solveur ACE utilisant la suite $\texttt{abd}_s^r$ pour définir sa politique ABD.

La Figure 2 montre le comportement de certaines variantes de $\texttt{ACE}_{\texttt{prev}}$ (figure de gauche) et de $\texttt{ACE}_{\texttt{exp}}$ (figure de droite). Les paramètres (r) utilisés pour les suites sont indiqués sous les figures. On peut observer que l'utilisation de n'importe quelle variante de Ψ' rend le solveur plus robuste, comparé au ACE par défaut (sauf pour les premières secondes de $\texttt{ACE}_{\texttt{prev}}^{10}$). Concernant $\texttt{ACE}_{\texttt{exp}}$, la meilleure variante correspond à $r = 2$, notre valeur initiale. Concernant $\texttt{ACE}_{\texttt{prev}}$, les meilleures variantes sont $\texttt{ACE}_{\texttt{prev}}^{1.2}$ et $\texttt{ACE}_{\texttt{prev}}^{1.6}$ qui dominent nettement pendant $1,200$ secondes. Les variantes les plus agressives $\texttt{ACE}_{\texttt{prev}}^{6}$, $\texttt{ACE}_{\texttt{prev}}^{8}$ et $\texttt{ACE}_{\texttt{prev}}^{10}$ sont moins efficaces.

Après cette nouvelle expérience, nous proposons de conserver trois principales politiques, car elles restent plutôt robustes dans le temps : $\texttt{ACE}_{\texttt{exp}}^{2}$, $\texttt{ACE}_{\texttt{prev}}^{1.2}$ et $\texttt{ACE}_{\texttt{prev}}^{1.6}$.

4.2.3 Étude de Ψ''.

Nous considérons maintenant le benchmark $\mathcal{I}_{\texttt{full}}$, et la sélection Ψ'' :



(a) Impact des politiques ABD sur `ACE`

(b) Carte chaude de Wilcoxon

FIGURE 3 – Comparaison des politiques ABD Ψ''

$$\Psi'' = \left\{ \texttt{abd}^{1.2}_{\texttt{prev}}, \texttt{abd}^{1.6}_{\texttt{prev}}, \texttt{abd}^{2}_{\texttt{exp}} \right\} \tag{15}$$

La Figure 3 nous permet de comparer, sur $\mathcal{I}_{\texttt{full}}$, le comportement de `ACE` sans et avec les meilleures politiques ABD (de Ψ'').

La tendance observée sur $\mathcal{I}_{\texttt{light}}$ reste la même sur $\mathcal{I}_{\texttt{full}}$, mais avec des performances légèrement inférieures. En effet, les trois politiques bénéficient de leur agressivité avec une amélioration de 12 à 13 pour-cent au début, avant une stagnation située autour de 8 à 10 pour-cent ; voir Figure 3a. Bien que $\texttt{ACE}^2_{\texttt{exp}}$ soit légèrement meilleur au tout début, $\texttt{ACE}^{1.6}_{\texttt{exp}}$ obtient les meilleures performances parmi les trois politiques ABD.

La Figure 3b confirme les précédentes observations. En effet, lorsque nous comparons chaque politique ABD au solveur par défaut de `ACE`, il apparaît que $\texttt{ACE}^{1.6}_{\texttt{prev}}$ a la meilleure p-value confirmant le rejet de l'hypothèse nulle. Avec une p-value légèrement supérieure et un meilleur score de Wilcoxon, $\texttt{ACE}^2_{\texttt{exp}}$ présente les meilleures performances. Enfin, bien que $\texttt{ACE}^{1.2}_{\texttt{prev}}$ montre de bonnes performances, sa p-value élevée ($> 10\%$) nous empêche de tirer des conclusions claires sur son intérêt.

En résumé, notre préférence va à $\texttt{ACE}^{1.6}_{\texttt{prev}}$ et à $\texttt{ACE}^2_{\texttt{exp}}$: $\texttt{ACE}^{1.6}_{\texttt{prev}}$ a de bonnes performances (et notamment, une p-value cohérente liée au test de Wilcoxon), et $\texttt{ACE}^2_{\texttt{exp}}$ est assez simple et non intrusif (sans contexte, car il n'y a pas besoin de gérer la séquence des gains liés).

4.2.4 Analyse par famille

Par manque d'espace, les figures ne sont pas affichées et sont brièvement décrites. Cette analyse consiste en une étude par famille de problème de la dernière campagne Ψ'' sur $\mathcal{I}_{\texttt{full}}$. Celle-ci met en valeur de très larges améliorations sur le problème *Fapp*, où une amélioration de 60 à 80 pour-cent est observée pour chacune des deux politiques $\texttt{ACE}_{\texttt{prev}}$. Seuls deux cas montrent que les politiques ABD produisent de moins bonnes performances que le solveur par défaut : *QueenAttacking* et *SteelMillSlab*. Pour les problèmes restants, l'usage d'ABD améliore la performance du solveur avec une amélioration approchant parfois les 40%. Aussi, nous remarquons toujours pour chaque problème, qu'ABD montre de plus claires performances dans les premiers temps d'exécution.

4.3 Optimisation Pseudo-Booléenne

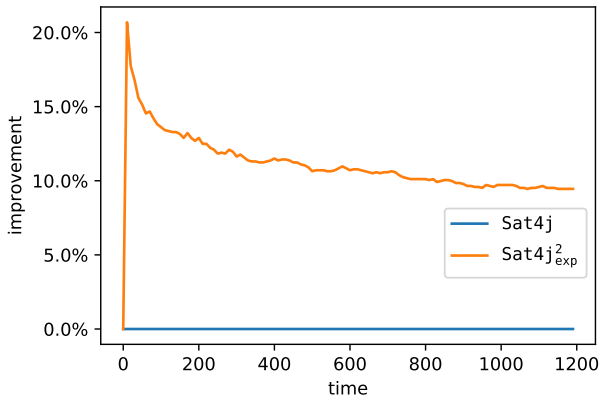


FIGURE 4 – Impact d'ABD sur `Sat4j`

Une dernière expérimentation montre les performances de l'implantation d'ABD sur le solveur Pseudo-Booléen `Sat4j`. Nous résumons, ici aussi, très brièvement les expérimentations et omettons la description du problème d'optimisation Pseudo-Booléen [2]. La Figure 4 montre le résultat de notre expérience avec le solveur par défaut `Sat4j` et le solveur $\texttt{Sat4j}^2_{\texttt{exp}}$, où $\texttt{Sat4j}^2_{\texttt{exp}}$

correspond à l'intégration d'une politique ABD basée sur la suite $\texttt{abd}^2_{\texttt{exp}}$. La Figure 4 affiche la même tendance que celle que nous avons déjà observée pour les solveurs COP : une forte croissance sur les 100 premières secondes (environ 20% d'amélioration), avant un gain plus modeste (environ 10% d'amélioration) à l'approche de la limite de temps.

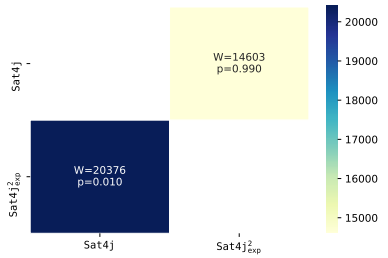


FIGURE 5 – Carte chaude de Wilcoxon

Ce résultat est confirmé par le test de Wilcoxon (Figure 5) où $\texttt{Sat4j}^2_{\texttt{exp}}$ a une p-value proche de 0 et un score plus élevé que le solveur par défaut.

5 Conclusion

Dans cet article, nous avons introduit ABD (*Aggressive Bound Descent*) qui est une technique modifiant de manière agressive la borne de la contrainte objectif. En prenant le risque d'exécuter périodiquement le solveur dans des parties non sûres de l'espace de recherche, nous montrons que des performances expérimentales intéressantes peuvent être obtenues sur des problèmes d'optimisation à la fois sous contraintes et Pseudo-Booléens. Cette politique permet d'obtenir plus rapidement de meilleures bornes que l'approche par défaut de recherche avec retour en arrière sur certains problèmes fortement structurés. Néanmoins, nous pensons que certains raffinements d'ABD pourraient être étudiés, comme par exemple, exploiter davantage l'historique des gains de bornes, ou identifier les suite pertinentes d'écarts de bornes à utiliser en fonction de la structure des problèmes.

Références

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

[2] Endre Boros et Peter L. Hammer : Pseudo-boolean optimization. *Discrete Applied Mathematics*, 123(1):155 – 225, 2002.

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

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

[5] E. Demirovic, G. Chu et 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.

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

[7] Daniel Le Berre et Anne Parrain : The sat4j library, release 2.2. *JSAT*, 7:59–6, 01 2010.

[8] C. Lecoutre, L. Sais, S. Tabary et V. Vidal : Reasonning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

[9] C. Lecoutre et N. Szczepanski : PyCSP^3 : Modeling combinatorial constrained problems in Python. Rapport technique, CRIL, 2020. Available from `https://github.com/xcsp3team/pycsp3`.

[10] M. Luby, A. Sinclair et D. Zuckerman : Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

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

[12] P. Shaw : Using constraint programming and local search methods to solve vehicle routing problems. *In Proceedings of CP'98*, pages 417–431, 1998.

[13] W.J. van Hoeve et M. Milano : Postponing branching decisions. *In Proceedings of ECAI'04*, pages 1105–1106, 2004.

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

[15] 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.

[16] T. Walsh : Search in a small world. *In Proceedings of IJCAI'99*, pages 1172–1177, 1999.

[17] H. Wattez, C. Lecoutre, A. Paparrizou et S. Tabary : Refining constraint weighting. *In Proceedings of ICTAI'19*, pages 71–77, 2019.

[18] F. Wilcoxon : Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

Contrainte de sac-à-dos à choix multiples dans les réseaux de fonctions de coûts

Pierre Montalbano[1*] **Simon de Givry**[1] **George Katsirelos**[2]

[1] Université Fédérale de Toulouse, ANITI, INRAE, UR 875, Toulouse, France
[2] Université Fédérale de Toulouse, ANITI, INRAE, MIA Paris, AgroParisTech,France
{pierre.montalbano,simon.de-givry}@inrae.fr gkatsi@gmail.com

Résumé

Les réseaux de contraintes pondérées (Cost Function Networks (CFNs)), aussi connus sous le nom de problèmes de satisfaction de contraintes pondérées (Weighted Constraint Satisfaction Problems (WCSPs)), peuvent représenter des fonctions décomposables de manière compacte, favorisant l'efficacité des algorithmes d'inférences. En particulier, la majorité des méthodes cherchant à calculer une borne duale (inférieur) pour minimiser un critère, passe par une solution faisable du dual de la relaxation linéaire. Ces méthodes se montrent plus efficaces que de résoudre la relaxation linéaire exactement, cependant elles sont exclusivement adaptées à la structure des relaxations linéaires des CFNs. Elles ne peuvent donc pas gérer les contraintes dont l'expression en extension est impossible, comme par exemple les contraintes linéaires de grande arité. Dans ce travail, nous montrons comment étendre les algorithmes de cohérences locales, de manière à pouvoir traiter des contraintes linéaires ainsi que des contraintes linéaires associées à des contraintes At Most One. Nous avons implémenté cet algorithme dans le solveur Toulbar2, un solveur exact de CFNs basé sur un algorithme de Séparation et Évaluation et sur la cohérence locale EDAC pour calculer les minorants pendant la recherche. Cela nous a permis de tester ce solveur sur différents problèmes qui jusqu'alors était hors de portée pour Toulbar2 comme les problèmes d'optimisation Pseudo-booléenne (PB) ou le problème du sac-à-dos avec un graphe de conflits (KPCG). Enfin nouc comparons nos résultats avec des solveurs PB et des solveurs de programmation linéaire en nombre entier, nous montrons des performances compétitives sur certaines familles des compétitions PB récentes et sur des instances KPCG. Nous améliorons également le calcul des séquences de solutions diverses pour le problème de conception de protéines (Computational Protein Design).

Abstract

Cost function networks (CFNs), also known as Weighted Constraint Satisfaction Problems, can compactly express large decomposable cost functions, which leads to efficient inference algorithms. In particular, most methods for computing dual (lower) bounds in minimization compute feasible dual solutions of a specific linear relaxation. These methods are more effective than solving the linear relaxation exacly, because its size is significantly larger than the original CFN. However, these algorithms are specialized to the structure of the linear relaxation of a CFN and cannot, for example, deal with constraints that cannot be expressed in extension, such as linear constraints of large arity.

In this work, we show how to extend soft local consistencies, a set of approximate inference techniques for CFNs, so that they handle linear constraints, as well as combinations of linear constraints with at-most-one constraints. We embedded the resulting algorithm in toulbar2, an exact Branch-and-Bound solver for CFNs which primarily relies on EDAC for dual bounds during search. This allows us to test this solver on problems on which it was previously not applicable, such as pseudo-Boolean optimization (PB) and knapsack with conflict graph (KPCG). Compared to dedicated PB solvers and integer linear programming solvers, we demonstrate state-of-the-art performance on some families from recent PB optimization evaluations and KPCG, and improve performance in computational protein design with diversity guarantees.

*Papier doctorant : Pierre Montalbano[1] est auteur principal.

1 Contexte

Definition 1. *Un problème de satisfaction de contraintes pondérées (WCSP) est un tuple* $(\boldsymbol{X}, \boldsymbol{D}, \boldsymbol{C}, \top)$ *où* $\boldsymbol{X}$ *est un ensemble de variables discrètes, de domaine* $\boldsymbol{D}(x)$ *pour* $x \in \boldsymbol{X}$. $\boldsymbol{C}$ *est un ensemble de fonctions de coûts, de portée* $scope(c) \subseteq \boldsymbol{X}$ *pour* $c \in \boldsymbol{C}$. $\top$ *définit un coût maximal indiquant une*

affectation interdite.

La taille de la portée est appelée arité. Une affectation partielle notée τ est une affectation de toutes les variables $x_i \in scope(\tau)$ à une valeur de leur domaine $\boldsymbol{D}(x_i)$. L'ensemble des affectations partielles d'une portée $\boldsymbol{S}$ est noté $\tau(\boldsymbol{S})$. L'évaluation du coût d'une affectation partielle τ par une contrainte $c_{\boldsymbol{S}}$ est notée $c(\tau|_{\boldsymbol{S}})$ avec $\boldsymbol{S} \subseteq scope(\tau)$. Dans le cadre du WCSP tous les coûts considérés sont des entiers positifs bornés par $\top$, de ce fait si $c(\tau|_{\boldsymbol{S}}) = \top$ alors l'affectation τ ne constitue pas une solution faisable. Une contrainte $c_{\boldsymbol{S}}$ est dite dure si pour toute affectation partielle τ, $c_{\boldsymbol{S}}(\tau) \in \{0, \top\}$, sinon on dit que la contrainte est souple. Un WCSP composé uniquement de contraintes dures est un CSP. Dans la suite les termes *fonction de coûts* et *contrainte* sont équivalents. Une affectation complète est donnée par $scope(\tau) = \boldsymbol{X}$ et le coût d'une affectation complète τ est donné par $c(\tau) = \sum_{c_{\boldsymbol{S}} \in \boldsymbol{C}} c(T|_{\boldsymbol{S}})$. Résoudre un WCSP revient à trouver l'affectation complète minimisant $c(\tau)$. Ce problème est défini comme NP-dur [32].

Les WCSP sont habituellement résolus avec un algorithme de Séparation et Évaluation (Brand-and-Bound ($B\&B$)). A chaque noeud de l'arbre issu du $B\&B$, le solveur calcule une borne et ferme le noeud si cette borne est plus grande que la solution courante ou si le noeud définit un problème infaisable. Les algorithmes présentés ici utilisent la fonction de coût $c_\varnothing$ pour représenter un minorant naturel du WCSP et cherchent une re-paramétrisation du problème augmentant $c_\varnothing$. Une re-paramétrisation P' d'un WCSP P est un WCSP de structure identique (contraintes de même portée) mais dont les coûts peuvent localement être différents, cependant $c_P(\tau) = c_{P'}(\tau)$ pour toute affectation complète τ. Il sera possible d'augmenter le minorant $c_\varnothing$ s'il existe une fonction de coûts $c_{\boldsymbol{S}}$ telle que pour tout $\tau \in \tau(\boldsymbol{S})$, $c_{\boldsymbol{S}}(\tau) > 0$. Toutes les re-paramétrisations considérées sont obtenues par une séquence de transformations préservant l'équivalence (en anglais, *Equivalence Preserving Transformation* (EPT)) menant à l'augmentation de $c_\varnothing$. Soit deux fonctions de coûts $c_{\boldsymbol{S}_1}, c_{\boldsymbol{S}_2}$ avec $\boldsymbol{S}_1 \subset \boldsymbol{S}_2$. La procédure MoveCost décrit comment déplacer les coûts entre ces 2 fonctions de coûts. On peut observer que si τ est utilisé dans une affectation complète, alors il y a exactement une extension de τ vers $\boldsymbol{S}_2$. De ce fait le coût de l'affectation complète est inchangé indépendemment de l'attribution du coût α à τ ou à toutes les extensions τ' dans $\boldsymbol{S}_2$. Dans la suite si α est positif, le mouvement s'opère d'une fonction d'arité plus grande vers une fonction d'arité plus faible et cette opération est appelée projection, notée $project(c_{\boldsymbol{S}_1}, c_{\boldsymbol{S}_2}, \tau_1, \alpha)$. Sinon si α est négatif, le coût se déplace vers une fonction d'arité plus grande et on appelle cette opération extension, notée $extend(c_{\boldsymbol{S}_1}\tau_1, c_{\boldsymbol{S}_2}, -\alpha)$. Le mouvement de $|\boldsymbol{S}_2| = 1$ vers $\boldsymbol{S}_1 = \varnothing$ est une projection unaire, notée $unaryProject(c_i, v, \alpha)$. Nous n'effectuons pas d'extension à partir de $c_\varnothing$, celui-ci ne pourra pas diminuer au cours de la recherche.

Procedure MoveCost($\boldsymbol{S}1, \boldsymbol{S}_2, \tau, \alpha$) : Déplace un coût α entre le tuple τ de la portée $\boldsymbol{S}_1$ et les tuples étendus de τ dans la portée $\boldsymbol{S}_2$

Data: Scopes $\boldsymbol{S}_1 \subset \boldsymbol{S}_2$
Data: $\tau \in \tau(\boldsymbol{S}_1)$
Data: cost α to move

$c(\tau|_{\boldsymbol{S}_1}) \leftarrow c(\tau|_{\boldsymbol{S}_1}) + \alpha$;
foreach $\tau' \in \tau(\boldsymbol{S}_2) \mid \tau'|_{\boldsymbol{S}_1} = \tau$ **do**
  $c(\tau|_{\boldsymbol{S}_2}) \leftarrow c(\tau|_{\boldsymbol{S}_2}) - \alpha$;

L'objectif est donc de trouver une séquence d'EPTs menant à l'augmentation de $c_\varnothing$, cependant trouver les opérations menant à une re-paramétrisation optimale n'est pas évident. Il a été prouvé qu'elle peut être obtenue à partir d'une solution duale optimale de la relaxation linéaire du WCSP [11], appelée *polytope local*.

$$\min \sum_{c_{\boldsymbol{S}} \in \boldsymbol{C}, \tau \in \tau(\boldsymbol{S})} c(\tau|_{\boldsymbol{S}}) \times y_\tau$$

s.t.

$$y_\tau = \sum_{\tau' \in \tau(\boldsymbol{S}_2), \tau'|_{\boldsymbol{S}_1} = \tau} y_{\tau'} \quad \forall c_{\boldsymbol{S}_1}, c_{\boldsymbol{S}_2} \in \boldsymbol{C} | \boldsymbol{S}_1 \subset \boldsymbol{S}_2, \tau \in \tau(\boldsymbol{S}_1), |\boldsymbol{S}_1| \geq 1$$

$$\sum_{\tau \in \tau(\boldsymbol{S})} y_\tau = 1 \qquad \forall c_{\boldsymbol{S}} \in \boldsymbol{C}, |\boldsymbol{S}| \geq 1$$

Cependant résoudre ce problème à l'optimalité est souvent très coûteux car même si l'on considère un WCSP uniquement composé de contraintes binaires, il y a $O(ed^2 + nd)$ variables et $O(end^2)$ valeurs non-nulles dans sa matrice, où e est le nombre de fonctions de coûts, n le nombre de variables et d la taille des domaines. De plus la structure du problème ne permet pas de définir un algorithme spécifiquement efficace, en effet il a été prouvé qu'il est aussi difficile de résoudre ce problème que de résoudre un problème linéaire arbitraire [31]. De ce fait en pratique les recherches se sont dirigées vers le calcul d'une solution faisable du problème dual. Différents algorithmes ont ainsi été proposé, certains issus du domaine du traitement d'image [22, 40, 38, 23, 37] ou encore de la programmation par contraintes [25, 11], ces derniers sont appelés *algorithmes de cohérences locales souples*. Nous ne détaillons pas précisément tous les algorithmes de cohé-

rences locales existants mais nous supposons que le WCSP vérifie la propriété suivante :

Definition 2. *Un WCSP est Node Consistent (NC) [25] si pour tout* $c_{\boldsymbol{S}} \in \boldsymbol{C}$ *avec* $|\boldsymbol{S}| = 1$ *il existe un* $\tau \in \tau(c_{\boldsymbol{S}})$ *tel que* $c_{\boldsymbol{S}}(\tau) = 0$ *et pour tout* $\tau \in \tau(c_{\boldsymbol{S}})$, $c_\varnothing + c(\tau|_{\boldsymbol{S}}) < \top$.

Definition 3. *Un WCSP est* $\emptyset$*-Inverse Consistent (*$\emptyset$*IC) [41] si pour tout* $c_{\boldsymbol{S}} \in \boldsymbol{C}$ *il existe* $\tau \in \tau(\boldsymbol{S})$ *tel que* $c(\tau|_{\boldsymbol{S}}) = 0$.

Definition 4. *Un WCSP est Existential Arc Consistent (EAC) [12] s'il est NC et si pour toute variable* $x_i \in \boldsymbol{X}$ *il existe une valeur* $v \in \boldsymbol{D}_{x_i}$ *telle que pour toute contrainte* $c_{\boldsymbol{S}} \in \boldsymbol{C}, |\boldsymbol{S}| > 1$, *il existe une affectation partielle* τ *vérifiant* $\tau|_{x_i} = v$ *et* $\sum_{x_j \in S} c_j(\tau|_{x_j}) + c(\tau|_{\boldsymbol{S}}) = 0$. *La valeur* v *est appelée support EAC.*

Cette dernière définition s'applique uniquement aux fonctions de coûts binaires [1]. Une version plus faible pouvant être appliquée aux fonctions de coûts globales est introduite dans [28] et permet notamment d'éviter le problème d'oscillation des coûts. Nous proposons ici une autre approche plus faible que EAC en nous fondant sur $\emptyset$IC. Nous renforçons la définition précédente en prenant en compte les coûts unaires.

Definition 5. *Un WCSP est Full* $\emptyset$*-Inverse Consistent (F*$\emptyset$*IC) si pour tout* $c_{\boldsymbol{S}} \in \boldsymbol{C}$ *il existe* $\tau \in \tau(\boldsymbol{S})$ *tel que* $\sum_{x_j \in S} c_j(\tau|_{x_j}) + c(\tau|_{\boldsymbol{S}}) = 0$.

F$\emptyset$IC est plus faible que T-DAC [2]. C'est aussi plus faible que EAC sur les réseaux de contraintes binaires, et incomparable avec EAC faible [28]. Il existe des algorithmes permettant d'établir les cohérences définies précédemment, ils sont basés sur une solution duale faisable du polytope locale. Cependant ils requièrent d'exprimer toutes les contraintes en extension [2], ce qui est impossible pour un grand nombre de contraintes classiques en programmation par contraintes. Ces contraintes dites *globales* peuvent impliquer un nombre arbitraire de variables, il est donc nécessaire de définir des algorithmes sur-mesure pour établir une cohérence locale souple sur un problème impliquant des contraintes globales. En plus de supprimer les valeurs n'apparaissant dans aucune solution faisable, ces algorithmes cherchent une solution optimale en utilisant les coûts unaires et une (parfois plusieurs) contraintes globales. Puis ils exploitent ce coût optimal en re-paramétrisant le problème.

Ici, nous traitons les contraintes Pseudo-Booléennes (PB). Ce sont des contraintes de la forme $\sum_{x_i \in S} w_i x_i \geq C$, où S est une portée, $x_i \in S$ sont des variables Booléennes, w_i et C sont des constantes. Les contraintes avec un opérateur $\leq$ peuvent être transformées en appliquant un facteur -1 de chaque coté de la contrainte.

Il est possible de détecter en un temps linéaire s'il existe au moins une solution, en testant si $\sum_{x_i \in S} \max(0, w_i) \geq C$. De même il est possible de vérifier que chaque valeur apparaît dans au moins une solution.

Une contrainte at-most-one (AMO) est une contrainte de la forme $\sum_{x_i \in S} x_i \leq 1$, ou $\sum_{x_i \in S} -x_i \geq -1$. C'est une contrainte exactly-one (EO) si $\sum_{x_i \in S} x_i = 1$.

2 Travaux connexes

Les problèmes définis par des contraintes linéaires pseudo-Booléennes, sont une généralisation du problème SAT. Les solveurs PB SAT utilisent des techniques d'apprentissages [29], soit en traduisant directement le problème au format CNF [17, 35], soit en généralisant le mécanisme d'apprentissage de clauses aux contraintes PBs[14, 18]. Ces solveurs ne calculent pas de minorant durant la recherche et reposent en grande partie sur l'analyse des conflits pour prouver l'optimalité du problème. Le travail de Devriendt et al [20] est une exception, en effet ils utilisent un solveur LP pour obtenir des bornes durant la recherche et apprendre des contraintes à partir d'une violation d'une borne. Ils limitent cependant le nombre d'itération du solveur LP afin de garder un temps de recherche raisonnable. C'est une différence majeure avec notre approche qui utilise une solution sub-optimale mais sans limiter les ressources déployées pour l'obtenir. Les solveurs PB peuvent aussi exploiter la présence de contraintes AMO ou EO pour renforcer la propagation des contraintes PB [5, 7].

La relaxation linéaire du polytope local permet d'utiliser les solveurs de programmation linéaire en nombre entier (Integer Linear Programming (ILP)) pour résoudre les WCSPs. De plus utilisant une méthode de résolution générale ils n'ont pas de difficulté à prendre en compte les contraintes linéaires ou des combinaisons de contraintes linéaires supplémentaires. Cependant même pour des implémentations hautement optimisées, la taille du polytope locale est trop large [19] pour une résolution efficace. En pratique les solveurs spécialisés en WCSP sont plus appropriés.

Du coté WCSP, un travail similaire a été effectué sur un cas particulier de contrainte PB appelé contrainte de clique [13], l'approche présentée ici généralise ce cas. Dlask et Werner [15, 16] ont montré comment gérer des LPs en utilisant un algorithme de descente de coordonées en bloc (BCD) et une généralisation

1. Une extension aux coûts ternaires a été définie dans [36].
2. Ou bien d'avoir un algorithme polynomial pour déterminer le tuple de coût minimum $\sum_{x_j \in S} c_j(\tau|_{x_j}) + c(\tau|_{\boldsymbol{S}})$ [2].

de VAC. Cependant malgré des avancées récentes [39], cette méthode est trop coûteuse pour être utilisée à tous les noeuds du solveur B&B.

Lee et Shum [26] ont montré que certaines contraintes globales peuvent être décrites par un ensemble de contraintes linéaires souples, cependant cela nécessite l'utilisation d'un solveur LP. Enfin il est possible de décomposer les contraintes linéaires en utilisant des fonctions de coûts d'arité 3 et des variables intermédiaires [3]. Cependant la taille des domaines des variables intermédiaires augmente linéairement avec les coefficients de la contrainte PB.

3 Contrainte pseudo-Booléenne dans le cadre WCSP

La contrainte que nous considérons ici est $\sum_{z_i \in S} w_i y_i \geq C$, associée à des contraintes AMO de portée P_i où $P_1, \ldots, P_k$ est un partitionnement des variables. De manière équivalente, soit $\boldsymbol{S}$ un ensemble de variables d'un WCSP de domaine arbitraire, et w_{iv} le poids associé à chaque valeur. Nous considérons la contrainte : $\sum_{x_i \in S, v \in \boldsymbol{D}(x_i)} w_{iv} x_{iv} \geq C$, où x_{iv} est la variable 0/1 qui prend la valeur 1 si $x_i = v$. Les variables du WCSP ne pouvant être affectées qu'à une seule valeur, elles définissent implicitement un partitionnement des variables 0/1. De plus il est possible d'associer une contrainte EO à chaque partition, en effet ajouter une variable 0/1 avec un poids nul permet de transformer une contrainte AMO en contrainte EO.

Nous voulons établir un niveau de cohérence équivalent à $F\emptyset IC$. Cependant c'est un problème NP-difficile, en effet minimiser la somme des coûts unaires tout en cherchant à vérifier une contrainte linéaire définit le problème du sac-à-dos (avec les signes inversés). De ce fait, nous n'établissons pas complètement $F\emptyset IC$, nous détectons simplement certains cas où la contrainte n'est pas $F\emptyset IC$ et procédons à des mouvements de coûts. Nous conservons une trace des différents mouvements de coûts en associant chaque variable x_{iv} à une quantité δ_{iv} et $c_\varnothing$ à $\delta_\varnothing$. Si l'EPT $project(c_i, c_{PB}, \{x_i = v\}, \alpha)$ est effectuée alors $\delta_{iv} \rightarrow \delta_{iv} - \alpha$, si l'EPT $extend(c_i, \{x_i = v\}, c_{PB}, \alpha)$ est effectuée alors $\delta_{iv} \rightarrow \delta_{iv} + \alpha$. Initialement $\delta_\varnothing = 0$ et $\delta_{iv} = 0$ pour tout i, v.

$$\min \sum_i \sum_v \delta_{iv} x_{iv} - \delta_0 \tag{1}$$

$$s.t.$$

$$\sum_i \sum_v w_{iv} x_{iv} \geq C \tag{2}$$

$$\sum_v x_{iv} \leq 1, \quad \forall i \tag{3}$$

$$x_{iv} \in \{0, 1\}, \quad \forall i \tag{4}$$

Nous appelons cette formulation $ILP_\emptyset$. A tout moment de la recherche les coûts δ permettent d'évaluer le coût d'un tuple sur la contrainte linéaire. De ce fait si $opt(ILP_\emptyset) > 0$, la contrainte n'est pas $\emptyset IC$ et il est possible d'augmenter $c_\varnothing$. Cependant, pour l'établissement de $F\emptyset IC$ il est nécessaire de prendre en compte les coûts unaires. De ce fait, bien que $ILP_\emptyset$ représente la contrainte, notre propagateur va s'intéresser à une version modifiée.

$$\min \sum_i \sum_v (\delta_{iv} + c_i(v)) x_{iv} - \delta_0 \tag{5}$$

Nous appelons ce problème $ILP_{F\emptyset}$. La contrainte est $F\emptyset IC$ si $opt(F\emptyset IC) = 0$. Dans la suite nous utilisons $p_{iv} = \delta_{iv} + c_i(v)$.

Les 2 problèmes $ILP_\emptyset$ et $ILP_{F\emptyset}$ étant NP-difficiles, nous relâchons la contrainte d'intégrité (4) : $0 \leq x_{iv} \leq 1$ et résolvons les programmes linéaire obtenus, respectivement $LP_\emptyset$ et $LP_{F\emptyset}$. Avec cette relaxation, nous savons que si $opt(LP_{F\emptyset}) > 0$ alors la contrainte n'est pas $F\emptyset IC$, de même avec $LP_\emptyset$ et $\emptyset IC$.

$LP_{F\emptyset}$ est un problème équivalent (exprimé en minimisation au lieu de maximisation) au problème de sac à dos à choix multiples (MCKP, [30]), ou encore du *knapsack problem with special ordered sets* [21].

3.1 Problème dual et solution optimale

On obtient une solution optimale $\boldsymbol{x}^*$ du problème $LP_{F\emptyset}$ en appliquant l'algorithme décrit par Pisinger [30]. Cet algorithme donne la solution optimale $\boldsymbol{x}^*$ en $O(n \log n)$ où n est le nombre de variables 0/1. De plus soit $\boldsymbol{x}^*$ est une solution entière soit il contient exactement deux valeurs fractionnelles, dans ce cas la variable $x_k \in \boldsymbol{X}$ vérifiant $\exists (v, v')$ tel que $0 < x^*_{kv}, x^*_{kv'} < 1$ se nomme *split class* et x^*_{kv}, x^*_{kv+1} sont les *splits variables*. Nous notons par o le coût optimal de la solution $\boldsymbol{x}^*$ $o = \sum_i \sum_v p_{iv} x^*_{iv}$. Le début de l'exemple 1 illustre l'utilisation de l'algorithme de Pisinger.

A présent, si l'on regarde le problème dual de $LP_{F\emptyset}$:

$$\begin{aligned} \max C \times y_{cc} + \sum_i y_i \quad & (6) \\ s.t. & \\ \forall i, v \quad & y_{cc} \times w_{iv} + y_i \leq p_{iv} \\ y_{cc} \geq 0 & \end{aligned}$$

Où y_{cc} est la variable duale correspondant à la contrainte de capacité et y_i correspond à la contrainte EO de x_i. En connaissant la solution optimale primale, il est aisé de calculer la solution optimale duale. Soit x_k la split class, $x_{ks}, x_{ks'}$ les splits variables et pour $i \neq k$, définissons la variable x_{is} comme la variable utilisée dans la solution optimale, *i.e.*, $x^*_{is} = 1$.

- $y_{cc} = \frac{p_{ks} - p_{ks'}}{w_{ks} - w_{ks'}}$
- $y_k = p_{ks} - y_{cc} \times w_{ks} = p_{ks'} - y_{cc} \times w_{ks'}$
- Pour $i \neq k$, $y_i = p_{is} - y_{cc} \times w_{is}$

La solution optimale duale va nous permettre de calculer les coûts réduits de chacune des variables x_{iv}.

Definition 6. *Le coût réduit d'une variable x sous une solution duale $\mathbf{y}$ est le slack de la contrainte duale correspondant à x, noté $cr^{\mathbf{y}}(x_{iv})$ ou $cr(x_{iv})$ si la solution duale est implicite.*

Le coût réduit de x peut être interprété comme le montant dont on doit diminuer le coefficient de x dans la fonction objective afin d'avoir $x > 0$ dans la solution optimale. Ces coûts vont nous permettre de définir notre re-paramétrisation. Dans le cas spécifique de $LP_{F\emptyset}$ nous avons :

- $\forall i,\ cr(x_{is}) = 0$
- $cr(x_{ks}) = cr(x_{ks'}) = 0$
- $\forall v \neq s : cr(x_{iv}) = p_{iv} - y_{cc} \times w_{iv} - y_i$

Observation 1. *Le problème $LP'_{F\emptyset}$ identique à $LP_{F\emptyset}$ mais avec $p'_{iv} = p_{iv} - cr(x_{iv})$ vérifie $opt(LP'_{F\emptyset}) = opt(LP_{F\emptyset})$.*

Démonstration. Les coefficients des variables avec une valeur > 0 dans $\mathbf{x}^*$ ont les mêmes coefficients dans $LP_{F\emptyset}$ et $LP'_{F\emptyset}$. De ce fait la solution $\mathbf{x}^*$ a pour coût o dans $LP_{F\emptyset}$ et $LP'_{F\emptyset}$. La solution duale optimale $\mathbf{y}^*$ reste faisable dans $LP'_{F\emptyset}$, de plus la fonction objective du problème dual n'ayant pas changée $\mathbf{y}^*$ a un coût o. Donc $opt(LP'_{F\emptyset}) = o = opt(LP_{F\emptyset})$. □

L'exemple 1 illustre l'obtention des coûts réduits à partir d'une solution duale optimale.

Exemple 1. *Considérons le problème de sac à dos à choix multiples suivant :*

$$\begin{aligned} \min 40x_{11} + 55x_{12} + 85x_{13} + 47x_{21} + 95x_{22} & \\ s.t. & \\ 4x_{11} + 14x_{12} + 24x_{13} + 16x_{21} + 40x_{22} >= 40 & \\ \forall i, \sum_v x_{iv} = 1 & \\ 0 \leq x_{ij} \leq 1 & \end{aligned}$$

L'algorithme de Pisinger donne la solution optimale $x^* = \{0, 1, 0, \frac{7}{12}, \frac{5}{12}\}$ avec un coût $o = 55 + \frac{7}{12} \times 47 + \frac{5}{12} \times 95 = 122$.

Nous déduisons la solution optimale duale suivante : $y_{cc} = 2, y_1 = 55 - 2 \times 14 = 27, y_2 = 47 - 2 \times 16 = 15$ on observe bien que $40y_{cc} + y_1 + y_2 = 122$.

Nous obtenons les coûts réduits suivants : $cr(x_{12}) = cr(x_{21}) = cr(x_{22}) = 0$ et $cr(x_{11}) = 40 - 2 \times 4 - 27 = 5$, $cr(x_{13}) = 85 - 2 \times 24 - 27 = 10$, nous en déduisons que remplacer la fonction objective par la fonction objective suivante ne change pas le coût de la solution optimale :

$$\min 35x_{11} + 55x_{12} + 75x_{13} + 47x_{21} + 95x_{22}$$

On observe que la solution $\boldsymbol{x^*} = \{0, 1, 0, \frac{7}{12}, \frac{5}{12}\}$ est toujours optimale.

3.2 Propagateur de contrainte pseudo-Booléenne

Nous allons prouver ici qu'il est possible d'augmenter le minorant d'au moins o (le coût de la solution optimale $\boldsymbol{x^*}$). Notre objectif est d'étendre le moins de coûts unaires possible tout en vérifiant que $opt(LP_\emptyset) = o = opt(LP_{F\emptyset})$ afin de pouvoir projeter o sur $c_\varnothing$. La séquence d'EPTs permettant d'obtenir une augmentation de $c_\varnothing$ est obtenue grâce aux coûts réduits.

Si l'on étend ou projette la quantité $|c_i(v) - cr(x_{iv})|$ entre les coûts unaires et la contrainte, on remarque que pour les variables vérifiant $x^*_{iv} > 0$ on a $cr(x_{iv}) = 0$, on étend donc la totalité du coût unaire $c_i(v)$. Ce qui signifie que l'on a $\sum_i \sum_v \delta_{iv} x^*_{iv} = o$. Pour tout i, v, $|c_i(v) - cr(x_{iv})| = |y_i + y_{cc} \times w_{iv}) - \delta_{iv}|$, nous obtenons donc les nouveaux coûts par les opérations décrites dans l'algorithme TransformPB.

Theorem 7. *L'algorithme TransformPB définit une séquence de transformations préservant l'équivalence.*

Démonstration. Rappelons que $p_{iv} = c_i(v) + \delta_{iv}$, et $cr(x_{iv}) \geq 0$. Si $|c_i(v) - cr(x_{iv}) \geq 0$ alors l'opération $extend(c_i, \{x_i = v\}, c_{PB}, c_{iv} - cr(x_{iv}))$ est valide.

Si $c_i(v) - cr(x_{iv}) < 0$ alors cela signifie que le coût de toutes solutions x' avec $x'_{iv} = 1$ est au moins $o - c_i(v) + cr(x_{iv})$ donc l'opération $project(c_i, c_{PB}, \{x_i = v\}, -c_i(v) + cr(x_{iv}))$ est valide.

Procedure TransformPB(c_{PB}, y_{cc}, y_i, o)

Data: c_{PB} : une contrainte
Data: y_{cc}, y_i, o : une solution optimale duale $LP_{F\emptyset}$

for *all the variables* x_{iv} **do**
  $c_i(v) \leftarrow c_i(v) - y_{cc} \times w_{iv} - y_i + \delta_{iv}$;
  $\delta_{iv} \leftarrow y_{cc} \times w_{iv} + y_i$;
$c_\varnothing \leftarrow c_\varnothing + o$;
$\delta_\varnothing \leftarrow \delta_\varnothing + o$;

Finalement pour vérifier que la séquence d'EPT justifie une augmentation de $c_\varnothing$ par o, nous calculons l'optimum de $LP_\emptyset$. D'après l'observation 3.1, $opt(LP_\emptyset) = o$, nous pouvons donc projeter o sur $c_\varnothing$ et augmenter $\delta_\varnothing$ pour obtenir $opt(LP_\emptyset) = opt(LP_{F\emptyset}) = 0$. □

Nous pouvons affiner le raisonnement en nous approchant de la solution optimale entière. Pour cela nous augmentons $c_\varnothing$ par l'arrondi supérieur de o, dans ce cas il faut également prendre l'arrondi supérieur pour tous les mouvements de coûts effectués. L'utilisation des arrondis ne nous permet plus d'utiliser l'observation 3.1, mais nous vérifions toujours que $opt(LP_\emptyset) = 0$.
Notre algorithme est donné par la procédure Propagate. Nous débutons par établir une cohérence des domaines sur la contrainte PB, puis résolvons $LP_{F\emptyset}$ à l'optimalité. S'il y a plusieurs solutions nous préférons celle minimisant les coûts réduits associés au support EAC de chaque variable. Pour finir nous utilisons la procédure TransformPB. Bien que $LP_\emptyset$ est utile pour expliquer l'algorithme, il n'est jamais résolu durant la recherche.

Theorem 8. *La complexité de la procédure Propagate est $O(nd \log(nd))$ où n est le nombre de variables WCSP impliqué dans la contrainte et d la taille maximale des domaines. (Notre problème a donc nd variables booléennes).*

Démonstration. L'opération la plus coûteuse est l'application de l'algorithme de Pisinger, qui a une complexité de $O(N \log N)$, où N est le nombre de variables. Dans notre cas $N = nd$, donc l'algorithme de Pisinger est en $O(nd \log nd)$. La cohérence de domaine comme dit en introduction peut se faire en temps linéaire. Enfin, la procédure TransformPB boucle une seule fois sur toutes les variables binaires, et effectue des opérations en temps constant. De ce fait la complexité totale est $O(nd \log nd)$. □

Procedure Propagate(c_{PB})

Data: c_{PB} : une contrainte pseudo-Booléenne avec une partition en contraintes EO
DomainConsistency(c_{PB}) ;
(y_{cc}, y_i, o) = DualSolve($LP_{F\emptyset}$) ;
TransformPB(c_{PB}, y_{cc}, y_i, o) ;

Exemple 2. *En reprenant l'exemple utilisé dans la section 4.1, nous avions les coûts réduits suivants : $cr(x_{12}) = cr(x_{21}) = cr(x_{22}) = 0$, $cr(x_{11}) = 5$, $cr(x_{13}) = 10$, le coût optimal était 122. On en déduit les mouvements de coûts suivants :*
$c_1(1) \leftarrow c_1(1) - 35 = 5$, *et* $\delta_{11} \leftarrow 35$
$c_1(2) \leftarrow c_1(2) - 55 = 0$, *et* $\delta_{12} \leftarrow 55$
$c_1(3) \leftarrow c_1(3) - 75 = 10$, *et* $\delta_{13} \leftarrow 75$
$c_2(1) \leftarrow c_2(1) - 47 = 0$, *et* $\delta_{21} \leftarrow 47$
$c_2(2) \leftarrow c_2(2) - 95 = 0$, *et* $\delta_{22} \leftarrow 95$
$c_\varnothing \leftarrow o = 122$
$\delta_\varnothing \leftarrow 122$
On remarque bien : $\frac{\delta_{12}-\delta_{11}}{w_{12}-w_{11}} = \frac{20}{10} = 2 = \frac{\delta_{13}-\delta_{12}}{w_{13}-w_{12}} = \frac{\delta_{22}-\delta_{21}}{w_{22}-w_{21}}$. *Si l'on construit la table des affections possibles pour le problème $LP_\emptyset$ obtenu après les extensions on observe :* $c(x_1 = 1, x_2 = 2) = 8$, $c_(x_1 = 2, x_2 = 2) = 28$, $c_(x_1 = 3, x_2 = 1) = 0$, $c_(x_1 = 3, x_2 = 2) = 48$ *les autres affectations ne vérifiant pas la contrainte.*
On observe que la solution optimale est 0, nos extensions justifient bien l'augmentation de $c_\varnothing$. Maintenant supposons que des opérations extérieures à la contrainte aient modifié les coûts unaires : $c_1(1) \rightarrow c_1(1)+16 = 21$, $c_1(2) \rightarrow c_1(2) + 30 = 30$, $c_1(3) \rightarrow c_1(3) - 9 = 1$. *Nous souhaitons calculer un nouveau minorant pour la contrainte PB en résolvant le problème $LP_{F\emptyset}$: :*

$$\begin{aligned} \min(21+35)x_{11} + (30+55)x_{12} + (1+75)x_{13} + 47x_{21} \\ +95x_{22} - 122 \\ s.t. \\ 4x_{11} + 14x_{12} + 24x_{13} + 16x_{21} + 40x_{22} >= 40 \\ \forall i, \sum_v x_{iv} = 1 \\ 0 \leq x_{ij} \leq 1 \end{aligned}$$

La solution optimale est $x^* = \{0, 0, 1, 1, 0\}$ et son coût est $o = 76 + 47 - 122 = 1$. On déduit la solution optimale duale suivante : $y_{cc} = e_{113} = 1, y_1 = 76 - 24 = 52, y_2 = 47 - 16 = 31$ et les coûts réduits sont $cr(x_{13}) = cr(x_{21}) = cr(x_{11}) = 0$ et $cr(x_{12}) = 19$, $cr(x_{22}) = 24$. On effectue les mouvements de coûts suivants :
$c_1(1) \leftarrow c_1(1) - 56 + 35 = 0$, *et* $\delta_{11} \leftarrow 56$
$c_1(2) \leftarrow c_1(2) - 66 + 55 = 19$, *et* $\delta_{12} \leftarrow 66$
$c_1(3) \leftarrow c_1(3) - 76 + 75 = 0$, *et* $\delta_{13} \leftarrow 76$
$c_2(2) \leftarrow c_2(2) - 71 + 95 = 24$, *et* $\delta_{22} \leftarrow 71$
$c_\varnothing \leftarrow c_\varnothing + 1 = 123$
$\delta_\varnothing \leftarrow \delta_\varnothing + 1 = 123$

4 Résultats expérimentaux

Nous avons implémenté cette approche dans TOULBAR2, un solveur exact de WCSP en C++ [3]. Pour les tests suivants nous avons imposé un temps limite de 30 minutes sur un seul coeur d'un Intel Xeon E5-2680 v3 à 2.50 GHz et 256 GB de RAM.

4.1 Compétition pseudo-Booléenne 2016

Nous avons testé TOULBAR2 sur les 1600 instances OPT-SMALLINT-LIN proposées lors de la compétition pseudo-Booléenne 2016 [4]. Toutes les instances sont booléennes et composées exclusivement de contraintes linéaires, le nombre de variables, de contraintes et l'arité des contraintes varient grandement. Nous comparons TOULBAR2 v1.1.2 aux solveurs PB NAPS v1.02b (vainqueur de la compétition en 2016), ROUNDINGSAT v2 et aux solveurs ILP, CPLEX v12.7.0.0 et SCIP v7.0.2. Nous avons appliqué les paramètres pour chercher une solution exacte avec CPLEX ($epagap = epgap = epint = 0$ et $eprhs = 10^{-9}$), les paramètres par défaut ont été utilisés pour les autres solveurs. Nous avons transformé les instances de la compétition PB en WCSP, en interprétant la fonction objective sous la forme de coûts unaires et en transformant les contraintes linéaires en contraintes de la forme $\sum_{x_i \in S} w_i x_i \geq C$. Sur toutes les instances, 1200 sont résolues par au moins un solveur. La table 1 récapitule pour chaque solveur le nombre totale d'instances résolues, le nombre de fois où il est le plus rapide [5], le nombre de fois qu'il est le seul à résoudre un problème et donne une comparaison deux à deux du temps de résolution. Notre approche ne semble pas idéale pour ce genre de problème, l'efficacité de TOULBAR2 dépend largement de la famille considérée, il est parfois compétitif (*caixa,primesdimacsnf...*) ou bien peu adapté (*rand,radar...*) avec parfois 6 fois moins d'instances résolues que les autres solveurs. D'après la table 1, TOULBAR2 est presque dominé par CPLEX et ROUNDINGSAT, cela peut être dû au fait que la compétition PB16 est principalement composée d'instances avec un grand nombre de contraintes de portée très grande. TOULBAR2 propage les contraintes une par une, le minorant ainsi obtenu est sous-optimal comparé à celui calculé en résolvant le LP comme le fait CPLEX. Et malgré une complexité de propagation raisonnable, ce rapport temps de calcul/qualité du minorant ne semble pas adapté ici. Les 4 instances uniquement résolues par TOULBAR2 appartiennent à la sous-famille *auto-corr_bern* (31 instances) de la famille *minlplib2-pb-0.1.0*, où les contraintes sont de faibles arités (max

TABLE 1 – Comparaison des temps de résolution sur la compétition PB OPT-SMALLINT-LIN 2016, données à lire en colonne.

	ROUNDINGSAT	NAPS	TB2	SCIP	CPLEX
ROUNDINGSAT	–	249	64	322	483
NAPS	618	–	205	537	677
TB2	850	643	–	746	881
SCIP	656	490	219	–	788
CPLEX	413	316	103	129	–
Instances résolues	1030	890	724	1057	1100
Le plus rapide	235	178	26	25	378
Seul à résoudre	17	21	4	2	31

5). Des tests similaires ont été effectué sur la famille lion9-single-obj composée de 126 instances de la compétition OPT-BIGINT-LIN, ces résultats montrent que notre approche peut aussi s'appliquer à des coefficients de grande taille sans pour autant dépasser les solveurs de l'état de l'art sauf à de rares exceptions.

4.2 Problème de placement des entrepôts à capacité limité

Dans le problème de placement des entrepôts (Warehouse Location Problem (WLP)), une entreprise veut ouvrir des entrepôts pour approvisionner ses magasins. L'objectif est de déterminer à partir d'une liste de potentiels entrepôts lesquels il faut ouvrir afin de minimiser le coût d'entretien et d'approvisionnement des différents magasins. Chaque entrepôt à une capacité qui lui est propre, de ce fait la demande des magasins qu'il approvisionne ne doit pas dépasser sa capacité. Il est possible de formaliser cette contrainte par une contrainte de sac-à-dos à choix multiples. Nous ajoutons également afin de faciliter la recherche une contrainte linéaire (redondante) imposant que la somme des capacités des entrepôts ouverts soit supérieure à la demande totale des magasins. Nous avons testé notre approche sur 15 instances [24] ayant 100 (5 *capmo* instances), 200(*capmp*), 300 (*capmq*) entrepôts et jusqu'à 90,901 fonctions de coûts parmi lesquels on trouve 301 contraintes linéaires d'arité 300. Nous utilisons la modélisation CFN proposée par [12] à laquelle nous ajoutons les contraintes PB. Les solveurs PB et ILP utilisent une formulation directe avec des contraintes linéaires. La table 2 récapitule le nombre d'instances résolues en moins de 10 heures. Nous donnons le temps moyen et le nombre de backtracks moyen (pour les instances résolues). Nous comparons notre approche avec NAPS v1.02b , ROUNDINGSAT v2 , CPLEX v12.7.0.0, SCIP v7.0.2 et le solveur CP OR-TOOLS v9.0.9048 (utilisant l'interface flatzinc). CPLEX obtient les meilleurs résultats, étant 19 (resp. 115) plus rapide que SCIP (resp. OR-TOOLS). TOULBAR2 ne résout pas 3/15 instances dans la limite de temps. Bien qu'il parcourt un

3. https ://github.com/toulbar2/toulbar2
4. http ://www.cril.univ-artois.fr/PB16/
5. Instance non prise en compte si au moins 2 solveurs l'ont résolue en moins de 0.005 secondes (357 cas).

TABLE 2 – Nombre d'instances résolues, temps CPU moyen, et nombre de retours-arrières moyen (ou noeuds de l'arbre de recherche pour les solveurs ILP) pour 15 problèmes de placement d'entrepôts à capacité limité.

NAPS	ROUNDINGSAT	TB2	OR-TOOLS	SCIP	CPLEX12.10
0	0	12	15	15	15
-	-	4,598 sec.	7,075 sec.	1,177 sec.	61.2 sec.
-	-	466,482 bt.	304,410 bt.	1,026 nd.	1,183 nd.

TABLE 3 – Nombre d'instances résolues, temps CPU moyen et nombre de retours-arrières moyen (ou noeuds de l'arbre de recherche pour les solveurs ILP) pour 50 problèmes de sac-à-dos.

NAPS	CHUFFED[27]	OR-TOOLS	TB2	ROUNDINGSAT	CPLEX	SCIP
0	50	50	50	50	50	50
-	180.9 sec.	0.134 sec.	0.086 sec.	0.035 sec.	0.011 sec.	0.01 sec.
-	990,631 bt.	24.2 bt.	204.7 bt.	215 bt.	16.2 nd.	1 nd.

plus grand nombre de noeuds ($\approx \times 1,000$), il résout plus rapidement que SCIP (le 2ème meilleur solveur) 3 instances (*capmp2, capmq2, capmq5*). Il est aussi plus rapide que OR-TOOLS sur 7 instances, dont 4/5 des plus grandes *capmq* (excepté *capmq1* qui n'est pas résolu). ROUNDINGSAT et NAPS n'ont résolu aucune instance en moins de 10 heures.

4.3 Problème de sac à dos

Pour l'état de l'art nous avons testé TOULBAR2 sur des problèmes de sac à dos ayant de 100 à 300 variables cités dans [9, 27]. La table 3 donne le nombre d'instances résolues et le temps moyen pour différents solveurs. TOULBAR2 résout les 50 instances assez efficacement, il est plus rapide que OR-TOOLS et CHUFFED (résultats pris dans [27] avec le modèle *length-3 nogood detection*). NaPS n'est pas adapté et ne résout aucune instance en moins de 1800s, tandis que RoundingSat est plus efficace que TOULBAR2. Sans surprise les solveurs ILP CPLEX et SCIP résolvent toutes les instances presque instantanément.

4.4 Problème de sac à dos avec graphe de conflits

Nous comparons ici les différents solveurs face aux problèmes Knapsack with Conflict Graph (KPCG) [6, 10]. Ces instances sont similaires à un problème de sac-à-dos mais associées à un grand nombre de contraintes binaires dures indiquant un conflit entre deux variables. Le nombre de variables Booléennes varie entre 120 et 1000, le poids associé à chaque variable est uniformément distribué entre $[250, 500]$ et la capacité est 1000. On définit 6 classes différentes $C1, C3, C10, R1, R3, R10$, les classes C signifient que le poids et le profit de chaque variable sont corrélés sinon le profit est aléatoire entre $[1, 1000]$ (instance R). Les nombres $1, 3, 10$ correspondent à un coefficient multiplicateur de la capacité, plus le multiplicateur est grand, plus l'instance est compliquée à résoudre. En plus de la variation du nombre de variable, la densité du graphe varie de $0,1$ à $0,9$, totalisant 720 instances par classe. Nous avons utilisé un encodage direct pour TOULBAR2 et un encodage de tuple [19] pour les autres solveurs afin de proposer une relaxation linéaire plus forte et favoriser la puissance de la propagation unitaire ainsi que des mécanismes d'apprentissage. La table 4 récapitule le nombre d'instances résolues par chaque solveur, encore une fois NAPS n'est pas adapté à ce genre de problème et ne résout que 350 instances de la classe $C1$ (une des plus faciles), de ce fait nous n'avons pas continué les expérimentations pour les autres classes. L'autre solveur PB ROUNDINGSAT n'est pas très efficace non-plus, il est dernier en terme de nombre d'instances résolus. Les solveurs ILP sont plus efficaces que les solveurs PB mais bien qu'offrant une relaxation linéaire plus forte, l'encodage en tuple ne s'avère pas payant ici. En effet l'approche de TOULBAR2 est plus efficace que SCIP et CPLEX utilisant l'encodage de tuple et compétitive avec CPLEX (version 12.6) utilisant un encodage direct (résultats repris de [6]) pour trois classes sur six. Cela peut s'expliquer par le nombre important de variables (trois par contrainte binaire) ajouté par l'encodage en tuple.

4.5 Séquence de solutions diverses pour CPD

Une protéine est une chaîne de molécules simples appelés acides aminés, cette séquence détermine en quelle forme 3D la protéine va se replier. Le problème de conception de protéine (Computational Protein Design (CPD) [4]) consiste à identifier une chaîne d'acides aminés à partir d'une forme 3D. Ce problème peut être modélisé par un CFN avec des coûts unaires et des fonctions de coûts binaires représentant l'énergie de la protéine. Cependant le critère ne fait qu'approcher la réalité, ainsi pour augmenter les chances de trouver la bonne chaîne d'acides aminés il est préférable de produire une séquence de solutions diverses. Chaque fois qu'une solution est trouvée, une contrainte de distance de Hamming est ajoutée au modèle pour imposer que la prochaine solution soit différente de la précédente. La distance de Hamming peut directement être encodée

TABLE 4 – Nombre d'instances KPCG résolues.

	RSat_t	TB2	SCIP_t	CPLEX_t	CPLEX[6]	NAPS
C1	621	708	663	694	**720**	350
C3	360	571	556	477	**622**	-
C10	255	**485**	396	316	443	-
R1	700	**720**	619	703	**720**	-
R3	559	**709**	628	566	695	-
R10	351	524	456	368	**538**	-

par une contrainte PB avec des partitions EO, cela à été implémenté dans TOULBAR2 et comparé à des encodages basés sur les automates (encodage ternaire, hidden, et dual) [33, 34] sur 30 instances [6]. Pour toutes les instances le temps limite était 1 heure et le solveur s'arrête après avoir trouvé 10 solutions diverses, la distance de Hamming est 10 et nous utilisons VAC en preprocessing sur les contraintes binaires et unaires (options -A -d : -a=10 -div=10 -divm=(0 pour dual, 1 pour hidden, 2 pour ternaire, et 3 pour encodage PB)). Le graph 1 récapitule le temps de résolution pour chaque encodage. L'encodage dual et ternaire ont échoué à donner 10 solutions pour 1 instance en moins de 1 heure, tandis que l'encodage PB et hidden ont réussi. De plus l'encodage PB est plus rapide sur 27 instances et en résout 24 en moins de 30 secondes tandis que l'encodage dual, hidden et ternaire en résolvent respectivement 12, 13 et 4 en moins de 30 secondes. Les encodages basés sur les automates ont le défaut d'introduire des variables supplémentaires qui peuvent perturber l'heuristique de choix de variable (ici *min domain size per weighted degree* [8]) et les algorithmes de cohérences locales (ici, EDAC [12] pendant la recherche). Tandis que l'encodage PB encode directement la distance de Hamming mais ralentit la propagation comme on peut l'observer en comparant le nombre de backtracks par seconde (1,094 pour l'encodage PB et 2,516 pour l'encodage dual).

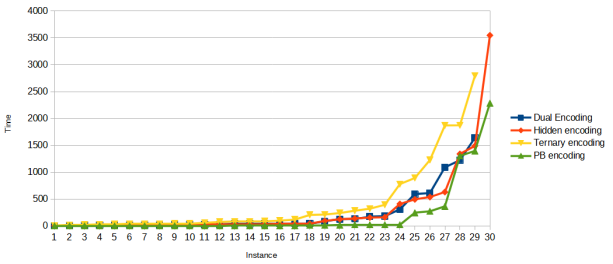


FIGURE 1 – Temps CPU de résolution pour différents encodages de la contrainte de distance de Hamming sur les 30 instances CPD.

5 Conclusion et travaux futurs

Il est à présent possible de modéliser une contrainte linéaire pseudo-Booléenne dans un WCSP. Cela offre une plus grande souplesse de modélisation et permet la résolution d'un plus grand nombre de problèmes par un solveur WCSP comme TOULBAR2. Les instances pseudo-Booléennes composées d'un faible nombre de contraintes linéaires de grande arité associées à des contraintes binaires est un cadre avantageux pour notre approche, sur ce type d'instance TOULBAR2 surpasse parfois les solveurs pseudo-Booléens et ILP. Cependant dans le cas général, ce n'est pas suffisant pour être compétitif face à ces mêmes solveurs. Une des faiblesses de notre approche est que l'algorithme propage une à une les contraintes pseudo-Booléennes sans prendre en compte les autres contraintes, produisant ainsi une solution sous-optimale du programme linéaire. Il y a plusieurs améliorations imaginables, en utilisant par exemple une relaxation Lagrangienne [23] ou une approche plus proche de VAC [15]. Cela ouvre aussi la possibilité d'utiliser les contraintes linéaires pour générer des coupes (ou autres mécanismes d'apprentissages).

6. 5 instances (*1ENH.matrix.36p.17aa, 1STN.matrix.120p.18aa, HHR.matrix.115p.19aa, 1PGB.matrix.31p.17aa, 2CI2.matrix.51p.18aa*) ont été enlevé car TOULBAR2 ne les résout pas en moins de 9,000 seconds même sans contrainte de diversité [1].

Financement

Ce travail a bénéficié d'aides de l'État gérées par l'Agence Nationale de la Recherche au titre du programme d'Investissements d'Avenir portant les références ANR-18-EURE-0021 et ANR-19-P3IA-0004.

Références

[1] David ALLOUCHE, Sophie BARBE, Simon de GIVRY, George KATSIRELOS, Yahia LEBBAH, Samir LOUDNI, Abdelkader OUALI, Thomas SCHIEX, David SIMONCINI et Matthias ZYTNICKI : *Operations Research and Simulation in Healthcare*, chapitre Cost Function Networks to Solve Large Computational Protein Design Problems. Springer, 2021.

[2] David ALLOUCHE, Christian BESSIÈRE, Patrice BOIZUMAULT, Simon de GIVRY, Patricia GUTIERREZ, Jimmy H.M. LEE, Ka Lun LEUNG, Samir LOUDNI, Jean-Philippe MÉTIVIER, Thomas SCHIEX et Yi WU : Tractability-preserving transformations of global cost functions. *Artificial Intelligence*, 238:166–189, 2016.

[3] David ALLOUCHE, Christian BESSIÈRE, Patrice BOIZUMAULT, Simon DE GIVRY, Patricia GUTIERREZ, Samir LOUDNI, Jean-Philippe METIVIER et Thomas SCHIEX : Filtering decomposable global cost functions. *In Proc. of AAAI-12*, Toronto, Canada, 2012.

[4] David ALLOUCHE, Jessica DAVIES, Simon de GIVRY, George KATSIRELOS, Thomas SCHIEX, Seydou TRAORÉ, Isabelle ANDRÉ, Sophie BARBE, Steve PRESTWICH et Barry O'SULLIVAN : Compu-

tational protein design as an optimization problem. *Artificial Intelligence*, 212:59–79, 2014.

[5] Carlos Ansótegui, Miquel Bofill, Jordi Coll, Nguyen Dang, Juan Luis Esteban, Ian Miguel, Peter Nightingale, András Z. Salamon, Josep Suy et Mateu Villaret : Automatic detection of at-most-one and exactly-one relations for improved SAT encodings of pseudo-boolean constraints. *In* Thomas Schiex et Simon de Givry, éditeurs : *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, volume 11802 de *Lecture Notes in Computer Science*, pages 20–36. Springer, 2019.

[6] Andrea Bettinelli, Valentina Cacchiani et Enrico Malaguti : A branch-and-bound algorithm for the knapsack problem with conflict graph. *INFORMS Journal on Computing*, 29(3):457–473, 2017.

[7] Miquel Bofill, Jordi Coll, Josep Suy et Mateu Villaret : An mdd-based SAT encoding for pseudo-boolean constraints with at-most-one relations. *Artif. Intell. Rev.*, 53(7):5157–5188, 2020.

[8] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre et Lakhdar Sais : Boosting systematic search by weighting constraints. *In ECAI*, volume 16, page 146, 2004.

[9] Geoffrey Chu et Peter J Stuckey : Dominance driven search. *In International Conference on Principles and Practice of Constraint Programming*, pages 217–229. Springer, 2013.

[10] Stefano Coniglio, Fabio Furini et Pablo San Segundo : A new combinatorial branch-and-bound algorithm for the knapsack problem with conflicts. *European Journal of Operational Research*, 289(2):435–455, 2021.

[11] Martin C Cooper, Simon De Givry, Martı Sánchez, Thomas Schiex, Matthias Zytnicki et Tomas Werner : Soft arc consistency revisited. *Artificial Intelligence*, 174(7-8):449–478, 2010.

[12] Simon De Givry, Federico Heras, Matthias Zytnicki et Javier Larrosa : Existential arc consistency : Getting closer to full arc consistency in weighted csps. *In IJCAI*, volume 5, pages 84–89, 2005.

[13] Simon de Givry et George Katsirelos : Clique cuts in weighted constraint satisfaction. *In International Conference on Principles and Practice of Constraint Programming*, pages 97–113. Springer, 2017.

[14] Heidi Dixon : *Automating pseudo-boolean inference within a dpll framework*. Citeseer, 2004.

[15] Tomás Dlask et Tomás Werner : Bounding linear programs by constraint propagation : Application to max-sat. *In* Helmut Simonis, éditeur : *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 de *Lecture Notes in Computer Science*, pages 177–193. Springer, 2020.

[16] Tomás Dlask et Tomás Werner : On relation between constraint propagation and block-coordinate descent in linear programs. *In* Helmut Simonis, éditeur : *Principles and Practice of Constraint Programming - 26th International Conference, CP 2020, Louvain-la-Neuve, Belgium, September 7-11, 2020, Proceedings*, volume 12333 de *Lecture Notes in Computer Science*, pages 194–210. Springer, 2020.

[17] Niklas Eén et Niklas Sörensson : Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

[18] Jan Elffers et Jakob Nordström : Divide and conquer : Towards faster pseudo-boolean solving. *In IJCAI*, volume 18, pages 1291–1299, 2018.

[19] Barry Hurley, Barry O'Sullivan, David Allouche, George Katsirelos, Thomas Schiex, Matthias Zytnicki et Simon De Givry : Multi-language evaluation of exact solvers in graphical model discrete optimization. *Constraints*, 21(3):413–434, 2016.

[20] Ambros Gleixner Jo Devriendt et Jakob Nordström : Learn to relax : Integrating 0-1 integer linear programming with pseudo-boolean conflict-driven search. *In Proceeding of CPAIOR 2020*, 2020.

[21] Ellis L Johnson et Manfred W Padberg : A note of the knapsack problem with special ordered sets. *Operations Research Letters*, 1(1):18–22, 1981.

[22] V Kolmogorov : Convergent tree-reweighted message passing for energy minimization. *IEEE transactions on pattern analysis and machine intelligence*, 28(10):1568–1583, 2006.

[23] Nikos Komodakis, Nikos Paragios et Georgios Tziritas : Mrf energy minimization and beyond via dual decomposition. *IEEE transactions on pattern analysis and machine intelligence*, 33(3): 531–552, 2010.

[24] J. Kratica, D. Tosic, V. Filipovic et I. Ljubic : Solving the Simple Plant Location Problems by Genetic Algorithm. *RAIRO Operations Research*, 35:127–142, 2001.

[25] J. Larrosa : On arc and node consistency in weighted CSP. *In Proc. AAAI'02*, pages 48–53, Edmondton, (CA), 2002.

[26] JHM Lee et Yu Wai Shum : Modeling soft global constraints as linear programs in weighted constraint satisfaction. *In 2011 IEEE 23rd International Conference on Tools with Artificial Intelligence*, pages 305–312. IEEE, 2011.

[27] Jimmy HM Lee et Allen Z Zhong : Automatic dominance breaking for a class of constraint optimization problems. *In Twenty-Ninth International Joint Conference on Artificial Intelligence*, pages 1192–1200, 2020.

[28] Jimmy Ho-Man Lee et Ka Lun Leung : Consistency techniques for flow-based projection-safe global cost functions in weighted constraint satisfaction. *Journal of Artificial Intelligence Research*, 43:257–292, 2012.

[29] Joao P Marques-Silva et Karem A Sakallah : Grasp : A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[30] David Pisinger et Paolo Toth : Knapsack problems. *In Handbook of combinatorial optimization*, pages 299–428. Springer, 1998.

[31] Daniel Prusa et Tomas Werner : Universality of the local marginal polytope. *In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1738–1743, 2013.

[32] Francesca Rossi, Peter Van Beek et Toby Walsh : *Handbook of constraint programming*. Elsevier, 2006.

[33] Manon Ruffini, Jelena Vucinic, Simon de Givry, George Katsirelos, Sophie Barbe et Thomas Schiex : Guaranteed diversity & quality for the weighted csp. *In 2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 18–25. IEEE, 2019.

[34] Manon Ruffini, Jelena Vucinic, Simon de Givry, George Katsirelos, Sophie Barbe et Thomas Schiex : Guaranteed diversity and optimality in cost function network based computational protein design methods. *Algorithms*, 2021.

[35] Masahiko Sakai et Hidetomo Nabeshima : Construction of an ROBDD for a PB-constraint in band form and related techniques for PB-solvers. *IEICE TRANSACTIONS on Information and Systems*, 98(6):1121–1127, 2015.

[36] M. Sánchez, S. de Givry et T. Schiex : Mendelian error detection in complex pedigrees using weighted constraint satisfaction techniques. *Constraints*, 13(1):130–154, 2008.

[37] D Sontag, D Choe et Y Li : Efficiently searching for frustrated cycles in MAP inference. *In Proc. of UAI*, pages 795–804, 2012.

[38] D Sontag, T Meltzer, A Globerson, Y Weiss et T Jaakkola : Tightening LP relaxations for MAP using message-passing. *In Proc. of UAI*, pages 503–510, Helsinki, Finland, 2008.

[39] Fulya Trösser, Simon de Givry et George Katsirelos : Relaxation-aware heuristics for exact optimization in graphical models. *In* Emmanuel Hebrard et Nysret Musliu, éditeurs : *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21-24, 2020, Proceedings*, volume 12296 de *Lecture Notes in Computer Science*, pages 475–491. Springer, 2020.

[40] Tomas Werner : A Linear Programming Approach to Max-sum Problem : A Review. *IEEE Trans. on Pattern Recognition and Machine Intelligence*, 29(7):1165–1179, juillet 2007.

[41] M. Zytnicki, C. Gaspin, S. de Givry et T. Schiex : Bounds Arc Consistency for Weighted CSPs. *Journal of Artificial Intelligence Research*, 35:593–621, 2009.

Actes JFPC 2021

Apprentissage d'arbres de décision optimaux grâce à la programmation par contraintes

Hélène Verhaeghe[1*] **Siegfried Nijssen**[1] **Gilles Pesant**[2]
Claude-Guy Quimper[3] **Pierre Schaus**[1]
[1] UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium
[2] Polytechnique Montréal, Montréal, Canada
[3] Université Laval, Québec, Canada
[1]{prenom.nom}@uclouvain.be [2]gilles.pesant@polymtl.ca
[3]claude-guy.quimper@ift.ulaval.ca

Résumé

Les arbres de décision sont parmi les modèles les plus populaires en apprentissage automatique. L'utilisation d'algorithmes gloutons pour les apprendre peut poser plusieurs désavantages : il est difficile de limiter la taille de l'arbre de décision tout en maintenant une bonne qualité de classification, et il est difficile d'imposer de nouvelles contraintes sur le modèle appris. Ces raisons sont à la base de l'émergence d'un intérêt pour des algorithmes exacts et flexibles dans l'apprentissage des arbres de décision. Ce papier résume notre papier "Learning Optimal Decision Trees using contraint Programming", accepté en version journal accéléré à CP2019 [5]. Dans ce papier, nous introduisons une nouvelle approche pour apprendre des arbres de décision en utilisant la programmation par contraintes.

1 Introduction

Les arbres de décision sont un modèle de classification populaire en apprentissage automatique. Leurs bénéfices incluent leur relative facile interprétation et le fait qu'ils donnent de bonnes performances de classification sûr de nombreux jeux de données.

De nombreuses méthodes ont été proposées dans la littérature pour apprendre des arbres de décision.Les méthodes gloutonnes sont les plus populaires.Ces méthodes partitionnent de manière récursive le jeu de données en deux sous-ensembles. Cette partition se base sur un attribut, sélectionné de manière gloutonne. Le partitionnement récursif s'arrête lorsque le critère d'arrêt est atteint (par exemple, un nombre minimum d'éléments du jeu de données initiales dans une feuille ou une classe unique pour ces éléments).Bien qu'en pratique ces méthodes arrivent à une bonne précision de classification pour beaucoup de types de données, malheureusement, elles ne donnent que peu de garantie.Résultant de cela, les arbres obtenus par ces méthodes peuvent être inutilement complexes, moins précis que possibles et il est également compliqué d'imposer de nouvelles contraintes sur ces arbres.

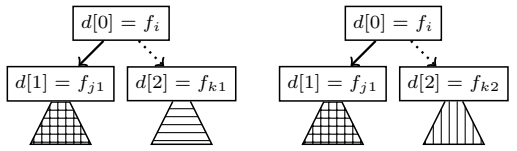
Pour pallier ces faiblesses, des chercheurs ont étudié l'inférence d'arbres de décision optimaux sous contraintes [3, 1]. Ces approches assurent que sous certaines contraintes bien définies et considérant un critère d'optimisation prédéfini, un arbre optimal est trouvé.

Notre papier propose une nouvelle approche, plus évolutive, basée sur la programmation par contraintes (PPC) pour apprendre des arbres de décision avec une profondeur maximale fixée qui minimise l'erreur de classification. Notre approche combine les idées clés suivantes : l'utilisation des mécanismes d'un solveur PPC, l'utilisation de la contrainte globale CoverSize [4], l'utilisation d'un arbre de recherche ET/OU [2] et l'utilisation d'un mécanisme de mémoization tel qu'introduit par DL8 [3]. Ces éléments permettent à notre approche par programmation par contraintes de profiter, de manière plus efficace, de la décomposition de ce problème d'apprentissage. Nous allons montrer que la combinaison de ces différentes idées mène à un modèle qui est plus efficace que les autres approches proposées dans la littérature.

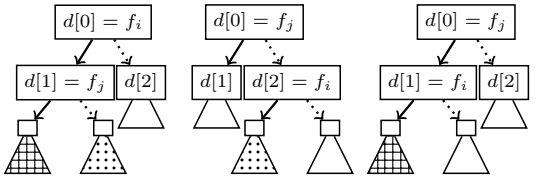
*Papier doctorant : Hélène Verhaeghe[1] est auteur principal.

2 Modèle et arbre de recherche

Les variables de décisions principales nécessaires pour modéliser notre arbre de décision sont les décisions prises à chaque noeud de l'arbre. Un autre ensemble de variables est utilisé pour compter le nombre de transaction correspondant à chaque feuille. L'intégrité de l'arbre est assurée par l'utilisation de contraintes AllDifferent, de contraintes CoverSize et de simples contraintes arithmétiques liant les différentes variables. D'autres contraintes redondante sont ajoutée pour accélérer la résolution.



(a) Indépendance des sous-arbres



(b) Sous-arbres redondants ; Redundant subtrees ; les sous-arbres mis en évidence de la même manière sont identiques

FIGURE 1 – Décompositions

Les solveurs traditionnels de PPC utilisent un arbre OU. Dans ceux-ci, pour chaque noeud, la solution se trouve dans l'une des branches. Pour trouver l'arbre de décision optimal, nous utilisons un arbre ET/OU. Un noeud ET va diviser le problème en sous problèmes indépendants. La solution est donc la combinaison des solutions des sous problèmes. Les noeuds OU sont les mêmes que classiquement utilisés. L'utilisation d'un tel arbre est rendue possible par le caractère indépendant de deux sous-arbres de décision pour lesquels la racine a été fixée (Fig.1a).

De plus, un mécanisme de mémoization permet le stockage de sous-arbres optimaux. En effet, si les ensembles des décisions pour arriver à deux sous-arbres sont équivalents, alors les sous-arbres optimaux sont les identiques (Fig.1b).

3 Résultats

Nous avons comparé notre méthode à DL8 [3] et BinOCT [6], deux méthodes résolvant le même problème. Notre méthode surpasse les deux autres sur la plupart des instances. Elle a pu prouver l'optimalité sur environ 80% des instances endéans la limite de temps. La meilleure solution trouvée est obtenue par notre méthode dans près de tous les cas. Cependant, DL8 performe mieux sur les petites instances. La grande différence entre BinOCT et notre méthode peut être expliquée par le bénéfice de la recherche ET/OU qui n'est pas utilisé par BinOCT. L'écart avec DL8 peut être partiellement expliqué par l'élagage du coût. Cela peut également venir de l'algorithme de recherche d'ensemble fréquent utilisé : DL8 ne profite pas des optimisations présentes dans la contrainte CoverSize. Nos expérimentations évaluent également les effets de certaines des techniques utilisées pour résoudre notre problème, tel que l'utilisation de la mémoization.

4 Conclusion

Pour résumer, notre papier présente une nouvelle approche pour créer de manière efficace un arbre de décision optimal avec profondeur limitée. Cette approche basée sur la PPC combine la contrainte globale CoverSize, le concept d'arbre de recherche ET/OU et la mémoization. Sur la plupart des jeux de données, notre algorithme trouve la meilleure solution dans le temps impartit et est le plus rapide à prouver l'optimalité. Nous pensons que notre approche peut être étendue de plusieurs manières (multi-classe, caractéristiques continues en utilisant la binarisation,...).

Références

[1] Gaël AGLIN, Siegfried NIJSSEN et Pierre SCHAUS : Learning optimal decision trees using caching branch-and-bound search. *In AAAI20*, 2020.

[2] R. DECHTER et R. MATEESCU : And/or search spaces for graphical models. *Artificial intelligence*, 171(2-3):73–106, 2007.

[3] S. NIJSSEN et E. FROMONT : Mining optimal decision trees from itemset lattices. *In SIGKDD2007*, 2007.

[4] P. SCHAUS, J. AOGA et T. GUNS : Coversize : a global constraint for frequency-based itemset mining. *In International Conference on Principles and Practice of Constraint Programming*, pages 529–546. Springer, 2017.

[5] Hélene VERHAEGHE, Siegfried NIJSSEN, Gilles PESANT, Claude-Guy QUIMPER et Pierre SCHAUS : Learning optimal decision trees using constraint programming. *Constraints*, 25(3):226–250, 2020.

[6] S. VERWER et Y. ZHANG : Learning optimal classification trees using a binary linear program formulation. *In AAAI19*, 2019.

Noname manuscript No.
(will be inserted by the editor)

Learning Optimal Decision Trees using Constraint Programming

Hélène Verhaeghe · Siegfried Nijssen · Gilles Pesant · Claude-Guy Quimper · Pierre Schaus

the date of receipt and acceptance should be inserted later

Abstract Decision trees are among the most popular classification models in machine learning. Traditionally, they are learned using greedy algorithms. However, such algorithms pose several disadvantages: it is difficult to limit the size of the decision trees while maintaining a good classification accuracy, and it is hard to impose additional constraints on the models that are learned. For these reasons, there has been a recent interest in exact and flexible algorithms for learning decision trees. In this paper, we introduce a new approach to learn decision trees using constraint programming. Compared to earlier approaches, we show that our approach obtains better performance, while still being sufficiently flexible to allow for the inclusion of constraints. Our approach builds on three key building blocks: (1) the use of AND/OR search, (2) the use of caching, (3) the use of the CoverSize global constraint proposed recently for the problem of itemset mining. This allows our constraint programming approach to deal in a much more efficient way with the decompositions in the learning problem.

Keywords Decision Tree, CoverSize, AND/OR search tree, Caching

1 Introduction

Decision trees are popular classification models in machine learning. Benefits of decision trees include that they are relatively easy to interpret and that they provide good classification performance on many datasets.

Several methods have been proposed in the literature for learning decision trees. The greedy methods are the most popular ones [6,19,20]. These methods

H. Verhaeghe, S. Nijssen and P. Schaus
UCLouvain, ICTEAM, Place Sainte Barbe 2, 1348 Louvain-la-Neuve, Belgium, E-mail: {firstname.lastname}@uclouvain.be

G.Pesant
Polytechnique Montréal, Montréal, Canada, E-mail: gilles.pesant@polymtl.ca

C.-G. Quimper
Université Laval, Québec, Canada, E-mail: claude-guy.quimper@ift.ulaval.ca

recursively partition a dataset into two subsets based on a greedily selected attribute until some stopping criterion is reached (such as a minimum number of examples in the leaf, or a unique class label in these examples). While in practice these methods obtain a good prediction accuracy for many types of data, unfortunately, they provide little guarantees. As a result, the trees learned using these methods may be unnecessarily complex, may be less accurate than possible, and it is hard to impose additional constraints on the trees, such as on the fairness of their predictions.

To address these weaknesses, researchers have studied the inference of *optimal* decision trees under constraints [1,3,4,15–17,26][1]. These approaches ensure that under well-defined constraints and optimization criteria, an optimal tree is found. Experiments conducted in earlier work [3,16,17] have shown that optimal decision trees computed with these exact methods can indeed obtain better classification performance while respecting constraints.

A problem that is solved by many of these earlier approaches [3,16,17,26] is the following. Given a dataset in which all examples are binary; the problem is to find the decision tree that optimizes prediction accuracy, while enforcing a constraint on the depth of the decision tree.

The key ideas behind this constraint are that it limits the complexity of the decision tree, hence making the predictions of the tree easier to interpret while preventing over-fitting.

Several papers have studied the addition of other constraints to these approaches, including support constraints on the leaves of the tree [3,17], on fairness [1], or on the preservation of privacy by these trees [17].

We suspect, as papers [3,17,26], that the problem of finding an optimal decision tree given a maximum depth is NP-complete even if no formal proof is available yet. Clearly the problem is in NP. As the depth is bounded, at most d tests are required to classify one of the transactions from the database. Therefore, the error can be computed in $\mathcal{O}(nd)$ (with n the total number of transactions and d the maximum depth), which is polynomial. Similar problems have been proven NP-Complete. Hyafil and Rivest [12] found that the problem of finding a decision tree which minimizes the expected number of tests to classify a new transaction is NP-complete. The problem of finding a decision tree minimizing the number of leaves is also proven NP-complete in [11].

Hence, approaches for this problem need to perform some form of exhaustive search through the space of possible trees. To explore this search space, earlier approaches have been built on existing technologies: Mixed Integer Programming (MIP) solvers, satisfiability (SAT) solvers, or itemset mining algorithms developed in the data mining literature.

This paper proposes a new, more scalable approach based on Constraint Programming (CP) for learning decision trees. Our approach combines these key ideas:

- the use of branch-and-bound in a CP solver to eliminate parts of the search space in which no solutions can be found;
- the use of the CoverSize global constraint, originally developed for itemset mining in CP, to calculate efficiently in which leaf examples end up [23];

[1] The problem of embedding a decision tree as a constraint into a CP model has been studied in [5].

– the use of an AND/OR search tree to exploit the fact that the optimal left-hand and right-hand subtrees of a node in a decision tree can be found independently from each other [8];
– the use of caching to store optimal decision trees for itemsets that have been considered in the past [16].

We will show that the combination of these different ideas leads to a model that is more efficient than other approaches proposed in the literature.

The paper is organized as follows. Section 2 presents the state of the art, followed by a formal definition of the problem in Section 3. Our CP model and CP search are detailed in Section 5. Finally, Section 6 presents empirical results about our algorithm.

2 Related Work

Most related to this work are the alternative approaches for finding optimal decision trees. There is a number of alternative definitions for the problem of finding optimal decision trees, each using different constraints and optimization criteria.

The most popular setting studied in recent papers [1,3,26] is the one in which a decision tree of bounded depth is learned by maximizing the accuracy on a given training dataset. The limit on depth allows to model the problem as a MIP problem with a fixed number of variables. Constraints can be added, as long as they are linear; this includes constraints on fairness [1] or on the number of examples in the leafs [3]. We will use this problem setting in this work.

A slightly different setting was studied in the DL8 algorithm [17]. DL8 builds on top of itemset mining algorithms to find decision tree paths, and uses dynamic programming to build a decision tree from these paths. Effectively, it uses itemsets as the key of a caching data structure. As a consequence of the use of itemset mining, DL8 does not require a specific constraint on the depth of the decision tree; it uses a minimum support constraint to limit the size of the search space. This approach can be used on constraints that are not linear in nature. From this approach, we will adapt its link to itemset mining, and its use of caching.

To the best of our knowledge, CP has not yet been used in the setting where accuracy is optimized. Two earlier studies [4,15] did, however, study the setting in which one finds the smallest decision tree consistent with a training dataset (i.e. the error of the decision tree has to be zero). As training data can be noisy and inconsistent, and hence finding a tree of zero error can be either impossible or undesirable, this setting is less common in the machine learning literature.

Similar to DL8, we will rely in this work on the fact that decision tree learning problems have many decompositions. We will exploit these using AND/OR search, which was studied extensively by Dechter et al. [8]. AND/OR search is not common in CP systems yet, and has not been used in decision tree learning yet; it has recently been exploited in the context of stochastic CP however [2].

3 Technical Background

3.1 Definition of the Problem

We restrict our attention to binary data. Continuous data can be discretized and binarized as proposed by Breiman et al. [6]; this observation was also exploited in earlier studies [16, 26].

We represent our data using an $n \times m$ binary matrix D. D_i represents the ith row of the data, or, following itemset mining terminology, the ith *transaction* of D. The number of transactions is thus n. The columns of the matrix represent the m *features* or *items* of the transactions. $D_{i,j}$ represents the value of the jth feature of D_i, i.e., 0 or 1 as we work with binary features. We assume in this work that each transaction belongs to one of two classes, represented by 0 and 1. The classes are stored in a vector v of size n. Hence, the database can be split into D^+, a matrix of size $n^+ \times m$, containing all the transactions from D associated to class 1, and D^-, a matrix of size $n^- \times m$, containing the ones associated to class 0.

In this work we are interested in finding decision trees. Each internal node w of a decision tree is associated to a feature (called the decision of the node) $d[w] \in \{1, \ldots, m\}$; each leaf is associated to a Boolean $b[w]$, representing the prediction for that leaf. We will use the function $F(r, t)$ to represent the predicted value for transaction t on a tree with root r, defined recursively as

$$F(w,t) = \begin{cases} b[w] & \text{if } w \text{ is a leaf;} \\ F(\text{left}(w), t) & \text{if } D_{t,d[w]} = 1; \\ F(\text{right}(w), t) & \text{if } D_{t,d[w]} = 0. \end{cases} \tag{1}$$

Here $\text{left}(w)$ (resp. $\text{right}(w)$) returns the left-hand (resp. right-hand) subtree of node w.

We focus on creating well-formed decision trees. This means that a given feature can be used at most once on each path from the root to a leaf. Also, each subtree rooted in one of the inner nodes should not have all its leaves assigned to the same class.

We define the *depth* d of a decision tree to be the maximum number of features on any path from the root of the tree towards a leaf. Given a maximum depth, our goal is to find a decision tree that minimizes the number of misclassified transactions (i.e. transactions where the class of the transaction, $v[t]$, is different from the classification by the tree, $F(r, t)$):

$$\min \sum_{t=1}^{n} [F(r,t) \neq v[t]]. \tag{2}$$

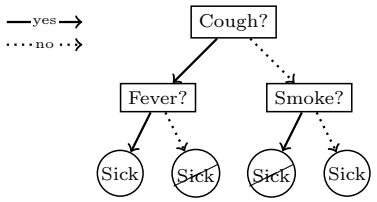
We allow for the additional specification of a constraint on the minimum number of examples $N_{\min}$ in each leaf of the tree [3, 17],

An extension of the problem is to consider more than two classes (multi-class decision trees). We will limit our discussion to binary classes, but the extension towards data with more than two classes is relatively straightforward.

Let us give a simple example. Given the dataset in Fig. 1a, a possible decision tree of depth 2 is provided in Fig. 1b. For patient 1, the tree predicts the correct class (she does not have a cough and does not smoke, which leads to the fourth leaf, which categorizes the patient as sick). For patient 4, the tree produces a

	Fever?	Cough?	Age > 60?	Smoke?	...	Sick from A?
Patient 1	✓	✘	✓	✘	...	✓
Patient 2	✓	✓	✘	✘	...	✓
Patient 3	✘	✘	✓	✓	...	✘
Patient 4	✘	✓	✓	✓	...	✓
⋮	⋮	⋮	⋮	⋮	⋮	⋮

(a) Dataset



(b) Decision tree

Fig. 1: Example

faulty classification (she has a cough and does not have a fever, which leads to the second leaf and categorizes the patient as not sick). The tree is the optimal decision tree of maximum depth 2 with respect to the classification error if the number of misclassified patients is the smallest among all the possible trees of maximum depth 2.

3.2 Constraint Programming

Constraint programming (CP) [21] is a computational paradigm aiming at solving combinatorial problems (satisfaction and optimization ones). The problem is stated as a set of *variables* and a set of *constraints* acting on the variables. The constraints represent the properties that should be respected in a solution (i.e. an assignment of a value to each variable). An example of a constraint is the *AllDifferent* constraint. It specifies that each variable involved should take a different value in a valid assignment. The variables and constraints form the *model* of the problem. A *search tree* is then explored (usually using backtracking depth first search) to discover an assignment of all the variables that satisfies all the constraints. A heuristic decides at each node of the search tree which variable and value should be assigned/removed in the next two alternative branches. For example, a first fail heuristic selects, at each node of the search tree, the unbound variable with the smallest remaining domain. The search tree is pruned by the constraints in charge of removing infeasible values from the domain of the variables during the fix-point computation triggered at each node of the search tree. A backtrack occurs when one domain becomes empty. We refer to [13] for more information about Constraint Programming technology.

3.3 AND/OR Search Trees

In a classical CP framework, the search tree is only composed of OR nodes. Each branch starting at an OR node adds a simple constraint in order to cut the search space. To keep the search complete, the branches should be complementary and together still represent the full search space. The optimal solution lies in one of the branches. An example could be to branch using $x = v$ and $x \neq v$. The first branch restricts the domain of x to the singleton $\{v\}$, while the other branch restricts it to $dom(x)\backslash\{v\}$. The union of the two branches covers all the possible values of x. Another example could be to create a branch for each of the values of the domain of a given variable, each assigning this value to the variable.

In AND/OR search trees [9,14], a second type of node is introduced: the AND nodes. Each branch starting at an AND node represents a distinct, independent subproblem. The set of unbound variables and the set of remaining constraints are partitioned into disjoint sets. The solution, if it exists, is therefore the conjunction of the partial solutions of all the branches. If one branch does not have a solution, then no solution exists for the node. An example: given a problem of 5 variables ($dom(A) = \{5\}$, $dom(B_1) = dom(B_2) = dom(C_1) = dom(C_2) = \{0, 1, 2, 3, 4, 5\}$) and two constraints ($A = B_1 + C_1$ and $A = B_2 + C_2$), since A is bound, the problem can be partitioned into two independent subproblems. On the first branch, B_1, C_1 and $A = B_1 + C_1$ form the first sub-problem, while B_2, C_2 and $A = B_2 + C_2$ forms the second, on the second branch.

An AND/OR search tree combines both OR and AND nodes in order to find solutions to the problem.

3.4 The CoverSize Constraint

To determine the accuracy of a decision tree, we need to decide in which nodes of the decision tree a transaction ends up. A correspondence can be drawn here with the *cover* of itemsets in itemset mining [16,17]. We exploit this correspondence by adapting the CoverSize global constraint [23] to the context of learning decision trees. The original CoverSize has the following parameters: an array of Boolean variables (one variable for each feature), the database, and a counter variable, and is defined as follows:

$$\textsc{CoverSize}([I_1, \ldots, I_m], D, c) \iff c = \left| \bigcap_{I_i=1} \{t \in \{1, \ldots, n\} \mid D_{t,i} = 1\} \right|. \quad (3)$$

The goal of the constraint is to link an *itemset* to the number of transactions containing the itemset. The itemset is represented by the Boolean array $[I_1, \ldots, I_m]$: Boolean I_i is true if and only if feature i is included in the itemset. A transaction contains an itemset if and only if every feature in the itemset has value 1 in the transaction. The algorithm described in [23] is bound consistent.

For example, reusing the dataset in Fig. 1a, each Boolean is linked to one of the features (*Fever?*, *Cough?*,...). Given the selected features (i.e. corresponding Boolean set to true) forming the pattern, the constraint ensures that the counter corresponds to how many transactions matches this pattern. If the only true Boolean are $I_{Fever?}$ and $I_{Age>60?}$, then on the displayed patients, only the first should be counted, since she is the only one to have both features.

4 Preliminaries: Adaptation of CoverSize

The CoverSize constraint, in its original form, does not suit our application right away. Our aim is to use it to link the number of transactions matching a given series of decisions (i.e. the path to the corresponding leaf). Given a tree, we know two things about each series of decisions leading to a leaf. One, there is a fixed number of decisions along the path and second, some are also rejecting decisions. This led to the modifications made to the CoverSize propagation algorithm.

The dense representation of an itemset using a bit vector is unnecessary and impractical in our application. Instead, we will use a sparse representation:

$$\text{CoverSizeS}(\{K_1, ..., K_a\}, D, c) \iff c = \left| \bigcap_{i=1}^{a} \{t \in \{1, ..., n\} \mid D_{t,K_i} = 1\} \right| \quad (4)$$

This constraint has the following parameters: a set of integer variables $\{K_1, ..., K_a\}$ (each representing the identifier of a selected feature), the database and the cover counter. Similar propagation is possible for this constraint as for CoverSize.

Reusing the example in Fig. 1a and a pattern of size 2, if the two values assigned to the variables are *Fever?* and *Age>60?*, then the constraint should ensure that only patient 1 is counted (from the four displayed).

Note that in the standard CoverSize constraint, we only test whether an item is included in a transaction ($D_{t,K_i} = 1$). In decision trees, we will also need to be able to test that an item is absent in a transaction. Neither with the initial CoverSize constraint nor its sparse version, is it possible to test for the absence of an item. To address this weakness, we propose the CoverSizeSR constraint, defined as follows:

$$\text{CoverSizeSR}(\underbrace{\{K_1, ..., K_a\}}_{\text{take set}}, \underbrace{\{L_1, ..., L_b\}}_{\text{drop set}}, D, c) \iff$$
$$c = \left| \left(\bigcap_{i=1}^{a} \{t \in \{1, ..., n\} \mid D_{t,K_i} = 1\} \right) \bigcap \left(\bigcap_{i=1}^{b} \{t \in \{1, ..., n\} \mid D_{t,L_i} = 0\} \right) \right| \quad (5)$$

The *take* (resp. *drop*) set defines the features that should (resp. should not) appear in the counted transactions.

Reusing the dataset in Fig. 1a, with a pattern consisting of two features, one in the take set and one in the drop set, if their values are respectively *Cough?* and *Fever?*, then only patient 4 (over the one displayed) matches the pattern and is therefore counted. In the decision tree of Fig. 1b, this pattern corresponds to the second leaf.

The pseudo-code of the CoverSizeSR propagator is given as Algorithm 1. Algorithm 2 details two methods used by the propagator. The key element of the algorithm is the cover. It represents the set of transactions corresponding to the features already selected in the take and drop sets. As in the original implementation of CoverSize, the cover is implemented using a reversible sparse bitset [10]. In this auto-backtracking structure, each bit is associated with one of the transactions of the database. If the transaction is still valid concerning the already selected features, then its associated bit is set to 1. It is set to 0 otherwise. To

help the computations, immutable bitsets are precomputed for each of the possible features. Each of these bitsets maintains the set of transactions containing the given feature.

Algorithm 1: PropagateCoverSizeSR

```
1  take: Set of variables                                   // set of take variable
2  drop: Set of variables                                   // set of drop variable
3  c: Variable                                              // counter variable
4  cover: Reversible Sparse Bitset                          // current cover
5  sizeCover: Integer                                       // size of current cover
6  support: Array of Bitset                                 // precomputed bitsets
7  Function propagate(take, drop, c, cover, sizeCover, support)
8      vars = take ∪ drop
9      takeUnbound ← { x | x ∈ take ∧ |dom(x)| > 1 }
10     remainingTakeVals ← ⋃_{x∈takeUnbound} dom(x)
11     dropUnbound ← { x | x ∈ drop ∧ |dom(x)| > 1 }
12     remainingDropVals ← ⋃_{x∈dropUnbound} dom(x)
13     isCoverChanged ← updateCover(take, drop, cover, sizeCover, support)
         // method defined at Algo.2
14     c.max ← min(sizeCover, c.max)
15     if { x | x ∈ vars ∧ |dom(x)| > 1 } = ∅ then
16         c.min ← sizeCover
17     else
18         filterValues(remainingTakeVals, takeUnbound, remainingDropVals,
             dropUnbound, c, cover, support)            // method defined at Algo.2
           /* compute cover as if every decision available for the take set were
              selected in the cover and every decision available for the drop
              set were rejected from the cover                                  */
19         if isCoverChanged ∧ ( remainingTakeVals ∩ remainingDropVals = ∅ )
             then
20             virtualCover ← cover
21             foreach i ∈ remainingTakeVals do
22                 virtualCover ← virtualCover ∩ support[i]
23             foreach i ∈ remainingDropVals do
24                 virtualCover ← virtualCover ∩ support[i]^C
25             lb ← | virtualCover |
26             c.min ← max(lb, c.min)
           /* if the counter variable is bounded to the current size of the
              cover, remove all values that would reduce the size of the cover
              */
27         if |dom(c)| = 1 ∧ c.min = sizeCover then
28             foreach i ∈ remainingTakeVals do
29                 if cover ∩ support[i] ≠ cover then
30                     foreach x ∈ takeUnbound do
31                         dom(x) ← dom(x)\{i}
32             foreach i ∈ remainingDropVals do
33                 if cover ∩ support[i]^C ≠ cover then
34                     foreach x ∈ dropUnbound do
35                         dom(x) ← dom(x)\{i}
```

Algorithm 2: PropagateCoverSizeSR: functions updateCover and filterValues

```
1  Function updateCover(take, drop, cover, sizeCover, support)
2      mask ← cover
       /* Update cover with the new values chosen in the take set          */
3      foreach x ∈ take do
4          if x newly bound then // bound since last propagation
5              mask ← mask ∩ support[x.value]
       /* Update cover with the new values rejected in the drop set        */
6      foreach x ∈ drop do
7          if x newly bound then // bound since last propagation
8              mask ← mask ∩ support[x.value]^C
9      if cover ≠ mask then
10         cover ← mask
11         sizeCover ← |cover|                          // cover upper bound
12         return true
13     else
14         return false
15 Function filterValues(remainingTakeVals, takeUnbound, remainingDropVals,
     dropUnbound, c, cover, support)
       /* Test remaining values available for the take set                 */
16     foreach i ∈ remainingTakeVals do
17         count ← | cover ∩ support[i] |
18         if count < c.min then // too few left to select
19             foreach x ∈ takeUnbound do
20                 dom(x) ← dom(x)\{i}
21             remainingTakeVals ← remainingTakeVals\{i}
       /* Test remaining values available for the drop set                 */
22     foreach i ∈ remainingDropVals do // Remove impossible values
23         count ← | cover ∩ support[i]^C |
24         if count < c.min then // too few left to reject
25             foreach x ∈ dropUnbound do
26                 dom(x) ← dom(x)\{i}
27             remainingDropVals ← remainingDropVals\{i}
```

The algorithm first updates the cover (Algo.2 line 1) for each of the variables newly bound. For each variable from the take set (Algo.2 line 3), the current cover is intersected with the support of the feature. For each variable from the drop set (Algo.2 line 6), the current cover is intersected with the complement of the support of the feature. An updated cover allows to compute the new value of the upper bound of the counter.

If all the variables from both take and drop sets have been assigned, then the cover cannot evolve (Algo.1 line 15). The previously computed upper bound is also the lower bound. Otherwise, the computation continues.

Then, some features may be now impossible to select or reject and should be filtered out of the domains (Algo. 2 line 15). This is triggered by a change in the cover or a change in the domain of the counter. To test this, a virtual inclusion (resp. rejection) of the feature is done (Algo. 2 line 17 (resp. line 23)) by doing the intersection between the cover and the concerned support (resp. the complement of

the concerned support). This intersection corresponds to the number of remaining transactions with (resp. without) the feature. Using the size of this intersection, we can easily prevent the feature from being used in the take (resp. drop) set. If the size is smaller than the current minimum, then the feature cannot be assigned to a take (resp. drop) set variable, i.e. not enough transactions with (resp. without) the feature to meet the current lower bound of the counter.

The next step is to compute a new lower bound (Algo. 1 line 19). This is done by virtually selecting all the remaining values from both take and drop set into the cover. All the supports of the available values for the take set are intersected with the cover and the complement of the supports of the available values for the drop set are intersected. The size of this virtual cover is the smallest cover possible. However, if a given value is still allowed in both the take and drop set, by the property stating that the intersection of a set and its complement is always empty, the virtual cover is empty and the lower bound is equal to 0. The computation of the lower bound can thus be avoided in such cases.

Finally, if the counter ends up taking the value of the size of the cover, the features which modify the cover, and thus its size, should be removed from the domains (Algo. 1 line 27).

The time complexity of CoverSizeSR is $\mathcal{O}(m\frac{n}{w})$ with w the size of the computer words (e.g., $w = 64$). This is the same complexity as the CoverSize propagator. The space complexity is $\mathcal{O}(m\frac{n}{w})$. The consistancy remains the same: bound consistent.

5 CP Modeling of the Problem

5.1 Model of the Problem

In this section, we will introduce the variables and constraints used in our model. Fig. 2 shows a visualization of our model for trees of a maximum depth of 3.

Note that in our model, we assume that a decision tree is a perfect tree. This assumption is motivated by the existence of a mapping of any proper binary tree (i.e. a tree where each node has exactly 0 or 2 children) into a perfect one (i.e. proper binary tree with all the leaves at the same level). We add a dummy feature f_0, not belonging to any of the transactions, to the model for unused decision nodes. Unlike other features used at most once per path, this feature is allowed multiple times to allow any tree shape. A node with this value therefore has no transactions from the database on its left branch. This assumption is also motivated by the CP framework. Each potentially used variable should be defined before the start of the search. Figure 3 shows how a proper tree can be made perfect by the use of the dummy feature.

The nodes ($\mathcal{N}$) of a perfect decision tree can be partitioned into two groups: the decision nodes ($\mathcal{N}^{\mathcal{D}}$), which are associated to a decision and which have children, and the leaves ($\mathcal{N}^{\mathcal{L}}$), which do not have children. The decision nodes ($\mathcal{N}^{\mathcal{D}}$) can be further partitioned into the end-nodes $\mathcal{N}^{\mathcal{E}}$, which do not have decision nodes as children, and the nodes $\mathcal{N}^{\mathcal{N}}$, which do. Variables and constraints are defined by the type of the node.

In our model, the number of variables and constraints are independent from the number of transactions in the database and the number of features. In fact,

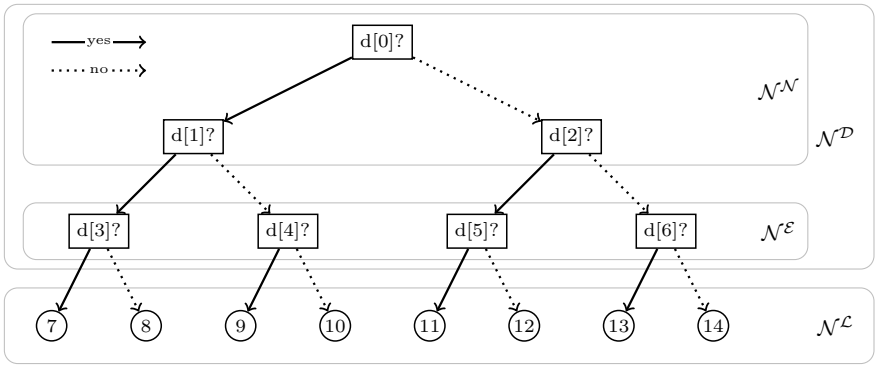
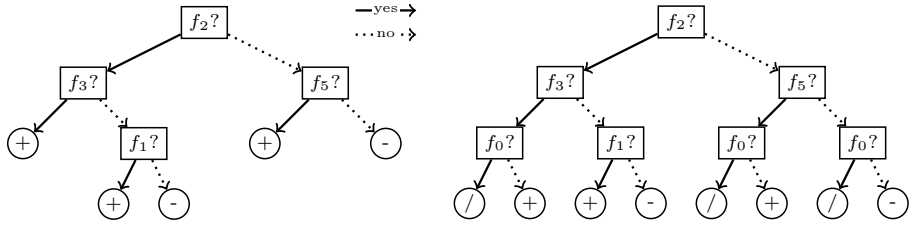


Fig. 2: Representation of a perfect decision tree of depth 3



(a) Proper binary tree (b) Equivalent perfect binary tree

Fig. 3: Example of the use of the dummy feature f_0 to transform the proper binary tree into a perfect binary tree

the number of variables and constraints only depends on the number of nodes in the tree.

5.1.1 Variables

In our model we have variables with the following domains:

$$\text{dom}(d[i]) = \{0, 1, \ldots, m\} \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (6)$$

$$\text{dom}(c^+[i]) = \{0, 1, \ldots, |D^+|\} \qquad \forall i \in \mathcal{N} \qquad (7)$$

$$\text{dom}(c^-[i]) = \{0, 1, \ldots, |D^-|\} \qquad \forall i \in \mathcal{N} \qquad (8)$$

$$\text{dom}(c[i]) = \{0\} \cup \{N_{\min}, N_{\min} + 1, \ldots, |D|\} \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \qquad (9)$$

$$\text{dom}(e[i]) = \{0, 1, \ldots, \min\{|D^+|, |D^-|\}\} \qquad \forall i \in \mathcal{N} \qquad (10)$$

Each decision node has a decision variable d (6) to model the decision feature. Its value can be 0 (representing the dummy feature f_0) or between 1 and m (representing one of the actual features f_1 to f_m). Two counters, c^+ (7) and c^- (8), are defined for each node of the tree. They are used to keep track of how many transactions respectively from D^+ and D^- match the decisions of the ancestors of the node. A third counter c (9), defined at the leaves, tracks the total number of transactions. The minimum number of transactions in each leaf is enforced by

constraining the domain of c from $N_{\min}$ to $|D|$. Value 0 also belongs to the domain and is meant to be used only when the parent of the node is inactive (i.e. when its decision is f_0). An additional variable e (10), defined for each node, keeps track of the error of the sub-tree rooted at that node. Our model does not have an explicit variable for the class of the leaves. However, this can be easily deduced from the solution by taking the class associated with the highest counter.

5.1.2 Constraints

On these variables, we define the following constraints:

$$c^+[i] + c^-[i] = c[i] \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \qquad (11)$$
$$c^+[i] = c^+[\text{left}(i)] + c^+[\text{right}(i)] \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (12)$$
$$c^-[i] = c^-[\text{left}(i)] + c^-[\text{right}(i)] \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (13)$$
$$e[i] = \min\{c^+[i], c^-[i]\} \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \qquad (14)$$
$$e[i] = e[\text{left}(i)] + e[\text{right}(i)] \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (15)$$
$$\textsc{CoverSizeSR}(\mathit{take}(i), \mathit{drop}(i), c^+[i], D^+) \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \qquad (16)$$
$$\textsc{CoverSizeSR}(\mathit{take}(i), \mathit{drop}(i), c^-[i], D^-) \qquad \forall i \in \mathcal{N}^{\mathcal{L}} \qquad (17)$$
$$\textsc{AllDifferentExcept0}(\{d[j] \mid j \in \text{ancestors}(i)\} \cup \{d[i]\}) \qquad \forall i \in \mathcal{N}^{\mathcal{E}} \qquad (18)$$
$$d[i] \neq 0 \Rightarrow \min\{c^+[i], c^-[i]\} > e[i] \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (19)$$
$$d[i] = 0 \Rightarrow (d[\text{left}(i)] = 0 \wedge d[\text{right}(i)] = 0) \qquad \forall i \in \mathcal{N}^{\mathcal{N}} \qquad (20)$$

First, constraint (11) links the counters at the leaves. Second, the counters at the decision nodes are linked to the counters of their children (12, 13). Third, the value of $e[i]$ is assigned to be the minimum between the class counters (14) at the leaves or to the sum of the errors from the children of i (15) for each of the decision nodes. To compute the values of the counters $c^+[i]$ and $c^-[i]$, we need to know which transactions match the decisions of the ancestors of the leaf. To this end, two CoverSizeSR global constraints (16, 17) are added at each leaf, one for each class. The decision variables of the ancestors (an ancestor is either the parent of a node, either the parent of an ancestor) are divided into two distinct sets: The *take* set $\mathit{take}(i) = \{d[j] \mid j \in \text{ancestors}(i) \wedge \text{left}(j) \in \text{ancestors}(i) \cup \{i\}\}$, containing the wanted features, and the *drop* set $\mathit{drop}(i) = \{d[j] \mid j \in \text{ancestors}(i) \wedge \text{right}(j) \in \text{ancestors}(i) \cup \{i\}\}$, containing the rejected features.

The next two constraints ensure the decision tree has no useless nodes. A node is useless if the decision taken in it was already taken in one of the ancestor nodes. An AllDifferentExcept0 (18) is used on the ancestors at each end-node to avoid this. A node is also useless if all the leaves below have the same class. This is avoidable if we constrain the error at the node to be strictly higher than the error of the subtree (19). Finally, when a decision node is inactive, all the decision nodes below should be inactive as well (20).

These constraints are enough to guarantee an optimal, well-formed tree (with no dummy decision feature being a parent from a non-dummy decision and with no decision leading to only one classification).

5.1.3 Objective

The objective is to minimize the sum of the errors at the leaves, which is stored in $e[\text{root}]$.

5.1.4 Redundant constraints

We add a number of redundant constraints to make the search more efficient:

$$\text{dom}(c[i]) = \{0\} \cup \{N_{\min}, N_{\min}+1, \ldots, |D|\} \qquad \forall i \in \mathcal{N} \qquad (21)$$

$$c^+[i] + c^-[i] = c[i] \qquad \forall i \in \mathcal{N} \qquad (22)$$

$$\textsc{CoverSizeSR}(take(i), drop(i), c^+[i], D^+) \qquad \forall i \in \mathcal{N} \backslash areRight(\mathcal{N}) \qquad (23)$$

$$\textsc{CoverSizeSR}(take(i), drop(i), c^-[i], D^-) \qquad \forall i \in \mathcal{N} \backslash areRight(\mathcal{N}) \qquad (24)$$

$$c^+[i] < N_{\min} \Rightarrow d[i] = 0 \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (25)$$

$$c^-[i] < N_{\min} \Rightarrow d[i] = 0 \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (26)$$

$$c[i] < 2N_{\min} \Rightarrow d[i] = 0 \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (27)$$

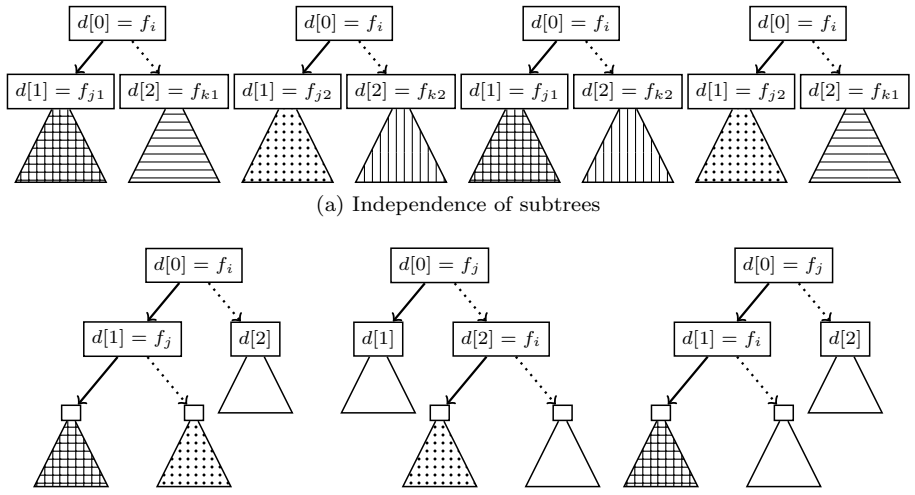
$$d[i] \neq 0 \Rightarrow (c[\text{left}(i)] \geq N_{\min} \wedge c[\text{right}(i)] \geq N_{\min}) \qquad \forall i \in \mathcal{N}^{\mathcal{D}} \qquad (28)$$

Here, $areRight(\mathcal{N}) = \{i \mid i \in \mathcal{N} \wedge i = \text{right}(\text{parent}(i))\}$; it represents the set of nodes being the right child of another node.

Adding a constraint CoverSizeSR for all of the nodes in the tree allows the computation of the exact values of the counters earlier in the tree and therefore helps prune earlier some candidate solutions. However adding them to all the decision nodes is not necessary. Constraints (12) and (13) can be relied on to compute the counters of one child based on the counters of the parent and the sibling. Constraints (23) and (24) are therefore used instead of (16) and (17). This allows a better propagation while using the same number of CoverSizeSR constraints. Constraints (25, 26) concern nodes with only transactions from one class left. When this arises, no decision should be taken in the node. As a minimum number of transactions should be in each activated node, if a given decision node does not have more than twice the threshold, no solution accepts a decision in the node (27). The contrapositives of (25), (26), (27) are also logically true. Combined together, they correspond to (28) which states that if the dummy decision is no longer in the domain, there should be enough transactions in each of the children. This constraint formulation requires to have the counter c (21) and the constraint linking the counters at each node (22).

5.2 Search

The motivation behind the use of a specific search strategy is to exploit the tree-decomposition into subproblems. During search each node of the search tree is associated to a subtree of the decision tree being built. This subtree, identified by the node id `currProblem`, is always rooted on a decision node. The assignment of the decision variables occurs in top-down fashion. Therefore in a given node of the search tree, we can always assume every node in $ancestors($`currProblem`$)$ has been assigned. Algorithm 3 details the pseudo-code of our algorithm.



(a) Independence of subtrees

(b) Redundant subtrees; identically shaded subtrees are identical

Fig. 4: Decompositions

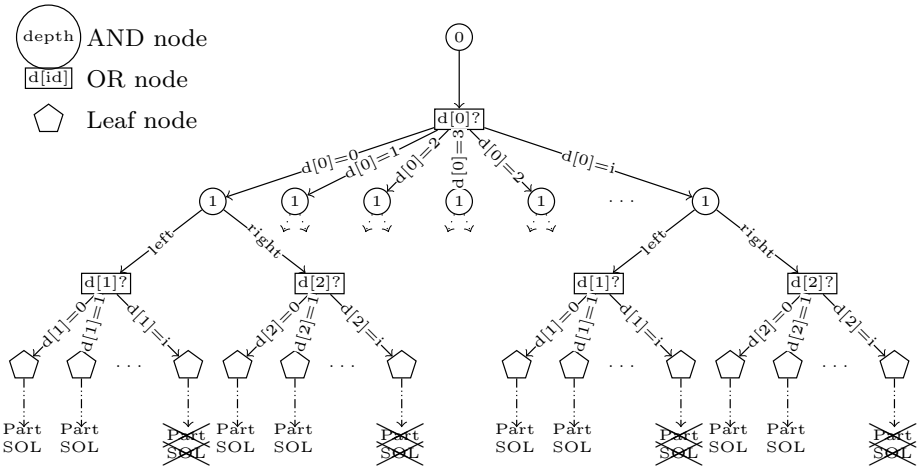


Fig. 5: AND/OR formulation of the search tree

5.2.1 Big picture.

Our search is the composition of three techniques: AND/OR search trees, branch-and-bound optimization, and memorization. Each of them aims to answer one of the specificities of the problem.

5.2.2 Subtree independence.

Given a subtree with its root decision and ancestors' decisions assigned, its two children are totally independent from one another. Any solution from the left child combined with any solution from the right child leads to a solution of the initial

Algorithm 3: AND/OR formulation with cache and minimum pruning

```
1  Function search(currProblem: ∈ 𝒩^𝒟):(Tree,Cost)
2      return ORnode(currProblem, ∞)
3  Function ORnode(currProblem: ∈ 𝒩^𝒟, cost_ub):(Tree,Cost)
4      prefix_hash ← getPrefixHash(currProblem)
5      if storage.contains(prefix_hash) then            // optimal already computed
6          (sol_best, cost_best) ← storage.get(prefix_hash)
7          return (sol_best, cost_best)
8      else
9          cost_best ← cost_ub
10         sol_best ← null
11         forall f ∈ dom(d[currProblem]) do            // following value ordering
12             trail.pushState()
13             try
14                 dom(d[currProblem]) ← {f}
15                 dom(e[currProblem]) ← {v|v < cost_best ∧ v ∈ dom(e[currProblem])}
                    // pruning by minimisation
16                 if currProblem ∈ 𝒩^𝒩 then
17                     (sol_tree, cost_tree) ← ANDnode(currProblem, cost_best, f)
18                 else
19                     sol_tree ← Tree(featureID : f left : null right : null)
20                     cost_tree ← e[currProblem].value
21                 if cost_best > cost_tree then
22                     cost_best ← cost_tree
23                     sol_best ← sol_tree
24             catch Inconsistency
                                                        // node have failed
25             trail.restoreState()
26         storage.add(prefix_hash, (sol_best, cost_best))      // new sol cached
27         return (sol_best, cost_best)
28 Function ANDnode(currProblem: ∈ 𝒩^𝒟, cost_ub, f_root):(Tree,Cost)
29     (sol_left, cost_left) ← ORnode(left(currProblem), cost_ub)          // 1st
30     if cost_left > cost_ub then
31         return (null, ∞)                  // pruning based on cost
32     (sol_right, cost_right) ← ORnode(right(currProblem), cost_ub − cost_left)   // 2nd
33     sol_tree ← Tree(featureID : f_root left : sol_left right : sol_right)
34     return (sol_tree, cost_left + cost_right)
```

subtree. This is illustrated at Fig. 4a. However our goal is to find the best solution and not one solution. Moreover our objective function is the sum of a cost computed in each of the leaves, independently. Therefore, the optimal solution, given a root and ancestors' decisions already assigned, can be computed independently by computing the optimal left child, then the optimal right child and finally combine them. The AND/OR search tree [9,14] framework is well suited for this kind of decomposable problem. The search is composed of two types of search nodes: the OR nodes (line 3) and the AND nodes (line 28). An example of the search tree for a decision tree of depth 2 is shown at Fig. 5.

The AND node is responsible for computing the optimal value of the left child (line 29), then the right child (line 32), and finally returns the composed solution

(line 33). The OR node tests all the possible values for the root decision variable of `currProblem` (line 11). The static ordering used to select the next value to test follows the principle of entropy [7]. The entropy of a set of transaction S is computed using the number of transactions from each class, and is a well-known heuristic in standard algorithms for learning decision trees:

$$Entropy(S) = -\frac{|\{t \in S : v[t] = 1\}|}{|S|} \log_2 \left(\frac{|\{t \in S : v[t] = 1\}|}{|S|} \right) - \frac{|\{t \in S : v[t] = 0\}|}{|S|} \log_2 \left(\frac{|\{t \in S : v[t] = 0\}|}{|S|} \right) \quad (29)$$

The information gain of a feature f is the difference between the initial entropy and the weighted entropy of a partition of the database into transactions with and without the feature:

$$Gain(f) = Entropy(D) - \frac{|\{t \in D : D_{t,f} = 1\}|}{|D|} Entropy(\{t \in D : D_{t,f} = 1\}) - \frac{|\{t \in D : D_{t,f} = 0\}|}{|D|} Entropy(\{t \in D : D_{t,f} = 0\}). \quad (30)$$

The classification is expected to be better when the gain is higher. We sort the values by decreasing gain. This ordering is computed once at the beginning of the search and is reused at every search node. After assigning the selected value, if the subtree still contains decision variables (i.e. if `currProblem` belongs to $\mathcal{N}^{\mathcal{N}}$), then the optimal subtrees are computed using an AND node (line 16). In the other case (i.e. if `currProblem` belongs to $\mathcal{N}^{\mathcal{E}}$), then we have already an optimal subtree (line 18). From all the values tested, the best sub-tree is kept (line 21) and returned (line 27).

5.2.3 Subtree equality.

Two subproblems are equivalent whenever the set of decisions on the paths towards these nodes (the itemsets corresponding to the sets of decisions) are identical. Figure 4b shows how some subtrees can be the same in two different solutions due to paths that represent the same itemset. This is taken care of by using a caching system similar to the one used in the DL8 dynamic programming approach [16]. Two subtrees are equivalent if they share the same assigned prefix. The prefix of node i is composed of the values assigned to the decisions of the ancestors. These values are separated in two distinct sets: The *take* set $\{d[j] \mid j \in \text{ancestors}(i) \wedge \text{left}(j) \in \text{ancestors}(i) \cup \{i\}\}$, and the *drop* set $\{d[j] \mid j \in \text{ancestors}(i) \wedge \text{right}(j) \in \text{ancestors}(i) \cup \{i\}\}$. Two subtrees with the same *take* and *drop* sets are thus equivalent. A hash is computed from these sets and serves as key to store and retrieve the optimal subtree from storage (hashMap). In addition to the decision in the root of the subtree, its cost is also stored, easing the computation. The search for an already computed solution happens at the beginning of an OR node (line 5). A new solution is stored when a new complete optimal subtree is computed, i.e. at the end of the OR node (line 26).

5.2.4 Minimization.

In order to decrease the number of explored search nodes, a pruning by minimization is added to the search. At each of the search nodes, the upper bound of the allowed cost is propagated from node to node. During an OR node, this upper bound is decreased each time a better solution is found (line 21) and the best cost found so far is set as upper bound of the error of the subtree (line 15). During an AND node, the propagated upper bound is first propagated to the computation of the first child. If the result of this first child is above this propagated upper bound, then there is no need to compute the right child since any solution would be above the propagated upper bound (line 30). This is triggered if the best solution was already cached and has a higher cost than the bound or if there is no solution with a cost smaller than the upper bound. An invalid subtree is then returned. If the first child is lower than the upper bound, the second child can be computed and the propagated upper bound for its computation is the difference between the propagated upper bound of the tree and the cost of the already computed tree (line 32). The search starts with an unbounded upper bound (∞). This is the default value if, prior to the search, no information is known about the optimal cost.

5.2.5 Implementation details

Oscar [18], the solver used in our experiment does not implement the AND/OR search tree framework. A simple AND/OR search can be easily implemented using a standard trail based solver [13]. The two main operations of a trailing system are the saveState() and the restoreState() methods. The first one is responsible for saving the current state of the solver and the second one to restore it. In a typical OR tree, a save is done before trying a new assignment and start a new OR node. The state is restored when the node is fully explored. In an AND/OR tree, the logic is the same. Algorithm 3 depicts, in the ORnode() method, where the save and restore are being done. Just before trying a new assignment, at line 12, a save of the state is performed. Then the assignment is tested and the node fully explored. The exploration of the node is embedded in an exception catching mechanism. In case of inconsistencies (i.e. a proof of no solutions) during the exploration of the node, an exception is thrown leading to the stop in the exploration. After the exploration, a restoration of the state is required (line 25). For further implementation details, the source code is available online[2].

6 Results

We compared our algorithm to two exact methods developed in earlier studies: BinOCT [26] and DL8 [16], both of which solve exactly the same optimization problem as our method. Notice that the optimal solution trees found by these algorithms are in most cases equal, therefore leading to the same model. Only in rare cases, two different trees are equally good. In any case, this can't be used as a criterion to efficiently differentiate the prediction efficiently between the exact

[2] `https://bitbucket.org/helene_verhaeghe/classificationtree`

methods since they are both able to output each an optimal tree. Both studies have already evaluated the quality of the resulting trees experimentally. It was shown in [3] that the more optimal is a tree on the training set, the more accurate it is on a test set. These results were confirmed in [26,16]. Therefore we decided to focus our experiments on the run time performance of our algorithm, and not on the validation of the quality of the trees.

6.1 First benchmark

Dataset	n	n^+	n^-	m
anneal	812	625	187	93
audiology	216	57	159	148
australian-credit	653	357	296	125
breast-wisconsin	683	444	239	120
diabetes	768	500	268	112
german-credit	1000	700	300	112
heart-cleveland	296	160	136	95
hepatitis	137	111	26	68
hypothyroid	3247	2970	277	88
ionosphere	351	225	126	445
kr-vs-kp	3196	1669	1527	73
letter	20000	813	19187	224
lymph	148	81	67	68
mushroom	8124	4208	3916	119
pendigits	7494	780	6714	216
primary-tumor	336	82	254	31
segment	2310	330	1980	235
soybean	630	92	538	50
splice-1	3190	1655	1535	287
tic-tac-toe	958	626	332	27
vehicle	846	218	628	252
vote	435	267	168	48
yeast	1484	463	1021	89
zoo-1	101	41	60	36

Table 1: Description of the instances

The benchmark[3] is composed of instances from the CP4IM[4] and UCI [5] websites. Their description is given at Table 1. BinOCT is a MIP-based approach running on CPLEX. It does not allow to give a specific value for $N_{\min}$. If a timeout is reached, the method outputs its best solution so far. We used the implementation available online with as arguments the depth, the timeout (10 min) and a polishing time (2.5 min). The polishing time is used to configure the CPLEX solver. At timeout minus the polishing time, CPLEX changes its search strategy. Polishing [22] is time consuming, but it allows improving a solution when the search stagnates. DL8 is a dynamic programming approach. It computes a subset of the frequent

[3] Available in the repository
[4] `https://dtai.cs.kuleuven.be/CP4IM/datasets/`
[5] `https://archive.ics.uci.edu/ml/index.php`

Dataset	Depth	$N_{\min} = 1$ DL8 obj	DL8 t	BinOCT obj	BinOCT t	CP obj	CP t
anneal	2	**137***	1	**137***	206	**137***	**< 1**
anneal	3	**112***	37	**112**	TO	**112***	**2**
anneal	4	∞	TO	121	TO	**91***	**142**
anneal	5	∞	TO	120	TO	**84**	TO
audiology	2	**10***	**< 1**	**10***	60	**10***	**< 1**
audiology	3	**5***	62	7	TO	**5***	**5**
audiology	4	∞	TO	**1**	TO	**1**	TO
audiology	5	∞	TO	4	TO	**0***	**3**
australian-credit	2	**87***	2	**87***	206	**87***	**< 1**
australian-credit	3	**73***	124	86	TO	**73***	**9**
australian-credit	4	∞	TO	85	TO	**57**	TO
breast-wisconsin	2	**22***	2	**22***	44	**22***	**< 1**
breast-wisconsin	3	**15***	103	16	TO	**15***	**6**
breast-wisconsin	4	∞	TO	15	TO	**7***	**493**
diabetes	2	**177***	1	180	TO	**177***	**< 1**
diabetes	3	**162***	93	171	TO	**162***	**8**
diabetes	4	∞	TO	169	TO	**137**	TO
german-credit	2	**267***	2	**267**	TO	**267***	**< 1**
german-credit	3	**236***	129	249	TO	**236***	**8**
german-credit	4	∞	TO	244	TO	**204**	TO
heart-cleveland	2	**60***	**< 1**	**60***	312	**60***	**< 1**
heart-cleveland	3	**41***	17	43	TO	**41***	**4**
heart-cleveland	4	**25***	515	39	TO	**25***	**265**
heart-cleveland	5	∞	TO	34	TO	**9**	TO
hepatitis	2	**16***	**< 1**	**16***	8	**16***	**< 1**
hepatitis	3	**10***	4	12	TO	**10***	**1**
hepatitis	4	**3***	54	10	TO	**3***	**49**
hepatitis	5	∞	TO	7	TO	**0***	**8**
hypothyroid	2	**70***	4	**70***	178	**70***	**< 1**
hypothyroid	3	**61***	122	62	TO	**61***	**4**
hypothyroid	4	∞	TO	62	TO	**53***	**183**
ionosphere	2	**32***	50	**32**	TO	**32***	**1**
ionosphere	3	∞	TO	29	TO	**22***	**328**
ionosphere	4	∞	TO	26	TO	**13**	TO
kr-vs-kp	2	**418***	2	**418**	TO	**418***	**< 1**
kr-vs-kp	3	**198***	74	301	TO	**198***	**2**
kr-vs-kp	4	∞	TO	877	TO	**144***	**107**
kr-vs-kp	5	∞	TO	675	TO	**81**	TO
letter	2	∞	TO	813	TO	**599***	**1**
letter	3	∞	TO	813	TO	**369***	**108**
letter	4	∞	TO	∞	TO	**294**	TO

Table 2: Results (part 1) Time out (TO) = 10 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (*t*, in seconds)) for a given $N_{\min} = 1$ in bold, optimal *obj* proven indicated with $*$

itemsets and then builds the optimal tree from it. This approach does not output any intermediate non-optimal tree. We used the implementation provided by the authors with as arguments the depth and the minimum support (value of $N_{\min}$).

The Table 2 and Table 3 shows the results for the three methods (DL8, BinOCT and ours) with $N_{\min} = 1$ using a timeout of 10 mins. The second part of Table 4 and Table 5 shows the results for two methods (DL8 and ours) and some variations of our approach (without the caching, labelled CP-c, and without the pruning

Dataset	Depth	$N_{\min} = 1$					
		DL8		BinOCT		CP	
		obj	t	obj	t	obj	t
lymph	2	**22***	**< 1**	**22***	17	**22***	**< 1**
lymph	3	**12***	2	13	TO	**12***	**1**
lymph	4	**3***	43	8	TO	**3***	**42**
lymph	5	∞	TO	8	TO	**0***	**1**
mushroom	2	**252***	27	520	TO	**252***	**< 1**
mushroom	3	∞	TO	396	TO	**8***	**4**
mushroom	4	∞	TO	160	TO	**0***	**< 1**
pendigits	2	∞	TO	**153**	TO	**153***	**1**
pendigits	3	∞	TO	496	TO	**47***	**50**
pendigits	4	∞	TO	780	TO	**14**	TO
primary-tumor	2	**58***	**< 1**	**58***	5	**58***	**< 1**
primary-tumor	3	**46***	**< 1**	49	TO	**46***	**< 1**
primary-tumor	4	**34***	**2**	39	TO	**34***	4
primary-tumor	5	**26***	**14**	37	TO	**26***	71
segment	2	**9***	49	**9**	TO	**9***	**< 1**
segment	3	∞	TO	6	TO	**0***	**2**
segment	4	∞	TO	21	TO	**0***	**1**
soybean	2	**55***	**< 1**	**55***	19	**55***	**< 1**
soybean	3	**29***	2	42	TO	**29***	**1**
soybean	4	**14***	33	16	TO	**14***	**14**
soybean	5	**8***	**315**	24	TO	**8***	497
splice-1	2	**508***	143	522	TO	**508***	**< 1**
splice-1	3	∞	TO	574	TO	**224***	**125**
splice-1	4	∞	TO	1087	TO	**141**	TO
tic-tac-toe	2	**282***	**< 1**	**282***	10	**282***	**< 1**
tic-tac-toe	3	**216***	**< 1**	231	TO	**216***	**< 1**
tic-tac-toe	4	**137***	**3**	169	TO	**137***	**3**
tic-tac-toe	5	**63***	**16**	128	TO	**63***	64
vehicle	2	**75***	23	**75**	TO	**75***	**< 1**
vehicle	3	∞	TO	60	TO	**26***	**45**
vehicle	4	∞	TO	84	TO	**13**	TO
vote	2	**17***	**< 1**	**17***	8	**17***	**< 1**
vote	3	**12***	2	13	TO	**12***	**1**
vote	4	**5***	23	11	TO	**5***	**16**
vote	5	**1***	**248**	5	TO	**1***	394
yeast	2	**437***	2	**437**	TO	**437***	**< 1**
yeast	3	**403***	74	430	TO	**403***	**6**
yeast	4	∞	TO	412	TO	**366***	**287**
zoo-1	2	**0***	**< 1**	**0***	**< 1**	**0***	**< 1**

Table 3: Results (part 2) Time out = 10 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (t, in seconds)) for a given $N_{\min} = 1$ in bold, optimal *obj* proven indicated with $*$

using bounds, labelled CP-m) with $N_{\min} = 5$ using a timeout of 10 mins. This comparison does not include BinOCT since its implementation cannot take into account $N_{\min}$. A value of 5 is chosen, as this yields results that are more statistically significant. Table 6 summarizes our results. For each of the algorithms, the number of instances where the optimality is proven, the solution found is the best among the tested algorithms, the algorithm was the fastest and timeout is reached are gathered.

Dataset	Depth	$N_{\min} = 5$							
		DL8		CP		CP-c		CP-m	
		obj	t	obj	t	obj	t	obj	t
anneal	2	**137***	**< 1**	**137***	**< 1**	**137***	**< 1**	**137***	**< 1**
anneal	3	**112***	31	**112***	**3**	**112***	**3**	**112***	4
anneal	4	**94***	591	**94***	**172**	**94***	257	**94***	296
anneal	5	∞	TO	**92**	TO	**92**	TO	**92**	TO
audiology	2	**11***	**< 1**	**11***	**< 1**	**11***	**< 1**	**11***	**< 1**
audiology	3	**7***	2	**7***	**1**	**7***	**1**	**7***	2
audiology	4	**4***	**43**	**4***	56	**4***	55	**4***	74
audiology	5	**1***	512	**1***	534	**1***	**475**	**1**	TO
australian-credit	2	**87***	2	**87***	**< 1**	**87***	**< 1**	**87***	**< 1**
australian-credit	3	**74***	90	**74***	**11**	**74***	12	**74***	14
australian-credit	4	∞	TO	**60**	TO	66	TO	66	TO
breast-wisconsin	2	**22***	3	**22***	**< 1**	**22***	**< 1**	**22***	**< 1**
breast-wisconsin	3	**15***	80	**15***	**8**	**15***	**8**	**15***	12
breast-wisconsin	4	∞	TO	**9**	TO	**9**	TO	**9**	TO
diabetes	2	**177***	1	**177***	**< 1**	**177***	**< 1**	**177***	**< 1**
diabetes	3	**162***	90	**162***	**10**	**162***	11	**162***	13
diabetes	4	∞	TO	**138**	TO	**138**	TO	**138**	TO
german-credit	2	**267***	2	**267***	**< 1**	**267***	**< 1**	**267***	**< 1**
german-credit	3	**236***	122	**236***	**11**	**236***	13	**236***	14
german-credit	4	∞	TO	**205**	TO	**205**	TO	**205**	TO
heart-cleveland	2	**60***	**< 1**	**60***	**< 1**	**60***	**< 1**	**60***	**< 1**
heart-cleveland	3	**41***	15	**41***	**5**	**41***	8	**41***	7
heart-cleveland	4	**27***	404	**27***	**333**	**27***	528	**27***	595
heart-cleveland	5	∞	TO	**17**	TO	**17**	TO	18	TO
hepatitis	2	**16***	**< 1**	**16***	**< 1**	**16***	**< 1**	**16***	**< 1**
hepatitis	3	**11***	2	**11***	**1**	**11***	2	**11***	2
hepatitis	4	**8***	**36**	**8***	62	**8***	86	**8***	99
hepatitis	5	**5***	**299**	6	TO	6	TO	8	TO
hypothyroid	2	**70***	3	**70***	**< 1**	**70***	**< 1**	**70***	**< 1**
hypothyroid	3	**62***	95	**62***	**4**	**62***	**4**	**62***	8
hypothyroid	4	∞	TO	**54***	**236**	**54***	323	**54***	570
ionosphere	2	**32***	48	**32***	**1**	**32***	**1**	**32***	**1**
ionosphere	3	∞	TO	**22***	**389**	**22***	443	**22**	TO
ionosphere	4	∞	TO	**16**	TO	**16**	TO	**16**	TO
kr-vs-kp	2	**418***	2	**418***	**< 1**	**418***	**< 1**	**418***	**< 1**
kr-vs-kp	3	**198***	63	**198***	**4**	**198***	**4**	**198***	7
kr-vs-kp	4	∞	TO	**144***	**214**	**144***	256	**144***	483
kr-vs-kp	5	∞	TO	**98**	TO	**98**	TO	132	TO
letter	2	∞	TO	**599***	**5**	**599***	**5**	**599***	**5**
letter	3	∞	TO	**369**	TO	**369**	TO	531	TO
letter	4	∞	TO	**296**	TO	**296**	TO	301	TO

Table 4: Results (part 1) Time out (TO) = 10 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (*t*, in seconds)) for a given $N_{\min} = 5$ in bold, optimal *obj* proven indicated with $*$

Our method outperforms the two others on most of the instances. It could find and prove optimality on roughly 83% of the instances within the time limit. The best solution found was reached by our method in almost every cases. However, DL8 performs better on small instances such as *hepatitis*, *lymph* or *primary-tumor*. The large difference between BinOCT and our method can be explained by the benefits of the AND/OR search that is not used by BinOCT. The gap with DL8 can be partially explained by the cost pruning. It can possibly also be explained by the itemset mining algorithms used: DL8 lacks the optimizations found in the CoverSize constraint [23].

Dataset	Depth	$N_{\min} = 5$							
		DL8		CP		CP-c		CP-m	
		obj	t	obj	t	obj	t	obj	t
lymph	2	**22***	**< 1**	**22***	**< 1**	**22***	**< 1**	**22***	**< 1**
lymph	3	**13***	**1**	**13***	**1**	**13***	2	**13***	2
lymph	4	**7***	**15**	**7***	34	**7***	40	**7***	59
lymph	5	**4***	**166**	**4***	595	**4**	TO	**4**	TO
mushroom	2	**252***	24	**252***	**< 1**	**252***	**< 1**	**252***	**< 1**
mushroom	3	∞	TO	**8***	**15**	**8***	**15**	**8***	44
mushroom	4	∞	TO	**0***	**< 1**	**0***	**< 1**	**0**	TO
pendigits	2	∞	TO	**153***	**2**	**153***	**2**	**153***	**2**
pendigits	3	∞	TO	**47***	**256**	**47***	268	**47***	415
pendigits	4	∞	TO	**15**	TO	19	TO	19	TO
primary-tumor	2	**58***	**< 1**	**58***	**< 1**	**58***	**< 1**	**58***	**< 1**
primary-tumor	3	**46***	**< 1**	**46***	**< 1**	**46***	**< 1**	**46***	**< 1**
primary-tumor	4	**40***	**1**	**40***	4	**40***	6	**40***	5
primary-tumor	5	**34***	**8**	**34***	65	**34***	162	**34***	104
segment	2	**9***	41	**9***	1	**9***	**< 1**	**9***	1
segment	3	∞	TO	**2***	**69**	**2***	71	**2***	136
segment	4	∞	TO	**0***	186	**0***	**181**	**0**	TO
soybean	2	**55***	**< 1**	**55***	**< 1**	**55***	**< 1**	**55***	**< 1**
soybean	3	**29***	2	**29***	**1**	**29***	**1**	**29***	**1**
soybean	4	**15***	27	**15***	**21**	**15***	26	**15***	35
soybean	5	**13***	**239**	**13**	TO	**13**	TO	**13**	TO
splice-1	2	**508***	89	**508***	**1**	**508***	**1**	**508***	**1**
splice-1	3	∞	TO	**225***	**156**	**225***	188	**225***	249
splice-1	4	∞	TO	**142**	TO	**142**	TO	**142**	TO
tic-tac-toe	2	**282***	**< 1**	**282***	**< 1**	**282***	**< 1**	**282***	**< 1**
tic-tac-toe	3	**216***	**< 1**	**216***	**< 1**	**216***	**< 1**	**216***	**< 1**
tic-tac-toe	4	**137***	**3**	**137***	7	**137***	9	**137***	5
tic-tac-toe	5	**63***	**16**	**63***	83	**63***	282	**63***	167
vehicle	2	**75***	20	**75***	**< 1**	**75***	**< 1**	**75***	1
vehicle	3	∞	TO	**28***	**83**	**28***	85	**28***	143
vehicle	4	∞	TO	**17**	TO	**17**	TO	**17**	TO
vote	2	**18***	**< 1**	**18***	**< 1**	**18***	**< 1**	**18***	**< 1**
vote	3	**13***	**1**	**13***	**1**	**13***	**1**	**13***	**1**
vote	4	**6***	**13**	**6***	17	**6***	20	**6***	41
vote	5	**3***	**118**	**3***	234	**3***	300	4	TO
yeast	2	**437***	2	**437***	**< 1**	**437***	**< 1**	**437***	**< 1**
yeast	3	**403***	70	**403***	**7**	**403***	9	**403***	**7**
yeast	4	∞	TO	**367***	**421**	**367**	TO	**367***	541
zoo-1	2	**0***	**< 1**	**0***	**< 1**	**0***	**< 1**	**0***	**< 1**

Table 5: Results (part 2) Time out = 10 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (t, in seconds)) for a given $N_{\min} = 5$ in bold, optimal *obj* proven indicated with $*$

	$N_{\min} = 1$			$N_{\min} = 5$			
	DL8	BinOCT	CP	DL8	CP	CP-c	CP-m
Proven optimality	49(61%)	13(16%)	**68**(85%)	54(67%)	**65**(81%)	63(79%)	59(74%)
Best solution found	49(61%)	21(26%)	**80**(100%)	54(67%)	**79**(99%)	77(96%)	72(90%)
Fastest	17(21%)	1(1%)	**63**(79%)	26(32%)	**52**(65%)	36(45%)	27(34%)
Time out	31(39%)	67(84%)	**12**(15%)	25(31%)	**15**(19%)	17(21%)	21(26%)

Table 6: Summary of the results

Finally, the effects of the cache and the pruning using the best known partial solutions can be observed. CP-c gives the results of our method when the cache system is not used and CP-m gives the results when the pruning using the best partial solution is not used. The cache becomes really useful at depth 4 (or more) and some instances greatly benefit from it (e.g. the *tic-tac-toe* benchmark with a depth of 5 improves its timing by 70% when adding the cache). The effect of the pruning is significant in some cases. On some benchmarks such as *mushroom*, *hypothyroid*, *ionosphere* or *vehicle*, the pruning improves greatly the solution (ex. on *hypothyroid* depth 4, the time is divided by 2.4). This improvement indicates that when searching for the best subtree, the best one is found early, prunning a fair amount of the search space concerning the subtree.

6.2 Second benchmark

Dataset	n	n^+	n^-	m
dexter	300	150	150	25736
dorothea	800	78	722	100000
dota2	92650	48782	43868	226

Table 7: Description of the instances (after binarization)

Dataset	Depth	$N_{\min} = 1$			
		DL8		CP	
		obj	t	obj	t
dexter	2	∞	TO	**135**	TO
dexter	3	∞	TO	**129**	TO
dexter	4	∞	TO	**124**	TO
dexter	5	∞	TO	**120**	TO
dorothea	2	∞	TO	**39**	TO
dorothea	3	∞	TO	**78**	TO
dota2	2	∞	TO	**42610***	33
dota2	3	∞	TO	**42107**	TO

Table 8: Results (part 3) Time out = 20 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (t, in seconds)) for a given $N_{\min} = 1$ in bold, optimal *obj* proven indicated with $*$

To evaluate the scaling of our method, we tested it on some bigger instances from the UCI website. The description of these instances is available in Tab. 7. The results with $N_{\min} = 1$ are available in Tab. 8 and with $N_{\min} = 5$ Tab. 8. A timeout of 20 minutes was used for these instances. These datasets required binarization first[6].

[6] Binarized versions available in the repository

Dataset	Depth	$N_{\min} = 5$							
		DL8		CP		CP-c		CP-m	
		obj	t	obj	t	obj	t	obj	t
dexter	2	**136***	13	**136***	12	**136***	12	**136***	15
dexter	3	**130***	99	**130***	264	**130***	260	**130***	198
dexter	4	**125***	100	**125***	849	**125***	829	**125***	602
dexter	5	**120***	1066	**120**	TO	**120**	TO	**120**	TO
dorothea	2	∞	TO	**39**	TO	**39**	TO	**39**	TO
dorothea	3	∞	TO	**78**	TO	**78**	TO	**78**	TO
dota2	2	∞	TO	**42610***	97	**42610***	100	**42610***	124
dota2	3	∞	TO	**42107**	TO	**42107**	TO	**42107**	TO

Table 9: Results (part 3) Time out = 20 min, best value (objective (*obj*, in number of wrongly classified transactions) or time (*t*, in seconds)) for a given $N_{\min} = 5$ in bold, optimal *obj* proven indicated with $*$

As shown by the *dota2* instance, in the line of the *letter* instance of the first benchmark, increasing the number of transactions affects less our algorithm than DL8. This is due to the use of the bitsets inside the CoverSize constraint. The *dexter* instance allows us to see that good performance can be achieved with a big number of features. Unfortunately, with too many features, as shown on the *dorothea* instance, even a depth-2 tree is not obtainaible within time out.

7 Conclusion

We presented a new approach for efficiently creating an optimal decision tree of limited depth. On most of the benchmarks, it gives the best solution within the allocated time and is the fastest to prove optimality.

We believe our approach can be extended in a number of different ways. It is straightforward to extend it to the multiclass setting, by adding counters and CoverSizeSR constraints for each of the additional classes. We assumed the input data was binary; if the data is not binary, it can be binarized beforehand [6]. Of particular interest can also be addition of further constraints and the use of other cost functions that can be expressed as a sum of costs at the leaves.

Remark

This paper is an extended and improved version of the paper initially accepted to CP2019 in the journal fast track. The initial work was also presented in a 2-page summary abstract [24] at BNAIC2019, a national conference, and as a 4-page summary abstract [25], in the sister conference track at IJCAI20.

References

1. Aghaei, S., Azizi, M.J., Vayanos, P.: Learning optimal and fair decision trees for non-discriminative decision-making (2019)

2. Babaki, B., Guns, T., De Raedt, L.: Stochastic constraint programming with and-or branch-and-bound. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017, pp. 539–545 (2017)
3. Bertsimas, D., Dunn, J.: Optimal classification trees. Machine Learning **106**(7), 1039–1082 (2017)
4. Bessiere, C., Hebrard, E., O'Sullivan, B.: Minimising decision tree size as combinatorial optimisation. In: I.P. Gent (ed.) Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings, *Lecture Notes in Computer Science*, vol. 5732, pp. 173–187. Springer (2009). DOI 10.1007/978-3-642-04244-7_16. URL `https://doi.org/10.1007/978-3-642-04244-7\_16`
5. Bonfietti, A., Lombardi, M., Milano, M.: Embedding decision trees and random forests in constraint programming. In: L. Michel (ed.) Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings, *Lecture Notes in Computer Science*, vol. 9075, pp. 74–90. Springer (2015). DOI 10.1007/978-3-319-18008-3_6. URL `https://doi.org/10.1007/978-3-319-18008-3\_6`
6. Breiman, L.: Classification and regression trees. Routledge (1984)
7. Cover, T.M., Thomas, J.A.: Elements of information theory. John Wiley & Sons (2012)
8. Dechter, R., Mateescu, R.: The impact of AND/OR search spaces on constraint satisfaction and counting. In: Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings, pp. 731–736 (2004)
9. Dechter, R., Mateescu, R.: And/or search spaces for graphical models. Artificial intelligence **171**(2-3), 73–106 (2007)
10. Demeulenaere, J., Hartert, R., Lecoutre, C., Perez, G., Perron, L., Régin, J.C., Schaus, P.: Compact-table: efficiently filtering table constraints with reversible sparse bit-sets. In: International Conference on Principles and Practice of Constraint Programming, pp. 207–223. Springer (2016)
11. Hancock, T., Jiang, T., Li, M., Tromp, J.: Lower bounds on learning decision lists and trees. Information and Computation **126**(2), 114–122 (1996)
12. Hyafil, L., Rivest, R.L.: Constructing optimal binary decision trees is np-complete. Inf. Process. Lett. **5**(1), 15–17 (1976). DOI 10.1016/0020-0190(76)90095-8. URL `https://doi.org/10.1016/0020-0190(76)90095-8`
13. Laurent Michel, Pierre Schaus, Pascal Van Hentenryck: MiniCP: A lightweight solver for constraint programming (2018). Available from `https://minicp.bitbucket.io`
14. Marinescu, R., Dechter, R.: And/or tree search for constraint optimization. In: Proc. of the 6th International Workshop on Preferences and Soft Constraints. Citeseer (2004)
15. Narodytska, N., Ignatiev, A., Pereira, F., Marques-Silva, J.: Learning optimal decision trees with SAT. In: J. Lang (ed.) Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden, pp. 1362–1368. ijcai.org (2018). DOI 10.24963/ijcai.2018/189. URL `https://doi.org/10.24963/ijcai.2018/189`
16. Nijssen, S., Fromont, E.: Mining optimal decision trees from itemset lattices. In: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 530–539. ACM (2007)
17. Nijssen, S., Fromont, É.: Optimal constraint-based decision tree induction from itemset lattices. Data Min. Knowl. Discov. **21**(1), 9–51 (2010)
18. OscaR Team: OscaR: Scala in OR (2012). Available from `https://bitbucket.org/oscarlib/oscar`
19. Quinlan, J.R.: Induction of decision trees. Mach. Learn. **1**(1), 81–106 (1986). DOI 10.1023/A:1022643204877. URL `https://doi.org/10.1023/A:1022643204877`
20. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
21. Rossi, F., Van Beek, P., Walsh, T.: Handbook of constraint programming. Elsevier (2006)
22. Rothberg, E.: An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS Journal on Computing **19**(4), 534–541 (2007)
23. Schaus, P., Aoga, J.O., Guns, T.: Coversize: a global constraint for frequency-based itemset mining. In: International Conference on Principles and Practice of Constraint Programming, pp. 529–546. Springer (2017)

24. Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C., Schaus, P.: Learning optimal decision trees using constraint programming. In: K. Beuls, B. Bogaerts, G. Bontempi, P. Geurts, N. Harley, B. Lebichot, T. Lenaerts, G. Louppe, P.V. Eecke (eds.) Proceedings of the 31st Benelux Conference on Artificial Intelligence (BNAIC 2019) and the 28th Belgian Dutch Conference on Machine Learning (Benelearn 2019), Brussels, Belgium, November 6-8, 2019, *CEUR Workshop Proceedings*, vol. 2491. CEUR-WS.org (2019). URL `http://ceur-ws.org/Vol-2491/abstract109.pdf`
25. Verhaeghe, H., Nijssen, S., Pesant, G., Quimper, C., Schaus, P.: Learning optimal decision trees using constraint programming (extended abstract). In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020, Tokyo, Japan, 2020 (2020)
26. Verwer, S., Zhang, Y.: Learning optimal classification trees using a binary linear program formulation. In: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019, pp. 1625–1632. AAAI Press (2019). DOI 10.1609/aaai.v33i01.33011624. URL `https://doi.org/10.1609/aaai.v33i01.33011624`

Exploitation de l'apprentissage par renforcement avec la Programmation par Contraintes ou la Recherche Locale - Cas d'application dans l'industrie automobile

V. Antuori[1,2] **E. Hébrard**[1,3] **M-J. Huguet**[1] **S. Essodaigui**[2] **A. Nguyen**[2]

[1] LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
[2] Renault, France
[3] ANITI, Université de Toulouse, France
{vantuori, hebrard, huguet}@laas.fr,{siham.essodaigui,alain.nguyen}@renault.com

Résumé

Le travail présenté porte sur un problème de transport de composants dans un atelier d'assemblage de l'industrie automobile. Deux premiers modèles de programmation par contraintes (PPC) ont été proposés et les évaluations expérimentales montrent qu'ils surpassent les performances d'une méthode de recherche locale actuellement utilisée par Renault sur un jeu de données industrielles. De plus, ces expérimentations mettent en évidence qu'une heuristique adéquate permet généralement de guider le solveur vers une solution avec un faible nombre de *backtracks*. Nous proposons alors d'*apprendre* une politique heuristique efficace via l'apprentissage par renforcement, et d'exploiter cette politique dans plusieurs méthodes : PPC avec redémarrage rapide, *recherche à divergences limitées* et recherche locale multi-départ. Ces méthodes sont plus efficaces que que les modèles initiaux de PPC sur les instances industrielles, mais également sur des instances générées aléatoirement.

1 Problème étudié

Le problème considéré dans cet article[1] provient d'un atelier d'assemblage du constructeur automobile Renault. Il consiste à déplacer des composants entre leurs points de production et leurs points de consommation. Ces composants sont transportés à l'aide de chariots dédiés, i.e. spécifiques à chaque composant. Il faut collecter ces chariots lorsqu'ils sont pleins pour les livrer à leurs lieux de consommation, mais également collecter les chariots lorsqu'ils sont vides pour les ramener à leur lieux de production. Les cadences de production et de consommation d'un même composant sont identiques. Ces cadences définissent alors des fenêtres de temps, il s'agit de faire ces requêtes (collecte et livraison du chariot plein, collecte et livraison du chariot vide), avant la fin du prochain cycle de production/consommation, ou ces requêtes devront être effectuées à nouveau, la fin d'une fenêtre de temps marquant le début de la fenêtre de temps des opérations du prochain cycle. De plus, chaque composant dispose de sa propre cadence de production/consommation. Les chariots peuvent s'assembler de manière à former un train de chariots, qui ne doit pas dépasser une certaine taille, chaque chariot ayant une taille différente. Le problème étudié se ramène à un problème de collectes et de livraisons à un véhicule, avec fenêtres de temps et contrainte de capacité, sans objectif à optimiser. La particularité du problème industriel est que les requêtes sont périodiques et que cette périodicité est propre à chaque composant. Le but est de planifier les différentes opérations, en respectant les contraintes de fenêtres de temps et de capacité, jusqu'à un horizon temporel donné (allant d'un *shift* d'une durée de 7h15 jusqu'à une semaine de 6 jours composés chacun de 3 *shifts*).

1. L'article original a été présenté à la conférence CP 2020[1]

2 Modèles PPC

Dans l'article nous présentons deux modèles de programmation par contraintes (PPC). Le premier modélise un problème d'ordonnancement disjonctif, avec précédences, fenêtres de temps et contraintes de ré-

servoir, et est basé sur des variables d'ordre entre les opérations. Le second correspond à un problème de type voyageur de commerce et s'appuie sur la contrainte Circuit. Nous comparons ces deux approches avec une méthode de recherche locale actuellement utilisée par Renault. Les résultats obtenus montrent qu'avec une bonne heuristique, il est possible d'obtenir des solutions quasiment sans *backtrack* pour la plupart des instances industrielles mais, il reste difficile d'obtenir des solutions pour des instances plus contraintes. Aussi, le modèle d'ordonnancement est le plus efficace.

3 Méthodes basées sur l'apprentissage d'une heuristique

Apprentissage par renforcement. Afin d'améliorer la résolution de ce problème, nous proposons d'*apprendre* une heuristique efficace via apprentissage par renforcement. Plus précisément, nous travaillons sur une relaxation du problème (minimisation du retard maximum), qui peut être décrite comme un processus de décision markovien (PDM). Dans ce PDM, un état est une séquence d'opérations (séquence vide à l'état initial), et l'ensemble des actions d'un état est l'ensemble des opérations qui peuvent étendre cette séquence sans violer les contraintes de précédence ni de capacité. On obtient une pénalité lorsqu'une action augmente le retard maximal, la pénalité étant l'accroissement marginal du retard maximum. Nous souhaitons alors apprendre une politique stochastique pour naviguer dans ce PDM, et qui minimise l'espérance de la somme des pénalités (et donc le retard maximum). L'espace des états étant exponentiellement large, nous avons besoin d'un descripteur d'une abstraction d'un état. Pour cela, nous définissons, à chaque état de notre PDM, une liste de 4 critères pour chacune des actions réalisables. Nous agrégeons ces critères par une combinaison linéaire, pour aboutir à un score de *fitness* pour chacune des actions. Ce vecteur de *fitness* est ensuite transformé en distribution de probabilité sur les actions par une fonction `softmax`. La politique stochastique consiste à choisir aléatoirement une action en suivant cette distribution. L'objectif de l'apprentissage par renforcement est alors de trouver les poids de cette combinaison linéaire qui vont minimiser l'espérance de la fonction objectif, à savoir le retard maximum, en suivant cette politique. Pour réaliser cet apprentissage, nous utilisons un algorithme de *policy gradient*, algorithme classique d'apprentissage par renforcement.

Exploitation de l'heuristique apprise. Nous proposons trois méthodes permettant de tirer profit de l'heuristique apprise lors de l'étape d'apprentissage par renforcement. La première méthode est basée sur une adaptation du modèle PPC d'ordonnancement dans lequel nous intégrons une stratégie de redémarrage très agressif, couplé avec l'heuristique stochastique, dans le but d'explorer très rapidement différentes parties de l'arbre. La seconde méthode est basée sur le même modèle PPC mais utilise une stratégie d'exploration de type *Limited Discreancy Search* (LDS) afin de dévier le moins possible de l'heuristique. Pour cela, nous utilisons une version déterministe de l'heuristique, qui consiste à choisir l'opération ayant le meilleur score de *fitness*. La troisième méthode est une recherche locale multi-départ consistant à répéter le processus glouton (basé sur l'heuristique stochastique) suivi d'une recherche locale. Nous avons proposé deux voisinages en $O(nm)$ où n est le nombre total d'opérations et m le nombre de composants différents. Le premier voisinage est un *swap* entre deux opérations de la séquence. Le second voisinage est un "double" *swap*, entre les deux opérations de collecte et de livraison d'un chariot plein, et celles d'un chariot vide.

Résultats expérimentaux. Pour compléter les instances industrielles, nous avons généré un ensemble d'instances aléatoires plus contraintes que les instances réelles. Les expérimentations montrent que les trois méthodes surpassent chacune les performances des premiers modèles de PPC, sur les instances industrielles mais également sur les instances générées aléatoirement. Malgré cela, de nombreuses instances restent non résolues.

À la suite de ces travaux, plusieurs perspectives se dessinent : d'abord nous proposons d'enrichir le modèle d'apprentissage, par l'ajout de critères, ou par le remplacement de la fonction linéaire par une fonction plus complexe comme par exemple un réseau de neurones. Ensuite nous visons à intégrer l'heuristique trouvée dans une approche de type recherche arborescente de Monte-Carlo, car cette approche peut s'appuyer fortement sur une heuristique gloutonne. Enfin le problème présenté est une partie d'un problème plus global, dans lequel on doit affecter les opérateurs aux composants, avant de planifier les routes, et nous projetons de nous attaquer à ce problème.

Références

[1] Valentin Antuori, Emmanuel Hébrard, Marie-José Huguet, Siham Essodaigui et Alain Nguyen : Leveraging Reinforcement Learning, Constraint Programming and Local Search : A Case Study in Car Manufacturing. *In Principles and Practice of Constraint Programming. CP 2020*, pages 657–672, 2020.

Leveraging Reinforcement Learning, Constraint Programming and Local Search: A Case Study in Car Manufacturing

Valentin Antuori[1,2], Emmanuel Hebrard[1,3], Marie-José Huguet[1], Siham Essodaigui[2], and Alain Nguyen[2]

[1] LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France
{vantuori,hebrard,huguet}@laas.fr
[2] Renault, France
{valentin.antuori,siham.essodaigui,alain.nguyen}@renault.com
[3] ANITI, Université de Toulouse, France

Abstract. The problem of transporting vehicle components in a car manufacturer workshop can be seen as a large scale single vehicle pickup and delivery problem with periodic time windows. Our experimental evaluation indicates that a relatively simple constraint model shows some promise and in particular outperforms the local search method currently employed at Renault on industrial data over long time horizon. Interestingly, with an adequate heuristic, constraint propagation is often sufficient to guide the solver toward a solution in a few backtracks on these instances. We therefore propose to learn efficient heuristic policies via reinforcement learning and to leverage this technique in several approaches: rapid-restarts, limited discrepancy search and multi-start local search. Our methods outperform both the current local search approach and the classical CP models on industrial instances as well as on synthetic data.

Keywords: Constraint Programming · Reinforcement Learning · Local Search · Scheduling · Travelling Salesman Problem

1 Introduction

Improving the production line is a constant concern in the industry. The car manufacturer Renault has long been interested in models and techniques from Operations Research and Constraint Programming to tackle the various routing and scheduling problems arising from the production process.

Recent advances in Artificial Intelligence and in particular in Machine Learning (ML) open up many new perspectives for solving large scale combinatorial optimization problems with promising results popularized by the success of AlphaGo and AlphaZero [14, 15]. In particular, several approaches combining reinforcement learning and deep neural networks to guide the reward strategy have been proposed for solving the traveling salesman problem (TSP) [2, 4]. Moreover, the combination of ML and classical combinatorial techniques seems very

promising. For instance, the integration of standard TSP heuristics with a neural network heuristic policy outperforms the pure ML approaches [5].

In Section 2 we introduce the problem of planning the flow of vehicle components across the assembly lines. More precisely, given fixed production cycles, logistics operators are in charge of collecting components from, and delivering them to, working stations in such a way that there is no break in the manufacturing process. This problem is in many ways similar to the *Torpedo Scheduling Problem* [7], from the 2016 ACP challenges [13]: pickup and delivery operations are to be scheduled, and these operations are repeated in time because the commodity is being produced constantly at a fixed rate. However, in our problem, components are carried over using trolleys that can be stacked into a "train" that should not exceed a given maximal length. It should be noted that in this paper we consider the problem associated to a single operator whose route is to be optimized. However, the more general problem for Renault is to assign components (or equivalently working stations) to operators as well as to plan the individual routes, and the longer term objective is to proactively design the layout of the assembly line to reduce the cost of logistics operations.

We first evaluate in Section 3 two basic constraint programming (CP) models in `Choco` [11] and compare them to the current method used by Renault: a local search (LS) method implemented in `LocalSolver`[4]. From this preliminary study, we observe that although the problem can be hard for both approaches, CP shows promising results compare to LS. Moreover, if solving the problem via backtracking search seems very unlikely given its size, and if stronger filtering techniques seem to be ineffective, greedy "dives" are often surprisingly successful.

We therefore propose in Section 4 to learn effective stochastic heuristic policies via reinforcement learning. Then, we show how these policies can be used within different tailored approaches: constraint programming with rapid restarts, limited discrepancy search, and multi-start local search approach.

Finally, since industrial benchmarks are easily solved by all new methods introduced in this paper, we generated a synthetic dataset designed to be more challenging. In Section 6 we report experiments on this dataset that further demonstrates the efficiency of the proposed methods.

2 Problem Definition

The Renault assembly line consists of a set of m components to be moved across a workshop, from the point where they are produced to where they are consumed. Each component is produced and consumed by two unique machines, and it is carried from one to the other using a specific trolley. When a trolley is filled at the production point for that component, an operator must bring it to its consumption point. Symmetrically, when a trolley is emptied it must be brought to the corresponding production point. A production cycle is the time c_i taken to produce (resp. consume) component i, that is, to fill (resp. empty) a trolley. The

[4] `https://www.localsolver.com/home.html`

end of a production cycle marks the start of the next, hence there are $n_i = \frac{H}{c_i}$ cycles over a time horizon H. There are two pickups and two deliveries at every cycle k of each component i: the pickup pe_i^k and delivery de_i^k of the empty trolley from consumption to production and the pickup pf_i^k and delivery df_i^k of the full trolley from production to consumption. The processing time of an operation a is denoted p_a and the travel time between operations a and b is denoted $D_{a,b}$.

Let P be the set of pickup operations and D the set of delivery operations: $P = \bigcup_{i=1}^{m} \left(\bigcup_{k=1}^{n_i} \{pe_i^k, pf_i^k\} \right)$, $D = \bigcup_{i=1}^{m} \left(\bigcup_{k=1}^{n_i} \{de_i^k, df_i^k\} \right)$. The problem is to compute the bijection σ (let $\rho = \sigma^{-1}$ be its inverse) between the $|A| = n$ first positive integers to the operations $A = P \cup D$ satisfying the following constraints.

Time windows. As production never stops, all four operations of the k-th cycle of component i must happen during the time window $[(k-1)c_i, kc_i]$. Let $r_{a_i^k}$ (resp. $d_{a_i^k}$) be the release date $(k-1)c_i$ (resp. due date kc_i) of operation a_i^k of the k-th cycle of component i. The start time of operation $\sigma(j)$ is:

$$s_{\sigma,j} = \begin{cases} r_{\sigma(j)} & \text{if } j = 1 \\ \max(r_{\sigma(j)}, s_{\sigma,j-1} + p_{\sigma(j-1)} + D_{\sigma(j-1),\sigma(j)}) & \text{otherwise} \end{cases}$$

Then, the completion time $e_{\sigma,j} = s_{\sigma,j} + p_{\sigma(j)}$ of operation $\sigma(j)$ must be lower than its due date:

$$\forall j \in [1, n],\ e_{\sigma,j} \leq d_{\sigma(j)} \tag{1}$$

Precedences. Pickups must precede deliveries.

$$\rho(pf_i^k) < \rho(df_i^k) \wedge \rho(pe_i^k) < \rho(de_i^k)\ \forall i \in [1, m]\ \forall k \in [1, n_i] \tag{2}$$

Notice that there are only two possible orderings for the four operations of a production cycle. Indeed, since the first delivery (of either the full or the empty trolley) and the second pickup take place at the same location, doing the second pickup before the first delivery is dominated (w.r.t. the train length and the time windows). Hence Equation 3 is valid, though not necessary:

$$\rho(df_i^k) < \rho(pe_i^k) \vee \rho(de_i^k) < \rho(pf_i^k)\ \forall i \in [1, m]\ \forall k \in [1, n_i] \tag{3}$$

Train length. The operator may assemble trolleys into a train[5], so a pickup need not be directly followed by its delivery. However, the total length of the train of trolleys must not exceed a length $T_{\max}$. Let t_i be the length of the trolley for component i, and let $t_{a_i^k} = t_i$ if $a_i^k \in P$ and $t_{a_i^k} = -t_i$ otherwise, then:

$$\forall j \in [1, n],\ \sum_{l=1}^{j} t_{\sigma(l)} \leq T_{\max} \tag{4}$$

[5] Trolleys are designed so that they can be extracted out of the train in any order

This is a particular case of the *single vehicle pickup and delivery problem with capacity and time windows constraints*. However, there is no objective function, and instead, feasibility is hard. Moreover, the production-consumption cycles entail a very particular structure: the four operations of each component must take place in the same time windows and this is repeated for every cycle. As a result, one of the best method, Large Neighborhood Search [12], is severely hampered since it relies on the objective to evaluate the moves and the insertion of relaxed requests is often very constrained by the specific precedence structure.

3 Baseline Models

We designed two CP models: a variant of a single resource scheduling problem and a variant of a TSP. Then, we describe the current `LocalSolver` model.

Scheduling-based model. The importance of time constraints in the problem studied, suggests that a CP model based on scheduling would be relevant [1]. The problem is a single resource (the operator) scheduling problem with four types of non overlapping operations (pickup and delivery of empty and full trolleys). For each operation $a \in A$, we define the variable $s_a \in [r_a, d_a]$ as the starting date of operation a. Moreover, for each pair of operations $a, b \in A$, we introduce a Boolean variable x_{ab} standing for their relative ordering. In practice, we need much fewer than n^2 Boolean variables as the time windows and Constraint (2) entails many precedences which can be either ignored or directly posted.

$$x_{ab} = \begin{cases} 1 \Leftrightarrow s_b \geq s_a + p_a + D_{a,b} \\ 0 \Leftrightarrow s_a \geq s_b + p_b + D_{b,a} \end{cases} \qquad \forall (a,b) \in A \qquad (5)$$

$$x_{df_i^k pe_i^k} \vee x_{de_i^k pf_i^k} \qquad \forall i \in [1,m]\ \forall k \in [1,n_i] \qquad (6)$$

Constraint (5) chanel the two sets of variables, and constraint (6) encodes Equation (3). Finally, Constraint (4) can be seen as a reservoir resource with limited capacity, which is filled by pickups, and emptied by deliveries. We use the algorithm from [9] to propagate it on starting date variables using the precedence graph implied by Boolean variables and precedences from Constraint (2).

TSP-based model. The second model is an extension of the first one, to wich we add extra variables and constraints from the model for TSP with time windows proposed in [6]. We need two fake operations, 0 and $n+1$, for the start and the end of the route. For each operation $a \in A \cup \{0, n+1\}$, there is a variable $next_a$ that indicates which operation directly follows a. Also, a variable pos_a indicates the position of the operation a in the sequence of operations. We need another variable $train_a \in [0, T_{\max}]$ which represents the length of the train before the operation a. The following equations express the constraints of the problem:

$$train_0 = 0 \wedge train_{next_a} = train_a + t_a \qquad \forall a \in A \cup \{0\} \qquad (7)$$

$$pos_{pf_i^k} < pos_{df_i^k} \wedge pos_{pe_i^k} < pos_{de_i^k} \qquad \forall i \in [1,m]\ \forall k \in [1,n_i] \qquad (8)$$

$$s_0 = 0 \wedge s_{next_a} \geq s_a + p_a + D_{a,next_a} \qquad \forall a \in A \cup \{0\} \qquad (9)$$

165

Constraint (7) encodes the train length at every point of the sequence using the ELEMENT constraint and constraint (8) ensures that pickups precede their deliveries. Constraint (9) ensure the accumulation of the time along the tour. Moreover, we use the CIRCUIT constraint to enforce the variables $next$ to form an Hamiltonian circuit. Additional redundant constraints are used to make the channeling between variables and improve filtering in addition to the constraint from the first model: $x_{ab} \implies next_b \neq a$, $pos_b > pos_a + 1 \implies next_a \neq b$, $pos_b > pos_a \Leftrightarrow x_{ab}$, $pos_a = \sum_{b \in A \cup \{0,n+1\}} x_{ab}$ and ALLDIFFERENT(pos).

Search strategy. Preliminary experiments revealed that branching on variables in an ordering "consistent" with the sequence of operations was key to solving this problem via CP. In the TSP model, we simply branch on the variables $next$ in ascending order, and choose first the operation with least earliest start time. In the scheduling model, we compute the set of pairs of operations a, b such that $\{s_a, s_b\}$ is Pareto-minimal, draw one pair uniformly at random within this set and assign x_{ab} so that the operation with least release date comes first. In conjunction with constraint propagation, this strategy acts as the "nearest city" heuristic in TSP. Indeed, since the sequence of past operations is known, the earliest start time of an operation depends primarily on the distance from the latest operation in the partial sequence (it also depends on the release date).

LocalSolver. The `LocalSolver` (LS) model is similar to the TSP model. It is based on a variable of type list $\boldsymbol{seq}$, a special decision variable type that represent the complete tour: $seq_j = a$ means operation a is performed at position j. This variable is channeled with another list variable $\boldsymbol{pos}$ with $pos_a = j \Leftrightarrow seq_j = a$. We need two other list variables: $train$ and s to represent respectively the length of the train and the start time of the operation at a given position.

$$s_1 = 0 \wedge$$

$$s_j = \max(r_{seq_j}, s_{j-1} + p_{seq_{j-1}} + D_{seq_{j-1},seq_j}) \qquad \forall j \in [2,n] \quad (10)$$

$$train_1 = 0 \ \wedge train_j = train_{j-1} + t_{seq_j} \qquad \forall j \in [2,n] \quad (11)$$

$$s_j + p_{seq_j} \leq d_{seq_j} \qquad \forall j \in [1,n] \quad (12)$$

$$pos_{pf_i^k} < pos_{df_i^k} \wedge pos_{pe_i^k} < pos_{de_i^k} \qquad \forall i \in [1,m]\ \forall k \in [1,n_i] \quad (13)$$

$$pos_{df_i^k} < pos_{pe_i^k} \vee pos_{de_i^k} < pos_{pf_i^k} \qquad \forall i \in [1,m]\ \forall k \in [1,n_i] \quad (14)$$

$$\text{COUNT}(\boldsymbol{seq}) = 0 \quad (15)$$

The last constraint (15) acts like the global constraint ALL DIFFERENT and therefore ensures that $\boldsymbol{seq}$ is a permutation. Surprisingly, relaxing the due dates and using the maximal tardiness as objective tends to degrade the performance of `LocalSolver`, hence we kept the satisfaction version.

3.1 Preliminary Experiments

The industrial data we have collected fall into three categories. In the industrial assembly line, each category is associated to one logistic operator who has the

charge of a given set of components. In practice, these datasets are relatively underconstrained, with potentially quite large production cycles (time windows) for each component. For each category, denoted by S, L and R, we consider three time horizons: 43 500, 130 500 and 783 000 *hundredths of a minutes* which corresponds to a shift of an operator (7 hours and 15 minutes), a day of work (made up of three shifts) and a week (6 days) respectively.

We then have 9 industrial instances from 400 to more than 10 000 operations. The main differences between those three categories are the number of components, and the synchronicity of the different production cycles. For S instances, there are only 5 components, and their production cycles are almost the same and very short. The other two instances have more than 30 components, with various production cycles (some cycles are short and others are very long).

The CP models were implemented using the constraint solver `Choco` 4.10 [11], and the `LocalSolver` version was 9.0. All experiments of this section were run on Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz with a timeout of 1 hour.

The results are given in Table 1. The first column (Cl) denotes the category of the instance and second column (H) denotes the temporal horizon. For the two CP models (Scheduling and TSP), two indicators are given, the CPU time (in seconds), and the numbers of fails for solved instances. For `LocalSolver`, we give the CPU time when the instances are solved before the time limit.

The CP scheduling-based model solves all of the industrial instances without any fail except for instances R and L on the whole week. Notice, however, that when using generic heuristics such as *Domain over Weighted Degree* [3], only 3 instances could be solved by the same model in less than an hour. Moreover, `LocalSolver` cannot solve the largest instances, and requires much more CPU time in general. Although industrial instances are clearly very underconstrained, they are not trivial. Moreover, even on underconstrained instances, wrong early choices may lead to infeasible subtrees from which we cannot easily escape. In particular, the number of fails for the largest instance of category R shows that the very deep end of the search tree is likely explored in a brute-force manner.

One key factor in solving these instances is for the variable ordering to follow the ordering of the sequence of operations being built. Indeed, the propagation is much more efficient in this way, and in particular, the earliest start times of future operations can be easily computed and, as mentioned in the description of the heuristic, it reflects the distance from the last operation in the route.

We observe that the scheduling-based model is the fastest. Moreover, the TSP-based model contains too many variables and constraints, and run out of memory for the bigger horizon. Instances without any fails show that the first branch can be very slow to explore as propagation in nodes close to the root can be very costly, in particular with the TSP-based model.

We draw two conclusions from these preliminary experiments: First, the basic CP (or LS) models cannot reliably solve hard instances[6]. It is unlikely that stronger propagation would be the answer, as the TSP model (with stronger

[6] There might be too few data points to make that claim on industrial instances, but it is clearly confirmed on the synthetic benchmark (see Table 2).

Table 1: Comparison on the industrial instances

Cl	H	Scheduling		TSP		`LocalSolver`
		CPU	#fail	CPU	#fail	CPU
S	Shift	0.275	0	3.999	0	26
S	Day	0.840	0	48.855	0	566
S	Week	17.747	0	memory out		timeout
L	Shift	7.129	0	36.033	8	121
L	Day	23.468	0	344.338	8	3539
L	Week	318.008	6	memory out		timeout
R	Shift	8.397	0	41.615	14	1065
R	Day	44.215	0	417.116	15	timeout
R	Week	timeout	773738	memory out		timeout

filtering) tends to be less effective, and does not scale very well. Second, building the sequence "chronologically" helps significantly, which explains why CP models outperform local search. As a consequence, greedy runs of the CP solver are surprisingly successful. Therefore, we propose to *learn* efficient heuristic policies and explore methods that can take further advantage of these heuristics.

4 Reinforcement Learning

The search for a feasible sequence can be seen as a Markov Decision Process (MPD) whose *states* are partial sequences σ, and *actions* $\mathcal{A}(\sigma)$ are the operations that can extend σ without breaking precedence (2) nor capacity constraints (4).

In order to apply Reinforcement Learning (RL) to this MDP, it is convenient to relax the due date constraints and replace them by the minimization of the maximum tardiness: $\max\{L(\sigma, j) \mid 1 \leq j \leq |\sigma|\}$ where $L(\sigma, j) = e_{\sigma,j} - d_{\sigma(j)}$ is the tardiness of operation $\sigma(j)$. We also define the maximum tardiness on intervals: $L(\sigma, j, l) = \max\{L(\sigma, q) \mid j \leq q \leq l\}$, and we write $L(\sigma)$ for $L(\sigma, 1, |\sigma|)$. Since in this case operations can finish later than their due dates, it is necessary to make explicit the precedence constraints due to production cycles:

$$\max(\rho(df_i^{k-1}), \rho(de_i^{k-1})) < \min(\rho(pe_i^k), \rho(pf_i^k)) \; \forall i \in [1, m] \; \forall k \in [2, n_i] \quad (16)$$

Then we can further restrict the set $\mathcal{A}(\sigma)$ to operations that would not violate Constraints (16), nor (3). As a result, any state of the MDP reachable from the empty state is feasible for the relaxation. Finally, we can define the *penalty* $R(\sigma, j)$ for the j-th decision as the marginal increase of the objective function when the j-th operation is added to σ: $R(\sigma, j) = L(\sigma, 1, j) - L(\sigma, 1, j-1))$.

Now we can apply standard RL to seek for an effective stochastic heuristic policy for selecting the next operation. Moreover, as the MPD defined above is exponentially large, it is common to abstract a state σ with a small descriptor,

namely, $\boldsymbol{\lambda}(\sigma, a)$ a vector of four criteria for operation $a \in A$ in state σ. Let $lst(a, \sigma)$ be the latest starting time of the task a in order to satisfy constraint (1) with respect to the tasks in σ and constraints (2) and (3). For each operation $a \in \mathcal{A}(\sigma)$, we compute $\boldsymbol{\lambda}(\sigma, a)$ as follows:

$$\boldsymbol{\lambda}_1(\sigma, a) = (lst(a, \sigma) - \max(r_a, e_{\sigma,|\sigma|} - L(\sigma) + D_{\sigma(|\sigma|),a}))/\max_{b \in A}\{d_b - r_b\} \quad (17)$$

$$\boldsymbol{\lambda}_2(\sigma, a) = \max(r_a - (e_{\sigma,|\sigma|} - L(\sigma)), D_{\sigma(|\sigma|),a})/\max_{b,c \in A^2}\{D_{b,c}\} \quad (18)$$

$$\boldsymbol{\lambda}_3(\sigma, a) = 1 - |t_a|/T_{\max} \quad (19)$$

$$\boldsymbol{\lambda}_4(\sigma, a) = \begin{cases} 1 & \text{if } a \in P \\ 0 & \text{otherwise} \end{cases} \quad (20)$$

Criterion (17) can be seen as the operation's *emergency*: the distance to the due date of the task. Criterion (18) is the *travel time* from the last task in the sequence. For both of these criterion, we use $e_{\sigma,|\sigma|} - L(\sigma)$ instead of $e_{\sigma,|\sigma|}$ as the end time of the partial sequence σ to offset the impact of previous choices. Criterion (19) is the *length* of the trolley. Indeed, since all operations must eventually be done, doing the operations which have the highest consumption of the "train" resource earlier leaves more freedom for later. Finally, criterion (20) penalizes pickups as leaving a trolley in the train for too long is wasteful.

We want to learn a stochastic policy $\pi_{\boldsymbol{\theta}}$, governed by a set of parameters $\boldsymbol{\theta}$ which gives the probability distribution over the set of actions $\mathcal{A}(\sigma)$ available at state σ. We first define a fitness function $f(\sigma, a)$ as a simple linear combination of the criteria: $f(\sigma, a) = \boldsymbol{\theta}^\intercal \boldsymbol{\lambda}(\sigma, a)$. Then, we use the `softmax` function to turn the fitness function into a probability distribution (ignore parameter β for now).

$$\forall a \in \mathcal{A}(\sigma)\ \pi_{\boldsymbol{\theta}}(a \mid \sigma) = \frac{e^{(1-f(\sigma,a))/\beta}}{\sum_{b \in \mathcal{A}(\sigma)} e^{(1-f(\sigma,b))/\beta}} \quad (21)$$

4.1 Policy Gradient

As we look for a stochastic policy, the goal is to find the value of $\boldsymbol{\theta}$ minimizing the expected maximum value (i.e., maximum tardiness) $J(\boldsymbol{\theta})$ of solutions σ produced by the policy $\pi_{\boldsymbol{\theta}}$. The basic idea of policy gradient methods is to minimize $J(\boldsymbol{\theta})$ by gradient descent, that is: iteratively update $\boldsymbol{\theta}$ by subtracting the gradient. We resort to the REINFORCE learning rule [16] to get the gradient of $J(\boldsymbol{\theta})$:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\sigma \sim \pi_{\boldsymbol{\theta}}(\sigma)}[L(\sigma) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\sigma)] \quad (22)$$

Which can then be approximated via Monte-Carlo sampling over S samples:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{S} \sum_{k=1}^{S} L(\sigma_k) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\sigma_k) \quad (23)$$

We can decompose Equation (23) in order to give a specific penalty to each decision (for every position j) of the generated solutions:

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{S} \sum_{k=1}^{S} \sum_{j=1}^{n} G(\sigma_k, j) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(\sigma_k(j) \mid \sigma_k(1, j-1)) \tag{24}$$

The penalty function takes into account that a decision can only affect the future marginal increase of tardiness: we replace the overall tardiness $L(\sigma)$ by:

$$G(\sigma, j) = \begin{cases} R(\sigma, j) & \text{if } j = n \\ R(\sigma, j) + \gamma * G(\sigma, j+1) & \text{otherwise} \end{cases}$$

The value of γ controls how much decisions impact future tardiness. For $\gamma = 0$, we only take into account the immediate penalty. Conversely $\gamma = 1$ means we consider the sum of the penalties from position j.

4.2 Learning algorithm

We learn a value of $\boldsymbol{\theta}$ for the synthetic dataset (Section 6) with the REINFORCE rule. Given a set of instances I, we learn by batch of size $S = q|I|$, in other words, we generate q solutions for each instance. The value of $\boldsymbol{\theta}$ is initialized at random, then we apply the following three steps until convergence or timeout:

1. Generate S solutions following $\pi_{\boldsymbol{\theta}}$.
2. Compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ according to Equation (24)
3. Update the value of θ as follows: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

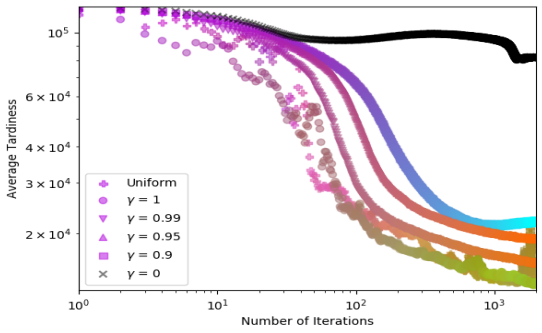
We found out that using a classic `softmax` function did not discriminate enough, and hence acts as random policy. To circumvent this, we use the parameter β in Equation (21) to control the trade off between the quality and the diversity of generated solutions. For a low value of β the policy always chooses the best candidate, whereas a large value yields a more "balanced" policy. It turns out that $\beta = 0.1$ was a good value for learning in our case.

Then, we have evaluated the impact of γ to compute the gradient in Equation (24). Recall that γ controls how much importance we give to the j-th decision in the total penalty: the overall increase of the tardiness from j to n for $\gamma = 1$ or only the instant increase for $\gamma = 0$. Moreover, we also tried to give the penalty $L(\sigma)$ uniformly for every decision j of the policy, instead of giving individual penalties $G(\sigma, j)$. We denote this penalty strategy "Uniform".

For each variant, we plot in Figure (1a) the average performance $L(\sigma)$ of the policy after each iteration (notice the log-scale both for X and Y). Here we learn on all generated instances of a day horizon (40 instances), for 2000 iterations.

High values for γ are always better. For low value of γ, the (local) optimum is higher, and in some cases the method may not even converge, e.g. for $\gamma = 0$. Using uniformly the overall tardiness gives similar results to $\gamma = 1$, however it is less stable, and in Fig. (1a) we observe that it diverges after 1000 iterations.

170



(a) Convergence for different values of γ

Cl	By Inst.	By Cat.	All
A	1011	1130	1128
B	6417	6766	6797
C	14622	15184	15069
D	20285	20578	20640

(b) Different training sets

Fig. 1: Behavior of reinforcement learning

Datapoints are colored with the vector $\boldsymbol{\theta}$ interpreted as a RGB value ($|\boldsymbol{\theta}| = 4$ but it can be characterized by three values after we normalize so that $\sum_{i=1}^{4} \boldsymbol{\theta}_i = 1$)[7]. We can see for instance that $\gamma = 0.9$ finds a value of $\boldsymbol{\theta}$ that is significantly different from all other methods ($\langle 0, 0.63, 0.29, 0.07 \rangle$). Interestingly, "Uniform" and $\gamma = 1$ not only converge to the same average tardiness, but to similar $\boldsymbol{\theta}$'s (respectively $\langle 0.30, 0.49, 0.16, 0.04 \rangle$ and $\langle 0.25, 0.56, 0.15, 0.04 \rangle$). However, $\gamma = 1$ finds values of $\boldsymbol{\theta}$ that seem closer to the target ("greener") earlier, although it is not really apparent from the value of $L(\sigma)$. These values of θ indicate that a good heuristic is mainly a compromise between the *emergency* and the *travel time*[8]. Moreover, the travel time tends to be more important on larger horizon, because it a longer term impact: all subsequent operations are affected.

Finally, we report in Figure (1b) the average tardiness for each instances following $\pi_{\boldsymbol{\theta}}$. The gain of learning specifically for a given (class of) instance(s) is at best marginal. This is not so surprising as we abstract states with a very simple model using a few criteria. However, it means that the value of $\boldsymbol{\theta}$ learnt on the full dataset is relevant to most instances.

5 Using the Heuristic Policy

We have implemented two types of approaches taking advantage of the stochastic policy learnt via RL: integrating it within CP as a randomized branching heuristic, and using it to generate sequences to be locally optimized via steepest descent in a multi-start local search. Here we describe the necessary modifications of the CP model, and we propose an efficient local search neighborhood.

[7] To highlight the differences we also normalize the RGB values and omit $\gamma = 0$

[8] Although the importance of a criterion also depends on the distribution of the values of λ after normalization, we are confident that the first two criteria are more important than the other two.

5.1 Constraint Programming

In order to use the stochastic policy described in Section 4 in a CP solver, we need to slightly change the scheduling model. We introduce a new set of variables seq_j, one for each position j, standing for the operation at that position, with the following channeling constraint: $x_{seq_j seq_l} = 1 \ \forall j < l \in [1, n]$

We branch only on the variables seq in lexicographic order. Therefore, the propagation for this constraint is straightforward: when branching on $seq_j = a$, as all variables $seq_l \ \forall l < j$ are already instantiated, we set $x_{ab} = 1, \ \forall b \in A \backslash \{seq_l \mid l < j\}$. Conversely, after assigning seq_j we can remove a from the domain of seq_{j+1} if there exists $b \in A \backslash \{seq_l \mid l \leq j\}$ such that the domain of the variable x_{ab} is reduced to $\{0\}$. Moreover, we can easily enforce *Forward Consistency* on $\{seq_1, \ldots, seq_j\}$ with respect to precedence and train size constraints (2) and (4) as well as Constraint (3), i.e., when $\{seq_1, \ldots, seq_{j-1}\}$ are all instantiated, we can remove all values of seq_j that cannot extend the current subsequence. Therefore, we do not need the *Reservoir resources* propagator anymore.

We propose two strategies based on this CP model using the learned policy.

1. *Softmax policy and rapid restart.* In this method we choose randomly the next operation according to the softmax policy (Eq. 21). In order to explore quickly different part of the search tree, we rely on a rapid restart strategy, following a Luby [10] sequence with a factor 15.
2. *Limited Discrepancy Search.* As the key to solve those instances is to follow good heuristics, and to deviate as little as possible from them, *limited discrepancy search* (LDS) [8] fits well with this approach. We run the LDS implementation of `Choco`, which is an iterative version: the discrepancy starts from 0, to a maximum discrepancy parameter incrementally. For this approach we use the deterministic version of the policy $\pi(\sigma) = \arg\min_a f(\sigma, a)$.

5.2 Local Search

The solutions found by the heuristic policy can often be improved by local search moves. Therefore, we also tried a multi-start local search whereby we generate sequences with the heuristic policy, and then improve them via steepest descent. Sequences are generated using the same model used for RL (i.e., with relaxed due dates). Therefore, generated sequences respect all constraints, except (1) and we consider a neighborhood that preserves all other constraints as well. Then we apply the local move that decrease the most the maximum tardiness $L(\sigma)$ until no such move can be found. We use two types of moves and the time complexity of an iteration (i.e., computing, and commiting to, the best move) is in $O(nm)$.

We recall that $L(\sigma, j, l) = \max\{L(\sigma, q) \mid j \leq q \leq l\}$ is the maximum tardiness among all operations between positions j and l in σ.

Swap moves. The first type of moves consists in swapping the values of $\sigma(j)$ and $\sigma(l)$. First, we need to make sure that the ordering of the operations within a

given component remains valid, i.e., satifies constraints (2) and (16): a pickup (resp. delivery) operation must stay between its preceding and following deliveries (resp. pickups) for the same component. For every operation a, a valid range between the position of its predecessor $pr(a)$ and of its successor $su(a)$ can be computed in constant time. Then, for all $j \in [1, n]$ we shall consider only the swaps between j and l for $l \in [j+1, su(\sigma(j))]$ and such that $pr(\sigma(l)) \leq j$.

The second condition for the move to be valid is that the swap does not violate constraint (4), i.e., the maximum length of the train. Let $\tau_j = \sum_{l=1}^{j-1} t_{\sigma(l)}$ be the length of the train before the j-th operation. After the swap we have $\tau_{j+1} = \tau_j + t_{\sigma(l)}$ which must be less than $T_{\max}$. At all other ranks until l, the difference will be $t_{\sigma(l)} - t_{\sigma(j)}$, we only need to check the constraint for the maximum train length, that is: $\max\{\tau_q \mid j \leq q \leq l\} + t_{\sigma(l)} - t_{\sigma(j)} \leq T_{\max}$. This can be done in constant (amortized) time for all the swaps of operations a by computing the maximum train length incrementally for each $l \in [j+1, su(\sigma(j))]$.

Then, we need to forecast the maximum tardiness of the sequence σ' where the operations at positions j and l are swapped, i.e., compute the marginal cost of the swap. The tardiness of operations before position j do not change. However, we need to compute the new tardiness $L(\sigma', j)$ and $L(\sigma', l)$ at positions j and l, respectively. Moreover, we need to compute $L(\sigma', j+1, l-1)$ the new maximum tardiness for operations strictly between j and l and $L(\sigma', l+1, n)$ the new maximum tardiness for operations strictly after l.

The new end time $e_{\sigma',j}$ of operation $\sigma'(j) = \sigma(l)$ and hence the tardiness at position j is $L(\sigma', j) = e_{\sigma',j} - d_{\sigma'(j)}$ can be computed in $O(1)$ as follows:

$$e_{\sigma',j} = p_{\sigma'(j)} + \max(r_{\sigma'(j)}, (e_{\sigma',j-1} + D_{\sigma'(j-1),\sigma'(j)}))$$

Next, operations $\sigma(j+1), \ldots, \sigma(l-1)$ remain in the same order and $\sigma(j+1)$ is shifted by a value $\Delta = e_{\sigma',j} + D_{\sigma'(j),\sigma'(j+1)} - e_{\sigma,j} - D_{\sigma(j),\sigma,j+1}$. However, subsequent operations may not all be equally time-shifted. Indeed, when $\Delta < 0$ there may exist an operation whose release date prevents a shift of Δ.

Let $g_j = r_{\sigma(j)} - s_{\sigma(j)}$ be the *maximum left shift* (negative shift of highest absolute value) for the j-th operation, and let $g_{j,l} = \max\{g_q \mid j \leq q \leq l\}$.

Proposition 1. *If the sequence does not change between positions j and l, a time-shift $\Delta < 0$ at position j yields a time-shift* $\max(\Delta, g_{j,l})$ *at position l.*

Let $L_\Delta(\sigma, j, l)$ be the maximum tardiness on the interval $[j, l]$ of sequence σ time-shifted by Δ from position j. We can define $L_{-\infty}(\sigma, j, l)$ the maximum tardiness on the interval $[j, l]$ for an infinite negative time-shift:

$$L_{-\infty}(\sigma, j, l) = \max\{L(\sigma, q, l) + g_{j,q} \mid j \leq q \leq l\} \tag{25}$$

Proposition 2. *If $\Delta < 0$ then* $L_\Delta(\sigma, j, l) = \max(\Delta + L(\sigma, j, l), L_{-\infty}(\sigma, j, l))$.

Conversely, when $\Delta > 0$ some of the time-shift may be "absorbed" by the waiting time before an operation. However, there is little point in moving operations coming before a position j with a non-negative waiting time (i.e., where

$s_{\sigma(j)} = r_{\sigma(j)}$), as this operation and all subsequent operations would not profit from the reduction in travel time. Therefore we consider only swaps whose earliest position j is such that $\forall q > j,\ g_q < 0$. As a results, if there is a positive time-shift Δ at $q > j$, we know that $L_\Delta(\sigma, q, l) = \Delta + L(\sigma, q, l)$. Moreover, the values of $L(\sigma, j, l)$, $g_{j,l}$ and $L_{-\infty}(\sigma, j, l)$ can be computed incrementally as:

$$\begin{aligned} L(\sigma, j, l+1) &= \max(L(\sigma, j, l), L(\sigma, l+1)) \\ g_{j,l+1} &= \max(g_{j,l}, g_{l+1}) \\ L_{-\infty}(\sigma, j, l+1) &= \max(L_{-\infty}(\sigma, j, l), g_{j,l+1} + L(\sigma, l+1)) \end{aligned}$$

Therefore, when $j < l-1$, we can compute the new tardiness $L(\sigma', j+1, l-1) = L_\Delta(\sigma, j+1, l-1)$ of the operations in the interval $[j+1, l-1]$ in constant (amortized) time since the query of Proposition 2 can be checked in $O(1)$.

The new tardiness $L(\sigma', l)$ at position l is computed in a similar way as for $L(\sigma', j)$ since we know the new start time of $\sigma'(l-1)$ from previous steps.

Finally, in order to compute the new maximum tardiness $L(\sigma', l+1, n)$ over subsequent operations, we precompute $L(\sigma, j, n)$, $g_{j,n}$ and $L_{-\infty}(\sigma, j, n)$ for every position $1 \leq j \leq n$ once after each move in $O(n)$. Then $L(\sigma', l+1, n)$ can be obtained in $O(1)$ for every potential move from Proposition 2.

Therefore, we can check the validity and forecast the marginal cost of a swap in constant amortized time and perform the swap in linear time. The time complexity for an iteration is thus in $O(nm)$ since, for a given component i, the sum of the sizes of the valid ranges for all pickups and deliveries of this component is in $O(n)$. Indeed, let $a_i^1, \ldots, a_i^{4n_i}$ be the operations component i ordered as in σ. Then $su(a_i^k) = \rho(a_i^{k+1})$ and $\sum_{k=1}^{4n_i} su(a_i^k) - \rho(a_i^k) = \rho(a_i^{4n_i}) - \rho(a_i^1) \in \Theta(n))$.

Toggle moves. As observed in Section 3, there are only two dominant orderings for the four operations of the k-th production cycle of component i. The second type of moves consists in changing from one to the other of these two orderings, by swapping the values of $\sigma(pf_i^k)$ and $\sigma(pe_i^k)$ and the values of $\sigma(df_i^k)$ and $\sigma(de_i^k)$. This change leaves the size of the train constant, hence all these moves are valid.

Let $j_1 = pf_i^k, j_2 = df_i^k, j_3 = pe_i^k, j_4 = de_i^k$ be the positions of the four operations of component i and cycle k in the current solution, and suppose, wlog, that $j_1 < j_2 < j_3 < j_4$. Let σ' denote the sequence obtained by applying a toggle move on component i and cycle k in σ. In order to forecast the marginal cost of the move, we need to compute the new tardiness at the positions of the four operations involved $L(\sigma', j_1)$, $L(\sigma', j_2)$, $L(\sigma', j_3)$ and $L(\sigma', j_4)$. Moreover, we need the new maximum tardiness on four time-shifted intervals:
$L(\sigma', j_1+1, j_2-1) = L_{\Delta_1}(\sigma, j_1+1, j_2-1)$, $L(\sigma', j_2+1, j_3-1) = L_{\Delta_2}(\sigma, j_2+1, j_3-1)$, $L(\sigma', j_3+1, j_4-1) = L_{\Delta_3}(\sigma, j_3+1, j_4-1)$ and $L(\sigma', j_4+1, n) = L_{\Delta_4}(\sigma, j_4+1, n)$.

Computing the marginal costs can be done via the same formulas as for swaps: we can first compute the new end time for the operation at position j_1, then from it compute the value of Δ_1 that we can use to compute $L_{\Delta_1}(\sigma, j_1+1, j_2-1)$ in $O(j_2-j_1-1)$ time, and so forth. The difference, however, is that there is fewer possible moves ($n/4$), although the computation of the marginal cost cannot be amortized. The resulting time complexity is the same: $O(nm)$.

6 Experimental Results

We generated synthetic instances[9] in order to better assess the approaches. Due to the time windows constraints, it is difficult to generate certifiably feasible instances. Their feasibility has been checked on the shortest possible horizon, i.e., the duration of the longest production cycle of any component, which is about 10 000 time units depending on the instances. There are four categories of instances parameterized by the number of components (15 in category A, 20 in B, 25 in C and 30 in D). In the real dataset, several components have similar production cycles. We replicates this feature: synthetic instances have from 2 or 3 distinct production cycles in category A, to up to 7 in category D. The latter are therefore harder because there are more asynchronous productions cycles. We generated 10 random instances for each category and consider the same three horizons (shift, day and week) for each, as industrial instances.

Experiments for this section were run on a cluster made up of Xeon E5-2695 v3 @ 2.30GHz and Xeon E5-2695 v4 @ 2.10GHz. For the basic CP models, we add randomization and a restart strategy following a Luby sequence, and we ran each of the 120 instances 10 times. We could not carry on experiments on synthetic data with `LocalSolver` because we were not granted a license. However, from the few tests we could do, we expect `LocalSolver` to behave similarly as on industrial benchmarks.

Table 2: Comparison of the methods on generated instances

Cl	H	Scheduling			TSP			CP-softmax			LDS			Multi-start LS		
		#S	CPU	#fail	#S	CPU	#fail	#S	CPU	#fail	#S	CPU	#fail	#S	CPU	L_{max}
A	shift	7.1	418	300K	4.0	56	366	**9.0**	2	15	**9.0**	2	9	**9.0**	0	1m
	day	4.0	29	213	3.6	802	1267	**9.0**	15	815	8.0	21	71	**9.0**	176	19m
	week	3.1	866	1976	0.0	mem. out		**8.0**	118	27	5.0	68	0.0	7.0	155	1h21
B	shift	2.1	389	150K	0.9	844	11K	**6.0**	4	77	**6.0**	15	85	**6.0**	2	11m
	day	1.0	201	15K	0.0	–		**5.2**	341	20K	4.0	12	19	4.6	346	1h
	week	0.0	–		0.0	mem. out		**3.5**	423	715	1.0	99	0.0	1.0	0	4h59
C	shift	0.0	–		0.0	–		**4.0**	103	5366	**4.0**	715	4090	**4.0**	255	32m
	day	0.0	–		0.0	–		**1.0**	12	7	**1.0**	18	27	**1.0**	1	1h45
	week	0.0	–		0.0	mem. out		**1.0**	807	366	0.0	–		0.0	–	11h51
D	shift	0.0	–		0.0	–		**1.9**	697	24K	1.0	442	1058	1.6	1165	31m
	day	0.0	–		0.0	–		0.0	–		0.0	–		0.0	–	2h19
	week	0.0	–		0.0	mem. out		0.0	–		0.0	–		0.0	–	17h52

For each of the methods using the learnt heuristics, we normalize $\boldsymbol{\theta}$ so that $\sum_{i=1}^{4} \boldsymbol{\theta}_i = 1$ and set β to 1/150. We learn the policy by batches of size 240 formed by 6 runs on each of the 40 day-long instances, during 2000 iterations.

[9] Avalaible at `https://gitlab.laas.fr/vantuori/trolley-pb`.

The learning rate depends on the size of the instances: $\alpha = 2^{-12}/\overline{n}$ where $\overline{n}$ is the average number of task in the batch. The rationale is that the magnitude of the gradient depends on the tardiness $L(\sigma)$ which tends to grow with the number of operations. Therefore, we use the learning rate α to offset this growth, which is key to have a stable convergence. For the two methods using the stochastic policy, we made the first run deterministic i.e. the policy becomes stochastic only after the first restart for the CP based one, and after the first iteration for the multi-start local search.

The results are presented in Table 2. We report the number of solved instances (among 10 instances for every time horizon) averaged over 10 randomized runs for each CP model in the column "#S". The synthetic dataset is more constrained than the industrial dataset and the two basic CP models fail to solve most of the instances ("–" in the table indicates a time out). However, the relative performance remains unchanged w.r.t. Table 1: the scheduling-based models shows better performance in terms of number of solved instances and CPU time while scaling better in memory. All three of the RL-based methods significantly outperform previous approaches. The results in Table 2 indicate that the rapid restarts approach dominates the others. However, it may not be as clear-cut as that: for other settings of the hyperparameters (α, β and γ) the relative efficiencies fluctuate and other methods can dominate. Moreover, one advantage of the multi-start local search method is that since due dates are relaxed, imperfect solutions can be produced, even for infeasible instances. We report the average maximum tardiness in column L_{max}.

This global $\boldsymbol{\theta}$ also works well with the industrial dataset. All instances are easily solved by all three methods, except "R" for the week horizon, which is only solved by the rapid restart approach. We learnt a dedicated policy for the industrial dataset with the same settings. It turns out that every instance was solved by the deterministic policy using the new value for $\boldsymbol{\theta}$, except the instance "L" for the week horizon. However, it is easily solved by all three methods.

7 Conclusion

In this paper we have applied reinforcement learning to design simple yet efficient stochastic decision policies for an industrial problem: planning the production process at Renault. Moreover, we have shown how to leverage these heuristic policies within constraint programming and within local search.

The resulting approaches significantly improve over the current local search method used at Renault. However, many instances on synthetic data remain unsolved. We plan on using richer machine learning models, such as neural networks, to represent states. Moreover, we would like to embed this heuristic in a Monte-Carlo Tree Search as it would fit well with our current approach since it relies on many rollouts. Finally, we would like to tackle the more general problem of assigning components to operators and then planning individual routes.

References

1. J. Christopher Beck, Patrick Prosser, and Evgeny Selensky. Vehicle Routing and Job Shop Scheduling: What's the Difference? In *Proceedings of the 13th International Conference on Automated Planning and Scheduling*, ICAPS, pages 267–276, 2003.
2. Irwan Bello, Hieu Pham, Quoc V. Le, Mohammad Norouzi, and Samy Bengio. Neural Combinatorial Optimization with Reinforcement Learning. In *5th International Conference on Learning Representations, Workshop Track Proceedings*, ICLR, 2017.
3. Frederic Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence*, ECAI, pages 146–150, 2004.
4. Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning Combinatorial Optimization Algorithms over Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS, page 6351–6361, 2017.
5. Michel Deudon, Pierre Cournut, Alexandre Lacoste, Yossiri Adulyasak, and Louis-Martin Rousseau. Learning Heuristics for the TSP by Policy Gradient. In *Proceedings of the 15th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, CPAIOR, pages 170–181, 2018.
6. Sylvain Ducomman, Hadrien Cambazard, and Bernard Penz. Alternative Filtering for the Weighted Circuit Constraint: Comparing Lower Bounds for the TSP and Solving TSPTW. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*, AAAI, pages 3390–3396, 2016.
7. Martin Josef Geiger, Lucas Kletzander, and Nysret Musliu. Solving the Torpedo Scheduling Problem. *Journal of Artificial Intelligence Research*, 66:1–32, 2019.
8. William D. Harvey and Matthew L. Ginsberg. Limited Discrepancy Search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, IJCAI, page 607–613, 1995.
9. Philippe Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143(2):151–188, 2003.
10. Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal Speedup of Las Vegas Algorithms. *Information Processing Letters*, 47:173–180, 1993.
11. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.
12. Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.
13. Pierre Schaus. The Torpedo Scheduling Problem. `http://cp2016.a4cp.org/program/acp-challenge/`, 2016.
14. David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

15. David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.
16. Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3):229–256, 1992.

Un solveur générique par intervalles pour le CSP différentio-algébrique

S. Rohou[1] **A. Bedouhene**[2] **G. Chabert**[3] **A. Goldzstzejn**[4]
L. Jaulin[1] **B. Neveu**[2] **V. Reyes**[5] **G. Trombettoni**[5]

[1] LabSTICC, ENSTA Bretagne, CNRS, France
[2] LIGM, Ecole des Ponts Paristech, France
[3] IRT Jules Verne, LS2N, France, [4] L2SN, CNRS, France
[5] LIRMM, Université de Montpellier, CNRS, France
{Simon.Rohou, Luc Jaulin}@ensta-bretagne.fr Bertrand.Neveu.@enpc.fr
Gilles.Chabert@imt-atlantique.fr Alexandre.Goldsztejn@gmail.com
Gilles.Trombettoni@lirmm.fr

Résumé

Cet article, publié à la conférence CP en 2020 [4], propose une approche pour traiter le CSP différentio-algébrique (DACSP), dont les systèmes sont composés de variables réelles et fonctionnelles liées par des contraintes différentielles et/ou algébriques. Nous présentons un solveur qui approxime de manière rigoureuse les solutions d'un DACSP. Il utilise les bibliothèques à intervalles IBEX et CODAC. Des premières expérimentations montrent que ce solveur peut traiter des problèmes de Cauchy sur intervalles (IIVP), des problèmes aux limites (BVP) et des équations intégro-différentielles.

1 Programmation par contraintes sur intervalles pour le problème DACSP

De nombreuses applications comportent à la fois des variables réelles et une composante dynamique (des fonctions d'une variable, le "temps" par exemple) : des systèmes biologiques, la dynamique de certains mécanismes, de nombreux problèmes d'automatique et de robotique, etc. Les nombreux travaux existants proposent des solutions dédiées à différentes variantes de ce problème.

L'aspect déclaratif de la programmation par contraintes permet de poser tous ces problèmes comme des instances du DACSP qui définit des variables réelles (statiques), des variables fonctionnelles (ou trajectoires), des contraintes portant sur ces variables, des domaines représentés par des intervalles ou des *tubes*.

Les solutions et les domaines sont représentés par des tubes qui contiennent un ensemble de trajectoires possibles (cf. figure 1). Ce modèle est mis en œuvre dans

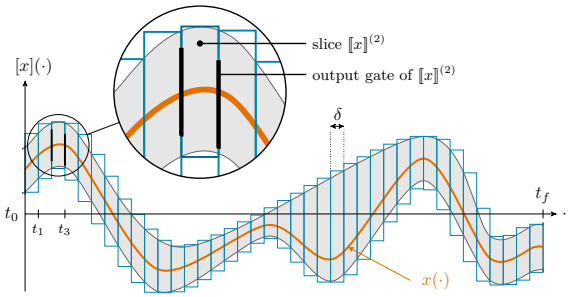


FIGURE 1 – Un tube $[x](\cdot)$, intervalle de deux fonctions $[\underline{x}(\cdot), \overline{x}(\cdot)]$, contenant une trajectoire $x(\cdot)$, en orange. Le tube est numériquement représenté par un ensemble de tranches bleues de largeur δ.

la bibliothèque CODAC [5] qui fournit des structures de données pour représenter des tubes et un catalogue de contracteurs, similaire au catalogue de *propagateurs* des solveurs PPC. L'approche a récemment été appliquée à des données réelles pour la localisation de robot sous-marin autonome [3]. L'évolution du robot est décrite par une équation différentielle ordinaire (EDO) et des mesures bornées, mais la condition initiale (position) du système n'est pas connue. Pendant la mission, son sonar détecte des rochers sur le fond de l'océan, rochers

non distinguables mais appartenant à une carte. La carte des coordonnées des rochers est aussi modélisée comme une contrainte. La propagation de contraintes peut croiser les informations venant de l'évolution du robot et des observations de rochers. Les rochers sont finalement associés aux coordonnées sur la carte et la trajectoire du robot est assez précisément approximée. Cette application souligne comment une approche PPC peut traiter un problème complexe mêlant contraintes discrètes, continues et différentielles.

Pour traiter ce type d'applications, la boucle de propagation suffit, car le problème contient de nombreuses contraintes "redondantes". La contribution de l'article présentée ci-dessous est un solveur générique basé sur Codac et déployant un arbre de recherche pour traiter les problèmes "bien contraints".

2 Solveur générique pour le DACSP

Le solveur effectue d'abord une étape de découpage temporel (*slicing*), puis déploie si nécessaire un arbre de recherche qui subdivise le tube (vectoriel). La résolution s'arrête quand une précision utilisateur est atteinte sur la largeur du tube. La phase de découpage alterne une étape de contraction/filtrage et la bissection temporelle d'une partie des tranches. L'arbre de recherche en profondeur alterne une étape de contraction et le découpage d'un tube en deux sous-tubes : en un instant donné, un intervalle (d'une des dimensions) est coupé en deux, créant deux sous-tubes étudiés dans deux branches de l'arbre.

La phase de contraction assez sophistiquée applique :

1. d'abord les contracteurs "classiques" (statiques),
2. puis contracte le tube de la variable (vectorielle) dynamique en entrelaçant (a) un contracteur développé dans Codac et spécialisé dans la contraction du sous-réseau de contraintes correspondant à une tranche et (b) un contracteur pilotant un outil d'intégration garantie spécialisé dans la résolution d'EDO avec une condition initiale intervalle (VNODE ou CAPD) ;
3. finit en appliquant Dyn3B, une variante de l'algorithme de consistance forte 3B [2] qui "force" des sous-intervalles aux bornes d'un intervalle à un instant donné pour espérer éliminer le sous-tube correspondant par propagation (intégration) dans le tube.

3 Expérimentations

Nous avons validé notre solveur sur dix systèmes DACSP de différentes catégories. Détaillons deux de ces systèmes. Le premier est un problème aux limites (BVP ; sans condition initiale fournie) dont la contrainte dynamique est une équation intégro-différentielle :

$$\begin{cases} \dot{x}(t) = 1 - 2x(t) - 5\int_0^t x(\tau)d\tau;\ t \in [0,1] \\ x(0)^2 + x(1)^2 = 1 \end{cases} \quad (1)$$

Le solveur est capable d'approximer les deux trajectoires solution en 8 s et 66 points de choix.

Le deuxième système bien connu dans la communauté PPC a été étudié par Cruz et Barahona [1] et comporte une contrainte d'observation incertaine.

$$\begin{cases} \dot{x}_1(\cdot) = -0.7x_1(\cdot) \\ \dot{x}_2(\cdot) = 0.7x_1(\cdot) - (\ln(2)/5)x_2(\cdot) \\ x_1(0) = 1.25 \\ x_2 \in [1.1, 1.3] \text{ during } [1,3] \\ [t_0, t_f] = [0,6] \end{cases} \quad (2)$$

Le solveur calcule la trajectoire solution "épaisse" en 7 s et un point de choix.

4 Discussion

Ce résumé a présenté un solveur construit au dessus de Codac permettant de traiter des systèmes DACSP mélangeant des variables réelles et des trajectoires manipulées comme des variables du système. Le solveur entrelace des opérations de contraction/filtrage, de découpage temporel et de découpage combinatoire du domaine de la trajectoire.

Depuis ces travaux en 2020, le pilotage du solveur d'intégration garantie VNODE a été amélioré, produisant un gain en temps d'un ordre de grandeur sur les systèmes IIVP et les BVP testés. Codac évolue pour prendre en compte d'autres variables "sophistiquées", comme les ensembles de $\mathbb{R}^n$.

Ce travail a été financé par le projet ANR CONTREDO [ANR-16-CE33-0024].

Références

[1] J. Cruz et P. Barahona : Constraint Satisfaction Differential Problems. *In Proc. CP*, pages 259–273, 2003.

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

[3] S. Rohou, B. Desrochers et L. Jaulin : Set-membership State Estimation by Solving Data Association. *In Proc. IEEE ICRA*, 2020.

[4] S. Rohou *et al.* : Towards a Generic Interval Solver for Differential-Algebraic CSP. *In Proc. CP, Springer, LNCS 12333*, pages 864–879, 2020.

[5] Simon Rohou, Benoit Desrochers *et al.* : The Codac library – Constraint-programming for robotics, 2021. http ://codac.io.

Towards a Generic Interval Solver for Differential-Algebraic CSP

Simon Rohou[1], Abderahmane Bedouhene[2], Gilles Chabert[3], Alexandre Goldsztejn[4], Luc Jaulin[1], Bertrand Neveu[2], Victor Reyes[5], and Gilles Trombettoni[5]

[1] Lab-STICC, ENSTA Bretagne, CNRS, France
[2] LIGM, Ecole des Ponts, Univ. Gustave Eiffel, CNRS, Marne-la-Vallée, France
[3] IRT Jules Verne, LS2N, France
[4] LS2N, CNRS, France
[5] LIRMM, University of Montpellier, CNRS, France

Abstract. In this paper, we propose an interval constraint programming approach that can handle the *differential-algebraic CSP* (DACSP), where an instance is composed of real and functional variables (also called dynamic variables or trajectories) together, and differential and/or "static" numerical constraints among those variables. Differential-Algebraic CSP systems can model numerous real-life problems occurring in physics, biology or robotics. We introduce a solver, built upon the `Tubex` and Ibex interval libraries, that can rigorously approximate the set of solutions of a DACSP system. The solver achieves temporal slicing and a tree search by splitting trajectories domains. Our approach provides a significant step towards a generic interval CP solver for DACSP that has the potential to handle a large variety of constraints. First experiments highlight that this solver can tackle interval Initial Value Problems (IVP), Boundary Value Problems (BVP) and integro-differential equations.

1 Introduction

Differential-Algebraic CSP (DACSP) systems include real variables, functional variables (also called trajectories or dynamical variables) describing the dynamics of the system, and differential and/or "static" numerical constraints among those variables. Because they are at the core of so many applications, such as biological systems, mechanism dynamics, astronomy, robotics, control, a lot of work has been dedicated to solving specific subclasses of the DACSP. Most of the approaches follow a probabilistic approach and are limited to linear constraints with Gaussian errors [17, 35].

There are a number of advantages to using interval methods for handling dynamical and/or static systems. They can manage nonlinear constraints and approximate the solutions rigorously, whatever the uncertainties on the parameters (*e.g.*, uncertainties related to measurements or to inaccurate physical models). Bounded intervals are used to characterize these uncertainties, as well as the errors due to operations over floating-point numbers.

With Jaulin *et al.* works [15, 16], a significant step has been made towards a declarative constraint programming approach for dynamical systems. Contrary to dominant constraint approaches dealing with differential equations such as [8, 30, 37], the trajectories are viewed as variables of the CSP. They consider trajectories as variables, differential equations as constraints and *tubes* as domains. The solution and the domain are given by a tube representing a set of possible trajectories (see Fig. 1). This increases the level of abstraction and simplifies the formalization of the problem. For estimating a trajectory, a set of *contractors*, similar to *propagators* in CP, are applied to filter (contract) the bounds of the domain/tube. A benefit of the contractor programming approach [3] is the variety of dynamical systems that can be handled. More recently, the `Tubex` library [32] has provided data structures for representing tubes and a catalogue of contractors, similarly to the catalogue of propagators available in CP solvers [29]. The user defines a sequence of contractors for modeling his problem. The set of contractors is applied iteratively until a quasi fixpoint in terms of contraction is reached. For instance, the resulting framework has been recently applied to actual data for autonomous robot localization. [31] describes the problem of localizing an underwater robot. Its evolution is depicted by an Ordinary Differential Equation (ODE) and bounded measurements, but the initial condition (position) of the system is unknown. During the mission, its sonar detects indistinguishable rocks on the seabed, that all look alike but are known to belong to a discrete point map embedded beforehand. The map of rocks positions is also modeled as a constraint. The constraint propagation approach is able to merge all data coming from the robot evolution and rocks observations. The identity of the rocks is finally associated to items in the map, and the trajectory of the robot is accurately estimated. This application highlights how a complex problem involving discrete, continuous and differential constraints can be solved easily following a CP/contractor approach.

Contractor programming is relevant when the problem is defined by heterogeneous constraints, provided they are redundant and numerous enough to enable the contraction phase alone to solve the problem.

Contributions. In this paper, we introduce a generic solver using the `Tubex` library that can handle a DACSP instance made of differential constraints and/or static continuous constraints. The numerical constraints relate real variables that either represent states of the trajectories at given instants or are independent from the dynamics (*e.g.*, the rocks positions in the previous example). Compared to the `Tubex` approach alone, the solver is endowed with an operator that can perform a choice point by splitting (bisecting) a tube in two at a chosen time. Thus, a tree search can accurately estimate distinct trajectories (for problems with several solutions) and can better handle several hard problems. To our knowledge, no other tool for solving dynamical systems performs a tree search.

Our solver can manage most of the contractors available in the literature. On the one hand, several contractors coming from CP, *e.g.*, 3B [22] and CID [36], are included naturally to improve the pruning of domains of functional variables, *i.e.*, tubes. On the other hand, our generic framework enables to wrap existing

solvers dedicated to specific types of (differential) constraints into contractors. These contractors efficiently reduce domains by taking into account space and time dependencies. In particular, we show in the experiments the benefits of using the contractor `CtcVnode` built upon the state-of-the-art VNODE guaranteed integration IVP solver [27, 28].

This generic CP framework allows separating the problem description, which includes here all combinations of differential and algebraic problems, from its resolution. We also wanted to solve difficult standard problems coming from numerical analysis, *e.g.*, BVP for integro-differential equations, that are out of the scope of previous CP/ODE frameworks and remain today difficult to solve in the numerical analysis framework.

Related Work. VNODE [28], CAPD [18], COSY [30] and `DynIbex` [4] are state of the art interval analysis solvers dedicated to IVPs.[6] They are fully relevant for determining the guaranteed solution of a system at a final time, that has crucial applications such as the position determination of celestial bodies in astronomy [39] or to characterize chaotic attractors [37]. They use different algorithms to reliably simulate the initial information over time. In particular, the VNODE tool used in our solver combines a high-order interval Taylor form to integrate the state from an instant to a next one, and a step limiting the wrapping effect implied by interval calculation: it encloses the solution at the discrete times by an envelope sharper than a box, such as rotated boxes, zonotopes or polygons [23].

A BVP is generally defined by an ODE, but the trajectory is not entirely determined at the initial or final times, which prevents a solving algorithm from propagating (integrating) the information from the known state to the rest of the trajectory. Instead, static constraints are defined on specific states (at specified instants). To deal with BVPs, the shooting method [12] is a sampling-and-optimization algorithm that runs several integration processes from the initial state while trying to minimize the distance to a solution satisfying the ODE. This makes the approach incomplete and unable to determine several solutions, if any. Instead, our rigorous and deterministic solver can explore the whole search space and isolate distinct solutions.

The constraint programming community has contributed to dynamical systems. Janssen *et al.* propose a strategy [8] dedicated to interval IVPs that has similarities with VNODE or CAPD. The integration step between two consecutive states (instants) used a box consistency algorithm [1] to better contract the output state. A first attempt to create a constraint language extended to ODEs was made a few years later [6, 7]. The language considers the entire ODE as a whole, but the unknown function has a special status that prevents direct manipulations: any constraint involving the unknown trajectory is managed by a specific operator. The fact that the function is bounded in an interval requires the introduction of an ad-hoc constraint called minimum/maximum restriction.

[6] An Initial Value Problem is composed of an ODE and an initial condition. Numerical integration propagates the initial value through the whole trajectory by integrating the evolution function of the ODE.

The link with real variables is also achieved via an ad-hoc constraint called value restriction. This language does not have the level of genericity targeted by our solver, where the concept of trajectories appears at the same level as other variables and ODEs are syntactic constructions among others. In our solver, we will rather use the $\mathcal{C}_{\text{eval}}$ contractor [33] to handle value restriction constraints.

Although [11] improves [6] in both modeling possibilities and solving efficiency, it is still restricted to ODE constraints relating solutions values at specified times. Finally, our model can accept various differential constraints (like an integro-differential equation) and static constraints such as distance relations between states at different instants, as long as the corresponding contractors exist.

Outline. Section 2 introduces the notations used in the paper and the background useful to understand the following sections. Sections 3 and 4 detail our DACSP solver and its different procedures and parameters: contractors, choice point heuristics, *etc.* Section 5 reports first experimental results obtained on interval IVPs, but also integro-differential equations and BVPs.

2 Background and Notations

We first provide some background about intervals, inclusion functions and contraction. We then present several differential constraints and interval techniques adapted to dynamical systems.

2.1 Intervals

Contrary to numerical analysis methods that work with single values, interval methods can manage sets of values enclosed in intervals. Interval methods are known to be particularly useful for handling nonlinear constraint systems.

Definition 1 (Interval, box, box width, box hull)
An interval $[x_i] = [\underline{x_i}, \overline{x_i}]$ *defines the set of reals* x_i *such that* $\underline{x_i} \leq x_i \leq \overline{x_i}$. $\mathbb{IR}$ *denotes the set of all intervals. A box* $[\boldsymbol{x}]$ *denotes a Cartesian product of intervals* $[\boldsymbol{x}] = [x_1] \times ... \times [x_n]$. *The size, width or diameter of a box* $[\boldsymbol{x}]$ *is given by* $w([\boldsymbol{x}]) \equiv \max_i(w([x_i]))$ *where* $w([x_i]) \equiv \overline{x_i} - \underline{x_i}$. *The hull of boxes approximates the union operator. It returns the smallest box enclosing all the boxes hulled.*

Interval arithmetic [26] has been defined to extend to $\mathbb{IR}$ the usual mathematical operators over $\mathbb{R}$ For instance, the interval sum is defined by $[x_1] + [x_2] = [\underline{x_1} + \underline{x_2}, \overline{x_1} + \overline{x_2}]$. When a function $\boldsymbol{f}$ is a composition of elementary functions, an *inclusion function* $[\boldsymbol{f}]$ of $\boldsymbol{f}$ must be defined to ensure a conservative image computation. There are several inclusion functions. The *natural* inclusion function of a real function f corresponds to the mapping of f to intervals using interval arithmetic. For instance, the natural inclusion function $[f]_N$ of $f(x) = x(x+1)$ in the domain $[x] = [0,1]$ computes $[f]_N([0,1]) = [0,1] \cdot [1,2] = [0,2]$. Another inclusion function is based on an interval Taylor form [14].

Interval arithmetics can be used for solving the *numerical CSP* (NCSP), *i.e.* finding solutions to an NCSP instance $P = (\boldsymbol{x}, [\boldsymbol{x}], \boldsymbol{c})$, where $\boldsymbol{x}$ is an n-set of variables taking their real values in the domain $[\boldsymbol{x}]$ and $\boldsymbol{c}$ is an m-set of numerical constraints using operators like $+$, $-$, $\times$, a^b, exp, log, sin, *etc.* NCSP solvers, like GlobSol [19], Gloptlab [9], RealPaver [13] or IBEX [2] to name a few, follow a Branch and Contract method to solve an NCSP. The branching operation subdivides the search space by recursively bisecting variable intervals into two subintervals and exploring both sub-boxes independently. The combinatorial nature of this tree search is not always observed thanks to the *contraction* (filtering) operations applied at each node of the search tree. Informally, a contraction applied to an NCSP instance can reduce the domain without losing any solution. A contractor used in this paper is the well-known `HC4-revise` [1, 25], also called *forward-backward*. This contractor handles a single numerical constraint and obtains a (generally non optimal [5]) contracted box including all the solutions of that constraint. To contract a box w.r.t. an NCSP instance, the HC4 algorithm performs a (generalized) AC3-like propagation loop applying iteratively the HC4-Revise procedure on each constraint individually until a quasi fixpoint is obtained in terms of contraction.

3B-consistency [22] and CID-consistency [36] are two other stronger consistencies, enforced on an NCSP, that are exploited by our solver. The corresponding 3B and CID algorithms should call their Var3B or VarCID procedure on all the NCSP variables for enforcing the 3B or CID consistency. In practice however the algorithms implemented apply these procedures on a subset of the variables to get a better tradeoff between contraction and performance. VarCID splits a variable interval in k subintervals, and runs a contractor, such as HC4, on the corresponding sub-boxes. The k sub-boxes contracted are finally hulled. Var3B is somehow a dual operator that tries to remove subintervals at the bounds of a variable interval. If a contraction, like HC4, applied to a sub-box, where the interval is replaced by a subinterval at a bound, leads to an empty domain, then it proves that the subinterval contains no solution and can be removed safely from the variable domain.

2.2 Trajectories and Tubes

A trajectory, denoted $\boldsymbol{x}(\cdot) = (x_1(\cdot), .., x_n(\cdot))$, is a function from $[t_0, t_f] \subset \mathbb{R}$ to $\mathbb{R}^n$. The input (argument) of $\boldsymbol{x}(\cdot)$ is named *time* in this article (and denoted $\cdot$ or t) while the output (image) is called *state*.

A tube $[\boldsymbol{x}](\cdot)$ is the interval counterpart of a trajectory and is defined as an envelope over the same temporal domain $[t_0, t_f]$. The concept appeared in [10,20] in the context of ellipsoidal estimations. In our solver, it is used as a domain on which we apply operations of contractions and bisections. We employ them as intervals of trajectories, which is consistent with the aforementioned tools.

Hence, we will use the definition given in [21] where a tube $[\boldsymbol{x}](\cdot) : [t_0, t_f] \to \mathbb{IR}^n$ is an interval of two trajectories $[\underline{\boldsymbol{x}}(\cdot), \overline{\boldsymbol{x}}(\cdot)]$ such that $\forall t \in [t_0, t_f],\ \underline{\boldsymbol{x}}(t) \leqslant \overline{\boldsymbol{x}}(t)$. We also consider empty tubes that depict an absence of solutions. A

trajectory $\boldsymbol{x}(\cdot)$ belongs to the tube $[\boldsymbol{x}](\cdot)$ if $\forall t \in [t_0, t_f]$, $\boldsymbol{x}(t) \in [\boldsymbol{x}](t)$. Fig. 1 illustrates a one-dimensional tube enclosing a trajectory $x(\cdot)$.

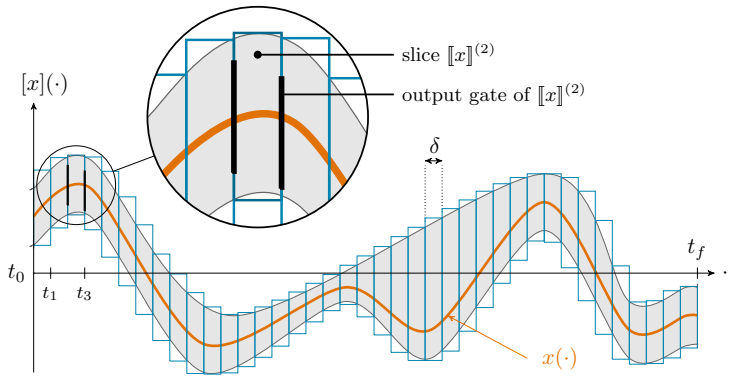


Fig. 1. A one-dimensional tube $[x](\cdot)$, interval of two functions $[\underline{x}(\cdot), \overline{x}(\cdot)]$, enclosing a random trajectory $x(\cdot)$ depicted in orange. The tube is numerically represented by a set of δ-width slices illustrated by blue boxes.

Our choice is to represent numerically a tube as a set of boxes corresponding to temporal slices. More precisely, an n-dimensional tube $[\boldsymbol{x}](\cdot)$ with a sampling time $\delta > 0$ is implemented as a box-valued function which is constant for all t inside intervals $[k\delta, k\delta + \delta]$, $k \in \mathbb{N}$. The box $[k\delta, k\delta + \delta] \times [\boldsymbol{x}](t_k)$, with $t_k \in [k\delta, k\delta + \delta]$, is called the k^{th} *slice* of the tube $[\boldsymbol{x}](\cdot)$ and is denoted by $[\![\boldsymbol{x}]\!]^{(k)}$. This implementation takes rigorously into account floating-point precision when building a tube: computations involving $[\boldsymbol{x}](\cdot)$ will be based on its slices, thus giving a reliable outer approximation of the solution set. The slices may be of same width as depicted in Fig. 1, but the tube can also be implemented with a customized temporal *slicing*. Finally, we endow the definition of a slice $[\![\boldsymbol{x}]\!]^{(k)}$ with the *slice (box) envelope* (blue painted in Fig. 1) and two input/output *gates* $[\boldsymbol{x}](t_k)$ and $[\boldsymbol{x}](t_{k+1})$ (black painted) that are intervals of $\mathbb{IR}^n$ through which trajectories are entering/leaving the slice.

Once a tube is defined, it can be handled in the same way as an interval. We can for instance use arithmetic operations as well as function evaluations. If f is an elementary function such as sin, cos or exp, we define $f([x](\cdot))$ as the smallest tube containing all feasible values: $f([x](\cdot)) = \left[\{f(x(\cdot)) \mid x(\cdot) \in [x](\cdot)\}\right]$.

2.3 Dynamical Systems and Differential-Algebraic CSP

A differential constraint relates one or several trajectories and/or real variables. Numerous types of differential constraints can be considered in our approach, including:

1. $\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{f}\big(\boldsymbol{x}(\cdot)\big)$ (ODE)
2. $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(\boldsymbol{x}(t)\big) + \int_{t_0}^{t} \boldsymbol{x}(\tau)d\tau$ (integro-differential equation)
3. $\boldsymbol{x}(t_k) = \boldsymbol{y}$, $\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{v}(\cdot)$ (evaluation constraint)
4. $\forall t \in [t]$, $\boldsymbol{x}(t) \notin [\boldsymbol{y}]$
5. $\boldsymbol{x}(t) = \boldsymbol{y}(t + \delta)$ (delay constraint)

The first one is the most widespread differential constraint (see Def. 2). The second constraint is a little more complicated in that the state at a given time depends on the sum (integral) of the previous states. The third evaluation constraint $\boldsymbol{y} = \boldsymbol{x}(t_k)$ states that the trajectory goes through an uncertain real value in $[\boldsymbol{y}]$ at an uncertain time in $[t_k]$. The fourth constraint is the complementary, although more complicated, constraint of the evaluation. The fifth one imposes a delay constraint of unknown real value δ between two trajectories and is particularly useful for clock calibration purposes in autonomous systems [38].

The idea behind our approach is to decompose a differential-algebraic system into a set of such primitive constraints associated to contractors (similar to propagators in CSP solvers [29]) that belong, or can be added to, the `Tubex` library. We formally define below the following differential constraints used in the experimental part.

Definition 2 (ODE and integro-differential equation)
Consider $\boldsymbol{x}(\cdot) : [t_0, t_f] \to \mathbb{R}^n$*, its derivative* $\dot{\boldsymbol{x}}(\cdot) : [t_0, t_f] \to \mathbb{R}^n$*, and an* evolution function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$*, possibly non-linear. An ODE*[7] *is defined by:*

$$\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{f}\big(\boldsymbol{x}(\cdot)\big)$$

An integro-differential equation is defined by:

$$\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(\boldsymbol{x}(t)\big) + \int_{t_0}^{t} \boldsymbol{x}(\tau)d\tau.$$

The ODEs considered are *explicit*, *i.e.* the evolution function $\boldsymbol{f}$ computes $\dot{\boldsymbol{x}}$ directly. These differential constraints can define dynamical systems.

Definition 3 (IVP, interval IVP, BVP)
The initial value problem (IVP) is defined by an ODE $\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{f}\big(\boldsymbol{x}(\cdot)\big)$ *and an initial condition* $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$*, where* $\boldsymbol{x}_0$ *is a constant in* $\mathbb{R}^n$*. In an interval IVP, the initial condition is bounded by an interval, i.e.* $\boldsymbol{x}(t_0) \in [\boldsymbol{x}_0]$*. A boundary value problem (BVP) is defined by an ODE and a numerical constraint* $\boldsymbol{c}\big(\boldsymbol{x}(t_1)..\boldsymbol{x}(t_n)\big) = \boldsymbol{0}$*, where* $\boldsymbol{c} : \mathbb{R}^n \to \mathbb{R}^n$ *and* $\forall i \in \{1..n\},\ t_i \in [t_0, t_f]$.

A BVP generalizes an IVP in that no initial condition fully determines the trajectory at a unique instant. Instead, n algebraic constraints relate several states

[7] Note that a high-order problem can be transformed automatically into a first-order ODE shown in Def. 2 by introducing auxiliary variables. Also note that non autonomous ODEs of the form $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(\boldsymbol{x}(t), t\big)$ can also be transformed into autonomous ODEs $\dot{\boldsymbol{x}}(t) = \boldsymbol{f}\big(\boldsymbol{x}(t)\big)$ whose derivative depends only on the state.

at times $t_1..t_n$ and enable the trajectory determination. Note that a condition known at any instant is equivalent to knowing the state at time t_0. Indeed, numerical integration can propagate the information forward or backward in time indifferently. We are now in position to define the DACSP.

Definition 4 (Differential-algebraic CSP – DACSP)
A DACSP network is defined by a quintuplet $(\boldsymbol{x}(\cdot), [\boldsymbol{x}](\cdot), \boldsymbol{y}, [\boldsymbol{y}], \boldsymbol{c})$*, where* $\boldsymbol{x}(\cdot)$ *is a trajectory variable of domain* $[\boldsymbol{x}](\cdot)$ *(a tube* $\mathbb{R} \to \mathbb{IR}^{n_1}$*, as defined in the previous section),* $\boldsymbol{y} \in \mathbb{R}^{n_2}$ *denotes the static numerical variables with a domain/box* $[\boldsymbol{y}]$ *and* $\boldsymbol{c}$ *denotes the set of static or differential constraints. Solving a DACSP instance consists in finding the set of values in* $[\boldsymbol{x}](\cdot)$ *and* $[\boldsymbol{y}]$ *satisfying* $\boldsymbol{c}$*.*

3 A Generic Solver for Differential-Algebraic CSP

In Algorithm 1, we give a description of a first generic solver for DACSP. The solver works on a network $P = (\boldsymbol{x}(\cdot), [\boldsymbol{x}](\cdot), \boldsymbol{y}, [\boldsymbol{y}], \boldsymbol{c})$ and returns a set of trajectories satisfying $\boldsymbol{c}$. The input tube $[\boldsymbol{x}](\cdot)$ is defined generally with one single slice $[t_0, t_f] \times [-\infty, \infty]^n$. We attempt to tackle a wide class of problems with pos-

```
Algorithm DACSP-Solver (P = (x(·), [x](·), y, [y], c), specialTimes,
 maxTubeDiam, #slicesMax, Integration, ε = (ε_fpt, ε_integr, ε_3B),
 slicingPolicy, bisectionPolicy)
    do
        /* Slicing loop: */
        ([x](·), [y]) ← Contraction(P, specialTimes, Integration, ε, false)
        [x](·) ← Slicing([x](·), slicingPolicy)
    while MaxDiam([x](·)) > maxTubeDiam and #Slices(tube) < #slicesMax
    if MaxDiam([x](·)) ≤ maxTubeDiam then return [x](·)
    L ← { ([x](·), null) }
    while L ≠ ∅ /* Depth-first tree search */ do
        ([x](·), gate) ← Pop(L)
        ([x](·), [y]) ← Contraction(P, specialTimes, Integration, ε, true,
          gate)
        if MaxDiam([x](·)) ≤ maxTubeDiam then
            solutionsList ← solutionsList ∪ {[x](·) }
        else
            ([x_1](·), [x_2](·), gate) ← Bisect ([x](·), bisectionPolicy)
            L ← {([x_1](·), gate)} ∪ {([x_2](·), gate)} ∪ L
    solutionsList ← TubeMerge(solutionsList, [x](·))
    return solutionsList
```

Algorithm 1: The DACSP solver.

sibly different behaviors. This may impair the effectiveness of a unique generic algorithm. In practice however, the user may already have an intuition of some

instants from which things should propagate. As a consequence, in addition to P, we allow the user to provide a set of *special times*, *i.e.* elements of the temporal domain that involve states of the trajectory $\boldsymbol{x}(\cdot)$ and other static constraints. It allows this first solver to perform contraction more incrementally.

The solver works in two main phases: a so-called *slicing* step splitting the temporal domain into time slices, followed by a tree search subdividing the vectorial tube $[\boldsymbol{x}](\cdot)$. The last `TubeMerge` function compensates a potential clustering effect and merges together pairs of solution tubes that intersect along the temporal domain (on all the slices) in all the $(\boldsymbol{x})$ dimensions. It is necessary when bisection is not used for identifying different solutions, but helps the solver to compute accurate trajectories.

The main precision parameter of the solver is $maxTubeDiam$, a size expressed as the maximum width over all the slices envelopes of the tube. The solver can indifferently compute "thin" trajectories of theoretical null volume (*e.g.*, when dealing with pure IVPs) or "thick" trajectories (*i.e.*, continua of trajectories, *e.g.* when dealing with interval IVPs). In the latter case, the user has to tune this precision parameter to get a good approximation of thick trajectories. A second user-defined $\#slicesMax$ parameter is a maximum number of slices created during the solving, especially during the slicing phase. A large slice number leads to a better trajectory accuracy at the cost of worse performance. Note that the CPU time generally grows linearly in the number of slices.

The first slicing phase is performed by the first `do..while` loop interleaving contraction of P and time slicing ("discretization"). The latter splits several slices of $[\boldsymbol{x}](\cdot)$ into two slices of equal temporal size. Three main slicing policies have been tested:

- (*all*) Split *all* the slices in two.
- (*median*) Compute for all the bounded slices a dx difference between the middle points of 2 consecutive gates, maximum over all the dimensions, *i.e.* $dx = \max_i |m([x_i^k]) - m([x_i^{k-1}])|$, where $m([x_i])$ denotes the middle of $[x_i]$. Split half of the slices with the largest dx and all the unbounded slices.
- (*average*) Split the slices having a dx greater than the average value and all the unbounded slices.

If the loop terminates because the number of slices reaches $\#slicesMax$, the tree search will be in charge to get the $maxTubeDiam$ precision. This slicing phase seems to contradict the principles of most numerical algorithms that decide to subdivide a given time step adaptively. However, the `Integration` procedure called by the `Contraction` method carries out these adaptive time discretization steps, so that both mechanisms, *i.e.* integration and slicing phases, perform time discretization in a complementary manner.

The second phase performed by the second `while` loop is combinatorial. It implements a tree search branching on the domains of the trajectory variables, *i.e.* tubes. Although depth-first search is well-known in the CP community, to our knowledge, no prior work proposed to make choice points on tubes, defined as follows.

Definition 5 (Tube bisection)
Let $[\boldsymbol{x}](\cdot)$ *be a tube of a trajectory* $\boldsymbol{x}(\cdot)$ *defined over* $[t_0, t_f]$.
Let t_k *be an instant in* $[t_0, t_f]$, *i a dimension in* $\{1..n\}$, *and* $[x_i]$ *the interval value of* $[x_i](\cdot)$ *at* t_k. *Let* $mid(x_i)$ *be* $\frac{\overline{x_i}+\underline{x_i}}{2}$.
The tube bisection (t_k, i) *of* $[\boldsymbol{x}](\cdot)$ *produces two tubes* $[\boldsymbol{x}^L](\cdot)$ *and* $[\boldsymbol{x}^R](\cdot)$ *equal to* $[\boldsymbol{x}](\cdot)$ *except at time* t_k, *where* $[x_i^L] = [\underline{x_i}, mid(x_i)]$ *and* $[x_i^R] = [mid(x_i), \overline{x_i}]$.

In practice, a bisection (t_k, i) is applied only to a gate of the tube. Two heuristics are proposed to the user for selecting the instant t_k. The first one picks randomly one instant among the "special times" specified by the user. The second one selects the t_k having the largest box $[\boldsymbol{x}](t_k)$. The dimension $i \in \{1..n\}$, on which the bisection is performed, is decided according to the largest component $[x_i]$.

Note that the DACSP solver is sound because no operator used in Algorithm 1 can eliminate a solution: `Contraction`, `Slicing`, `Bisect`, `TubeMerge`.

4 Contractions in the Solver

The `Contraction` function consists of a simple propagation loop that calls the contractors corresponding to constraints in $\boldsymbol{c}$ until the relative gain in contraction volume is less than ε_{fp}. Contractors can be of any type: HC4-Revise for a numerical constraint or the "map" contractor mentioned in introduction for the robotic application. The propagation loop is followed by a call to a contractor `Dyn3B` enforcing a strong consistency on the tube (see Algorithm 3).

Let us detail in Algorithm 2 an important contractor, called `ExplicitDE`, that carries out tube contractions based on ODEs or integro-differential equations. The procedure is mainly responsible for launching integration steps forward and backward in time through the tube. The actual integration method used is a parameter of Algorithm 2. Note that a guaranteed integration algorithm inferring a new information, like a value of the state known at a specific instant, is incremental in that it may contract only a subset of a tube if no more contraction is obtained at a given gate. The tube contraction is not incremental in the first slicing phase or at the top of the search tree ($gateBis = null$) because the `Slicing` procedure can subdivide numerous slices everywhere in the tube (parameter $isIncremental$ set to false). Therefore integration is run from t_0 to t_f (forward) and from t_f to t_0 (backward). Conversely, during the tree search, integration is triggered by a tube bisection or a domain modification at a special time (whose state is related with a static variable). That is why incremental integrations start from each of these instants.[8] Our solver is endowed with two possible `Integration` procedures. The first one is an "internal" generic integration algorithm that will be incorporated in the Tubex library. Its signature is close to the procedure `Integration` shown in Algorithm 2. It can be triggered from any specified time in the tube, forward or backward, and with the

[8] The actual code is a little bit more complicated. An instant is skipped if it is handled by the previous integration step.

Algorithm `ExplicitDE`($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, *specialTimes*, `Integration`, ε, *isIncremental*, *gateBis*)
  if *gateBis* = *null* **or not** *isIncremental* **then**
    $[\boldsymbol{x}](\cdot) \leftarrow$ `Integration`($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, t_0, FORWARD, ε, *false*)
    $[\boldsymbol{x}](\cdot) \leftarrow$ `Integration`($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, t_f, BACKWARD, ε, *false*)
  else
    gates $\leftarrow$ `Sort`($\{$*gateBis*$\} \cup$ *specialTimes*)
    forall *gate* $\in$ *gates* **do**
      `// forward and incremental simulation:`
      $[\boldsymbol{x}](\cdot) \leftarrow$ `Integration`($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, *gate*, FORWARD, ε, *true*)
    forall *gate* $\in$ *gates, in reverse order* **do**
      `// backward and incremental simulation:`
      $[\boldsymbol{x}](\cdot) \leftarrow$ `Integration`($\boldsymbol{f}$, $[\boldsymbol{x}](\cdot)$, *gate*, BACKWARD, ε, *true*)

Algorithm 2: A generic contractor for ODE and integro-diff equation

possibility of running the simulation incrementally, *i.e.* stopping it if no sufficient contraction volume gain has been obtained in a gate box. This procedure `Integration` is generic in that it can accept an evolution function $\boldsymbol{f}$ describing either an ODE or an integro-differential equation (see Definition 2). It can also be specialized by a "slice integration contractor" called at each time step of the simulation. Two slice contractors are highlighted in this paper. The first one, called `DynBasic` hereafter, wraps at the slice level two simple contractors available since the very first version of `Tubex`: `CtcDeriv` and an evaluation of the evolution function $\boldsymbol{f}$ called iteratively. `CtcDeriv` (denoted $\mathcal{C}_{\frac{d}{dt}}$ in the literature [34]) is a tube contractor treating the constraint $\dot{\boldsymbol{x}}(\cdot) = \boldsymbol{v}(\cdot)$, where $\boldsymbol{x}(\cdot)$ and $\boldsymbol{v}(\cdot)$ are two trajectories and $\boldsymbol{v}(\cdot)$ is the derivative of $\boldsymbol{x}(\cdot)$ over time. The fundamental theorem of calculus that relates differentiation and integration, is used by `CtcDeriv` for contracting the tube $[\boldsymbol{x}](\cdot)$. The second slice contractor, called `DynCIDGuess` hereafter, generates for each integration step a "slice" contractor graph, where the variables correspond to the two gates and the slice envelope (see Sec. 2.1). Based on the input gate box and the envelope, `DynCIDGuess` can improve the output gate box using sophisticated singleton consistencies based on 3B and CID (see Section 2.1). This contractor will be detailed in another article. This generic integration contractor starts by calling a Picard operator that allows one to set non-infinite initial bounds on some tube $[\boldsymbol{x}](\cdot)$, which is required for engaging contraction, and can create new slices adaptively [27].

A second `Integration` procedure wraps directly the state-of-the-art VNODE [28] guaranteed integration solver into a `CtcVnode` contractor. During the slicing phase, it calls VNODE simulations forward and backward from the smallest gate. After each bisection, it calls VNODE simulations forward and backward starting from the bisected gate. To make `CtcVnode` a contractor, the results obtained by the VNODE simulations are intersected with the current tube. VNODE performs its own slicing, especially in the first iterations, and the slices produced are added to those from the slicing phase. Finally, as we will see

in the experiments, the contractor `CtcVnode`, and a slice integration contractor as `DynBasic` or `DynCIDGuess`, can be called successively inside the contraction loop performed by the `Contraction` procedure.

Another new and useful dynamic contractor is the `Dyn3B` contractor described in Algorithm 3. This is a dynamic adaption of the 3B algorithm described in Section 2.1. It selects iteratively the instant (gate) t_k with the largest interval (the tube is thus not contracted at all instants) and applies a `VarDyn3B` shaving procedure to all the $[x_i]$ intervals at t_k. `VarDyn3B` is a straightforward adaptation of the standard `Var3B` shaving procedure (see Section 2.1) to tubes. Subintervals at the bounds of $[x_i]$ can be safely eliminated if an integration starting from the corresponding sub-tube leads to an empty domain. This integration procedure can be achieved by `DynBasic`, or `CtcVnode` followed by `DynBasic`.

Algorithm `Dyn3B` *(P, $\boldsymbol{\varepsilon}$)*
  do
    volumeSave $\leftarrow$ volume($[\boldsymbol{x}](\cdot)$)
    $t_k \leftarrow$ `SelectGate`($[\boldsymbol{x}](\cdot)$)
    forall $i \in \{1..n\}$ **do**
      $[\boldsymbol{x}](\cdot) \leftarrow$ `VarDyn3B`(P, t_k, i)
  while *VolumeGain*($[\boldsymbol{x}](\cdot)$, *volumeSave*) $> \varepsilon_{3B}$

Algorithm 3: The Dynamic 3B algorithm

5 Experiments

The goal of this section is to highlight that the DACSP model, the contractors available via `Tubex` and our DACSP solver can handle a large variety of systems that no competitor or a few ones can deal with. All the results have been obtained on a CPU computer using an `x86-64` processor (1.6 GHz).

5.1 BVP for Integro-Differential Equation

Let us illustrate the versatility of our DACSP solver on the following problem. It combines an integro-differential equation defined on the domain $[0, 1]$ and a constraint between the initial and final values, as follows:

$$\begin{cases} \dot{x}(t) = 1 - 2x(t) - 5\int_0^t x(\tau)d\tau;\ t \in [0,1] \\ x(0)^2 + x(1)^2 = 1 \end{cases} \tag{1}$$

Our solver can find both solutions in 8.35 seconds and needs to resort to 66 bisections (and a search tree depth of 25) to isolate them at a good accuracy. For both solutions, Table 1 reports some details. Note that only our generic `Integration` algorithm can be used in the solver for this particular problem

Table 1. Solutions obtained on the integro-differential based system. The table reports the diameters of the initial and final gates, the tube volume and the slices number.

Solution	Diam. of gate $t_0 = 0$	Diam. of gate $t_f = 1$	Tube volume	#slices
1	0.015	0.030	0.018	400
2	0.034	0.022	0.024	400

since there is no ODE, contrarily to the following DACSP systems. It has been run with $maxTubeDiam = 0.02$ and $\#slicesMax = 400$.

For the next two DACSP categories tested, we show the best combination of the `CtcVnode` (refered by `vnode` in the tables), `DynBasic` (`basic`), `DynCIDGuess` (`CIDG`) and `Dyn3B` (`3Bvnode` or `3Bbasic`) contractors.

5.2 BVPs and Cruz & Barahona System

We have tested and reported in Table 2 five BVPs and the Cruz system close to a BVP because no state is fully determined at a given instant. However, note that this system has a thick tube solution.

$$\begin{cases} \dot{x}(\cdot) = x(\cdot) \\ x(0)^2 + x(1)^2 = 1 \\ [t_0, t_f] = [0, 1] \\ maxTubeDiam = 0.0005 \end{cases} \quad (2) \qquad \begin{cases} \ddot{x}(\cdot) = -x(\cdot) \\ x(0) = 0; x(\pi/2) = 2 \\ [t_0, t_f] = [0, \pi/2] \\ maxTubeDiam = 0.0005 \end{cases} \quad (3)$$

$$\begin{cases} \ddot{x}(\cdot) = 5\dot{x}(\cdot) \\ x(0) = 1;\ x(1) = 0 \\ \dot{x}(0) \in [-10, 10];\ \dot{x}(1) \in [-10, 10] \\ [t_0, t_f] = [0, 1] \\ maxTubeDiam = 0.02 \end{cases} \quad (4) \qquad \begin{cases} \ddot{x}(\cdot) = -10(\dot{x}(\cdot) + x(\cdot)^2) \\ x(0) = 0; x(1) = 0.5 \\ \dot{x}(0) \in [-20, 20]; \dot{x}(1) \in [-20, 20] \\ [t_0, t_f] = [0, 1] \\ maxTubeDiam = 0.05 \end{cases} \quad (5)$$

$$\begin{cases} \ddot{x}(\cdot) = -\exp(x(\cdot)) \\ x(0) = 0;\ x(1) = 0 \\ \dot{x}(0) \in [-20, 20] \\ \dot{x}(1) \in [-20, 20] \\ [t_0, t_f] = [0, 1] \\ maxTubeDiam = 0.05 \end{cases} \quad (6) \qquad \begin{cases} \dot{x}_1(\cdot) = -0.7x_1(\cdot) \\ \dot{x}_2(\cdot) = 0.7x_1(\cdot) - (\ln(2)/5)x_2(\cdot) \\ x_1(0) = 1.25 \\ x_2 \in [1.1, 1.3] \text{ during } [1, 3] \\ [t_0, t_f] = [0, 6] \\ maxTubeDiam = 0.04 \end{cases} \quad (7)$$

Fig. 2. Five BVPs and the Cruz system. (2) A one-dimensional problem with an algebraic constraint between the initial and final states; (3) Classical linear example cited in Wikipedia; (4) and (5) denote resp. Systems 2 and 23 in the BVPSolve benchmark [24]; (6) the Bratu system, the only one with two solutions in the BVPSolve benchmark, (7) the Cruz system with a partial information in the middle of the temporal domain.

5.3 Interval IVPs

Although solving interval IVPs is not the primary purpose of the DACSP solver, we present results obtained on three interval IVPs (results in Table 3).

Table 2. Solutions obtained on BVP systems. For each system, strategy and solution (s_1 and/or s_2), we report the diameters of the two unknown states in t_0 and t_f (most of the systems tested are 2-dimensional, but 2 of the 4 bounds are provided as initial conditions), the volume of the solution tubes, the number of slices, the computational time and the number of choice points required (#bis.).

Sys.	Best strategy	#sol	t_0 diam.	t_f diam.	Tube vol.	#slices	Time	#bis.
(2)	vnode+basic	s_1	2e-8	5e-8	2.e-4	5,000	7.63s	1
		s_2	2e-8	5e-8	2.e-4	5,000		
(3)	vnode+basic	s_1	7e-15	7e-15	6e-4	12,288	12.7s	0
(4)	vnode+CIDG+3Bvnode	s_1	2e-9	4e-7	5e-3	1,216	3.05s	0
(5)	vnode+CIDG+3Bbasic	s_1	3.e-2	2.e-4	0.012	5,000	81s	6
(6)	vnode+basic	s_1	3.e-6	2.e-6	7.e-4	2,000	75s	62
		s_2	5.e-3	5.e-3	0.025	2,000		
(7)	CIDG	s_1	0.0644	0.0282	0.2637	10,000	6.85s	1

$$\begin{cases} \dot{x} = -x^2 \\ x(0) \in [0.1, 0.4] \\ [t_0, t_f] = [0, 5] \\ eps = 0.2 \end{cases} \quad (8) \qquad \begin{cases} \dot{x}_1 = -x_1 - 2x_2 \\ \dot{x}_2 = -3x_1 - 2x_2 \\ x_1(0) \in [5.9, 6.1] \\ x_2(0) \in [3.9, 4.1] \\ [t_0, t_f] = [0, 1] \\ eps = 0.5 \end{cases} \quad (9) \qquad \begin{cases} \dot{x}_1 = -x_2 + 0.1\, x_1\, (1 - x_1^2 - x_2^2) \\ \dot{x}_2 = x_1 + 0.1\, x_2\, (1 - x_1^2 - x_2^2) \\ x_1(0) \in [0.7, 1.3] \\ x_2(0) = 0.0 \\ [t_0, t_f] = [0, 5] \\ eps = 0.15 \end{cases} \quad (10)$$

Fig. 3. Three interval IVPs tested. (8) and (9) were introduced in [8]. (10) describes a limit cycle and is particularly sensitive to the wrapping effect caused by interval computation.

For the 2 examples from [8], the exact solution is known, so we also report the relative error (column Gap in Table 3) on interval width of the final gates.

5.4 Discussion on Experiments

Note first that the `VNODE` solver alone (outside the DACSP solver) cannot cope with BVPs and is not efficient on the interval IVPs selected. `CtcVnode` (inside the DACSP solver) often provides a very good accuracy on the gates. The good performance is probably due to the high-order interval Taylor form used (order 11 has been set for the experiments). However, `CtcVnode` does generally not obtain good contraction on the whole tube (slice envelopes). Additional work is required to envisage obtaining a better tube volume accuracy using `CtcVnode`.

Table 3. Solutions obtained on interval IVPs systems. For each system, we report the best strategy, the diameters of the state at t_f, the volume of the solution tube, the number of slices, the computational time and the number of bisections.

Sys.	**Strategy**	**t_f diam.**	**Tube vol.**	**Gap**	**#slices**	**Time**	**#bis.**
(8)	CIDG	0.06668	0.6934	0.02%	40,000	9.35 s	1
(9)	vnode+CIDG	(0.544;0.544)	0.700	(0.01%;0.01%)	2,000	3.93 s	1
(10)	vnode+basic	(0.0695;0.2273)	2.54		1,000	13.3 s	7

`DynCIDGuess` alone is generally not efficient, except on Systems (7) and (8), because it requires too many slices to reach the precision (recall that the CPU time generally grows linearly in the number of slices). Finally, the best option for the `Integration` procedure is generally to call first `CtcVnode` and then `DynBasic` or `DynCIDGuess`. A final call to `Dyn3B` is useful for Systems (4) (using `CtcVnode` as subcontractor) and (5) (using `DynBasic`).

Overall, different solver strategies provide the best results on the different systems tested. All the devices offered by the solver can be useful on different instances: contractors, slicing, choice points. When the best strategy includes a number of bisections, this means that the $\#slicesMax$ has been reached and the solver resorts to choice points to better approximate the solution tube. An issue for future work is to better study the interplay between slicing and bisection in order to obtain a more generic DACSP solver that can work without the $\#slicesMax$ parameter.

6 Conclusion and Future Work

We have presented a new generic solver that can handle together differential and static numerical constraints. The originality of the approach lies both in the underlying model considering trajectories as variables and in a novel backtracking mechanism applicable to DACSP. Our DACSP solver is endowed with an exploration operator that enables to bisect a tube at a chosen time. This allows the DACSP solver to better handle hard DACSP systems and accurately estimate distinct trajectories of problems having several solutions. We have shown on first experimental results that our solver is versatile enough to solve DACSP instances for which no or a few algorithmic solutions currently exist. We have also demonstrated the benefits of wrapping the state-of-the-art VNODE in a `CtcVnode` contractor implemented in `Tubex`.

With regard to future work, we will first try to limit the number of user-defined parameters, in particular remove $\#slicesMax$. Also, we want to propose ideally only one combination of contractors in the DACSP solver for every DACSP subclass. Second, we will study a more general search tree branching static and dynamical variables domains indifferently, though we need to explore new ideas on large-scale problems. Finally, in the current solver, the propagation between functional and real variables domains is somewhat naive and is partly ensured by the "special times" specified by the user (see Algorithm 2). The quite recent Tubex 3.0 accepts bi-level slice/tube variables, which will enable a fully incremental contraction achieved by a propagation engine.

Supplementary materials including the sources of the solver and the experiments are available on http://simon-rohou.fr/research/dacsp-solve/.

Acknowledgements. This work was supported by the French Agence Nationale de la Recherche (ANR) [grant number ANR-16-CE33-0024].

References

1. F. Benhamou, F. Goualard, L. Granvilliers, and J.-F. Puget. Revising Hull and Box Consistency. In *Proc. of International Conference on Logic Programming (ICLP)*, pages 230–244, 1999.
2. G. Chabert. IBEX – an Interval-Based EXplorer, 2020. http://www.ibex-lib.org/.
3. G. Chabert and L. Jaulin. Contractor Programming. *Artificial Intelligence*, 173:1079–1100, 2009.
4. A. Chapoutot, J. Alexandre dit Sandretto, and O. Mullier. *Dynibex*. 2015. http://perso.ensta-paristech.fr/ chapoutot/dynibex/.
5. H. Collavizza, F. Delobel, and M. Rueher. Comparing Partial Consistencies. *Reliable Computing*, 5(3):213–228, 1999.
6. J. Cruz and P. Barahona. Constraint Satisfaction Differential Problems. In *Principles and Practice of Constraint Programming - CP 2003.*, pages 259–273, 2003.
7. J. Cruz and P. Barahona. Constraint Reasoning with Differential Equations. *Applied Numerical Analysis & Computational Mathematics*, 1(1):140–154, 2004.
8. Y. Deville, M. Janssen, and P. VanHentenryck. Consistency Techniques in Ordinary Differential Equations. In *Proc. of CP98*, pages 162–176, 1998.
9. F. Domes. GLOPTLAB: A configurable framework for the rigorous global solution of quadratic constraint satisfaction problems. *Optimization Methods & Software*, 24:727–747, 10 2009.
10. T. F. Filippova, A. B. Kurzhanski, K. Sugimoto, and I. Vályi. Ellipsoidal State Estimation for Uncertain Dynamical Systems. In *Bounding Approaches to System Identification*, pages 213–238. Springer US, Boston, MA, 1996.
11. A. Goldsztejn, O. Mullier, D. Eveillard, and H. Hosobe. Including Ordinary Differential Equations Based Constraints in the Standard CP Framework. In *Proc of CP 2010*, pages 221–235. Springer Berlin, Heidelberg, 2010.
12. T.R Goodman and G.N. Lance. The Numerical Solution of Two-Point Boundary Value Problems. *Mathematical Tables and Other Aids to Computation*, 10:82–86, 1956.
13. L. Granvilliers and F. Benhamou. RealPaver: An Interval Solver using Constraint Satisfaction Techniques. *ACM Transactions on Mathematical Software - TOMS*, 32:138–156, 2006.
14. E. R. Hansen. *Global Optimization using Interval Analysis*. Marcel Dekker, New York, NY, 1992.
15. L. Jaulin. Nonlinear Bounded-error State Estimation of Continuous-Time Systems. *Automatica*, 38:1079–1082, 2002.
16. L. Jaulin. Range-Only SLAM with Indistinguishable Landmarks: A Constraint Programming Approach. *Constraints*, 21(4):557–576, 2016.
17. R.E. Kalman. Contributions to the Theory of Optimal Control. *Bol. Soc. Mat. Mex.*, 5:102–119, 1960.
18. T. Kapela, M. Mrozek, P. Pilarczyk, D. Wilczak, and P. Zgliczynski. CAPD – a rigorous toolbox for Computer Assisted Proofs in Dynamics. `http://capd.ii.uj.edu.pl/`, 2010.
19. R. Kearfott. GlobSol: History, Composition, and Advice on Use. In *Proc of COCOS2002*, LNCS 2861, pages 17–31. Springer, 10 2002.
20. A. B. Kurzhanski and T. F. Filippova. On the Theory of Trajectory Tubes - A Mathematical Formalism for Uncertain Dynamics, Viability and Control. In *Advances in Nonlinear Dynamics and Control: A Report from Russia*, pages 122–188. Birkhäuser, Boston, MA, 1993.

21. F. Le Bars, J. Sliwka, L. Jaulin, and O. Reynet. Set-membership state estimation with fleeting data. *Automatica*, 48(2):381–387, 2012.
22. O. Lhomme. Consistency Techniques for Numeric CSPs. In *IJCAI*, pages 232–238, 1993.
23. R. Lohner. Enclosing the solutions of ordinary initial and boundary value problems. In E. Kaucher, U. Kulisch, and Ch. Ullrich, editors, *Computer Arithmetic: Scientific Computation and Programming Languages*, pages 255–286. BG Teubner, Stuttgart, Germany, 1987.
24. F. Mazzia, J.R. Cash, and K. Soetaert. Solving boundary value problems in the open source software R: Package bvpSolve. *Opuscula mathematica*, 34(2):387–403, 2014.
25. F. Messine. *Méthodes d'optimisation globale basées sur l'analyse d'intervalle pour la résolution des problèmes avec contraintes*. PhD thesis, LIMA-IRIT-ENSEEIHT-INPT, Toulouse, 1997.
26. R. E. Moore. *Interval Analysis*, volume 4. Prentice-Hall Englewood Cliffs, 1966.
27. N. Nedialkov, K. Jackson, and G. Corliss. Validated Solutions of Initial Value Problem for Ordinary Differential Equations. *Applied Mathematics and Applications*, 105(1):21–68, 1999.
28. N.S. Nedialkov. VNODE-LP, A Validated Solver for Initial Value Problems in Ordinary Differential Equations. Technical Report CAS-06-06-NN, Department of Computing and Software, McMaster University,Hamilton, Ontario, Canada, 2006.
29. C. Prud'homme, J.-G. Fages, and X. Lorca. Choco documentation. 2014. http://www. choco-solver. org.
30. N. Revol, K. Makino, and M. Berz. Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. *Journal of Logic and Algebraic Programming*, 64:135–154, 2005.
31. S. Rohou, B. Desrochers, and L. Jaulin. Set-membership State Estimation by Solving Data Association. In *IEEE International Conference on Robotics and Automation*, 2020.
32. S. Rohou et al. The Tubex library – Constraint-programming for robotics, 2020. http://simon-rohou.fr/research/tubex-lib/.
33. S. Rohou, L. Jaulin, L. Mihaylova, F. Le Bars, and S. M. Veres. Reliable Nonlinear State Estimation Involving Time Uncertainties. *Automatica*, 93:379–388, 2018.
34. S. Rohou, L. Jaulin, L. Mihaylova, F. Le Bars, and S. M. Veres. Guaranteed computation of robot trajectories. *Robotics and Autonomous Systems*, 93:76–84, 2017.
35. S. Thrun, W. Bugard, and D. Fox. *Probabilistic Robotics*. MIT Press, Cambridge, M.A., 2005.
36. G. Trombettoni and G. Chabert. Constructive Interval Disjunction. In *Proc. CP, Constraint Programming, LNCS 4741*, pages 635–650. Springer, 2007.
37. W. Tucker. A Rigorous ODE Solver and Smale's 14th Problem. *Foundations of Computational Mathematics*, 2(1):53–117, 2002.
38. R. Voges and B. Wagner. Timestamp offset calibration for an IMU-camera system under interval uncertainty. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 377–384, 2018.
39. D. Wilczak and P. Zgliczynski. Heteroclinic connections between periodic orbits in planar restricted circular three-body problem–a computer assisted proof. *Communications in mathematical physics*, 234(1):37–75, 2003.

Utiliser la PPC pour générer des structures de benzénoïdes en chimie théorique*

Yannick Carissan[1] **Denis Hagebaum-Reignier**[1]
Nicolas Prcovic[2] **Cyril Terrioux**[2] **Adrien Varet**[2]
[1] Aix Marseille Univ, CNRS, Centrale Marseille, ISM2, Marseille, France
[2] Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{prénom.nom}@univ-amu.fr

Résumé

Les benzénoïdes sont une sous-famille d'hydrocarbures (molécules composées uniquement d'atomes d'hydrogène et de carbone) dont les atomes de carbone forment des hexagones. Ces molécules ont fait l'objet de nombreuses études en chimie théorique et possèdent de nombreuses applications concrètes. La génération de benzénoïdes ayant certaines propriétés structurelles (par exemple, ayant un nombre donné d'hexagones ou ayant une structure particulière du point de vue graphique) est un problème intéressant et important. Il constitue une étape préliminaire à l'étude de leurs propriétés chimiques. Dans cet article, nous montrons que modéliser ce problème dans Choco Solver et laisser son moteur de recherche générer les solutions constitue une approche rapide et très flexible. Elle permet notamment de générer des structures répondant aux besoins des chimistes simplement en ajoutant de nouvelles variables et/ou contraintes tout en évitant d'avoir à développer des méthodes algorithmiques ad-hoc.

Ce papier est un résumé de [2].

1 Introduction

Les *hydrocarbures aromatiques polycycliques (HAP)* sont des hydrocarbures dont les atomes de carbone forment des cycles de différentes tailles (de taille 6 dans le cas des *benzénoïdes*). Ils ont été très étudiés dans divers domaines (nanoélectronique moléculaire, synthèse organique, chimie interstellaire, ...) en raison de leur stabilité énergétique, de leurs structures moléculaires ou de leurs spectres optiques. Ils présentent une grande variété de propriétés physico-chimiques en fonction de leur taille et de leur structure. Ainsi, générer des structures de benzénoïdes respectant certaines propriétés est d'un grand intérêt du point de vue de la chimie théorique.

Dans la littérature, des approches ont été proposées pour générer des structures quelconques ou satisfaisant des propriétés particulières [1]. Il s'agit d'approches ad-hoc qui s'avèrent très efficaces en pratique mais qui sont difficiles à adapter aux besoins des chimistes et qui requièrent un travail de développement significatif. Dans cet article, nous exploitons la PPC pour proposer une nouvelle approche qui soit plus flexible tout en étant relativement compétitive par rapport aux méthodes ad-hoc. En effet, d'une part, une fois le problème modélisé dans toute sa généralité, nous pouvons aisément le spécialiser en ajoutant des variables et/ou des contraintes afin de répondre aux besoins exprimés par les chimistes. D'autre part, la PPC offre des outils de résolution très efficaces.

2 Génération de benzénoïdes

Le *benzène*, représenté à la figure 1(a), est une molécule dont les six atomes de carbone forme un hexagone (appelé *cycle benzénique*). Les benzénoïdes sont les molécules qui peuvent être obtenues en fusionnant des cycles benzéniques Par exemple, l'anthracène (voir figure 1(b)) contient trois cycles benzéniques fusionnés. En exploitant cette caractéristique, nous représentons la structure d'un benzénoïde B par le graphe non orienté B_h (appelé *graphe d'hexagones*) dont chaque sommet correspond à un hexagone (cycle benzénique) de B et tel que deux sommets sont reliés par une arête si les hexagones correspondants partagent une arête. La figure 1(d) présente ce graphe pour le coronène. Nous définissons le problème de génération ainsi : Étant donné un ensemble de propriétés structurelles $\mathcal{P}$, générer toutes les structures satisfaisant les propriétés

*Ce travail est soutenu par l'Agence Nationale de la Recherche dans le cadre du projet DEMOGRAPH (ANR-16-CE40-0028).

de $\mathcal{P}$. Ces propriétés peuvent porter sur le nombre de carbone ou d'hexagone ou sur des formes particulières du graphe d'hexagones (arbre, rectangle, ...).

Nous présentons maintenant une modélisation CSP permettant de générer toutes les structures ayant n hexagones. Elle repose sur la propriété que tout benzénoïde de n hexagones peut être placé dans un coronénoïde de taille au plus $k(n) = \lfloor \frac{n}{2} + 1 \rfloor$. Un coronénoïde de taille k est une molécule de benzène à laquelle on a ajouté successivement $k-1$ couronnes d'hexagones (le coronène est le coronénoïde de taille 2). Par la suite, on note $B_h^{c(k(n))}$ le graphe d'hexagones du coronénoïde de taille $k(n)$, n_c son nombre d'hexagones et m_c celui d'arêtes. Nous numérotons arbitrairement les hexagones de $B_h^{c(k(n))}$ à partir de 1. Tout d'abord, nous considérons une variable de graphe x_G pour représenter le graphe d'hexagones de la structure souhaitée. Son domaine est l'ensemble de tous les sous-graphes entre le graphe vide et $B_h^{c(k(n))}$. Nous exploitons également un ensemble de n_c variables booléennes $\{x_1, \ldots, x_{n_c}\}$. La variable x_i vaut 1 si le i-ème hexagone de $B_h^{c(k(n))}$ est utilisé dans x_G, 0 sinon. De même, nous considérons un ensemble de m_c variables booléennes $y_{i,j}$. La variable $y_{i,j}$ vaut 1 si l'arête $\{i,j\}$ de $B_h^{c(k(n))}$ est utilisée dans x_G, 0 sinon. Ensuite, nous modélisons les propriétés suivantes à l'aide de contraintes :

- *Lien entre x_G et x_i (resp. $y_{i,j}$)* : nous utilisons une contrainte `channeling` qui impose $x_i = 1 \iff x_G$ *contient le sommet* i (resp. $y_{i,j} = 1 \iff x_G$ *contient l'arête* $\{i,j\}$).
- *x_G est un sous-graphe induit de $B_h^{c(k(n))}$* : Toute valeur de x_G n'est pas nécessairement un graphe d'hexagones valide. Pour garantir sa validité, il doit correspondre à un sous-graphe de $B_h^{c(k(n))}$ induit par les sommets appartenant à x_G. Ainsi, pour chaque arête $\{i,j\}$ de $B_h^{c(k(n))}$, on ajoute une contrainte $x_i = 1 \wedge x_j = 1 \Leftrightarrow y_{i,j} = 1$. En d'autres termes, l'arête $\{i,j\}$ existe dans x_G si et seulement si les sommets i et j apparaissent dans x_G.
- *La structure a n hexagones* : $\sum_{i \in \{1,\ldots,n_c\}} x_i = n$.
- *Le graphe d'hexagones est connexe* : nous appliquons la contrainte de graphe `connected` sur x_G.
- *Six hexagones formant un cycle génèrent un hexagone (et non un trou)* : Pour chaque hexagone u, nous considérons l'ensemble $N(u)$ de voisins de u dans le graphe d'hexagones. Pour chaque sommet u, on pose la contrainte $\sum_{v \in N(u)} x_v = 6 \Rightarrow x_u = 1$.

Enfin, nous ajoutons plusieurs contraintes pour éviter les redondances. D'abord x_G doit avoir au moins un sommet sur le bord supérieur (resp. gauche) de $B_h^{c(k(n))}$ afin d'éviter les symétries par translation. Nous posons donc une contrainte qui spécifie que la somme des variables binaires x_i associées au bord supérieur (resp. gauche) est strictement positive. Ensuite, il faut s'assurer que le graphe décrit par x_G est le seul représentant de sa classe de symétrie. Il existe jusqu'à douze solutions symétriques : six symétries de rotation de 60 degrés combinées à une éventuelle symétrie axiale. Ces symétries sont cassées grâce à la contrainte `lex-lead`. Pour chacune des douze symétries, il faut ajouter n_c variables booléennes (chacune associée à une variable x_i) et un total de $3.n_c$ clauses ternaires.

Ce modèle peut facilement être mis en œuvre avec le solveur *Choco* [3]. Il peut également être spécialisé pour prendre en compte les besoins des chimistes en ajoutant des variables et/ou des contraintes. Par exemple, générer des structures ayant une forme arborescente (appelées benzénoïdes *catacondensés*) nécessite simplement d'ajouter au modèle général la contrainte de graphe `tree` appliquée à x_G. D'autres propriétés ont été modélisées afin de générer des structures ayant une forme rectangulaire, possédant un trou ou symétriques.

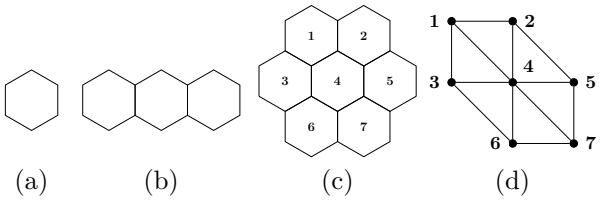


FIGURE 1 – Le benzène (a), l'anthracène (b), le coronène (c) (les atomes d'hydrogène sont omis) et son graphe d'hexagones (d).

3 Conclusion

Nous avons présenté une approche basée sur la PPC permettant de générer des structures de benzénoïdes ayant des propriétés structurelles. Elle s'avère flexible et permet de répondre aux besoins des chimistes tout en se révélant compétitive vis-à-vis des méthodes ad-hoc existantes.

Références

[1] G. BRINKMANN, G. CAPOROSSI et P. HANSEN : A Constructive Enumeration of Fusenes and Benzenoids. *Journal of Algorithms*, 45(2), 2002.

[2] Y. CARISSAN, D. HAGEBAUM-REIGNIER, N. PRCOVIC, C. TERRIOUX et A. VARET : Using Constraint Programming to Generate Benzenoid Structures in Theoretical Chemistry. *In CP*, pages 690–706, 2020.

[3] J.-G. FAGES, X. LORCA et C. PRUD'HOMME : Choco solver user guide documentation.

Using Constraint Programming to Generate Benzenoid Structures in Theoretical Chemistry*,**

Yannick Carissan[1][0000−0002−9876−0272], Denis Hagebaum-Reignier[1][0000−0001−8761−1047], Nicolas Prcovic[2], Cyril Terrioux[2][0000−0002−9779−9108], and Adrien Varet[2]

[1] Aix Marseille Univ, CNRS, Centrale Marseille, ISM2, Marseille, France
[2] Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{firstname.name}@univ-amu.fr

Abstract. Benzenoids are a subfamily of hydrocarbons (molecules that are only made of hydrogen and carbon atoms) whose carbon atoms form hexagons. These molecules are widely studied in theoretical chemistry and have a lot of concrete applications. Therefore, generating benzenoids which have certain structural properties (e.g. having a given number of hexagons or having a particular structure from a graph viewpoint) is an interesting and important problem. It constitutes a preliminary step for studying their chemical properties. In this paper, we show that modeling this problem in Choco Solver and just letting its search engine generate the solutions is a fast enough and very flexible approach. It can allow to generate many different kinds of benzenoids with predefined structural properties by posting new constraints, saving the efforts of developing bespoke algorithmic methods for each kind of benzenoids.

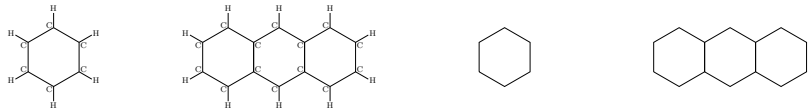
Keywords: Constraint programming · modeling · Graph variables and constraints · Chemistry

1 Introduction

Polycyclic aromatic hydrocarbons (PAHs) are hydrocarbons whose carbons are forming cycles of different sizes. *Benzenoids* are a subfamily of PAHs made of 6-membered carbon rings (i.e. each cycle is a hexagon). To fill carbon valency, each atom of carbon is bonded to either two other carbons and one hydrogen or three carbons. For example, Figures 1(a) and (b) are representing two benzenoids: benzene and anthracene.

PAHs are well-studied in various fields because of their energetic stability, molecular structures or optical spectra. In natural environment, these molecules

* This work has been funded by the Agence Nationale de la Recherche project ANR-16-C40-0028.
** The final authenticated version is available online at https://doi.org/10.1007/978-3-030-58475-7_40.



(a) (b) (c) (d)

Fig. 1. Two small benzenoids: benzene (a) and anthracene (b) with their graphical representations (c) and (d).

are created by the incomplete combustion of carbon contained in combustibles [18]. They are popular research subjects in material sciences, e.g. molecular nanoelectronics where they are used to store or transport energy [29,2] or in organic synthesis [25,22], where the controlled design of specific shapes remains challenging. PAHs are also intensively studied in interstellar chemistry because of their suspected presence in various interstellar and circumstellar environments where they are believed to act as catalysts for chemical reactions taking place in space [14].

PAHs exhibit a large variety of physicochemical properties depending on size and, more specifically, on edge and bond topologies. In the astrophysical community, the so-called "PAH hypothesis" formulated more than 30 years ago, of whether the existence of PAHs in space could explain some unidentified mid-infrared emission bands in astrophysical environments, has motivated numerous observational, experimental and theoretical investigations. It is now accepted that mixtures of free PAHs of different size, shapes and ionic states can account for the overall appearance of the widespread interstellar infrared emission spectrum. But the question of relative abundance of PAHs with a given size and/or shape remains open. Many studies are devoted to explore the effect of the size, shapes in terms of compacity and symmetry of PAHs, on band positions and intensities of the infrared spectra [1,3,23]. Very recently, a systematic investigation of a series of 328 PAHs containing up to 150 carbon atoms showed that PAHs with armchair edges that maximize the Clar number (i.e. the maximum number of non-adjacent rings containing 6 electrons called a sextet) are potential emitters of a certain class of astrophysical infrared sources [24]. For their study, the authors needed to systematically generate all PAHs having armchair edge topology and selecting a subclass of PAHs whose structure maximizes the Clar number. They used the algorithm of Caporossi and Hansen [8]. Constraint-programming is particularly well suited for the generation of such families of PAHs.

Another important example where the generation of specific shapes is relevant for chemists deals with so-called "non-Kekulean" benzenoids [10]. These benzenoids cannot be represented by Kekulé structures, i.e. structures that have only simple and double bonds. From a graph-theoretical point of view, Kekulé structures are covered by the maximal number of disjoint (double) edges so that all vertices are incident to one of the disjoint edges. It was accepted among chemists until recently that "non-Kekulean" benzenoids should be very unstable

due to their open-shell electronic structure (i.e. one or more electron(s) remain unpaired, contrary to a closed-shell structure where all electrons are paired) and thus their synthesis would be a real challenge. The experimental realization and in-depth characterization of small "non-Kekulean" benzenoids was very recently achieved on gold surfaces [21,20]. These studies opened the way to the synthesis of new classes of compounds which show unconventional magnetism induced by their topology, with promising applications in various fields like molecular electronics, nonlinear optics, photovoltaics and spintronics. Moreover, it was shown that some PAHs with specific topologies (e.g. rhombus shapes) may "prefer" having an open-shell structure when reaching a certain size, although they could have a closed-shell structure and could thus be described by a set of Kekulé structures [27]. From a quantum theoretical point of view, the proper description of the electronic structure of open-shell benzenoids is a difficult task. The use of a constraint programming approach for the systematic search of larger non-Kekulean or Kekulean benzenoids having an open-shell electronic structure is undoubtedly advantageous.

In this context, many approaches have been proposed in order to generate benzenoids having or not a particular shape or satisfying a particular property (e.g. [5,6]). These are bespoke approaches which have the advantage of being efficient, but are difficult to adapt to the needs of chemists. Moreover, designing a new bespoke method for each new desired property often requires a huge amount of efforts. So, in this paper, we prefer to use an approach based on constraint programming. With this aim in view, we present a general model which can be refined depending on the desired properties by simply adding variables and/or constraints. By so doing, our approach benefits from the flexibility of CP and requires less efforts of implementation. In the meantime, CP offers efficient solvers which can be quite competitive with respect to bespoke algorithms.

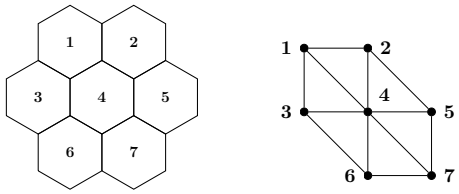
The paper is organized as follows. First, we recall some definitions about benzenoids and constraint programming in Section 2. Section 3 introduces the fastest existing algorithm for generating benzenoids. Then Section 4 presents a new approach using constraint programming, explains its advantages and gives some examples. Finally, we conclude and provide some perspectives in Section 5.

2 Preliminaries

2.1 Theoretical Chemistry

Benzene, represented in Figure 1(a) is a molecule made of 6 carbon atoms and 6 hydrogen atoms. Its carbon atoms form a hexagon (also called *benzenic cycle* or *benzenic ring*) and each of them is linked to a hydrogen atom. *Benzenoids* are a subfamily of PAHs containing all molecules which can be obtained by aggregating benzenic rings. For example, Figure 1(b) shows anthracene, which contains three benzenic rings.

By definition of the *valence* (i.e. the number of bonds that an atom can establish) of carbon and hydrogen atom, we know that each carbon atom is



(a) (b)

Fig. 2. Coronene (a) and its hexagon graph (b).

linked to either two other carbon atoms and one hydrogen atom or three other carbon atoms. So benzenoids can be perfectly defined by describing only the interactions between carbon atoms. Hydrogen atoms can then be deduced since each hydrogen atom is linked to a carbon atom which is only bonded to two other carbon atoms. As a consequence, any benzenoid can be represented as an undirected graph $B = (V, E)$, with V the set of vertices and E the set of edges. Every vertex in V represents a carbon atom and every edge of E represents a bond between the two corresponding carbons. Moreover, this kind of graph, is connected, planar and bipartite. Figures 1(c) and (d) represent the graphs related to the molecules of benzene and anthracene.

In the following, for any benzenoid B, we need to consider some of its faces. A *face* of a planar graph is an area of the plan bounded by edges. Figure 2(a) presents the graph corresponding to coronene (a well-known benzenoid). This graph has eight faces namely the seven numbered faces and the external face. Note that in the sequel, we do not take into account the external face. For this example, the numbered faces correspond exactly to the hexagons of coronene. However, we will see later that this property does not hold for all the benzenoids.

Then, given a benzenoid, we consider another graph, namely the hexagon graph. The *hexagon graph* of a benzenoid $B = (V, E)$ is the undirected graph $B_h = (V_h, E_h)$ such that there is a vertex v_h from V_h per hexagonal face h of B (the external face and "holes" in the benzenoid are excluded) while there is an edge $\{v_h, v_{h'}\}$ in E_h if the corresponding hexagonal faces h and h' of B share an edge of E. Figure 2(b) presents the hexagon graph of coronene. The hexagon graph allows us to express the interaction between the hexagons of the considered benzenoid.

2.2 Constraint Programming

An instance I of the *Constraint Satisfaction Problem (CSP)* is a triplet (X, D, C). $X = \{x_1, \ldots, x_n\}$ is a set of n *variables*. For each variable $x_i \in X$, there exists an associated domain $D_{x_i} \in D = \{D_{x_1}, \ldots, D_{x_n}\}$ which represents the values that x_i can take. $C = \{c_1, ..., c_e\}$ represents a set of e *constraints*. Constraints represent the interactions between the variables and describe the allowed combinations of values.

Solving a CSP instance $I = (X, D, C)$ amounts to find an assignment of all the variables of X with a value contained in their associated domain which satisfies all the constraints of C. Such assignment is called a *solution*. This problem is NP-hard.

Many libraries are available to represent and solve efficiently CSP instances. In this paper, we exploit the open-source Java library *Choco* [15]. This choice is highly guided by our need to be able to define *graph variables* and directly apply graph-related constraints (e.g. connected or cyclic constraints). Graph variables have as domain a set of graphs defined by a lower bound (a sub-graph called GLB) and an upper bound (a super-graph called GUB). Moreover, Choco implements the usual global constraints which make the modeling easier and its solver is efficient and configurable.

3 Generating Benzenoids

We can define the benzenoid generation problem (denoted BGP in the future) as follows: given a set of structural properties $\mathcal{P}$, generate all the benzenoids which satisfy each property of $\mathcal{P}$. For instance, these structural properties may deal with the number of carbons, the number of hexagons or a particular structure for the hexagon graph. Of course, the more interesting instances of the BGP problem combine several properties. For example, Figure 5 shows benzenoids having a tree as hexagon graph. Such a property-based instances design allows for the search of benzenoids with chemically relevant properties. Our interest lies in the search of benzenoids with radical electronic structures (as in the work of Malrieu and Trinquier [27]), which arise from their geometrical arrangement.

Now, we present an existing method proposed by Brinkmann et al. [5]. Given an integer n, this method is able to generate all the benzenoids with n hexagons by generating all the hexagon graphs with at most n vertices. This is done by adding successively new vertices to the hexagon graph (which is equivalent to generate all the wanted molecules by successively adding new hexagons).

This method is really efficient. For instance, it could generate the 669,584 benzenoids having 12 hexagons in 1.2 seconds and 1,000 billions of benzenoids having 21 hexagons in two weeks when launched on an old computer (Intel Pentium, 133 MHz, 2002). However it has some disadvantages. Indeed it is not complete in the sense that it is unable to generate benzenoids with "holes". By hole, we mean a face which does not correspond to a hexagon or the external face. For example, Figure 3(a) depicts the smallest benzenoid (in terms of number of hexagons) which admits a hole. Such a benzenoid cannot be produced by this method. Indeed, when this method wants to add a new hexagon, it checks whether the added hexagon allows to close a cycle of hexagons. If so, the hexagon is not added and so benzenoids with holes cannot be generated. Benzenoids with holes are quite seldom. There is a single one for 8 hexagons (among 1,436 benzenoids), 5 for 9 hexagons (among 6,510). Note that this proportion grows as we increase the number of hexagons (see Table 1). Furthermore, this method is unable to take into account other properties natively and cannot easily be tuned

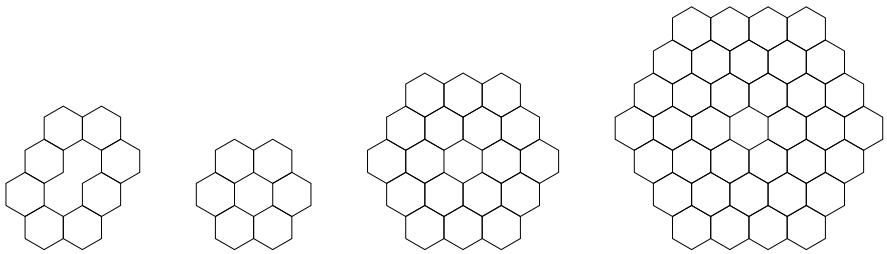


Fig. 3. The smallest benzenoid with hole (a) and coronenoids of size 2 (b), 3 (c) and 4 (d).

to fit the needs of chemists. Indeed, it is based on an augmenting procedure that decides how to add a vertex. So this procedure should be changed and proven adequate to avoid generating non canonical graphs (i.e. redundant isomorphic graphs) each time we want to change the structural property of the benzenoids we wish to generate. It is therefore a relatively heavy task even for the addition of a basic property.

In the next section, we present a new method using constraint programming which is able to generate any benzenoid structure and benefits from the flexibility of constraint programming.

4 Generating Benzenoid Structures Thanks to CP

In this section, we see how to model a BGP instance as a CSP instance. We first present a general model which considers the generation of all the benzenoids having a given number of hexagons. This property is the minimal property to ensure. Then we provide some examples showing how the model can be easily specialized to take into account some additional structural properties.

4.1 General Model

In this part, we want to generate all the benzenoids having a given number n of hexagons. Before modeling this problem as a CSP instance, we highlight some useful properties. A *coronenoid* of size k is a molecule of benzene (i.e. a hexagon) to which we successively add $k-1$ crowns of hexagons. Benzene corresponds to the coronenoid of size 1 (see Figure 1(c)). Figures 3(b)-(d) present the coronenoids of size 2, 3 and 4. Note that the diameter (i.e. the number of hexagons of the central line) of a coronenoid of size k is $2 \times k - 1$. Our interest for coronenoids lies in the fact that they are useful to "embed" benzenoids of a given number of hexagons:

Property 1. Any benzenoid involving n hexagons can be embedded in a coronenoid of size at most $k(n) = \left\lfloor \frac{n+1}{2} + 1 \right\rfloor$.

So if we reason in terms of hexagon graph, obtaining all the benzenoids with n hexagons is equivalent to find all the connected sub-graphs of the hexagon graph of coronenoid of size $k(n)$. The model we propose relies on this property.

So, given an integer n, we model the BGP problem where $\mathcal{P}$ is reduced to "having n hexagons" as a CSP instance $I = (X, D, C)$. First, we consider a graph variable x_G which represents the possible hexagon graph of the built benzenoid. Its domain is the set of all the sub-graphs between the empty graph and the hexagon graph of coronenoid of size $k(n)$ (see Figure 4(a)). We also exploit a set of n_c Boolean variables $\{x_1, \ldots, x_{n_c}\}$ where n_c is the number of hexagons of coronenoid of size $k(n)$. The variable x_i is set to 1 if the ith hexagon of coronenoid of size $k(n)$ is used in the hexagon graph depicted by x_G, 0 otherwise. For sake of simplicity, hexagons are numbered from top to bottom and from left to right like in Figure 2. Likewise, we consider a set of m_c Boolean variables $\{y_1, \ldots, y_{m_c}\}$ where m_c is the number of edges of the hexagon graph of coronenoid of size $k(n)$. The variable y_j is set to 1 if the jth edge of the hexagon graph of coronenoid of size $k(n)$ is used in the hexagon graph depicted by x_G, 0 otherwise. We must emphasize that the set of x_i and y_i variables and the channeling constraints maintaining the consistency between their values and the value of x_G are automatically generated by Choco Solver through the call of a predefined method.

Finally, we model the following properties by constraints:

- *Link between the graph variable x_G and the variables x_i* As mentioned above, the variable x_i specifies if the ith hexagon of coronenoid of size $k(n)$ is used in the graph represented by x_G. So we must ensure that their respective values are consistent each others. For this aim in view, we consider a channeling constraint per variable x_i which involves x_i and x_G and imposes that $x_i = 1 \iff x_G$ *contains the vertex* i.
- *Link between the graph variable x_G and the variables y_j* Like previously, we consider a channeling constraint per variable y_j which involves y_j and x_G and imposes that $y_j = 1 \iff x_G$ *contains the edge* j.
- *x_G is an induced sub-graph of the coronenoid hexagon graph.* Any value of x_G is not necessarily a valid hexagon graph. For example, in Figure 2(b), removing only edge $\{1, 2\}$ does not produce a valid hexagon graph. To ensure that the hexagon graph is valid, we must add a constraint for every triplet $(h_{j_1}, h_{j_2}, h_{j_3})$ of hexagons which are pairwise adjacent in the coronenoid hexagon graph. This constraint imposes that if two of these edges exists, then the third one exists too. This can be achieved by posting a set of ternary clauses of the form $\{\neg y_{j_1} \vee \neg y_{j_2} \vee y_{j_3}, y_{j_1} \vee \neg y_{j_2} \vee \neg y_{j_3}, \neg y_{j_1} \vee y_{j_2} \vee \neg y_{j_3}\}$ for each possible triple of pairwise adjacent hexagons.
- *Benzenoids have n hexagons* It can be easily done by using a sum global constraint involving all the variables x_i: $\sum_{i \in \{1,\ldots,n_c\}} x_i = n$.
- *Benzenoids correspond to connected graphs* Variable graphs come with particular constraints. Among them, we consider the `connected` constraint which applies on the variable x_G ensures that only connected graphs are allowed values for x_G.

– *Six hexagons forming a cycle generate a hexagon* When six hexagons form a cycle, the face contained in the interior of the cycle is not a hole but a hexagon. For instance, if we consider the cycle forms by the hexagons 1, 2, 5, 7, 6 and 3 of coronene (see Figure 2), we have necessarily a hexagon in the middle of the crown, namely the hexagon 4. To ensure this property, we add a set of constraints which specify that G cannot have a hole whose size is exactly one hexagon. For each hexagon u, we consider the set $N(u)$ of the neighbors of u in the hexagon graph. Then, for each vertex u having 6 neighbors, we add a constraint between x_u and the variables corresponding to its neighbors which imposes: $\sum_{v \in N(u)} x_v = 6 \Rightarrow x_u = 1$.

This model allows us to enumerate all the benzenoids having n hexagons, possibly with holes. However, some benzenoids may be generated multiple times due to the existence of symmetries. So we add several additional constraints in order to break as many symmetries as possible:

– Two constraints which specify that G must have at least one vertex respectively on the top-border and the left-border in order to avoid the symmetries by translation. So, we have to create a constraint which specifies that the sum of the binary variables associated to the top border (resp. left border) is strictly positive.
– A set of constraints which specify that G must be the only representative of its class of symmetry by axis and rotation. There are up to twelve symmetric solutions : six 60 degrees rotation symmetries combined with a possible axis symmetry. Symmetries are broken thanks to the compact `lex-lead` constraint described in [11]. For each of the twelve symmetries, it requires n_c new Boolean variables (each associated with a x_i Boolean variable representing a hexagon) and a total of $3n_c$ ternary clauses.

This model can be easily implemented with the open-source Java library *Choco* [15]. Indeed, Choco natively proposes graph variables and the more usual graph-related constraints (notably `connected` constraint).

4.2 How to Specialize the Model

The first advantages of our approach is that it is able to generate all the benzenoids, including those with holes unlike the method described in the previous section. Moreover, using constraint programming makes it easier the addition of most of structural properties wished by the chemists. Indeed, starting from the general model, for each new desired property, we simply have to model it by posting new constraints and eventually by adding new variables.

For example, let us consider that chemists are interested by benzenoids whose structure is a path of hexagons. Such benzenoid structures can easily generated by exploiting the general model I and adding the graph constraint `path` on x_G. Now, if chemists are more interested by *catacondensed benzenoids*, that is benzenoids whose structure is a tree, we can just add the graph constraint `tree`

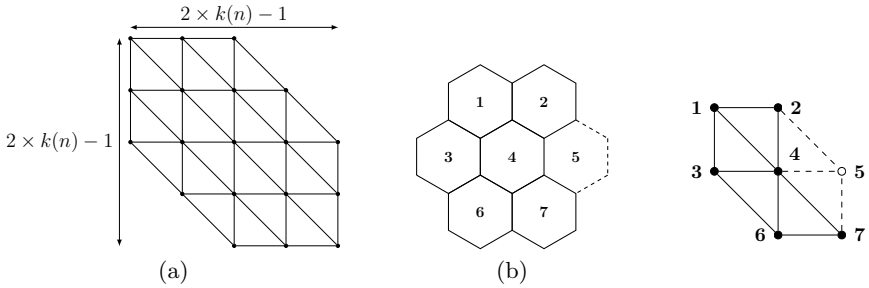


Fig. 4. Upper bound of the domain of the graph variable (a), rectangle benzenoid (in solid line) of dimension 3×2 embedded in coronenoid of size 2 (b) and its related hexagon graph (c).

on x_G to the general model I. Figure 5 shows nine (among twelve possible) examples of 5-hexagon benzenoids obtained by just adding the `tree` constraint of Choco on x_G.

Of course, depending on the desired property the model may be more complex. Especially, it may require to add new variables or the property cannot be directly expressed by a single existing constraint. In next subsections, we give such examples.

4.3 Generating Rectangle Benzenoids

In this part, we present how we can model the property *"all the built benzenoids have a rectangle shape"*, in addition to the property *"having n hexagons"*, and add it to the model we describe previously. For instance, Figure 6(i) shows a rectangle benzenoid with the dimensions 3×3.

First, remember that the general model described in the previous part takes in input the number n of hexagons, and embeds any generated benzenoid in a coronenoid of size $k(n)$. We can easily see that the largest rectangle benzenoid which can be embedded in a coronenoid of size $k(n)$ has a width w_{max} equal to $k(n)$ and a height h_{max} equal to $2 \times k(n) - 1$ (i.e. the diameter of coronenoid of size $k(n)$). Figure 4 shows the rectangle benzenoid of dimensions 2×3 embedded in coronenoid of size 2 (b) and its hexagon graph (c).

Then, starting from model I, we must add new variables to model the desired property. Namely, we add two integer variables x_w and x_h whose domain is respectively $\{1, ..., w_{max}\}$ and $\{1, ..., h_{max}\}$. These variables represent respectively the number of columns and lines of the built benzenoid. In addition, we denote L_i (resp. C_i) the set of variables x_i which appear in the ith line (resp. ith column) in the coronenoid of order $k(n)$. We assume that lines (resp. columns) are numbered from top to bottom (resp. from left to right). For example, if we consider the hexagon graph of the rectangle benzenoid of dimensions 3×2

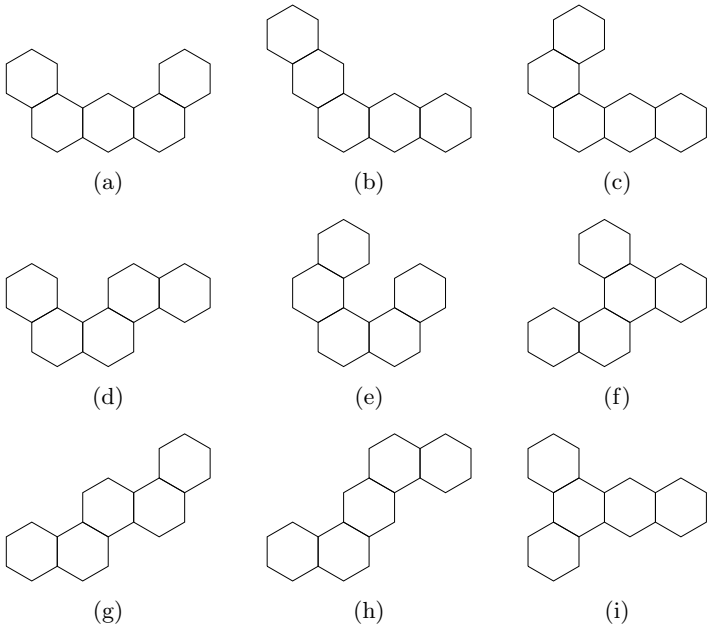


Fig. 5. Catacondensed benzenoids with 5 hexagons.

described in Figure 4(b), we have the following sets:

$$\begin{cases} L_1 = \{x_1, x_2\} \\ L_2 = \{x_3, x_4, x_5\} \\ L_3 = \{x_6, x_7\} \\ C_1 = \{x_1, x_3, x_6\} \\ C_2 = \{x_2, x_4, x_7\} \end{cases}$$

Now we add several constraints to the general model in order to model the following properties:

- *The hexagons of each line are positioned contiguously* We want to avoid to have a Boolean variable equal to 0 between two Boolean variables equal to 1. For the ith line, this can be modeled by imposing an arithmetic constraint $x_{i_1} \geq x_{i_2} \ldots \geq x_{i_{wmax}}$ if $L_i = \{x_{i_1}, x_{i_2}, \ldots, x_{i_{wmax}}\}$. We can also use instead a global constraint `ordered` applied on the variables of L_i with operator $\geq$.
- *The hexagons of each column are positioned contiguously* We proceed as for the lines by considering C_i instead of L_i.
- *Lines have a consistent size* Each line must be empty or have a size equal to the current width of the rectangle. The size of a line can be defined as the number of hexagons it contains since we know that all the hexagons are contiguous. For the ith line, we add a constraint linking x_w to all the variables in L_i and imposing $\sum_{x_{i_j} \in L_i} x_{i_j} = 0 \vee \sum_{x_{i_j} \in L_i} x_{i_j} = x_w$. As such a

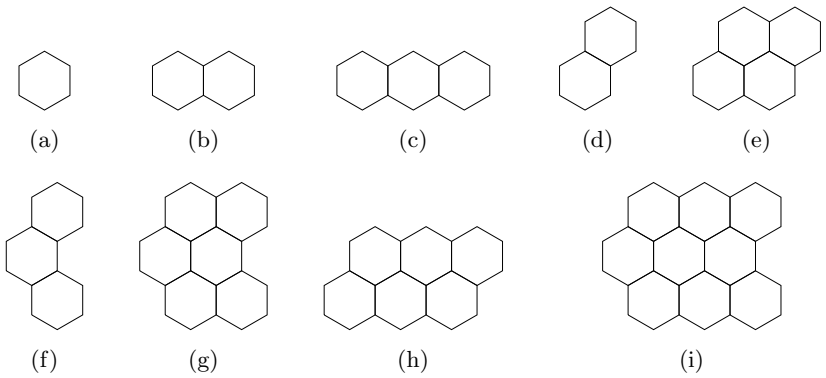


Fig. 6. Rectangle benzenoids generated with $w = h = 3$.

constraint is added for each line, we are sure that all the lines have the same width.

– *Columns have a consistent size* We proceed as for the lines by considering C_i instead of L_i and x_h instead of x_w.

Figure 6 shows the 9 rectangle benzenoids generated with the parameters $w = h = 3$.

In this extended model, we can note that some variables are now useless. Indeed, only the leftmost hexagons of coronenoid of order $k(n)$ covered by the rectangle of dimensions $w_{max} \times h_{max}$ are required. The other hexagons will only lead to produce symmetrical structures. So, we can refine our model by removing useless variables. Likewise, we can filter the domain of x_G in order that GUB is the hexagon graph of the rectangle $w_{max} \times h_{max}$ by posting the adequate unary constraint.

This extension of our general model is given as a simple illustration of our approach. Of course, we can easily generate benzenoids having a rectangle shape with a bespoke algorithm. What is interesting in our approach is its flexibility. For instance, if some chemists are interested by identifying the rectangular benzenoids which has a given Clar number, we have only to model the property "having a given Clar number" by adding some variables and/or constraints to be able to find the wished benzenoids. The Clar number of a benzenoid is the maximum number of non-adjacent hexagons (i.e. hexagons which have no bond in common) which admit three double bonds [9]. Unfortunately, due to page limit, we cannot detail the corresponding extended model.

4.4 Generating Coronoids

Chemists refer to benzenoids with at least one hole as *coronoids* (not to be confused with coronenoids). These molecules are promising model structures of graphene with well-defined holes [12,4,13]. Their enumeration and generation gave rise to several studies (e.g., [6] which enumerates 2-hole coronoids and

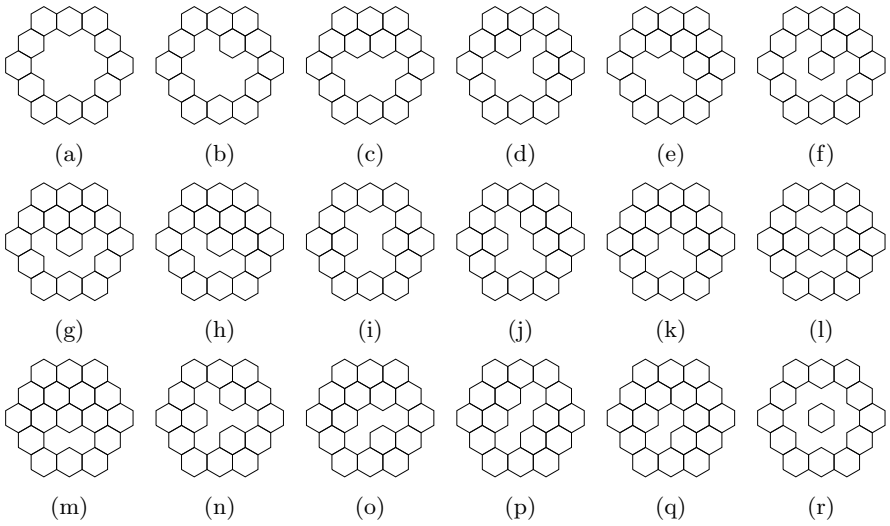


Fig. 7. All the ways of digging holes in the coronenoid of size 3. The last one (r) is not a valid coronoid: the hole in the coronenoid is cyclic and has disconnected the "coronoid" into two benzenoids. Hence, the constraint that x_C must be connected.

generates the smallest 18 and 19-hexagon 3-hole coronoids). The methods for generating them are quite inefficient or too specific. The first kind of approach tries to build specific kinds of coronoids by considering cycles of hexagons and try all possible ways to add hexagons around. The second kind of approach consists of generating all the benzenoids with n hexagons and then detects the ones with holes. Another possible approach consists in generating benzenoids without holes (e.g. with the method of Brinkmann et al. [5]) and then digging holes in the obtained benzenoids. However we can note that the two latter approaches can quickly become too time-consuming with respect to an approach which would directly generate coronoids. Indeed, if the number of hexagons is increased by one, the number of benzenoids is multiplied by approximately 5 (as well as the time to generate them [5]), whereas the time to generate the coronoids with the direct approach we describe below appears to be only twice longer (see Table 1). So, in this part, we present how we can model the property *"all the built benzenoids have a hole and are contained in a benzenoid with n hexagons"*. This allows to generate easily all kinds of coronoids with any number of holes.

Any coronoid can be seen as a benzenoid that has lost several contiguous hexagons (which created holes). So, the vertices of the benzenoid can be split into the vertices belonging to a coronoid and the vertices forming holes. To model this problem, we consider our general model. First, we define two new graph variables x_C, which represents an underlying coronoid of x_G, and x_H the holes to dig in x_G to form x_C. x_C and x_H have the same domain as x_G. So, we can have all the possible coronoid x_C by generating all the pairs (x_G, x_H). There

Table 1. Number of coronoids obtained by digging holes from all the benzenoids with n hexagons.

n	#coronoids	time (s)	#benzenoids without holes
10	1	43	30,086
11	4	114	141,229
12	38	262	669,584
13	239	533	3,198,256
14	1,510	1,076	15,367,577

can be several values of x_H for a value of x_G, as illustrated in Figure 7. Then we consider two sets of n_c Boolean variables $\{x_1^C, \ldots, x_{n_c}^C\}$ and $\{x_1^H, \ldots, x_{n_c}^H\}$ (with n_c the number of hexagons of coronenoid of size $k(n)$). Like x_i for x_G, the variable x_i^C (resp. x_i^H) is set to 1 if the ith hexagon of coronenoid of size $k(n)$ is used in the graph depicted by x_C (resp. x_H), 0 otherwise. Likewise, we define the set of m_c Boolean variables $\{y_1^H, \ldots, y_{m_c}^H\}$ (with m_c the number of edges in the hexagon graph of coronenoid of size $k(n)$). The variable y_j^H is set to 1 if the jth edge of coronenoid of size $k(n)$ is used in the graph depicted by x_H, 0 otherwise.

Finally, we add the following constraints ensuring that variables x_H and x_C have the right properties:

- x_H *is a sub-graph of* x_G This is enforced thanks to the `subgraph` constraint of Choco applied on variables x_H and x_G.
- *Only fully surrounded vertices of* x_G *can be vertices of* x_H For all vertices of x_H if the degree of a vertex in x_H is strictly positive then the degree of the same vertex in x_G is 6. Indeed, only the vertices/hexagons in x_G surrounded by six hexagons can belong to a hole. This constraint is enforced thanks to clauses on the y_i and y_i^H Boolean variables.
- *A single hexagon does not form a hole* Each vertex/hexagon of x_H must have a degree strictly greater than 0. This constraint eliminates holes that would be a sole hexagon and allows multiple holes. We simply use the `minDegrees` graph constraint of Choco applied on x_H.
- x_H *involves at least two hexagons.* We post the constraint $\sum_{i \in \{1,\ldots,n_c\}} x_i^H > 1$.
- x_C *and* x_H *form a partition of* x_G *w.r.t. hexagons* For all vertices of x_G, a vertex is in x_C iff this vertex is in x_G and not in x_H. This ensures that any vertex of x_G is either in x_C or in x_H. With this aim in view, we add a clause $x_i^C \leftrightarrow (x_i \wedge \overline{x_i^H})$ on x_i, x_i^C and x_i^H for any $i \in \{1, \ldots, n_c\}$.
- x_C *is connected* If x_C is not connected, we may obtain two benzenoids instead of one (see Figure 7(r) for instance). Again, this can be achieved by exploiting the graph constraint `connected` applied on x_C.

For example, Figure 8(a) shows how the coronoid of Figure 9(a) can be embedded in the coronenoid of size 3. Then, we depict in dashed line the value of x_G, x_C and x_H respectively in Figures 8(b)-(d).

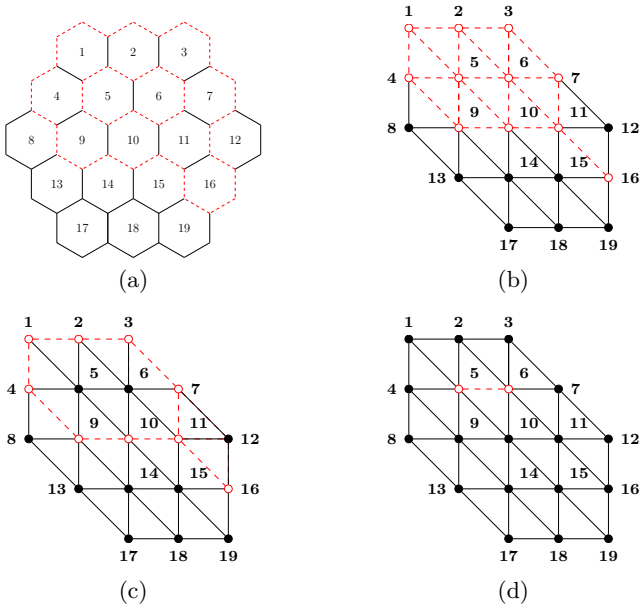


Fig. 8. The coronoid of Figure 9(a) embedded in the coronenoid of size 3 (a),the corresponding value of x_G (b), x_C (c) and x_H (d).

Table 1 shows the results of the experiments we ran on a 3.4 GHz Intel Core i7 iMac with a 12 Gb RAM. We implemented our CSP model in Java 8 with Choco Solver 4.0.4 using the choco-graph 4.2.3 module. We did not specify any search strategy or heuristic, so the default ones were used by the search engine. We generated the coronoids by digging holes in different sizes of benzenoids. Among all the benzenoids with n hexagons, we show the number of coronoids we can produce by removing hexagons. For example, the only coronoid produced from the 10-hexagon benzenoids is the 8-hexagon coronoid of Figure 3(a). Figure 9 lists the four coronoids for $n = 11$. To show how rare coronoids are, Table 1 also provides the number of benzenoids without holes [5]. Of course, thanks to the model we propose for coronoid generation, we do not consider all these benzenoids. Indeed, they are not generated by Choco Solver because it filters out benzenoids that cannot have holes through constraint propagation.

4.5 Generating Symmetric Benzenoids

Benzenoids are also classified by chemists by their classes of internal symmetries (symmetries that let a benzenoid invariant by rotation and/or mirroring). We can generate such classes of benzenoids by adding the constraints for enforcing internal symmetries. When searching for all the possible benzenoids embeddable in a coronenoid of size 3, we obtain the 11,578 benzenoids (with at most 19

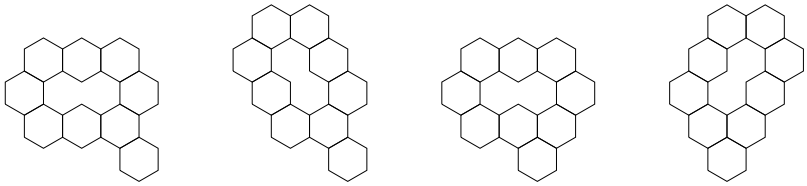


Fig. 9. Coronoids for $n = 11$.

hexagons) in 36 minutes. Enforcing invariance by 60 degree rotation (to obtain the 4 corresponding benzenoids), by 120 degree rotation (16 benzenoids) and 180 degree rotation (70 benzenoids) takes less than one second for each task. This strengthens the idea that constraint propagation in nowadays solvers is efficient enough to allow these theoretical chemistry problems to be modeled and solved with CP without having to define and use bespoke methods. Moreover, interestingly, note that this extension of our general model may be combined with any extension described above.

5 Conclusions and Perspectives

In this paper, we addressed the problem of generating benzenoid structures, which is an interesting and important problem in theoretical chemistry. In this context, we presented an approach using constraint programming able to generate benzenoids which satisfy a certain amount of properties. Its main advantage w.r.t. existing methods in the literature lies in its flexibility. Indeed, from a general model, we can express additional properties by simply adding variables and/or constraints while existing bespoke methods rely on more rigid and complex notions and cannot be adapted without requiring heavy tasks. Moreover, our approach turns to be more general, making it possible to generate benzenoids with holes for instance.

Chemists are interested in generating benzenoids with particular shapes (e.g. rectangle or rhombus shapes [27]). We have already dealt with the rectangle shapes in this paper. So a natural extension of this work relies in taking into account other specific properties related to the needs of chemists. Another step consists in studying the limit of our approach both in terms of properties we can express and our ability to generate benzenoids of large size. Furthermore, this paper shows how, once again, constraint programming can be useful to tackle and solve problems related to theoretical chemistry [19,28,26,16,17]. In particular, many questions about benzenoids can be modeled as decision or optimization problems under constraints (e.g. computing their aromaticity or finding the closest structure to a Kekulé structure) and can correspond to difficult tasks (e.g. computing the Clar number is NP-hard [7]). It could be of interest for both communities to study them.

References

1. Allamandola, L.J., Hudgins, D.M., Sandford, S.A.: Modeling the Unidentified Infrared Emission with Combinations of Polycyclic Aromatic Hydrocarbons. The Astrophysical Journal **511**(2), L115–L119 (1999). https://doi.org/10.1086/311843
2. Aumaitre, C., Morin, J.F.: Polycyclic Aromatic Hydrocarbons as Potential Building Blocks for Organic Solar Cells. The Chemical Record **19**(6), 1142–1154 (2019). https://doi.org/10.1002/tcr.201900016, https://onlinelibrary.wiley.com/doi/abs/10.1002/tcr.201900016
3. Bauschlicher, Jr., C.W., Peeters, E., Allamandola, L.J.: The Infrared Spectra of Very Large, Compact, Highly Symmetric, Polycyclic Aromatic Hydrocarbons (PAHs). The Astrophysical Journal **678**(1), 316–327 (2008). https://doi.org/10.1086/533424
4. Beser, U., Kastler, M., Maghsoumi, A., Wagner, M., Castiglioni, C., Tommasini, M., Narita, A., Feng, X., Müllen, K.: A C216-Nanographene Molecule with Defined Cavity as Extended Coronoid. Journal of the American Chemical Society **138**(13), 4322–4325 (2016). https://doi.org/10.1021/jacs.6b01181
5. Brinkmann, G., Caporossi, G., Hansen, P.: A Constructive Enumeration of Fusenes and Benzenoids. Journal of Algorithms **45(2)** (2002)
6. Brunvoll, J., Cyvin, R.N., Cyvin, S.J.: Enumeration and Classification of Double Coronoid Hydrocarbons – Appendix: Triple Coronoids. Croatica Chemica Acta **63**(4), 585–601 (1990)
7. Bérczi-Kovács, E., Bernáth, A.: The complexity of the Clar number problem and an exact algorithm. J Math Chem **56**, 597–605 (2018). https://doi.org/10.1007/s10910-017-0799-8
8. Caporossi, G., Hansen, P.: Enumeration of polyhex hydrocarbons to h = 21. Journal of Chemical Information and Computer Sciences **38**(4), 610–619 (1998). https://doi.org/10.1021/ci970116n, https://doi.org/10.1021/ci970116n
9. Clar, E.: The Aromatic Sextet. Wiley (1972)
10. Cyvin, J., Brunvoll, J., Cyvin, B.N.: Search for Concealed Non-Kekuliian Benzenoids and Coronoids. J. Chem. Inf. Comput. Sci. **29**(4), 237 (1989)
11. Devriendt, J., Bogaerts, B., Bruynooghe, M., Denecker, M.: Improved static symmetry breaking for sat. In: Creignou, N., Le Berre, D. (eds.) Theory and Applications of Satisfiability Testing – SAT 2016. pp. 104–122 (2016)
12. Di Giovannantonio, M., Yao, X., Eimre, K., Urgel, J.I., Ruffieux, P., Pignedoli, C.A., Müllen, K., Fasel, R., Narita, A.: Large-Cavity Coronoids with Different Inner and Outer Edge Structures. Journal of the American Chemical Society **142**(28), 12046–12050 (2020). https://doi.org/10.1021/jacs.0c05268
13. Dias, J.R.: Structure and Electronic Characteristics of Coronoid Polycyclic Aromatic Hydrocarbons as Potential Models of Graphite Layers with Hole Defects. The Journal of Physical Chemistry A **112**(47), 12281–12292 (2008). https://doi.org/10.1021/jp806987f
14. Draine, B.T.: Astronomical Models of PAHs and Dust. EAS Publications Series **46**, 29–42 (2011). https://doi.org/10.1051/eas/1146003
15. Fages, J.G., Lorca, X., Prud'homme, C.: Choco solver user guide documentation. https://choco-solver.readthedocs.io/en/latest/
16. Ismail, I., Stuttaford-Fowler, H.B.V.A., Ochan Ashok, C., Robertson, C., Habershon, S.: Automatic Proposal of Multistep Reaction Mechanisms using a Graph-Driven Search. The Journal of Physical Chemistry A **123**(15), 3407–3417 (2019). https://doi.org/10.1021/acs.jpca.9b01014

17. Kim, Y., Kim, J.W., Kim, Z., Kim, W.Y.: Efficient prediction of reaction paths through molecular graph and reaction network analysis. Chemical Science **9**(4), 825–835 (2018). https://doi.org/10.1039/C7SC03628K
18. Luch, A.: The Carcinogenic Effects of Polycyclic Aromatic Hydrocarbons. Imperial College Press, London (2005), https://www.worldscientific.com/worldscibooks/10.1142/p306
19. Mann, M., Nahar, F., Schnorr, N., Backofen, R., Stadler, P.F., Flamm, C.: Atom mapping with constraint programming. Algorithms for Molecular Biology **9**(1), 23 (2014). https://doi.org/10.1186/s13015-014-0023-3
20. Mishra, S., Beyer, D., Eimre, K., Kezilebieke, S., Berger, R., Gröning, O., Pignedoli, C.A., Müllen, K., Liljeroth, P., Ruffieux, P., Feng, X., Fasel, R.: Topological frustration induces unconventional magnetism in a nanographene. Nature Nanotechnology **15**(1), 22–28 (2020). https://doi.org/10.1038/s41565-019-0577-9
21. Mishra, S., Beyer, D., Eimre, K., Liu, J., Berger, R., Gröning, O., Pignedoli, C.A., Müllen, K., Fasel, R., Feng, X., Ruffieux, P.: Synthesis and Characterization of π-Extended Triangulene. Journal of the American Chemical Society **141**(27), 10621–10625 (2019). https://doi.org/10.1021/jacs.9b05319
22. Narita, A., Wang, X.Y., Feng, X., Müllen, K.: New advances in nanographene chemistry. Chemical Society Reviews **44**(18), 6616–6643 (2015). https://doi.org/10.1039/C5CS00183H
23. Ricca, A., Bauschlicher, C.W., Boersma, C., Tielens, A.G.G.M., Allamandola, L.J.: The Infrared spectroscopy of compact polycyclic aromatic hydrocarbons containing up to 384 carbons. The Astrophysical Journal **754**(1), 75 (2012). https://doi.org/10.1088/0004-637X/754/1/75
24. Ricca, A., Roser, J.E., Peeters, E., Boersma, C.: Polycyclic Aromatic Hydrocarbons with Armchair Edges: Potential Emitters in Class B Sources. The Astrophysical Journal **882**(1), 56 (2019). https://doi.org/10.3847/1538-4357/ab3124
25. Rieger, R., Müllen, K.: Forever young: Polycyclic aromatic hydrocarbons as model cases for structural and optical studies. Journal of Physical Organic Chemistry **23**(4), 315–325 (2010). https://doi.org/10.1002/poc.1644
26. Simoncini, D., Allouche, D., de Givry, S., Delmas, C., Barbe, S., Schiex, T.: Guaranteed Discrete Energy Optimization on Large Protein Design Problems. Journal of Chemical Theory and Computation **11**(12), 5980–5989 (2015). https://doi.org/10.1021/acs.jctc.5b00594
27. Trinquier, G., Malrieu, J.P.: Predicting the Open-Shell Character of Polycyclic Hydrocarbons in Terms of Clar Sextets. The Journal of Physical Chemistry A **122**(4), 1088–1103 (2018). https://doi.org/10.1021/acs.jpca.7b11095
28. Wu, C.W.: Modelling Chemical Reactions Using Constraint Programming and Molecular Graphs. In: Principles and Practice of Constraint Programming. pp. 808–808 (2004)
29. Wu, J., Pisula, W., Müllen, K.: Graphenes as Potential Material for Electronics. Chemical Reviews **107**(3), 718–747 (2007). https://doi.org/10.1021/cr068010r

Actes JFPC 2021

Prise en compte de motifs et génération de structures de benzénoïdes*

Yannick Carissan[1] **Denis Hagebaum-Reignier**[1]
Nicolas Prcovic[2] **Cyril Terrioux**[2] **Adrien Varet**[2]
[1] Aix Marseille Univ, CNRS, Centrale Marseille, ISM2, Marseille, France
[2] Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{prénom.nom}@univ-amu.fr

Résumé

Les benzénoïdes sont une sous-famille d'hydrocarbures (molécules composées uniquement d'atomes d'hydrogène et de carbone) dont les atomes de carbone forment des hexagones. Ces molécules ont fait l'objet de nombreuses études en chimie théorique et peuvent posséder différentes propriétés physico-chimiques (résistance mécanique, conductivité électronique, ...) desquelles découlent de nombreuses applications concrètes. Ces propriétés peuvent notamment reposer sur l'existence ou l'absence de fragments de la molécule correspondant à un motif donné (certains motifs imposent la nature de certaines liaisons, ce qui a un impact sur la structure électronique totale). Générer des structures de benzénoïdes tout en maîtrisant la présence ou non d'un certain motif constitue donc une problématique importante en chimie théorique.

Dans cet article, nous montrons comment la programmation par contraintes peut aider les chimistes à répondre à différentes questions autour de cette problématique. Pour ce faire, nous proposons différentes modélisations dont une basée sur une variante du problème d'isomorphisme de sous-graphes et nous générons les structures souhaitées à l'aide du solveur Choco.

Abstract

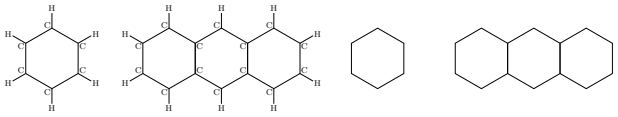
Benzenoids are a subfamily of hydrocarbons (molecules that are only made of hydrogen and carbon atoms) whose carbon atoms form hexagons. These molecules are widely studied in theoretical chemistry and can have various physicochemical properties (mechanical resistance, electronic conductivity, ...) from which a lot of concrete applications are derived. These properties can rely on the existence or absence of fragments of the molecule corresponding to a given pattern (some patterns impose the nature of certain bonds, which has an impact on the whole electronic structure). Generating benzenoid structures while controlling the presence or absence of a certain pattern is therefore an important problem in theoretical chemistry.

In this paper, we show how constraint programming can help chemists to answer different questions around this problem. To do so, we propose different models including one based on a variant of the subgraph isomorphism problem and we generate the desired structures using the Choco solver.

*Ce travail est soutenu par l'Agence Nationale de la Recherche dans le cadre du projet DEMOGRAPH (ANR-16-CE40-0028).

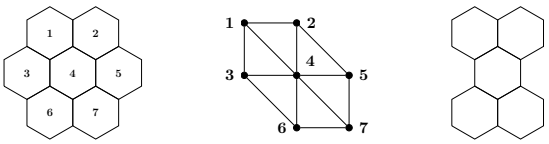
1 Introduction

Les *hydrocarbures aromatiques polycycliques (HAP)* sont des hydrocarbures dont les atomes de carbone forment des cycles de différentes tailles (de taille 6 dans le cas des *benzénoïdes*). Ils ont été très étudiés dans divers domaines (nanoélectronique moléculaire, synthèse organique, chimie interstellaire, ...) en raison de leur stabilité énergétique, de leurs structures moléculaires ou de leurs spectres optiques. Ils présentent une grande variété de propriétés physico-chimiques en fonction de leur taille et de leur structure. Par exemple, ils peuvent allier une forte résistance mécanique à une conductivité électronique élevée. Ces propriétés peuvent notamment reposer sur l'existence ou l'absence de fragments de la molécule correspondant à un motif donné. Certains motifs imposent la nature de certaines liaisons, ce qui impacte la structure électronique dans son ensemble. Par exemple, le pérylène (voir figure 2(c)) peut être vu comme deux triangles de trois cycles fusionnés se chevauchant avec pour conséquence que les liaisons centrales ont un caractère essentiellement simple. Ces dernières années, les motifs et les propriétés qui en découlent se retrouvent au cœur de nombreux travaux en chimie théorique [12, 4, 9]. Ainsi, être capable de générer des structures de benzénoïdes tout en maîtrisant



(a) (b) (c) (d)

FIGURE 1 – Exemples de benzénoïdes : le benzène (a) et l'anthracène (b) accompagnés de leur représentation graphique (c) et (d).



(a) (b) (c)

FIGURE 2 – Le coronène (a), son graphe d'hexagones (b) et le pérylène (c).

la présence ou non d'un certain motif constitue donc une problématique importante en chimie théorique.

Dans la littérature, des approches ad-hoc ont été proposées pour générer des structures quelconques ou satisfaisant des propriétés particulières [1]. Elles s'avèrent très efficaces en pratique mais sont difficiles à adapter aux besoins des chimistes et ne considèrent que des propriétés portant sur l'ensemble de la molécule. Dans [2], nous avons proposé une nouvelle approche basée sur la PPC qui est plus flexible tout en étant compétitive. Cette approche ne traitant initialement que de propriétés globales, nous décrivons, dans ce papier, comment y intégrer la notion de motif. Plusieurs modélisations étant envisageables, nous les étudions sur la base des différentes questions se posant autour des motifs avant de les comparer expérimentalement.

Cet article est organisé comme suit. La section 2 introduit les notions nécessaires à la compréhension du papier. Puis, dans la section 3, nous rappelons comment générer des structures de benzénoïdes, notamment à l'aide de la PPC. Ensuite, nous formalisons la problématique qui nous intéresse dans la section 4 et abordons différentes questions pouvant en découler dans les sections 5 à 7. Enfin, nous comparons expérimentalement certaines modélisations, dans la section 8, avant de conclure dans la section 9.

2 Préliminaires

2.1 Chimie théorique

Le benzène (représenté à la figure 1(a)) est une molécule constituée de 6 atomes de carbone et de 6 atomes d'hydrogène. Ses atomes de carbone forment un hexagone (appelé *cycle benzénique*) et vont chacun être reliés à un atome d'hydrogène. Les *benzénoïdes* sont une sous-famille des *HAP* qui regroupe toutes les molécules obtenues en fusionnant des cycles benzéniques. La figure 1(b) présente, par exemple, l'anthracène qui est constitué de trois cycles benzéniques.

Les atomes établissent entre eux des liaisons qui peuvent notamment être simples ou doubles selon le nombre d'électrons impliqués dans la liaison. Dans un benzénoïde, chaque atome de carbone est relié soit à deux atomes de carbone et un atome d'hydrogène, soit à trois atomes de carbone. Dans la suite du propos, les atomes d'hydrogène ne jouent aucun rôle (et leur présence peut être déduite si nécessaire). Aussi, ils peuvent être omis dans la représentation que nous allons considérer. Ainsi, un benzénoïde peut être représenté comme un graphe non orienté $B = (V, E)$ dans lequel chaque sommet de V correspond à un atome de carbone et chaque arête de E traduit l'existence d'une liaison entre les deux atomes de carbone correspondant. Ce graphe est connexe, biparti et planaire. Les figures 1(c) et (d) présentent les graphes correspondant au benzène et à l'anthracène.

Par ailleurs, les benzénoïdes pouvant se définir comme une combinaison de cycles benzéniques fusionnés, nous considérons, pour chaque benzénoïde B, un deuxième graphe appelé *graphe d'hexagones*. Ce graphe $B_h = (V_h, E_h)$ est un graphe non orienté dont chaque sommet correspond à un hexagone (c'est-à-dire à un cycle benzénique) de B et tel que deux sommets sont reliés par une arête si les hexagones correspondants partagent une arête dans le graphe B. La figure 2 présente le graphe correspondant au coronène ainsi que son graphe d'hexagones.

2.2 Programmation par contraintes

Une instance *CSP (Constraint Satisfaction Problem)* P se définit comme un triplet (X, D, C). $X = \{x_1, \ldots, x_n\}$ 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. Résoudre une instance CSP $P = (X, D, C)$ revient à essayer de trouver une affectation de toutes les variables de X qui ne viole aucune contrainte de C. Il s'agit d'un problème NP-difficile.

De nombreux outils existent pour modéliser et résoudre efficacement des instances CSP. Par la suite, nous optons pour l'outil *Choco* [5]. Ce choix se justifie par la possibilité offerte par Choco de pouvoir définir des variables de graphe et de disposer de contraintes prédéfinies adaptées à ces variables (comme par exemple des contraintes de connexité ou de cyclicité). Ces va-

riables ont comme domaine des ensembles de graphes définis par une borne inférieure (un sous-graphe, noté GLB) et une borne supérieure (un super-graphe, noté GUB). De plus, Choco permet une modélisation assez simple et rapide d'un grand nombre de problèmes et le solveur est paramétrable à la guise de l'utilisateur.

3 Génération de benzénoïdes

La génération de benzénoïdes ayant certaines propriétés structurelles (par exemple, ayant un nombre donné d'hexagones ou ayant une structure particulière du point de vue graphique) est un problème intéressant et important en chimie théorique [3, 7, 8, 10]. Ce problème constitue une étape préliminaire à l'étude de leurs propriétés chimiques. Il peut se définir formellement ainsi : Étant donné un ensemble de propriétés structurelles $\mathcal{P}$, générer toutes les structures de benzénoïdes satisfaisant les propriétés de $\mathcal{P}$. Ces propriétés peuvent porter sur le nombre de carbones ou d'hexagones ou sur des formes particulières du graphe d'hexagones (arbre, rectangle, présence de « trous », ...).

Dans la littérature, des méthodes ad-hoc ont été proposées (par exemple [1]). Si elles s'avèrent souvent efficaces en pratique, elles possèdent l'inconvénient d'être difficilement adaptables aux besoins des chimistes. Dans [2], nous avons proposé une modélisation CSP de ce problème et montré combien il était facile de répondre aux souhaits exprimés par les chimistes en spécialisant ce modèle par simple ajout de variables et de contraintes. Par ailleurs, au-delà de sa flexibilité, cette approche se révèle relativement rapide du fait de l'efficacité du solveur Choco.

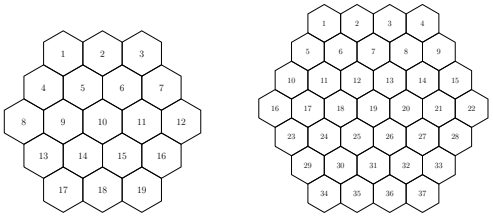
Nous rappelons maintenant la modélisation CSP permettant de générer toutes les structures ayant n hexagones. Elle repose sur la propriété que tout benzénoïde de n hexagones peut être placé dans un coronénoïde de taille au plus $k(n) = \lfloor \frac{n}{2} + 1 \rfloor$. Un coronénoïde de taille k est une molécule de benzène à laquelle on a ajouté successivement $k-1$ couronnes d'hexagones. Le coronène (voir figure 2) est le coronénoïde de taille 2. La figure 3 présente les coronénoïdes de tailles 3 et 4. Par la suite, on note $B_h^{c(k(n))}$ le graphe d'hexagones du coronénoïde $c(k(n))$ de taille $k(n)$, n_c son nombre d'hexagones et m_c celui d'arêtes. Nous numérotons arbitrairement les hexagones de $B_h^{c(k(n))}$ à partir de 1. Tout d'abord, dans notre modèle (noté $\mathcal{M}$), nous considérons une variable de graphe x_G pour représenter le graphe d'hexagones de la structure souhaitée. Son domaine est l'ensemble de tous les sous-graphes entre le graphe vide et $B_h^{c(k(n))}$. L'intérêt d'utiliser des variables de graphe est de pouvoir exprimer certaines notions (comme la connexité) de façon plus aisée et plus compacte. Nous exploitons également un ensemble de n_c variables booléennes $\{x_1, \ldots, x_{n_c}\}$. La variable x_i vaut 1 si le i-ème hexagone de $B_h^{c(k(n))}$ est utilisé dans x_G, 0 sinon. De même, nous considérons un ensemble de m_c variables booléennes $y_{i,j}$. La variable $y_{i,j}$ vaut 1 si l'arête $\{i,j\}$ de $B_h^{c(k(n))}$ est utilisée dans x_G, 0 sinon.

Ensuite, nous modélisons les propriétés suivantes à l'aide de contraintes :

- *Lien entre x_G et x_i (resp. $y_{i,j}$)* : nous utilisons une contrainte `channeling` qui impose $x_i = 1 \iff x_G$ *contient le sommet* i (resp. $y_{i,j} = 1 \iff x_G$ *contient l'arête* $\{i,j\}$).
- *x_G est un sous-graphe induit de $B_h^{c(k(n))}$* : Toute valeur de x_G n'est pas nécessairement un graphe d'hexagones valide. Pour garantir sa validité, il doit correspondre à un sous-graphe de $B_h^{c(k(n))}$ induit par les sommets appartenant à x_G. Ainsi, pour chaque arête $\{i,j\}$ de $B_h^{c(k(n))}$, on ajoute une contrainte $x_i = 1 \wedge x_j = 1 \Leftrightarrow y_{i,j} = 1$. En d'autres termes, l'arête $\{i,j\}$ existe dans x_G si et seulement si les sommets i et j apparaissent dans x_G.
- *La structure a n hexagones* : $\sum_{i \in \{1,\ldots,n_c\}} x_i = n$.
- *Le graphe d'hexagones est connexe* : nous appliquons la contrainte de graphe `connected` sur x_G.
- *Six hexagones formant un cycle génèrent un hexagone (et non un trou)* : Pour chaque hexagone u, nous considérons l'ensemble $N(u)$ de voisins de u dans le graphe d'hexagones. Pour chaque sommet u, on pose la contrainte $\sum_{v \in N(u)} x_v = 6 \Rightarrow x_u = 1$.

Enfin, nous ajoutons plusieurs contraintes pour éviter les redondances. D'abord x_G doit avoir au moins un sommet sur le bord supérieur (resp. gauche) de $B_h^{c(k(n))}$ afin d'éviter les symétries par translation. Nous posons donc une contrainte qui spécifie que la somme des variables binaires x_i associées au bord supérieur (resp. gauche) est strictement positive. Ensuite, il faut s'assurer que le graphe décrit par x_G est le seul représentant de sa classe de symétrie. Il existe jusqu'à douze solutions symétriques : six symétries de rotation de 60 degrés combinées à une éventuelle symétrie axiale. Ces symétries sont cassées grâce à la contrainte `lex-lead`. Pour chacune des douze symétries, il faut ajouter n_c variables booléennes (chacune associée à une variable x_i) et un total de $3.n_c$ clauses ternaires.

Ce modèle peut facilement être mis en œuvre avec le solveur Choco. Il peut également être spécialisé pour prendre en compte les besoins des chimistes en ajoutant des variables et/ou des contraintes. Par exemple, générer des structures ayant une forme arborescente (appelées benzénoïdes *catacondensés*) nécessite simplement d'ajouter au modèle général la contrainte de graphe `tree` appliquée à x_G. D'autres propriétés ont été modélisées afin de générer des structures ayant une



(a) (b)

FIGURE 3 – Les coronénoïdes de taille 3 (a) et 4 (b).

forme rectangulaire, possédant un trou ou symétriques.

4 Prise en compte de motifs

Le modèle $\mathcal{M}$, présenté dans [2] et rappelé dans la section 3, permet d'exprimer le problème de génération de structures de benzénoïdes dans toute sa généralité. Si plusieurs spécialisations de ce modèle ont été proposées ensuite, toutes correspondent à des propriétés structurelles portant sur l'ensemble de la molécule. Ces propriétés pourraient donc être qualifiées de *globales*. Toutefois, dans certains cas de figure, il peut être utile de raisonner en termes de propriétés *locales* que pourraient satisfaire ou non certaines parties (appelées *fragments*) des structures générées.

Les propriétés locales que nous considérons dans cet article peuvent être définies en « dessinant » une forme dont les briques de base sont des hexagones. Ces hexagones peuvent être de trois natures différentes :

(i) Les hexagones *positifs* dont la présence est requise dans la propriété,

(ii) Les hexagones *négatifs* dont l'absence est requise dans la propriété,

(iii) Les hexagones *neutres* dont la présence ou l'absence n'ont pas d'influence sur la propriété.

Si l'utilisation des hexagones positifs est une évidence, on peut se poser la question de l'intérêt des hexagones négatifs ou neutres. Les hexagones négatifs (respectivement neutres) sont utiles, par exemple, pour indiquer qu'il n'y a rien entre deux hexagones positifs (resp. pour garantir un certain écart entre deux hexagones positifs). Afin de représenter les formes souhaitées, nous introduisons la notion de graphe d'hexagones étendu :

Définition 1 (graphe d'hexagones étendu) *Un graphe d'hexagones étendu est un graphe d'hexagones dont les sommets et les arêtes sont étiquetés par les symboles $+$ (pour positif), $-$ (pour négatif) et $\circ$ (pour neutre) ainsi :*

(i) Chaque sommet a pour étiquette la nature de l'hexagone qu'il représente.

(ii) Une arête est étiquetée $-$ si au moins un de ses sommets est étiqueté $-$. Sinon, elle est étiquetée $\circ$ si au moins un de ses sommets est étiqueté $\circ$. Sinon, elle est étiquetée $+$.

Comme pour les hexagones (ou les sommets), les étiquettes associées aux arêtes qualifient le statut que doit avoir l'interaction entre deux hexagones dans la propriété locale que l'on souhaite définir. Formellement une propriété locale peut être définie par un *motif* :

Définition 2 (motif) *Un motif M se définit par la donnée d'un triplet $(M_+, M_-, M_\circ)$ et d'un graphe étendu d'hexagone M_h tels que :*

(i) M_+, M_- et $M_\circ$ désignent respectivement l'ensemble des hexagones positifs, négatifs et neutres,

(ii) ces trois ensembles sont deux à deux disjoints et

(iii) M_h est un graphe connexe portant sur l'ensemble des hexagones de $M_+ \cup M_- \cup M_\circ$.

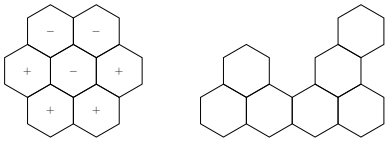
Son ordre k_M est la longueur maximale (exprimée en nombre d'arêtes) des plus court chemins de M_h séparant un hexagone négatif ou neutre d'un hexagone positif.

En d'autres termes, un motif est défini par une collection d'hexagones positifs, négatifs et neutres dont l'agencement est décrit par un graphe d'hexagones étendu. À titre d'exemple, la figure 4(a) présente un motif, appelé *baie profonde* [12], d'ordre 1 composé de quatre hexagones positifs et trois négatifs. Nous définissons enfin la notion d'inclusion de motif :

Définition 3 *Étant donné k un entier positif ou nul, on note B_h^k le graphe d'hexagones étendu représentant le benzénoïde B entouré de k couches d'hexagones négatifs (c'est-à-dire le graphe étendu de B augmenté de tous les hexagones négatifs situés à une distance au plus k d'un hexagone de B). Un* fragment *F^k d'ordre k d'un benzénoïde B est un sous-ensemble d'hexagones de B_h^k dont le graphe d'hexagones étendu est connexe. Il* satisfait *le motif M si $k = k_M$ et s'il existe une bijection qui à chaque hexagone positif de F^k associe un hexagone positif ou neutre de M et à chaque hexagone négatif de F^k associe un hexagone négatif ou neutre de M. Un benzénoïde B* contient *le motif M s'il possède un fragment d'ordre k_M satisfaisant M.*

Considérer B_h ou B_h^k ne change pas la nature du benzénoïde B. B_h^k permet simplement de matérialiser le vide qui l'entoure, ce qui s'avère nécessaire pour certaines propriétés. Par exemple, le benzénoïde de la figure 4(b) satisfait le motif « baie profonde » de la figure 4(a). Pour cela, nous sommes obligés de prendre en compte l'absence d'hexagone en bordure du benzénoïde pour identifier un fragment convenable.

Dans cet article, notre objectif est de générer des structures de benzénoïdes satisfaisant des propriétés locales exprimées à l'aide des motifs introduits ci-dessus. Ces propriétés locales peuvent revêtir différentes formes.



(a) (b)

FIGURE 4 – Le motif « *baie profonde* »(a) et un benzénoïde satisfaisant ce motif (b).

La plus simple consiste à inclure un motif donné. Ensuite, on peut s'intéresser à généraliser l'approche en incluant plusieurs motifs différents ou un certain nombre de fois le même motif. À l'opposé, on peut également désirer exclure un motif. Les sections suivantes traitent de ces différentes questions. Dans tous les cas, l'idée est de générer des structures de benzénoïdes en partant de notre modèle général $\mathcal{M}$. Il s'ensuit qu'il est tout à fait possible de considérer à la fois des propriétés globales et des propriétés locales.

5 Structures incluant un motif

Soit un motif M portant sur n_M hexagones, ces hexagones pouvant être positifs, négatifs ou neutres. On numérote arbitrairement chaque hexagone du motif M de 1 à n_M. Les ensembles M_+, M_- et $M_\circ$ sont définis en conséquence. Dans cette section, nous souhaitons modéliser le problème qui consiste à générer toutes les structures de benzénoïdes ayant n hexagones et incluant le motif M. Nous considérons d'abord toutes les variables et les contraintes du modèle général $\mathcal{M}$ auxquelles nous allons ajouter des variables et des contraintes pour exprimer le fait que le motif doit être présent dans les structures genérées. À ce niveau, nous avons plusieurs possibilités selon le point de vue considéré.

5.1 Première modélisation

Nous partons du modèle $\mathcal{M}$ et donc d'un coronénoïde de taille $k(n)$. Dans cette première modélisation (notée $\mathcal{M}_{i_1}$), nous identifions, en amont, tous les fragments possibles du motif M dans ce coronénoïde. Leur nombre étant en $O(c(k(n)) = O(n^2)$, ce calcul peut être réalisé efficacement à l'aide de rotations, de symétries axiales et de translations. Pour chacun de ces fragments F_i, nous définissons les ensembles F_{i+}, $F_{i\circ}$ et F_{i-} de ses hexagones positifs, neutres et négatifs. Nous associons à chaque fragment F_i une variable booléenne e_i. Puis, nous garantissons que si e_i est vraie alors le fragment F_i est présent dans la structure en cours de construction via la contrainte $e_i = 1 \Rightarrow \bigwedge_{j \in F_{i-}} x_j = 0 \wedge \bigwedge_{j \in F_{i+}} x_j = 1$. Notons que, pour les motifs d'ordre non nul, il n'est pas nécessaire de considérer un coronénoïde de taille supérieure. Le fragment peut être placé en bordure du coronénoïde avec des hexagones négatifs ou neutres se retrouvant hors de ce coronénoïde et, de ce fait, étant considérés comme absents. Dans ce cas, ces hexagones ne seront pas représentés dans F_{i-}, ni dans $F_{i\circ}$, mais placés dans un ensemble F_{i*}. Enfin, nous posons la contrainte de somme $\sum_j e_j = 1$ pour garantir l'existence d'au moins un exemplaire du motif.

5.2 Deuxième modélisation

Dans cette seconde modélisation (notée $\mathcal{M}_{i_2}$), nous cherchons à exprimer l'existence d'un fragment correspondant au motif M en raisonnant sur le voisinage de chaque hexagone. Pour cela, partant du modèle $\mathcal{M}$, nous ajoutons une variable f_i par hexagone du coronénoïde de taille $k(n)$. Chaque variable f_i a pour domaine $\{0, 1, \ldots, n_M\}$. La variable f_i est affectée à une valeur j positive si l'hexagone i du coronénoïde de taille $k(n)$ participe au fragment recherché en tant qu'hexagone occupant la position j dans M, 0 sinon. Ensuite, la génération des structures de benzénoïdes et la recherche d'un fragment se faisant simultanément, nous devons assurer leur concordance. Il faut, en particulier, garantir que les hexagones positifs (resp. négatifs) soient effectivement présents (resp. absents) dans la structure générée. Pour rappel, cette structure est notamment représentée par la variable de graphe x_G et par les variables booléennes x_i. Aussi, pour chaque hexagone i du coronénoïde de taille $k(n)$, on pose les contraintes suivantes (sous forme de clauses) :

- $x_i = 1 \Rightarrow f_i \in \{0\} \cup M_+ \cup M_\circ$,
- $x_i = 0 \Rightarrow f_i \in \{0\} \cup M_- \cup M_\circ$,
- $f_i \in M_+ \Rightarrow x_i = 1$,
- $f_i \in M_- \Rightarrow x_i = 0$.

Il nous faut ensuite définir la bijection permettant d'établir que le fragment construit satisfait le motif M. Autrement dit, il nous faut garantir qu'exactement n_M hexagones de la structure doivent correspondre aux n_M hexagones du motif M. Aussi, pour chaque hexagone $j \in \{1, \ldots, n_M\}$ du motif, on pose la contrainte globale [1] $Count(\{f_1, \ldots, f_{n_c}\}, \{j\}) = 1$ si $j \in M_+$ (≤ 1 sinon). La valeur 0 est obtenue dans le cas où un hexagone négatif ou neutre se trouve hors du coronénoïde de taille $k(n)$. En d'autres termes, le motif dépasse de ce coronénoïde mais uniquement pour des hexagones négatifs ou neutres (qui seraient alors absents). En procédant ainsi, cela évite d'introduire des variables supplémentaires (et les contraintes associées) pour représenter les k_M couches d'hexagones absents utilisés dans la définition formelle de fragment.

Il reste maintenant à définir le motif en lui-même. Pour cela, on va décrire les liens de voisinage entre

1. Pour rappel, la contrainte $Count(Y, V) \odot k$ est satisfaite ssi le nombre de variables de Y affectées avec une valeur dans V satisfait la condition vis-à-vis de l'opérateur $\odot$ et de la valeur k.

chaque hexagone du motif. Un hexagone peut posséder jusqu'à six hexagones voisins. Pour un hexagone h donné, nous considérons ses voisins potentiels v_1 à v_6 dans le sens horaire en commençant par le voisin situé en haut à droite. À partir de là, nous listons les différentes configurations prises par les voisins selon que l'hexagone h participe au fragment ou non. Plus précisément, chaque configuration est un tuple composé d'un entier par voisin. Cet entier est une valeur j non nulle si le voisin participe au fragment en tant qu'hexagone j du motif, 0 sinon. Pour chaque position de l'hexagone h dans le motif M, nous considérons six configurations possibles afin de tenir compte des rotations de 60° du motif. Cela est nécessaire pour générer toutes les structures car le modèle $\mathcal{M}$ impose l'existence d'hexagone(s) sur les bordures en haut et à gauche du coronénoïde considéré. Notons qu'à partir d'une configuration donnée, appliquer une rotation de 60° revient à effectuer une permutation circulaire au niveau du tuple. Par exemple, dans la table 1, nous listons toutes les configurations de voisinage possibles quand l'hexagone est en position 1 dans le motif « *baie profonde* », la numérotation des hexagones étant celle de la figure 2(a). Pour les autres, nous n'en donnons qu'une par manque de place. Ces configurations vont être utilisées pour définir la relation associée à des contraintes de tables concises [11]. Nous considérons une telle contrainte par hexagone h du coronénoïde de taille $k(n)$ avec pour portée la variable f_h et chaque variable f_i associée à un voisin de h dans $B_h^{c(k(n))}$. Pour les hexagones en bordure du coronénoïde, on ne conserve que les lignes de la table dont les voisins participant au fragment correspondent à des hexagones (quelle que soit leur nature) à l'intérieur du coronénoïde ou à des hexagones négatifs ou neutres hors du coronénoïde. Puis, on fait une projection de ces lignes sur les voisins présents et la variable f_i.

f_i	f_{v_1}	f_{v_2}	f_{v_3}	f_{v_4}	f_{v_5}	f_{v_6}
0	*	*	*	*	*	*
1	0	2	4	3	0	0
1	2	4	3	0	0	0
1	4	3	0	0	0	2
1	3	0	0	0	2	4
1	0	0	0	2	4	3
1	0	0	2	4	3	0
2	0	0	5	4	1	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮
3	1	4	6	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮
4	2	5	7	6	3	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
5	0	0	0	7	4	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮
6	4	7	0	0	0	3
⋮	⋮	⋮	⋮	⋮	⋮	⋮
7	5	0	0	0	6	4
⋮	⋮	⋮	⋮	⋮	⋮	⋮

TABLE 1 – La contrainte de table concise décrivant le voisinage pour le motif « *baie profonde* ».

5.3 Troisième modélisation

Un fragment d'ordre k d'un benzénoïde B correspond à un sous-graphe connexe de B_h^k. Aussi, déterminer s'il existe un fragment satisfaisant un motif M dans un benzénoïde B revient, d'une certaine manière, à déterminer s'il existe un sous-graphe dans $B_h^{k_M}$ isomorphe à M_h. Toutefois, il ne s'agit pas exactement du traditionnel problème d'isomorphisme de sous-graphes, mais, d'une de ses variantes prenant en compte l'étiquetage des sommets et des arêtes. Cela ne change en rien la complexité du problème de décision qui reste NP-complet. Fort heureusement, nous n'avons pas besoin de nous attaquer à ce problème car, dans notre approche, nous allons, par construction, produire directement des structures satisfaisant le motif.

Nous présentons maintenant notre modèle $\mathcal{M}_{i_3}$. Partant du modèle général $\mathcal{M}$, nous ajoutons une variable s_i par hexagone du motif M (quelle que soit sa nature). Chaque variable s_i a pour domaine $\{1, \ldots, n'_c\}$ avec n'_c le nombre d'hexagones du coronénoïde de taille $k(n) + k_M$. Nous exploitons un coronénoïde de taille $k(n) + k_M$, et non $k(n)$, car nous avons besoin d'entourer le coronénoïde de taille $k(n)$ de k_M couronnes d'hexagones absents. Notons que cela n'a aucun impact sur la variable de graphe x_G ou sur les variables x_i car nous ajoutons des hexagones dont on sait qu'ils ne sont pas présents dans la structure considérée. La variable s_i a pour valeur j si le i-ème hexagone du motif M est le j-ème hexagone du coronénoïde de taille $k(n) + k_M$. Par convention, les valeurs de j comprises entre 1 et n_c correspondent à des hexagones présents dans le coronénoïde de taille $k(n)$. Nous ajoutons ensuite les contraintes suivantes pour exprimer la notion d'isomorphisme :

— *Injectivité* : Les hexagones participant au fragment doivent être tous différents deux à deux. Ceci est imposé grâce à la contrainte $\texttt{alldifferent}(\{s_1, \ldots, s_{n_M}\})$. Cela assure également qu'on a bien n_M hexagones de x_G qui participent au fragment.

— *Préservation des arêtes* : On doit garantir qu'à deux sommets voisins de M_h correspondent deux sommets voisins dans le graphe d'hexagones du coronénoïde de taille $k(n)$. Aussi, pour chaque arête $\{i, i'\}$ de M_h (quelle que soit sa nature), on pose une contrainte de table portant sur s_i et $s_{i'}$ dont la relation contient tous les couples

(j, j') tels que $\{j, j'\}$ est une arête du graphe d'hexagones du coronénoïde de taille $k(n) + k_M$.

Cette partie du modèle est inspirée de la modélisation du problème d'isomorphisme de sous-graphes présenté dans [6]. Cependant, il faut noter que, dans notre cas, le graphe dans lequel le sous-graphe est recherché n'est pas connu à l'avance, car il s'agit du graphe que nous souhaitons construire. Aussi, dans notre modèle, nous contournons cette difficulté en considérant le graphe d'hexagones du coronénoïde de taille $k(n) + k_M$.

Concernant les étiquetages, par définition, l'étiquetage des arêtes découle de celui des sommets. Celui des sommets est directement pris en compte par définition des variables s_i. Il ne reste donc qu'à exprimer l'adéquation entre l'étiquetage des sommets et l'existence des hexagones grâce aux contraintes suivantes (exprimées sous forme de clauses) :

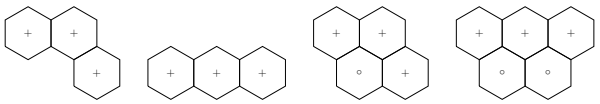
— $\forall i \in M_+, \forall j \in \{1, \ldots, n_c\}, s_i = j \Rightarrow x_j = 1$,
— $\forall i \in M_+, \forall j \in \{1, \ldots, n_c\}, x_j = 0 \Rightarrow s_i \neq j$,
— $\forall i \in M_-, \forall j \in \{1, \ldots, n_c\}, s_i = j \Rightarrow x_j = 0$, et
— $\forall i \in M_-, \forall j \in \{1, \ldots, n_c\}, x_j = 1 \Rightarrow s_i \neq j$.

Nous abordons maintenant les limites du raisonnement en termes d'isomorphisme de sous-graphes du point de vue de la chimie. Les figures 5(a)-(b) décrivent deux motifs basés sur trois hexagones positifs dont les graphes d'hexagone sont isomorphes. Il s'avère que les deux molécules correspondantes n'ont pas les mêmes propriétés chimiques. Or, si nous demandons à Choco de produire les structures correspondant à chacun de ces deux motifs sur la base du modèle $\mathcal{M}_{i_3}$ nous obtiendrons les mêmes solutions. Aussi, pour pallier ce problème, nous ajoutons une phase de prétraitement du motif en amont de la génération de l'instance à résoudre. Cette phase consiste à rajouter des hexagones neutres de sorte à ce que toute arête du graphe d'hexagones du motif intervienne dans au moins un triangle (c'est-à-dire une clique de taille 3). Un triangle dans le graphe d'hexagones représente trois hexagones qui sont deux à deux voisins. Il caractérise donc une configuration unique (à une symétrie axiale ou à une rotation de 60° près). Ce prétraitement peut être mis en œuvre en parcourant une fois les hexagones du motif initial du haut vers le bas et de la gauche vers la droite. Par manque de place, nous ne détaillons pas cet algorithme. Les figures 5(c)-(d) présentent les motifs ainsi complétés associés aux motifs de (a) et (b). Notons qu'il n'est pas toujours nécessaire d'ajouter des hexagones neutres. Par exemple, le motif « baie profonde » reste inchangé car chaque arête de son graphe d'hexagones participe déjà à au moins un triangle.

6 Structures incluant plusieurs motifs

Dans cette section, nous nous intéressons à générer des structures contenant simultanément plusieurs motifs. Soit $E_M = \{M^1, \ldots, M^\ell\}$ l'ensemble de ces motifs. L'existence de plusieurs motifs pose la question de comment ils peuvent interagir entre eux. Nous listons ici trois cas ayant du sens du point de vue chimique :

(1) Les motifs peuvent partager des hexagones (quelle que soit leur nature),
(2) Les motifs ne peuvent partager que des hexagones absents (autrement dit du « vide »),
(3) Les motifs sont disjoints deux à deux.



(a) (b) (c) (d)

FIGURE 5 – Les limites de l'isomorphisme de sous-graphes (a) et (b). Les motifs (a) et (b) après prétraitement (c) et (d).

Une première approche naïve pour résoudre ce problème multi-motifs consiste à résoudre une collection de problèmes à un seul motif. Il faudrait, pour cela, énumérer tous les motifs uniques qu'on pourrait construire sur la base des motifs de E_M. Mais, vu la combinatoire, cette approche semble exclue. Nous proposons donc ci-dessous des modélisations adaptées sur la base des modèles vus précédemment.

6.1 Première modélisation

Nous partons du modèle général $\mathcal{M}$. Puis, nous ajoutons, pour chaque motif M^j de E_M, un ensemble de variables e_i^j équivalentes aux variables e_i pour un motif M dans le modèle $\mathcal{M}_{i_1}$ ainsi que la contrainte de somme associée. Bien entendu, cela suppose d'avoir calculé, en amont, tous les fragments possibles de chaque motif de E_M. Cela définit le modèle $\mathcal{M}^1_{m_1}$.

Pour obtenir des motifs disjoints (modèle $\mathcal{M}^3_{m_1}$), il faut ajouter au modèle $\mathcal{M}^1_{m_1}$ des clauses d'exclusion mutuelle $e_i^j = 0 \vee e_{i'}^{j'} = 0$ pour chaque paire de fragments $\{F_i^j, F_{i'}^{j'}\}$ s'intersectant (c'est-à-dire tel que $(F^j_{i+} \cup F^j_{i-} \cup F^j_{i\circ} \cup F^j_{i*}) \cap (F^{j'}_{i'+} \cup F^{j'}_{i'-} \cup F^{j'}_{i'\circ} \cup F^{j'}_{i'*}) \neq \emptyset$).

Pour ne partager que du vide (modèle $\mathcal{M}^3_{m_1}$), nous ajoutons, au modèle $\mathcal{M}^1_{m_1}$, des contraintes de la forme $e_i^j = 0 \vee e_{i'}^{j'} = 0$ si F_i^j et $F_{i'}^{j'}$ peuvent partager un hexagone présent (c'est-à-dire si $(F^j_{i+} \cap F^{j'}_{i'+}) \cup (F^j_{i+} \cap F^{j'}_{i'\circ}) \cup (F^j_{i\circ} \cap F^{j'}_{i'+}) \neq \emptyset$). Sinon, s'ils partagent des hexagones neutres, ceux-ci doivent être absents de la structure, ce qui est assuré en posant la contrainte $(e_i^j = 1 \wedge e_{i'}^{j'} = 1) \Rightarrow x_h = 0$ pour chaque hexagone $h \in F^j_{i\circ} \cap F^{j'}_{i'\circ}$.

6.2 Deuxième modélisation

Nous partons du modèle général $\mathcal{M}$. Puis, nous ajoutons, pour chaque motif M^j de E_M, un ensemble de variables f_i^j équivalentes aux variables f_i pour un motif M dans le modèle $\mathcal{M}_{i_2}$ ainsi que toutes les contraintes associées. Toutefois, dans chaque contrainte de table définissant le motif M^j, nous introduisons, dans sa portée, une variable booléenne t^j. Cette variable vaut 1 si la configuration associée est obtenue après l'application d'une symétrie axiale sur M^j, 0 sinon. Ainsi, la table va lister toutes les configurations valides obtenues à partir du motif M^j ou de son image par une symétrie axiale. La prise en compte des symétries axiales dans le cas de l'inclusion de plusieurs motifs est requise afin d'énumérer toutes les possibilités de combiner les motifs entre eux. Plusieurs axes de symétries sont possibles. Toutefois, il suffit d'en considérer un seul, les autres pouvant être obtenus par combinaisons avec des rotations de 60°. L'utilisation de la variable t^j au sein de chacune des contraintes de tables définissant M^j garantit que globalement, on exploite soit le motif M^j si t^j vaut 0, soit son image par symétrie axiale sinon. Cela évite donc de considérer des fragments erronés dont une partie correspondrait à M^j et une autre à son image par symétrie. Notons que, dans le cas d'un motif unique, l'utilisation de cette variable t^j ne ferait que rajouter des solutions équivalentes à celles déjà produites. Le modèle que nous venons de décrire correspond au cas (1). Nous le notons $\mathcal{M}^1_{m_2}$.

Ensuite, pour traiter le cas (2) autorisant le partage uniquement d'hexagones absents, nous reprenons le modèle $\mathcal{M}^1_{m_2}$ et ajoutons des contraintes d'exclusion mutuelle pour les hexagones présents. Cela revient à poser la contrainte suivante pour chaque hexagone h du coronénoïde de taille $k(n)$: $x_h = 1 \Rightarrow Count(\{f_h^1, \ldots, f_h^\ell\}, \{1 \ldots, n_{E_M}\}) \leq 1$ avec $n_{E_M} = \max_{M^j \in E_M} n_{M^j}$. Nous notons $\mathcal{M}^2_{m_2}$ ce modèle.

Enfin, pour pouvoir considérer des motifs disjoints (cas (3)), nous devons tenir compte des hexagones qui pourraient être partagés à l'extérieur du coronénoïde de taille $k(n)$. Pour cela, nous définissons l'ordre k_{E_M} de l'ensemble E_M comme l'ordre maximum d'un motif M^j de E_M. Puis, nous considérons le modèle $\mathcal{M}^1_{m_2}$ mais dans un coronénoïde de taille $k(n) + k_{E_M}$. Autrement dit, nous ajoutons à $\mathcal{M}^1_{m_2}$ une variable f_i^j par hexagone se trouvant à l'extérieur du coronénoïde de taille $k(n)$ et par motif M^j. Tous les hexagones étant représentés explicitement, les contraintes de tables sont définies en prenant en compte ces nouvelles variables et les contraintes $Count$ de $\mathcal{M}^1_{m_2}$ pour des hexagones j' négatifs ou neutres du motif M^j sont désormais de la forme $Count(\{f_1^j, \ldots, f_{n_c'}^j\}, \{j'\}) = 1$. Pour terminer, nous ajoutons une contrainte d'exclusion mutuelle $Count(\{f_h^1, \ldots, f_h^\ell\}, \{1 \ldots, n_{E_M}\}) \leq 1$ pour chaque hexagone h du coronénoïde de taille $k(n) + k_{E_M}$. Nous notons $\mathcal{M}^3_{m_2}$ ce modèle.

6.3 Troisième modélisation

Le principe est le même que pour la modélisation précédente. Pour chaque motif M^j de E_M, nous ajoutons au modèle $\mathcal{M}$ un ensemble de variables s_i^j équivalentes aux variables s_i utilisées pour un motif M dans le modèle $\mathcal{M}_{i_3}$ ainsi que toutes les contraintes associées. Ce faisant, nous obtenons le modèle $\mathcal{M}^1_{m_3}$ correspondant au cas (1). Le modèle $\mathcal{M}_{i_3}$ dépendant de l'ordre du motif, les structures générées devront être inscrites dans un coronénoïde de taille $k(n) + k_{E_M}$. Bien entendu, comme dans $\mathcal{M}_{i_3}$, chaque motif doit être prétraité en amont.

Puis, nous pouvons étendre ce modèle au modèle $\mathcal{M}^3_{m_3}$ afin de tenir compte des motifs disjoints, en ajoutant la contrainte d'exclusion mutuelle $\texttt{alldifferent}(\{s_1^1, \ldots, s_{n_{M^1}}^1\} \cup \ldots \cup \{s_1^\ell, \ldots, s_{n_{M^\ell}}^\ell\})$.

Enfin, le modèle $\mathcal{M}^2_{m_3}$ correspondant au cas (2) est obtenu à partir du modèle $\mathcal{M}^1_{m_3}$, en ajoutant les contraintes suivantes :

- $\texttt{alldifferent}(\{s_i^j | j \in \{1, \ldots \ell\}, i \in M_+^j\})$ qui exprime que les hexagones positifs dans x_G sont deux à deux disjoints,
- $\forall j, j' \in \{1, \ldots \ell\}, j < j', \forall i \in M_\circ^j, \forall i' \in M_\circ^{j'}, s_i^j = s_{i'}^{j'} \Rightarrow (s_i^j > n_c \vee Element(\{x_z | z \in \{1, \ldots, n_c\}\}, s_i^j) = 0)$ [2] qui exprime le fait que si deux hexagones neutres désignent le même hexagone de x_G, alors le sommet correspondant ne figure pas dans x_G.
- $\forall j, j' \in \{1, \ldots \ell\}, j \neq j', \forall i \in M_\circ^j, \forall i' \in M_+^{j'}, s_i^j \neq s_{i'}^{j'}$ qui interdit d'avoir le même hexagone de x_G pour un hexagone neutre et un hexagone positif de deux motifs différents.

7 Autres questions autour des motifs

Nous abordons maintenant quelques problématiques voisines autour des motifs. D'abord, nous traitons de l'exclusion d'un motif avant de montrer comment imposer des contraintes sur le nombre d'exemplaires d'un motif donné.

7.1 Structures excluant un motif

Nous souhaitons, à présent, générer toutes les structures à n hexagones ne contenant pas un motif M donné. Le raisonnement suivi pour les modèles $\mathcal{M}_{i_2}$ et $\mathcal{M}_{i_3}$ semble peu adapté car il faudrait garantir qu'il n'existe aucune numérotation f_i ou aucun sous-graphe

2. Pour rappel, la contrainte $Element(Y, j) \odot k$ est satisfaite ssi la valeur de la j-ème variable de Y satisfait la condition vis-à-vis de l'opérateur $\odot$ et de la valeur k.

isomorphe convenable. Aussi, nous suivons ici le même raisonnement que celui du modèle $\mathcal{M}_{i_1}$. Plus précisément, nous partons du modèle $\mathcal{M}$ et y ajoutons une variable e_i par fragment possible dans un coronénoïde de taille $k(n)$. Chaque variable e_i est vraie ssi la contrainte $\bigwedge_{j \in F_{i-}} x_j = 0 \wedge \bigwedge_{j \in F_{i+}} x_j = 1$ est satisfaite (c'est-à-dire le fragment présent dans la structure). Nous posons enfin une contrainte de somme $\sum_j e_j = 0$. Une formulation équivalente consiste à ne pas exploiter les variables e_i et à représenter directement chaque fragment F_i sous la forme d'un nogood $\bigvee_{j \in F_{i-}} x_j = 1 \vee \bigvee_{j \in F_{i+}} x_j = 0$.

7.2 Contraindre le nombre d'exemplaires

Certaines contraintes sur le nombre d'exemplaires d'un motif M sont faciles à modéliser. Par exemple, pour générer des structures de benzénoïdes ayant au moins k exemplaires disjoints du motif M, on peut utiliser les modélisations $\mathcal{M}^3_{m_1}$, $\mathcal{M}^3_{m_2}$ ou $\mathcal{M}^3_{m_3}$ et un ensemble E_M constitué de k fois le motif M. D'autres sont un peu plus délicates. Pour faciliter l'expression de telles contraintes, nous définissons une variable n_e qui représente le nombre d'exemplaires du motif M contenus dans la structure générée et sur laquelle nous poserons les contraintes adéquates selon les besoins des chimistes. La variable n_e a pour domaine $\{0, \ldots, k_{max}\}$ avec k_{max} le nombre maximum d'exemplaires que peut ou doit contenir la structure. Par défaut, si aucune information n'est donnée en entrée sur k_{max}, on prend $k_{max} = \left\lfloor \frac{n}{|M_+|} \right\rfloor$.

Une nouvelle fois, l'approche suivie dans le modèle $\mathcal{M}_{i_1}$ semble être la plus appropriée. Ainsi, partant du modèle $\mathcal{M}_{i_1}$, nous intégrons la variable n_e. En plus des variables e_i introduites pour chaque fragment possible, nous ajoutons une variable booléenne e'_i par fragment. La variable e'_i est vraie si le fragment F_i est présent dans le motif. Cela est assuré en ajoutant, pour chaque fragment F_i la contrainte $\bigwedge_{j \in F_{i-}} x_j = 0 \wedge \bigwedge_{j \in F_{i+}} x_j = 1 \Rightarrow e'_i = 1$. Ensuite, certains fragments partageant des hexagones, nous ajoutons des contraintes d'exclusion mutuelle, avec, pour chaque hexagone h, la contrainte $\sum_{i|h \in F_i} e'_i \geq 1 \Rightarrow \sum_{i|h \in F_i} e_i = 1$ (en considérant que $F_i = F_{i+} \cup F_{i-} \cup F_{i\circ} \cup F_{i*}$). Ainsi, cela nous garantit qu'un seul fragment est considéré comme présent pour un hexagone participant simultanément à plusieurs fragments. Enfin, la contrainte $\sum_j e_j = n_e$ nous permet alors de calculer le nombre d'exemplaires présents sur lequel nous pouvons ensuite poser aisément n'importe quelle contrainte arithmétique. Il est également possible d'utiliser cette variable afin de trouver les structures maximisant le nombre d'exemplaires du motif.

8 Expérimentations

Nous comparons ici nos différentes modélisations. Notre implémentation repose sur Choco (v. 4.10.7) lequel est utilisé avec ses réglages par défaut. Nous considérons les huit motifs issus de [12] et décrits dans les figures 4(a) et 6 et faisons varier le nombre n d'hexagones présents dans les structures du nombre d'hexagones positifs du motif à 9. Cela nous permet de produire 55 instances (resp. 135) du problème de génération de structures contenant un motif (resp. deux motifs). Les expérimentations sont réalisées sur des serveurs DELL PowerEdge R440 dotés de processeur Intel Xeon 4112 à 2,6 GHz et de 32 Go de mémoire. Le temps est limité à 2h.

D'après la figure 7, le modèle $\mathcal{M}_{i_1}$ s'avère être légèrement plus performant pour générer les structures de benzénoïdes contenant un motif donné que les modèles $\mathcal{M}_{i_2}$ et $\mathcal{M}_{i_3}$. Une explication possible réside dans le fait que les trois modèles reposent sur le modèle général $\mathcal{M}$ auquel quelques variables et contraintes sont ajoutées. Au final, les trois modèles ont un nombre similaire de contraintes pour un nombre de variables plus important pour le modèle $\mathcal{M}_{i_1}$.

Pour deux motifs, le modèle $\mathcal{M}^3_{m_1}$ se révèle le meilleur (voir la figure 8), suivi par le modèle $\mathcal{M}^3_{m_3}$. Ce résultat semble lié au nombre de contraintes qui est deux fois plus important pour le modèle $\mathcal{M}^3_{m_2}$ tandis que les modèles $\mathcal{M}^3_{m_1}$ et $\mathcal{M}^3_{m_3}$ conduisent à un nombre similaire de contraintes.

9 Conclusion et perspectives

Nous avons présenté une approche basée sur la PPC permettant de générer des structures de benzénoïdes satisfaisant certaines contraintes autour de l'inclusion de motifs. Pour cela, plusieurs modélisations ont été considérées et comparées. La première modélisation, basée sur la liste des fragments possibles, se révèle globalement la plus robuste, mais aussi celle ayant le plus grand pouvoir d'expression (inclusion/exclusion de motif, nombre d'occurrences, ...).

Une première perspective de ce travail sera d'étudier ses retombées du point de vue de la chimie théorique. Concernant la modélisation, une étude plus poussée de la notion d'ordre d'un motif devrait permettre de l'affiner avec, à la clé, une réduction de l'espace de recherche à explorer. Enfin, plusieurs voies peuvent être explorées pour améliorer l'efficacité pratique de l'approche (définition de contraintes globales dédiées, étude des heuristiques de choix de variables, ...).

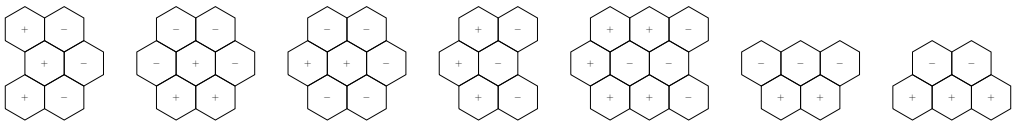


FIGURE 6 – Sept des huit motifs considérés pour les expérimentations.

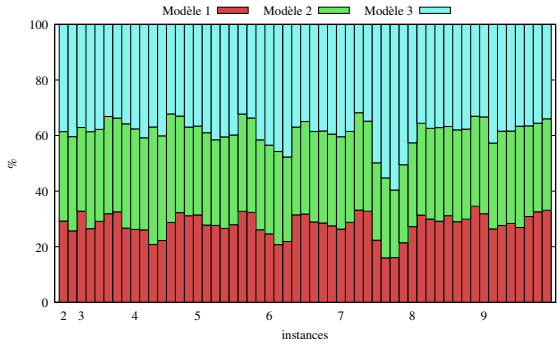


FIGURE 7 – Rapport en pourcentage du temps consommé par chaque modèle sur le temps cumulé des modèles $\mathcal{M}_{i_1}$, $\mathcal{M}_{i_2}$ et $\mathcal{M}_{i_3}$ en fonction des instances classées selon le nombre n d'hexagones souhaités.

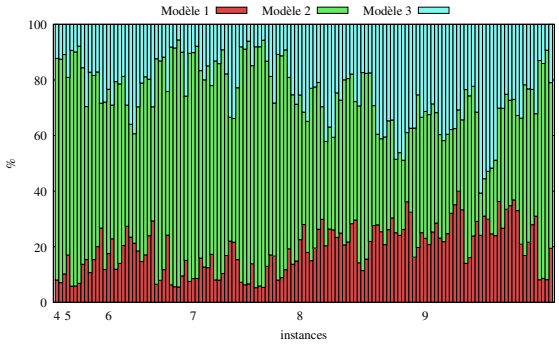


FIGURE 8 – Rapport en pourcentage du temps consommé par chaque modèle sur le temps cumulé des modèles $\mathcal{M}_{m_1}$, $\mathcal{M}_{m_2}$ et $\mathcal{M}_{m_3}$ en fonction des instances classées selon le nombre n d'hexagones souhaités.

Références

[1] G. BRINKMANN, G. CAPOROSSI et P. HANSEN : A Constructive Enumeration of Fusenes and Benzenoids. *Journal of Algorithms*, 45(2), 2002.

[2] Y. CARISSAN, D. HAGEBAUM-REIGNIER, N. PRCOVIC, C. TERRIOUX et A. VARET : Using Constraint Programming to Generate Benzenoid Structures in Theoretical Chemistry. *In CP*, pages 690–706, 2020.

[3] J. CYVIN, J. BRUNVOLL et B. N. CYVIN : Search for Concealed Non-Kekuléan Benzenoids and Coronoids. *J. Chem. Inf. Comput. Sci.*, 29(4):237, 1989.

[4] T. DUMSLAFF, Y. GU, G. M. PATERNÒ, Z. QIU, A. MAGHSOUMI, M. TOMMASINI, X. FENG, F. SCOTOGNELLA, A. NARITA et K. MÜLLEN : Hexa-Peri-Benzocoronene with Two Extra K-Regions in an Ortho-Configuration. *Chem. Sci.*, 11:12816–12821, 2020.

[5] J.-G. FAGES, X. LORCA et C. PRUD'HOMME : Choco solver user guide documentation. https://choco-solver.readthedocs.io/en/latest/.

[6] C. LECOUTRE et O. ROUSSEL, éditeurs. *Proceedings of the 2018 XCSP3 Competition*, 2018.

[7] S. MISHRA, D. BEYER, K. EIMRE, S. KEZILEBIEKE, R. BERGER, O. GRÖNING, C. A. PIGNEDOLI, K. MÜLLEN, P. LILJEROTH, P. RUFFIEUX, X. FENG et R. FASEL : Topological frustration induces unconventional magnetism in a nanographene. *Nature Nanotechnology*, 15(1):22–28, 2020.

[8] S. MISHRA, D. BEYER, K. EIMRE, J. LIU, R. BERGER, O. GRÖNING, C. A. PIGNEDOLI, K. MÜLLEN, R. FASEL, X. FENG et P. RUFFIEUX : Synthesis and Characterization of π-Extended Triangulene. *Journal of the American Chemical Society*, 141(27):10621–10625, 2019.

[9] W. NIU, J. MA, P. SOLTANI, W. ZHENG, F. LIU, A. POPOV, J. WEIGAND, H. KOMBER, E. POLIANI, C. CASIRAGHI, J. DROSTE, M. HANSEN, S. OSELLA, D. BELJONNE, M. BONN, H. WANG, X. FENG, J. LIU et Y. MAI : A Curved Graphene Nanoribbon with Multi-Edge Structure and High Intrinsic Charge Carrier Mobility. *J. Am. Chem. Soc.*, 142:18293–18298, 2020.

[10] Georges TRINQUIER et Jean-Paul MALRIEU : Predicting the Open-Shell Character of Polycyclic Hydrocarbons in Terms of Clar Sextets. *The Journal of Physical Chemistry A*, 122(4):1088–1103, 2018.

[11] H. VERHAEGHE, C. LECOUTRE et P. SCHAUSS : Extending Compact-Table to Negative and Short Tables. *In AAAI*, pages 3951–3957, 2017.

[12] N. WOHNER, P. K. LAM et K. SATTLER : Systematic energetics study of graphene nanoflakes : From armchair and zigzag to rough edges with pronounced protrusions and overcrowded bays. *Carbon*, 82:523–537, 2015.

Actes JFPC 2021

Stratégies de résolution anytime de problèmes de satisfaction de contraintes numériques

Thomas Richard de Latour[1]*, **Raphaël Chenouard**[2], **Laurent Granvilliers**[1]

Laboratoire des Sciences du Numérique de Nantes, UMR CNRS 6004
Université de Nantes[1], France
Centrale Nantes[2], France
thomas.richard-de-latour@ls2n.fr raphael.chenouard@ec-nantes.fr
laurent.granvilliers@univ-nantes.fr

Résumé

Un algorithme Branch-and-Prune calcule un pavage de l'ensemble des solutions d'un problème de satisfaction de contraintes numériques à une précision donnée. Un algorithme anytime stoppe le calcul prématurément, par exemple en limitant le temps de calcul. Cet article revisite les algorithmes Branch-and-Prune Anytime et introduit deux nouvelles stratégies de recherche ainsi que trois indicateurs de qualité pour les comparer. Une étude de complexité et une analyse expérimentale montrent que ces nouvelles stratégies sont pertinentes.

Abstract

A Branch-and-Prune algorithm computes a paving of the solution set of a numerical constraint satisfaction problem to a given precision. An anytime algorithm prematurely stops the computation to a given criterion (for example a maximum running time). This article revises anytime Branch-and-Prune algorithms and introduces two new search strategies and three quality indicators to compare them. The evaluation of their complexity and an experimental analysis enabled to prove the efficiency of those new strategies.

1 Introduction

Un problème de satisfaction de contraintes numériques (NCSP) est défini par un ensemble de contraintes (équations et inégalités) non linéaires sur des variables réelles dont les domaines sont des intervalles. C'est un cadre utilisé pour modéliser des applications dans de nombreux domaines comme en robotique [5], en automatique [6] et en conception de systèmes [2]. Résoudre un NCSP peut prendre plusieurs sens : prouver qu'il n'existe pas de solutions, trouver une solution, toutes les solutions, la meilleure selon une fonction objectif ou un sous-ensemble de solutions diverses. C'est ce dernier sens que nous privilégions ici. Nous visons des applications en conception de systèmes où un système est souvent modélisé par un NCSP sous-contraint admettant un nombre de solutions infini. Le but est de trouver des architectures différentes et représentatives dans un tel espace de solutions pour aider au choix des bons concepts.

Un algorithme de type Branch-and-Prune (BP) sur les intervalles a pour but de calculer des boîtes (produits cartésiens d'intervalles) encadrant les solutions d'un NCSP à une précision $\epsilon > 0$ fixée [16]. Un arbre de recherche est généré en alternant des phases de réduction des domaines au moyen de techniques de cohérence locale sur les intervalles [8, 1, 15] et des phases de découpage des domaines. Une exploration complète permet de calculer une couverture de l'espace des solutions à la précision voulue mais ce n'est pas réalisable en un temps raisonnable si l'espace des solutions est infini et si la précision ϵ est faible.

Notre travail s'intéresse aux algorithmes Branch-and-Prune anytime (ABP). En fixant un critère d'arrêt comme un temps de calcul limite ou un nombre de solutions maximum, un tel algorithme réalise une exploration partielle de l'espace avec le but d'obtenir des solutions diverses, représentatives, les plus éloignées possibles les unes des autres. Ce type d'algorithme est notamment utilisé pour le calcul de points bien répartis sur un front de Pareto en optimisation multi-objectif [7]. Dans ce but, les parcours classiques de

*Papier doctorant : Thomas Richard de Latour[1] est auteur principal.

l'arbre de recherche ne fonctionnent pas : un parcours en profondeur (DFS, Depth-First Search) convergerait rapidement vers des solutions proches ; un parcours en largeur (BFS, Breadth-First Search) serait très lent pour atteindre des boîtes de précision ϵ. Il existe des stratégies de parcours hybrides pour contrer ces problèmes. Par exemple, la stratégie présentée dans [11] alterne des étapes de recherche en profondeur et des étapes de recherche en largeur pour diversifier (toutefois pour des problèmes discrets). Dans le même esprit, la stratégie DMDFS [3] a pour but de maximiser la distance entre les solutions calculées. Elle alterne des phases de recherche en profondeur pour trouver une solution et des phases de sélection de la prochaine boîte à explorer la plus éloignée des solutions déjà calculées. Dans le cadre de l'optimisation, la stratégie Best-First [12] consiste à sélectionner le prochain nœud de l'arbre selon un critère donné.

Dans ce travail, nous proposons un cadre pour décrire les stratégies ABP ainsi que deux nouvelles stratégies. Elles sont comparées à l'heuristique DMDFS [3] sur le plan de la complexité algorithmique. Une analyse expérimentale est également conduite au moyen d'un prototype développé au sein de la librairie IBEX. Pour qualifier les résultats obtenus, différents indicateurs de qualité sont définis en s'inspirant de l'existant en optimisation multi-objectif [14, 9].

Après un bref rappel des notions de NCSP et de calcul par intervalles dans la section 2, nous présentons les algorithmes ABP et les différentes stratégies de recherche existantes dans la section 3. La section 4 introduit les nouvelles stratégies. Dans la section 5 nous présentons le contenu des expérimentations et les indicateurs de qualité pour qualifier les résultats.

2 Notions Préliminaires

2.1 Arithmétique des intervalles

On note $[x]$ un *intervalle* fermé de $\mathbb{R}$ défini par une borne inférieure $\underline{x}$ et une borne supérieure $\overline{x}$ telles que $[x] = \{y \in \mathbb{R} : \underline{x} \leq y \leq \overline{x}\}$. La *largeur* de $[x]$ est définie par le nombre réel $w([x]) = \overline{x} - \underline{x}$. L'ensemble des intervalles est noté $\mathbb{I}$.

Une *boîte* $[\mathbf{x}]$ de dimension n est un produit cartésien d'intervalles $[\mathbf{x}] = [x_1] \times [x_2] \times \cdots \times [x_n] \subseteq \mathbb{R}^n$. La *largeur* de $[\mathbf{x}]$ est le maximum des largeurs de ses composants. Le *volume* de $[\mathbf{x}]$ est le produit des largeurs de ses composants. Étant donné un nombre réel $\epsilon > 0$, on dit que $[\mathbf{x}]$ est une ϵ-*boîte* si $w([\mathbf{x}]) \leq \epsilon$.

L'*enveloppe* d'un ensemble $S \subseteq \mathbb{R}$ est le plus petit intervalle contenant S noté $\Box S$. L'*enveloppe* d'un ensemble $S \subseteq \mathbb{R}^n$ est la plus petite boîte (en volume) contenant S notée $\Box S$. Par extension, l'enveloppe d'un ensemble de boîtes correspond à l'enveloppe de l'union de ces boîtes. Un *pavage* d'un ensemble $S \subseteq \mathbb{R}^n$ est un ensemble de boîtes de $\mathbb{I}^n$ dont l'union contient S.

La distance de Hausdorff entre deux intervalles est définie par $d_H([x],[y]) = max\{|\overline{x} - \overline{y}|, |\underline{x} - \underline{y}|\}$. La distance de Hausdorff entre deux boîtes $[\mathbf{x}], [\mathbf{y}] \subseteq \mathbb{R}^n$ est définie par

$$d_H([\mathbf{x}],[\mathbf{y}]) = \max\{d_H([x_i],[y_i]) : 1 \leq i \leq n\}.$$

Les opérations arithmétiques réalisent des calculs d'enveloppes. Ainsi, une opération binaire $\diamond$ est étendue aux intervalles de manière à vérifier la propriété $[x] \diamond [y] = \Box\{x \diamond y : x \in [x], y \in [y]\}$ et on a par exemple $[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$.

Les fonctions réelles sont également étendues aux intervalles. Étant données deux fonctions $f : \mathbb{R}^n \to \mathbb{R}$ et $[f] : \mathbb{I}^n \to \mathbb{I}$, on dit que $[f]$ est une *extension* de f si pour toute boîte $[\mathbf{x}]$ de dimension n et tout point $x \in [\mathbf{x}]$ on a $f(x) \in [f]([\mathbf{x}])$. L'extension dite *naturelle* consiste à évaluer une expression de f au moyen des opérations intervalles. Par exemple, considérons la fonction $f(x_1, x_2) = x_1 \times (x_2 - 1) + 2$ et $[\mathbf{x}] = [1,3] \times [0,2]$. On calcule

$$[f]([\mathbf{x}]) = [1,3] \times ([0,2] - 1) + 2 = [-1,5]$$

et l'intervalle ainsi obtenu a la propriété d'être un sur-ensemble de l'ensemble image de $[\mathbf{x}]$ par f.

2.2 CSP numérique

Definition 2.1 (NCSP). *Un problème de satisfaction de contraintes numérique ou NCSP est un triplet* $\mathcal{P} = \langle \mathcal{X}, [\mathbf{x}^{init}], \mathcal{C} \rangle$ *défini par :*

- *un ensemble de variables* $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ *;*
- *une boîte* $[\mathbf{x}^{init}]$ *de dimension* n *telle que* $x_j \in [x_j^{init}]$ *pour tout* j *;*
- *un ensemble de contraintes* $\mathcal{C} = \{c_1, c_2, \ldots, c_m\}$ *où chaque contrainte est définie comme une équation ou une inéquation sur* $\mathcal{X}$*.*

L'ensemble des solutions de $\mathcal{P}$ *est défini par* $\Sigma_{\mathcal{P}} = \{x \in [\mathbf{x}^{init}] : c_1(x)$ *et* $c_2(x)$ *et* $\ldots$ *et* $c_p(x)\}$.

Résoudre un NCSP $\mathcal{P}$ pour une précision donnée $\epsilon > 0$ consiste à calculer un pavage de $\Sigma_{\mathcal{P}}$ composé de ϵ-boîtes. Ce pavage peut être obtenu par des algorithmes BP alternant des étapes de propagation pour contracter les domaines et des étapes de recherche pour séparer les problèmes en sous-problèmes. Le paragraphe suivant introduit les méthodes de filtrage sur les intervalles. La section 3 concerne les stratégies de recherche.

2.3 Contracteur et propagation

Les contraintes d'un NCSP permettent de filtrer les domaines des variables de manière à supprimer des

valeurs incohérentes. Dans ce but, on définit les notions de *contracteur* et de *propagation*.

Definition 2.2 (Contracteur)**.** *Un contracteur associé à une contrainte c est un opérateur $\gamma_c : \mathbb{I}^n \to \mathbb{I}^n$ qui vérifie les propriétés suivantes pour tout $[\mathbf{x}] \in \mathbb{I}^n$:*

1. $\gamma_c([\mathbf{x}]) \subseteq [\mathbf{x}]$.
2. $\forall x \in [\mathbf{x}] \setminus \gamma_c([\mathbf{x}])$ *: non* $c(x)$.

Il existe différentes techniques de contraction [8, 1, 15]. En particulier, l'algorithme HC4 repose sur un mécanisme d'inversion des contraintes combiné à des évaluations sur les intervalles. Considérons par exemple l'équation $f(x_1, x_2) = x_1 \times (x_2 - 1) + 2 = 0$ et $[\mathbf{x}] = [1, 3] \times [0, 2]$. Il vient $x_2 = 1 - 2/x_1$ et on réduit le domaine de x_2 de la manière suivante :

$$[x_2] \leftarrow [0, 2] \cap \left(1 - \frac{2}{[1, 3]}\right) = \left[0, \frac{1}{3}\right].$$

En associant un contracteur à chaque contrainte d'un NCSP, on peut appliquer ces contracteurs pour réduire les domaines jusqu'à un état stable. Ainsi, les réductions de domaines obtenues avec une contrainte sont propagées aux autres. La stratégie de propagation est en général de type AC3 [10].

3 Travaux connexes

3.1 Branch-and-Prune

L'algorithme 1 calcule un pavage de l'ensemble des solutions d'un NCSP à une précision donnée ϵ. La liste L est composée de sous-boîtes de la boîte initiale $[\mathbf{x}^{init}]$ correspondant aux feuilles de l'arbre de recherche à traiter. Une boîte $[\mathbf{x}]$ est extraite de L à chaque itération. Cette boîte est contractée (filtrée) puis, si elle n'est pas vide, découpée si sa largeur est supérieure à ϵ ou insérée dans le pavage en sortie sinon.

3.2 Algorithme Anytime

Un algorithme BP Anytime a pour but de résoudre partiellement un NCSP $\mathcal{P}$ en fixant généralement un seuil sur un critère comme le temps de calcul, le nombre de solutions, le nombre d'itérations, etc. Le résultat est donc un pavage d'un sous-ensemble de l'ensemble des solutions $\Sigma_\mathcal{P}$, c'est-à-dire un *sous-pavage* de $\Sigma_\mathcal{P}$.

Pour évaluer les algorithmes BP, on mesure par exemple le temps de calcul ou le nombre de boîtes générées. Pour les algorithmes Anytime, on peut s'inspirer des travaux en optimisation multiobjectif [14, 9]. Nous retenons les trois qualités suivantes :

— l'uniformité de la distribution des boîtes du sous-pavage calculé ;

Algorithme 1 Algorithme Branch-and-Prune

Entrées : NCSP $\mathcal{P} = \langle \mathcal{X}, [\mathbf{x}^{init}], \mathcal{C} \rangle$, précision $\epsilon > 0$
Sortie : pavage Σ de l'ensemble $\Sigma_\mathcal{P}$

$L \leftarrow \{[\mathbf{x}^{init}]\}$;
$\Sigma \leftarrow \emptyset$;
tant que $(L \neq \emptyset)$ **faire**
  $[\mathbf{x}] \leftarrow$ **ExtrairePremier**(L) ;
  Contracter$_\mathcal{C}([\mathbf{x}])$;
  si $[\mathbf{x}]$ n'est pas vide **alors**
    si $w([\mathbf{x}]) \leq \epsilon$ **alors**
      $\Sigma \leftarrow \Sigma \cup \{[\mathbf{x}]\}$;
    sinon
      $\{[\mathbf{x}_1], [\mathbf{x}_2]\} \leftarrow$ **Diviser**$([\mathbf{x}])$;
      $L \leftarrow L \cup \{[\mathbf{x}_1], [\mathbf{x}_2]\}$;
    finsi
  finsi
fin tant que

— le taux de couverture de l'ensemble des solutions du NCSP ;
— la cardinalité du sous-pavage calculé.

La figure 1 illustre le résultat obtenu avec un ABP pour la résolution du NCSP

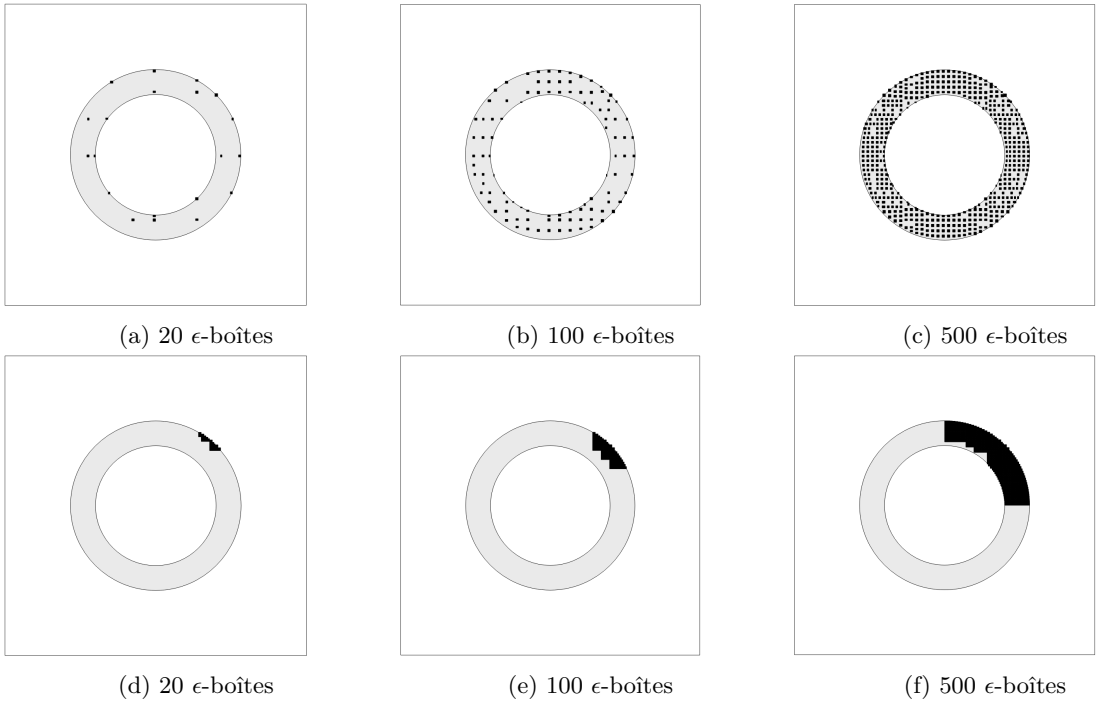
$$\mathcal{P}_1 \begin{cases} x^2 + y^2 \leq 4 \\ x^2 + y^2 \geq 2 \\ -5 \leq x, y \leq 5 \end{cases} \quad (1)$$

selon deux stratégies différentes. Les figures 1d, 1e, 1f sont obtenues par la stratégie Depth-First Search (DFS) de parcours en profondeur et les figures 1a, 1b, 1c par l'algorithme Anytime IDFS, présenté dans la section suivante.

3.3 Stratégies de recherche

Dans l'algorithme 1, la stratégie de recherche est paramétrée par les algorithmes **ExtrairePremier** sélectionnant la prochaine boîte à traiter et **Diviser** séparant la boîte courante en sous-boîtes. Notre étude se concentre sur les stratégies pour sélectionner la prochaine boîte à traiter dans un algorithme BPA et on utilise par exemple une technique de bissection du plus grand domaine pour diviser les boîtes.

Best-First Search La stratégie de recherche *Best-First* (BestFS) a été introduite dans [4] pour diriger l'exploration de graphes au moyen d'une fonction d'évaluation ρ. BestFS est surtout utilisé en optimisation pour accélérer la convergence vers les solutions optimales. Dans le cas des NCSP, il s'agit de maximiser la qualité du sous-pavage calculé au fur et à mesure de



(a) 20 ϵ-boîtes (b) 100 ϵ-boîtes (c) 500 ϵ-boîtes

(d) 20 ϵ-boîtes (e) 100 ϵ-boîtes (f) 500 ϵ-boîtes

FIGURE 1 – Approximations de la résolution du NCSP $\mathcal{P}_1$ (1). Les figures a, b et c illustrent une stratégie de recherche Anytime. Les figures d, e, et f ont été obtenue avec la stratégie DFS

l'exploration. L'algorithme 2 montre l'implémentation de cette stratégie dans un algorithme ABP. Les boîtes non traitées de la liste L sont évaluées par $\rho : \mathbb{I}^n \to \mathbb{R}$ et cette liste est ordonnée sur ce critère.

Les heuristiques BestFS trouvent un intérêt particulier dans l'exploration Anytime d'espace de recherche. La fonction d'évaluation ρ doit tendre vers l'amélioration de la qualité de l'ensemble Σ à chaque nouvelle solution calculée.

En pratique ces stratégies donnent de bonnes approximations mais manquent d'efficacité pour trouver les ϵ-boîtes. Une manière de contourner ce problème est d'implémenter une version hybride DFS/BestFS de ces heuristiques [13]. L'algorithme 3 en découle. L'idée est de trier la liste L seulement après la découverte d'une nouvelle ϵ-boîte solution.

Depth and Most-Distant-First Search Notre travail se positionne dans le sillage de la stratégie *Most-Distant-First Search* (MDFS) [3] associant comportement Anytime et exploration d'espace de recherche. La prochaine boîte sélectionnée par MDFS est celle qui maximise la fonction d'évaluation ρ_M définie par

$$\rho_M([\mathbf{x}], \Sigma) = min\{d_H([\mathbf{x}], \sigma) : \sigma \in \Sigma\} \qquad (2)$$

où Σ est le sous-pavage courant de l'ensemble des solutions et $[\mathbf{x}] \in L$.

Algorithme 2 Algorithme Best-First Search Anytime Branch & Prune

Entrées : NCSP $\langle \mathcal{X}, [\mathbf{x}^{init}], \mathcal{C} \rangle$, Précision ϵ, Critère d'arrêt ϕ
Sortie : sous-pavage Σ de l'ensemble $\Sigma_{\mathcal{P}}$

$L \leftarrow \{[\mathbf{x}^{init}]\}$;
$\Sigma \leftarrow \emptyset$;
tant que ($L \neq \emptyset$ et $\neg\phi$) **faire**
  $[\mathbf{x}] \leftarrow$ **ExtrairePremier**(L) ;
  Contracter$_C([\mathbf{x}])$;
  si $[\mathbf{x}]$ n'est pas vide **alors**
    si w$([\mathbf{x}]) \leq \epsilon$ **alors**
      $\Sigma \leftarrow \Sigma \cup [\mathbf{x}]$;
    sinon
      $\{[\mathbf{x}_1], [\mathbf{x}_2]\} \leftarrow$ **Diviser**$([\mathbf{x}])$;
      $L \leftarrow L \cup \{[\mathbf{x}_1], [\mathbf{x}_2]\}$;
      Evaluer$(\rho([\mathbf{x}_1]))$;
      Evaluer$(\rho([\mathbf{x}_2]))$;
      Trier(L, ρ) ;
    finsi
  finsi
fin tant que
Sortie : Σ
Fin

Algorithme 3 Algorithme hybride BestFS/DFS statique

```
...
  si w([x]) ≤ ϵ alors
    Σ ← Σ ∪ {[x]} ;
    Trier(L, ρ) ;
  sinon
    {[x1], [x2]} ← Diviser([x]) ;
    L ← L ∪ {[x1], [x2]} ;
    Evaluer(ρ([x1])) ;
    Evaluer(ρ([x2])) ;
  finsi
...
```

MDFS peut être assimilée à une stratégie BestFS ayant pour fonction d'évaluation ρ_M. Afin de limiter le nombre d'évaluations et obtenir rapidement les premières solutions, une version hybride DFS/MDFS est proposée dans [3]. Cette version appelée *Depth and Most-Distant-First Search*(DMDFS) est implémentée par l'algorithme 4 qui consiste à évaluer les boîtes non explorées avant l'étape de tri.

Complexité des stratégies Deux catégories de stratégies BestFS peuvent être identifiées : le cas *statique* où la valeur associée à une boîte par la fonction ρ est constante pour toute l'exploration ; le cas *dynamique* qui demande de réévaluer les critères de l'ensemble des boîtes non explorées par ρ à chaque itération (ligne 12 dans l'algorithme 2). Les stratégies dynamiques permettent de prendre en compte des informations qui évoluent au fur et à mesure de l'exploration et ainsi adapter l'exploration à l'aide de ρ, à l'image de DMDFS qui réordonne les boîtes de L selon ρ_M après chaque nouvelle solution trouvée. Ainsi, si ρ est à coût constant, l'étape d'évaluation d'une heuristique dynamique a un coût linéaire en fonction du nombre de boîtes non explorées contre un coût constant pour une heuristique statique.
Pour DMDFS la complexité de ρ_M dépend du nombre de variables n, du nombre de solutions déjà calculées m et de la taille des liste des boîtes non-explorées p. Pour cette étape on distingue :

1. l'évaluation des boîtes issues de la bissection précédente, pour lesquelles on calcule entièrement la valeur de ρ_M. Complexité en $O(nm)$;
2. la mise à jour de l'évaluation des boîtes non-explorées, où seule la distance à la dernière solution est calculée. Complexité en $O(np)$.

La complexité de l'étape d'évaluation de DMDFS est donc en $O(nm + np)$.

Algorithme 4 Algorithme hybride BestFS/DFS dynamique

```
...
  si w([x]) ≤ ϵ alors
    Σ ← Σ ∪ {[x]} ;
    Evaluer(ρ([x])), ∀ [x] ∈ L ;
    Trier(L, ρ) ;
  sinon
    {[x1], [x2]} ← Diviser([x]) ;
    L ← L ∪ {[x1], [x2]} ;
  finsi
...
```

4 Stratégies de recherche pour algorithmes BPA

Cette section introduit de nouvelles stratégies adaptées à l'exploration Anytime de CSP numériques. Elles s'appuient sur l'algorithme hydride BestFS/DFS dont les fonctions d'évaluations maximisent la qualité des approximations.

4.1 Interleaved Depth-First Search

Cette stratégie s'inspire des travaux de [11], une heuristique d'exploration d'arbres de recherche appelée *Interleaved Depth-First Search* (IDFS). IDFS propose une exploration hybride entre un parcours en profondeur et un parcours en largeur de l'arbre. Le parcours en profondeur permet d'atteindre rapidement les premières solutions, tandis que le parcours en largeur est réalisé par ordre croissant de la profondeur des nœuds dans l'arbre. Concrètement IDFS est assimilable à une stratégie hybride DFS/BestFS dont la fonction d'évaluation est :

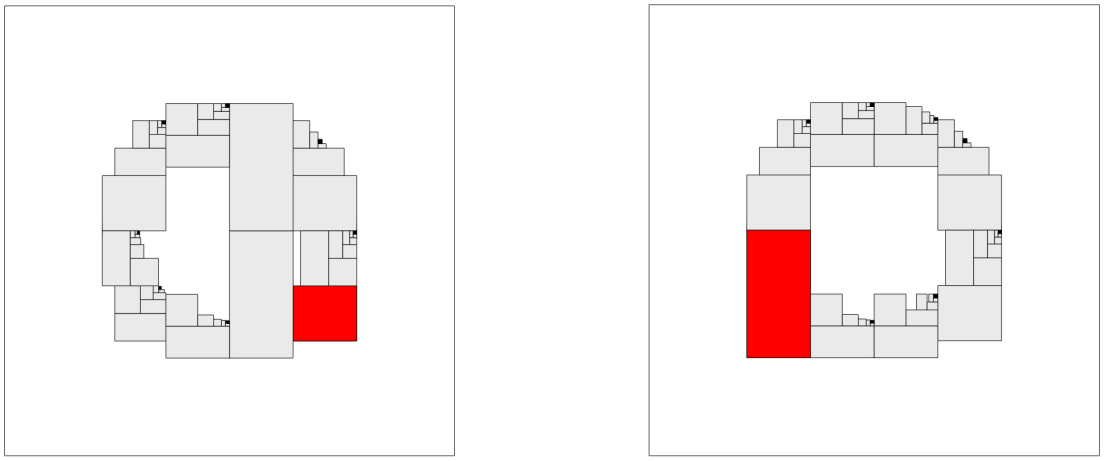
$$\rho_I([\mathbf{x}]) = \text{Profondeur}([\mathbf{x}]) \tag{3}$$

La profondeur dans l'arbre est une propriété statique de chaque boîte et IDFS suit le schéma de l'algorithme 3. L'évaluation de ρ_I a une complexité de $O(1)$ si cette information est mémorisée à chaque nœud de l'arbre de recherche.

4.2 Depth and Largest-First Search

Pour cette autre stratégie, nous exploitons l'hypothèse qui considère qu'entre deux boîtes non-explorées, celle de plus grande largeur contient probablement davantage de solutions. L'heuristique Largest-First (LF) traduit cette hypothèse et peut être évaluée par la fonction d'évaluation $\rho_L : \mathbb{I}^n \to \mathbb{R}$:

$$\rho_L([\mathbf{x}]) = w([\mathbf{x}]) \tag{4}$$



(a) DMDFS (b) DLFS

FIGURE 2 – Comportement des stratégies DMDFS et DLFS sur le problème $\mathcal{P}_1$ et identification de la prochaine boîte extraite après 7 solutions calculées.

La liste des boîtes non explorées est triée selon les valeurs de ρ_L. De la même façon que pour IDFS, l'implémentation de cette stratégie est une version statique (algorithme 3) de BestFS/DFS désignée par *Depth and Largest-First search* (DLFS). L'étape d'évaluation correspondant au calcul de ρ_L a une complexité en $O(n)$.

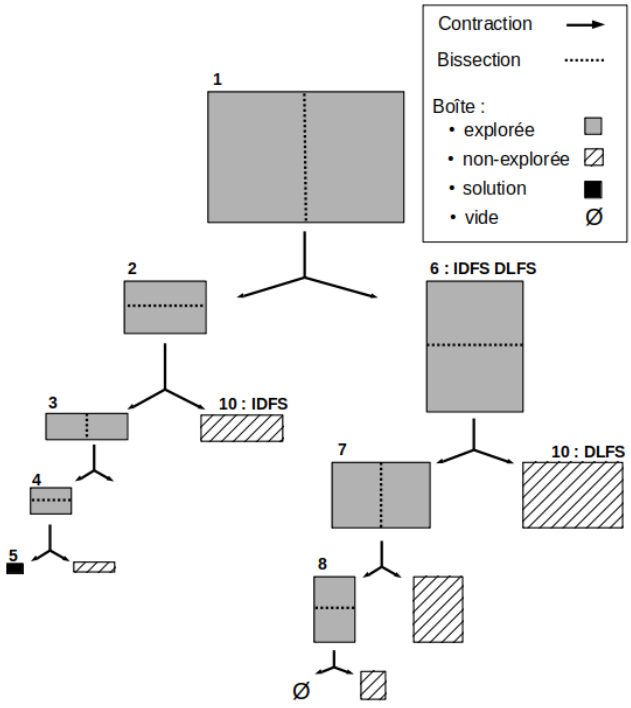


FIGURE 3 – Illustration de la différence entre IDFS et DLFS dans le parcours d'un arbre de recherche.

La figure 2 montre la différence entre DLFS et DMDFS lors de l'étape **ExtrairePremier** après 7 solutions trouvées sur le NCSP $\mathcal{P}_1$ ($\epsilon = 10^{-1}$). La figure 3 illustre la différence entre le comportement de DLFS et de IDFS lors de l'exploration d'un espace de recherche matérialisé par la boîte 1. Après le premier parcours en profondeur (1 à 5) les deux stratégies sélectionnent la boîte 6 qui est à la fois la boîte la plus large et celle de plus haut niveau dans l'arbre de recherche. Le second parcours en profondeur (6 à 9) donne une boîte vide. A ce stade, les deux stratégies divergent pour explorer la boîte la plus large pour DLFS et la boîte de plus haut niveau pour IDFS.

5 Évaluation expérimentale

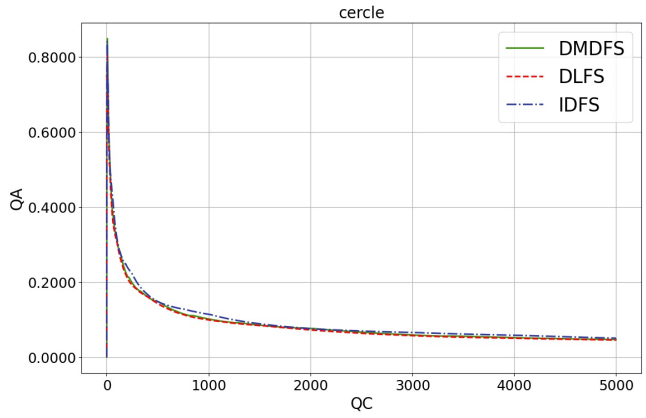
5.1 Indicateurs de qualité

Trois indicateurs de qualité sont introduits pour évaluer la performance des stratégies mises en place Soit $\Sigma = \{\sigma_1, ..., \sigma_m\}$ un ensemble de ϵ-boîtes en sortie d'un ABP. On définit :

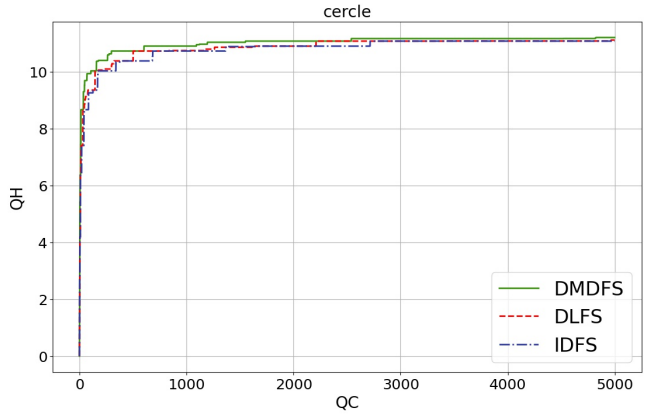
— Q_C la cardinalité m de Σ ;
— Q_H le volume de l'enveloppe $\square\Sigma$;
— Q_A la distance minimale moyenne entre les σ_k, soit :

$$Q_A = \frac{1}{m} \times \sum_{i=1}^{m} \min\{d_H(\sigma_i, \sigma_j) : 1 \leq j \leq m, j \neq i\}.$$

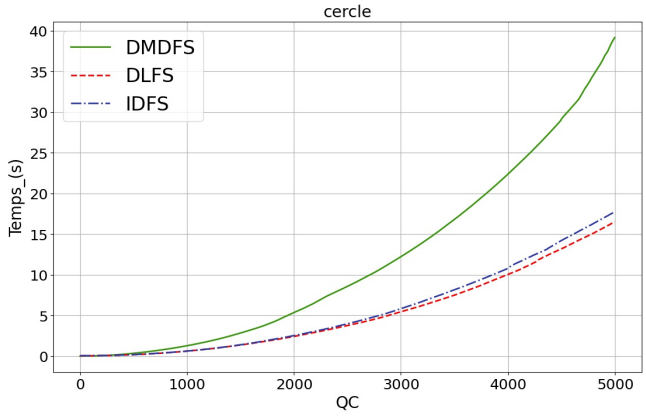
Ces indicateurs traduisent les propriétés attendues d'un algorithme de résolution anytime. Q_C représente le nombre de solutions calculées. Q_H décrit le degré de couverture de l'espace des solutions du NCSP. L'uniformité de la distribution des solutions calculées est évaluée par Q_A.



(a)



(b)



(c)

FIGURE 4 – Profils de performances des stratégies DMDFS, IDFS et DLFS en fonction de Q_C pour le problème simple $\mathcal{P}_1$ avec le scenario 0.

Nous verrons que Q_A ne peut-être comparé qu'entre deux sous-ensembles de même taille (Q_C équivalent).

5.2 Protocole

Les tests ont été réalisés sur une machine Linux Core i7-9850H 2.6 GHz (16 GB). Les algorithmes sont implémentés en C++ avec la librairie IBEX [1] basée sur l'arithmétique par intervalles. Les stratégies de contraction et de découpage des boîtes sont fixées : HC4, propagation de type AC3 et bissection de type RoundRobin (par ordre cyclique des variables).

Les NCSPs utilisés pour les tests sont tirés de la librairie GAMS [2]. Ce sont initialement des problèmes d'optimisation pour lesquels nous explorons uniquement l'espace réalisable sans tenir compte de l'objectif. Pour les problèmes non-contraints (cuspidal, ackley, rosenbrock), nous avons fixé un seuil sur l'objectif pour se ramener à un NCSP. Les tests ont été réalisés selon trois scenarii :

— Scenario 0 : nombre de solutions limité à $Q_C^{max} = 5000$, précision $\epsilon = 10^{-3}$;
— Scenario 1 : temps de résolution limité à t_{max}, précision $\epsilon = 10^{-3}$;
— Scenario 2 : nombre de solutions limité à $Q_C^{max} = 800$, précision $\epsilon = 10^{-3}$.

On distingue les problèmes dit *simples* dont les contraintes ne sont que des inéquations ($\mathcal{P}_1$, ackley, chance, cuspidal, ex7_2_10, rosenbrock), et les problèmes *complexes* qui comportent en plus des équations (bearing, alkyl, ship et himmel16).

5.3 Résultats

Les courbes de la figure 4 donnent l'évolution des indicateurs Q_H, Q_A et du temps de résolution en fonction de Q_C pour le NCSP $\mathcal{P}_1$ sur le scénario 0. Ce problème illustre les tendances des profils de performances pour les problèmes simples. On observe sur la figure 4c que la stratégie DMDFS nécessite des temps de calcul nettement plus importants pour calculer un même nombre de solutions. Ce résultat est une conséquence directe du coût de la fonction d'évaluation ρ_M par rapport aux stratégies statiques IDFS et DFLS.

La première partie des courbes 4a et 4b montrent une pente rapide suivie d'un plateau asymptotique. La qualité d'une approximation est donc fortement déterminée par les premières solutions calculées. On retrouve ce résultat sur l'ensemble des problèmes simples.

Le tableau 1 résume les résultats des scénarios 1 et 2 pour les problèmes simples avec : le nombre de variables (#V), le nombre de contraintes (#C). Les valeurs des indicateurs Q_A et Q_H ont été normalisées par $Q_i^{norm} = Q_i/Q_{max}$. Sur le scénario 1, t_{max} est déterminé comme le temps nécessaire à l'heuristique

1. http://www.ibex-lib.org/
2. https://www.gams.com/latest/gamslib_ml/libhtml/index.html#gamslib

				Scénario 1				Scénario 2		
Problème	**#V**	**#C**	**Stratégie**	**Temps (s)**	Q_C	Q_A	Q_H	**Temps (s)**	Q_A	Q_H
			DLFS	1,00	993	0,69	0,87	0,70	0,84	0,84
ackley_3	3	1	DMDFS	1,00	586	1,00	**1,00**	1,71	**1,00**	**1,00**
			IDFS	1,00	914	0,79	0,97	0,79	0,93	0,93
			DLFS	7,50	898	0,27	0,00	6,19	0,24	10^{-4}
ackley_12	12	1	DMDFS	7,50	616	0,22	0,06	12,21	0,25	0,07
			IDFS	7,51	855	1,00	**1,00**	6,77	**1,00**	**1,00**
			DLFS	20,01	834	0,01	0,00	19,11	0,01	10^{-4}
ackley_18	18	1	DMDFS	20,01	630	0,03	0,44	27,30	0,03	0,44
			IDFS	20,02	952	1,00	**1,00**	16,25	**1,00**	**1,00**
			DLFS	0,20	759	0,82	0,99	0,22	0,98	0,98
chance	4	3	DMDFS	0,20	459	1,00	0,99	0,57	**1,00**	**1,00**
			IDFS	0,20	902	0,51	**1,00**	0,16	0,66	0,99
			DLFS	1,00	792	0,83	0,88	1,02	0,94	0,88
cuspidal	3	1	DMDFS	1,00	541	1,00	0,94	2,32	**1,00**	0,94
			IDFS	1,00	882	0,73	**1,00**	0,83	0,87	**1,00**
			DLFS	5,00	786	0,73	0,65	5,19	0,79	0,66
ex7_2_10	11	9	DMDFS	5,00	431	1,00	**1,00**	18,22	**1,00**	**1,00**
			IDFS	5,00	829	0,68	0,65	4,66	0,75	0,66
			DLFS	0,30	673	0,77	0,94	0,42	0,95	0,94
rosenbrock_3	3	1	DMDFS	0,30	411	1,00	**1,00**	1,18	**1,00**	**1,00**
			IDFS	0,30	646	0,82	0,94	0,46	0,98	0,94
			DLFS	3,56	750	0,86	0,93	3,69	0,92	0,95
rosenbrock_6	6	1	DMDFS	3,50	580	1,00	**1,00**	4,68	**1,00**	**1,00**
			IDFS	3,50	699	0,93	0,89	3,81	0,99	0,87
			DLFS	-	811	0,62	0,66	4.57	0,71	0,66
Valeurs moyennes	-	-	DMDFS	-	532	0,78	0,81	8.53	0,78	0,81
			IDFS	-	835	0,81	0,93	4.22	0,90	0,92

Tableau 1 – Résultats normalisés des expérimentations sur les problèmes simples (inéquations uniquement). Scénario 1 : temps de résolution limité à t_{max}. Scénario 2 : nombre de solutions limité à $Q_C^{max} = 800$.

la plus performante pour atteindre le plateau asymptotique. En dehors de ackley_12 et ackley_18 DMDFS obtient de meilleurs résultats mais calcule moins de solutions (35% de moins en moyenne). Rapporté au temps de calcul, IDFS obtient en moyenne de meilleurs résultats sur les trois indicateurs.

L'analyse sur Q_A est moins immédiate. En effet cet indicateur décroît avec le nombre de solutions trouvées et nous cherchons l'approximation le maximisant. De fait, entre deux sous-ensembles de solutions, celui avec le Q_C le plus grand donnera, sauf exception, le Q_A le plus faible. Ainsi, on ne comparera Q_A qu'à Q_C équivalent et le scénario 1 ne permet pas de conclure sur l'indicateur Q_A.

Les tests du scénario 2 à $Q_C = 800$ montrent de meilleures performances pour DMDFS sur Q_A sauf pour les problèmes ackley_12 et ackley_18. Ce résultat est attendu puisque la fonction d'évaluation de DMDFS revient à maximiser Q_A au fur et à mesure de l'exploration. Sur l'ensemble des tests IDFS est plus performant avec en moyenne $Q_A = 0,9$. Pour l'indicateur Q_H DMDFS obtient la meilleure approximation sur 5 problèmes parmi les 8. IDFS est cependant plus efficace pour obtenir rapidement ces approximations.

Les courbes de la figure 5 illustrent les profils de performances pour les problèmes complexes. On observe que les temps de calcul dépendent énormément du problème choisi. Par exemple pour himmel16, DLFS est la stratégie la moins efficace, pour le problème alkyl il s'agit de IDFS. Au niveau des indicateurs les résultats sont disparates, on ne peut pas conclure ici. Cette observation peut être expliquée par la génération importante de boîtes non explorées lors de certains backtracks, augmentant la taille de l'arbre de recherche. De plus, une grande partie de ces boîtes vont donner des boîtes vides à l'issue de la contraction du fait des contraintes d'égalité.

Finalement, l'heuristique dynamique DMDFS présente le meilleur comportement Anytime pour des problèmes simples sans contraintes d'égalité et à nombre de solutions équivalent. Néanmoins ce type de stratégie peut conduire rapidement à des temps de calculs impor-

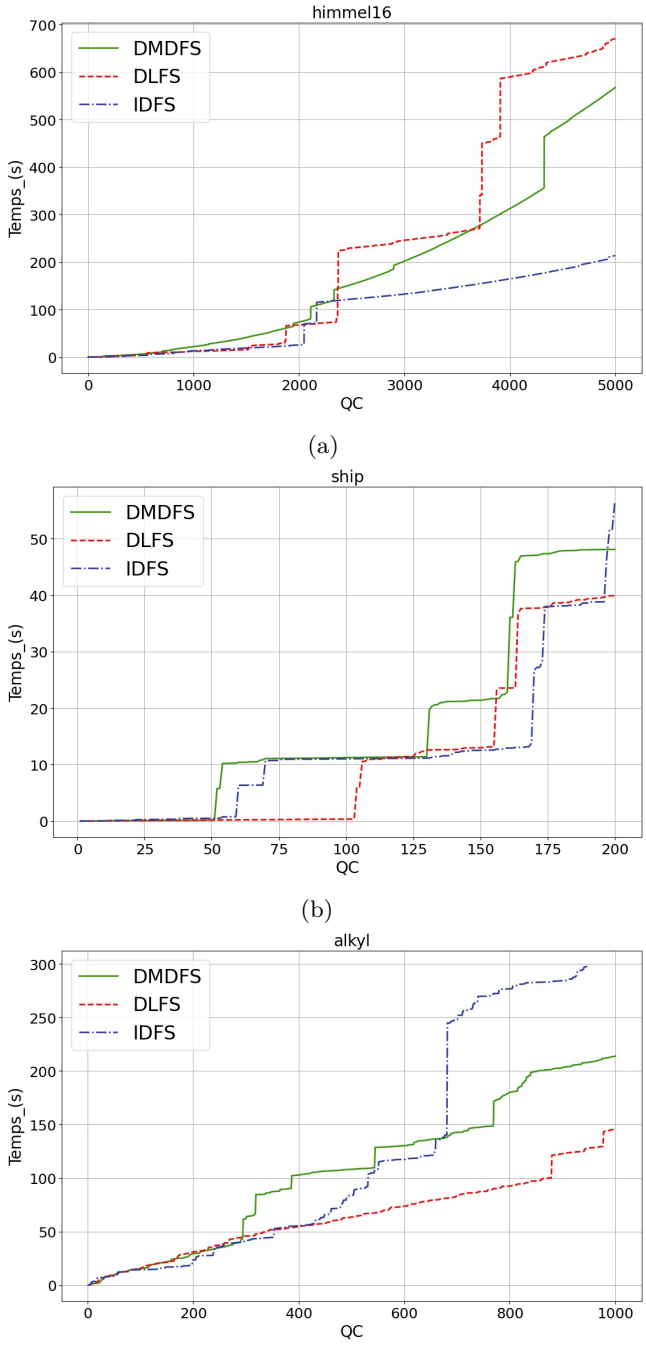


FIGURE 5 – Temps de calcul des stratégies DMDFS, IDFS et DLFS en fonction du nombre de solutions obtenue pour trois problèmes complexes avec une précision de 10^{-3}.

tants dès que l'on augmente le nombre de ϵ-boîtes dans l'approximation attendue. Sur l'ensemble de l'étude, IDFS représente le meilleur compromis entre la qualité de l'approximation (Q_H et Q_A), le temps de calcul et un nombre de solutions suffisamment grand (Q_C). Dans les conditions de notre étude et pour des NCSP non-linéaires complexes les performances des stratégies ne sont pas uniformes et dépendent largement du problème choisi.

6 Conclusion

Ce travail propose d'élargir le domaine des problèmes de satisfaction de contraintes numériques à une approche de résolution partielle, dite Anytime, déjà utilisée en optimisation. Ce nouveau paradigme demande la construction de stratégies de recherche spécifiques pour calculer des approximations représentatives de l'espace des solutions. Des indicateurs de qualité sont proposés pour évaluer ces stratégies entre elles. Q_H et Q_A traduisent l'exploration diversifiée (couverture et uniformité) de l'espace des solutions.

Deux nouvelles stratégies de recherches pour algorithmes Branch-and-Prune Anytime sont proposées : IDFS et DLFS. Elles sont comparées à la stratégie existante DMDFS. Cet article a permis de distinguer deux comportements : dynamique et statique. Cette différence se traduit par une complexité temporelle plus importante pour les stratégies dynamiques comme DMDFS. Si le temps de calcul n'est pas considéré, DMDFS obtient de meilleures évaluations pour les indicateurs Q_H et Q_A à Q_C (nombre de solutions) fixé. Dans le cas d'un nombre restreint de solutions, DMDFS peut être préférable si la taille de l'arbre de recherche ne croît pas trop vite (difficilement prévisible). L'évaluation de la complexité et les résultats observés sur un ensemble de problèmes simples, sans contrainte d'égalité, montrent les propriétés supérieures de la stratégie IDFS.

En revanche, les expérimentations ne permettent pas de conclure sur l'exploration anytime de problèmes plus complexes. Plusieurs pistes peuvent être envisagées pour attaquer ce point, notamment l'impact des étapes de division des domaines et de l'étape de tri des boîtes non-explorées pour ces stratégies BestFS. Il sera aussi pertinent d'adapter la phase de descente en profondeur DFS pour contrôler davantage l'exploration vers des solutions diversifiées.

Parmi les perspectives à court terme, l'étude de ces stratégies sur des problèmes mixtes permettra de traiter des problèmes issues de la conception préliminaire. IDFS et DLFS sont de bons candidats car la profondeur ne s'appuie pas sur les domaines des variables et la notion de largeur peut-être aisément étendue aux entiers.

Les indicateurs de comparaison proposés ici se focalisent principalement sur les ϵ-boîtes et pourraient aussi s'appliquer aux différents nœuds de l'arbre de recherche.

Dès lors, nous pouvons définir des usages beaucoup plus interactifs des solveurs. Tous ces éléments ont pour objectif, in fine, de permettre une utilisation plus active des outils d'exploration de concepts.

Références

[1] F. Benhamou, F. Goualard, L. Granvilliers et J-F Puget : Revising hull and box consistency. *International Conference on Logic Programming*, pages 230–244, 01 1999.

[2] P. Chenouard, R.and Sébastian et L. Granvilliers : Solving an Air Conditioning System Problem in an Embodiment Design Context Using Constraint Satisfaction Techniques. 4741/2007:18–32, septembre 2007.

[3] R. Chenouard, A. Goldsztejn et C. Jermann : Search strategies for an anytime usage of the branch and prune algorithm. *IJCAI International Joint Conference on Artificial Intelligence*, 01 2009.

[4] R. Dechter et J. Pearl : Generalized best-first search strategies and the optimality of a*. *Journal of the ACM*, 32(3), 1985.

[5] Merlet J-P., Chablat D., Wenger P. et Majou F. : An interval analysis based study for the design and the comparison of three-degrees-of-freedom parallel kinematic machines. *Int. J. Robotics Res.*, 23(6):615–624, 2004.

[6] L. Jaulin, O. Kieffer, M.and Didrit et E. Walter : *Applied Interval Analysis*. Springer London Ltd, août 2001.

[7] A. Jesus, L. Paquete et A. Liefooghe : A model of anytime algorithm performance for bi-objective optimization. *Journal of Global Optimization*, 05 2020.

[8] Olivier Lhomme : Consistency techniques for numeric csps. pages 232–238, 01 1993.

[9] M. Li et X. Yao : Quality evaluation of solution sets in multiobjective optimisation : A survey. *ACM Computing Surveys*, 52:1–38, 03 2019.

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

[11] P. Meseguer : Interleaved depth-first search. *In Proceedings of the Fifteenth International Joint Conference on Artifical Intelligence - Volume 2*, IJCAI'97, page 1382–1387, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[12] D. R. Morrison, Sheldon H. Jacobson, Jason J. Sauppe et Edward C. Sewell : Branch-and-bound algorithms : A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79 – 102, 2016.

[13] B. Neveu, G. Trombettoni et I. Araya : Node selection strategies in interval Branch and Bound algorithms. *Journal of Global Optimization*, 64(2): 289–304, février 2016.

[14] S. Sayin : Measuring the quality of discrete representations of efficient sets in multiple objective mathematical programming. *Mathematical Programming, Series B*, 87:543–560, 05 2000.

[15] G. Trombettoni et G. Chabert : Constructive Interval Disjunction. *In CP'07 - 13th International Conference on Principles and Practice of Constraint Programming*, 2007.

[16] P. Van Hentenryck, D. Mcallester et D. Kapur : Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34, 07 2001.

Actes JFPC 2021

Tables segmentées : un outil de modélisation efficace pour le raisonnement par contraintes

Gilles Audemard[1] **Christophe Lecoutre**[1] **Mehdi Maamar**[1]

[1] CRIL, Univ. Artois & CNRS, 62300 Lens, France
{audemard,lecoutre}@cril.fr

Résumé

Ces dernières années, il y a eu un intérêt croissant pour les structures de tables et de diagrammes de décision en programmation par contraintes (CP). Ceci est dû au caractère universel de ces structures, permettant la représentation de toute (ou groupe de) contrainte sous forme extensionnelle, et aux algorithmes de filtrage efficaces développés pour des contraintes basées sur des tables (ordinaires / étoilées / compressées / smart) et des diagrammes de décision à valeurs multiples. Dans cet article, nous proposons le concept de tables segmentées où les entrées dans les tables peuvent combiner des valeurs ordinaires, des valeurs universelles ($*$) et des sous-tables. Les tables segmentées peuvent être considérées comme une généralisation des tables compressées. Nous proposons un algorithme établissant la cohérence d'arc généralisée (GAC) sur des contraintes de tables segmentées, et montrons l'intérêt pratique, ainsi qu'en terme de modélisation, sur un problème difficile.

Abstract

These last years, there has been a growing interest for structures like tables and decision diagrams in Constraint Programming (CP). This is due to the universal character of these structures, enabling the representation of any (group of) constraints under extensional form, and to the efficient filtering algorithms developed for constraints based on (ordinary/short/compressed/smart) tables and multi-valued decision diagrams. In this paper, we propose the concept of segmented tables where entries in tables can combine ordinary values, universal values ($*$) and sub-tables. Segmented tables can be seen as a generalization of compressed tables. We propose an algorithm enforcing Generalized Arc Consistency (GAC) on segmented table constraints, and show their modeling and practical interests on a realistic problem.

La programmation par contraintes (CP) est un paradigme de modélisation qui s'est avéré efficace pour résoudre diverses formes de problèmes combinatoires, au moyen d'algorithmes d'inférence et de recherche hautement optimisés [4, 2, 14, 11]. La force de CP est sa capacité à prendre en compte tout type d'information, au stade de la modélisation, en raison de la disponibilité de structures permettant une forme universelle de représentation. Ces structures permettent d'introduire des contraintes énumérant, en extension, ce qui peut être accepté (ou non) : elles sont appelées contraintes *table*. Par exemple, en fouille de données, un utilisateur peut rechercher des motifs fréquents avec certaines fonctionnalités, ce qui peut être exprimé par une combinaison de contraintes (généralement, des contraintes arithmétiques ou de table) qui peuvent être facilement ajoutées au modèle en raison de la flexibilité de CP [8].

Il est intéressant de noter que l'efficacité pratique des algorithmes de filtrage pour les contraintes table a été considérablement améliorée au cours de la dernière décennie, conduisant à un algorithme état de l'art appelé Compact-Table [6]. Cependant, un problème majeur reste l'espace nécessaire pour stocker les tables, c'est-à-dire l'ensemble des tuples acceptés (ou interdits) par les contraintes. Pour y remédier, plusieurs formes compactes de tables ont été introduites dans la littérature, notamment les tables *étoilées* [9, 16], permettant l'utilisation de la valeur universelle '*', et les tables *compressées* [10, 17, 15], autorisant l'inclusion de sous-ensembles de valeurs dans les tuples. Les tables *sliced* [7] et les tables *smart* [13] sont deux autres représentations compactes sophistiquées.

Il est important de noter que, parfois, les tables sont simplement des choix simples et naturels pour faire face à des situations délicates. C'est le cas lorsqu'aucune contrainte (globale) adéquate n'existe ou lorsqu'une combinaison logique de (petites) contraintes doit être représentée sous forme d'une contrainte table unique pour des raisons d'efficacité. Si nécessaire, un autre

argument montrant l'importance des structures universelles comme les tables, mais aussi les MDD (Multi-valued Decision Diagrams), est la montée en puissance des techniques de tabulation, c'est-à-dire le processus de conversion des sous-problèmes en tables, à la main, à l'aide d'heuristiques [1] ou par annotations [5].

Les tables compressées généralisent les tables étoilées puisque chaque élément d'un tuple compressé peut être donné par n'importe quel sous-ensemble de valeurs (et par conséquent, l'ensemble complet de valeurs, tout comme $*$). Par exemple, le tuple compressé $\tau = (a, \{b, c\}, b, \{a, b, c\})$ capture 6 tuples *ordinaires*, parmi lesquels on trouve (a, b, b, a) et (a, c, b, c). Dans cet article, nous introduisons des tables segmentées qui généralisent les tables compressées puisque les sous-ensembles peuvent être étendus à plusieurs variables. Par exemple, $\Gamma = (\{(a, a), (a, c), (b, b)\}, *, \{(a, b), (c, c)\})$ est un tuple segmenté composé d'un premier segment représentant une (sous-)table sur deux variables, un deuxième segment avec la valeur universelle $*$ et un troisième segment représentant à nouveau une (sous-)table sur deux variables. Tout tuple ordinaire obtenu à partir du produit cartésien implicite de ces segments est représenté de manière compacte par le tuple segmenté, comme par exemple (a, a, a, a, b) et (b, b, c, c, c) . Notez qu'une forme générale de raisonnement logique avec plusieurs contraintes (segments) a été proposée dans [12].

Pour tout savoir sur les tables segmentées, nous invitons le lecteur à se référer à l'article publié dans les actes de la conférence ECAI'20 [3]. Dans cet article, le concept de contraintes de tables segmentées est introduit, ainsi qu'une vue synthétique, suivie d'une description plus détaillée, d'un algorithme appliquant GAC sur des contraintes de tables segmentées. L'intérêt en terme de modélisation et d'efficacité pratique est démontré sur un problème d'optimisation difficile consistant à générer des grilles de mots croisés (avec position libre des cases noires) appelé CD (Crosswords Design), utilisé lors des compétitions XCSP3.

Remerciements Ce travail a reçu le soutien du projet CPER Data de la région Hauts-de-France.

Références

[1] O. Akgun, I. Gent, C. Jefferson, I. Miguel, P. Nightingale et A. Salamon : Automatic discovery and exploitation of promising subproblems for tabulation. *In Proceedings of CP'18*, 2018.

[2] K.R. Apt : *Principles of Constraint Programming*. Cambridge University Press, 2003.

[3] G. Audemard, C. Lecoutre et M. Maamar : Segmented tables : An efficient modeling tool for constraint reasoning. *In Proceedings of ECAI'20*, pages 315–322, 2020.

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

[5] J. Dekker, G. Bjordal, M. Carlsson, P. Flener et J.-N. Monette : Auto-tabling for subproblem presolving in minizinc. *Constraints*, 22(4):512–529, 2017.

[6] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin et P. Schaus : Compact-Table : efficiently filtering table constraints with reversible sparse bit-sets. *In Proceedings of CP'16*, pages 207–223, 2016.

[7] N. Gharbi, F. Hemery, C. Lecoutre et O. Roussel : Sliced table constraints : Combining compression and tabular reduction. *In Proceedings of CPAIOR'14*, pages 120–135, 2014.

[8] T. Guns, A. Dries, S. Nijssen, G. Tack et L. De Raedt : Miningzinc : A declarative framework for constraint-based mining. *Artificial Intelligence*, 244:6–29, 2017.

[9] C. Jefferson et P. Nightingale : Extending simple tabular reduction with short supports. *In Proceedings of IJCAI'13*, pages 573–579, 2013.

[10] G. Katsirelos et T. Walsh : A compression algorithm for large arity extensional constraints. *In Proceedings of CP'07*, pages 379–393, 2007.

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

[12] O. Lhomme : Arc-consistency filtering algorithms for logical combinations of constraints. *In Proceedings of CPAIOR'04*, pages 209–224, 2004.

[13] J.-B. Mairy, Y. Deville et C. Lecoutre : The smart table constraint. *In Proceedings of CPAIOR'15*, pages 271–287, 2015.

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

[15] H. Verhaeghe, C. Lecoutre, Y. Deville et P. Schaus : Extending Compact-Table to basic smart tables. *In Proceedings of CP'17*, pages 297–307, 2017.

[16] H. Verhaeghe, C. Lecoutre et P. Schaus : Extending Compact-Table to negative and short tables. *In Proceedings of AAAI'17*, pages 3951–3957, 2017.

[17] W. Xia et R. Yap : Optimizing STR algorithms with tuple compression. *In Proceedings of CP'13*, pages 724–732, 2013.

Segmented Tables: an Efficient Modeling Tool for Constraint Reasoning

Gilles Audemard[1] and **Christophe Lecoutre**[1] and **Mehdi Maamar**[1]

Abstract. These last years, there has been a growing interest for structures like tables and decision diagrams in Constraint Programming (CP). This is due to the universal character of these structures, enabling the representation of any (group of) constraints under extensional form, and to the efficient filtering algorithms developed for constraints based on (ordinary/short/compressed/smart) tables and multi-valued decision diagrams. In this paper, we propose the concept of segmented tables where entries in tables can combine ordinary values, universal values ($*$) and sub-tables. Segmented tables can be seen as a generalization of compressed tables. We propose an algorithm enforcing Generalized Arc Consistency (GAC) on segmented table constraints, and show their modeling and practical interests on a realistic problem.

1 Introduction

Constraint Programming (CP) is a modeling paradigm that has been shown quite effective for solving various forms of combinatorial problems, by means of highly optimized inference and search algorithms [10, 2, 29, 19]. The strength of CP is its ability to take any kind of information into consideration, at modeling time, because of the availability of structures permitting a universal form of representation. These structures allow us to introduce constraints enumerating, in extension, what can be accepted (or not): they are called *table* constraints. For example, in data mining, a user can ask for frequent patterns together with some specific features, which can be expressed as a combination of user's constraints (typically, arithmetic or table constraints) that can be easily added to the model due to the flexible nature of CP [14].

Interestingly, the practical efficiency of filtering algorithms for table constraints has been substantially improved over the past decade, leading to the state-of-the-art Compact-Table [12], and STRbit [33]. However, one major issue remains the space required to store the tables, i.e., all the tuples that are accepted (or forbidden) by the constraints. To address it, several compact forms of tables have been introduced in the literature, notably *short* tables [17, 32], allowing the use of the universal value '*', and *compressed* tables [18, 34, 31], allowing subsets of values in tuples. Sliced tables [13] and *smart* tables [26] are two other sophisticated compact representation.

It is important to note that, sometimes, tables simply happen to be simple and natural choices for dealing with tricky situations. This is the case when no adequate (global) constraint exists or when a logical combination of (small) constraints must be represented as a unique table constraint for efficiency reasons. If ever needed, another argument showing the importance of universal structures like tables, and also MDDs (Multi-valued Decision Diagrams), is the rising of tabulation techniques, i.e., the process of converting sub-problems into tables, by hand, using heuristics [1] or by annotations [11].

Compressed tables generalize short tables since each element of a compressed tuple can be any subset of values (and consequently, the full set of values, just like '*'). For example, the compressed tuple $\tau = (a, \{b, c\}, b, \{a, b, c\})$ captures 6 *ordinary* tuples, among which we find (a, b, b, a) and (a, c, b, c). In this paper, we introduce segmented tables that generalize compressed tables since subsets are possibly extended over several variables. For example, $\Gamma = (\{(a, a), (a, c), (b, b)\}, *, \{(a, b), (c, c)\})$ is a segmented tuple composed of a first segment representing a (sub-)table over two variables, a second segment being the universal value $*$ and a third segment representing again a (sub-)table over two variables. Any ordinary tuple obtained from the implicit Cartesian product of these segments is compactly represented by the segmented tuple, as for example (a, a, a, a, b) and (b, b, c, c, c). Note that a general form of logically reasoning with several constraints (segments) was proposed in [24], but on an AC-6 [3] basis.

The paper is organized as follows. After some technical background, we introduce segmented tables and constraints. Then, we provide a synthetic view, followed by a fully detailed description, of an algorithm enforcing GAC on segmented table constraints. Before giving some perspectives and conclusions, we show the modeling and practical interest of segmented tables on a challenging problem called CD (Crosswords Design), used in XCSP3 Competitions.

2 Technical Background

A *Constraint Network* (CN) P is composed of a sequence $\texttt{vars}(P)$ of distinct variables and a set $\texttt{ctrs}(P)$ of constraints. Each *variable* x has an associated domain, $\texttt{dom}(x)$, that contains the finite set of values that can be assigned to it. Each *constraint* c involves a sequence of distinct variables, called the *scope* of c and denoted by $\texttt{scp}(c)$, and is semantically defined by a *relation*, $\texttt{rel}(c)$, that contains the set of tuples allowed for the variables involved in c. When a tuple τ is allowed (accepted) by a constraint c, we say that c is *satisfied* by τ. The *arity* of a constraint c is $|\texttt{scp}(c)|$. An *instantiation* of a sequence of variables X maps each variable $x \in X$ to a value in $\texttt{dom}(x)$. An instantiation is *complete* for P iff $X = \texttt{vars}(P)$. A *solution* of P is a complete instantiation satisfying all constraints of P; $\texttt{sols}(P)$ denote the set of solutions of P; when $\texttt{sols}(P) \neq \emptyset$, P is *satisfiable*. If $X = \langle x_1, \ldots, x_p \rangle$ and $Y = \langle y_1, \ldots, y_q \rangle$ are two sequences of p and q variables, the sequence $\langle x_1, \ldots, x_p, y_1, \ldots, y_q \rangle$ of $p + q$ variables is denoted by $X \odot Y$.

For simplicity, a pair (x, a) such that $x \in \texttt{vars}(P)$ and $a \in \texttt{dom}(x)$ is called a *literal* (of P). Let $\tau = (a_1, \ldots, a_r)$ be a tuple of values associated with a sequence of variables $\texttt{vars}(\tau) =$

[1] CRIL, Univ. Artois & CNRS, Lens, France

$\langle x_1, \ldots, x_r \rangle$. The ith value a_i of τ is denoted by $\tau[i]$ or $\tau[x_i]$. τ is *valid* iff $\forall i \in 1..r, \tau[i] \in \texttt{dom}(x_i)$. τ is a *support* on a constraint c iff $\texttt{vars}(\tau) = \texttt{scp}(c)$ and τ is a valid tuple allowed by c. When a support exists on c, c is said to be *satisfiable*. If τ is a support on a constraint c involving a variable x and such that $\tau[x] = a$, we say that τ is a *support for* the literal (x, a) on c; equivalently, we say that the literal (x, a) is supported (on c). Enforcing Generalized Arc Consistency (GAC) on a constraint c means removing all literals (values) without any support on c.

A *table constraint* c is a constraint such that $\texttt{rel}(c)$ is explicitly defined by listing the tuples that are allowed by c. Over the years, there have been many developments about compact forms of tables. Ordinary tables contain *ordinary* tuples, i.e., classical sequences of values as in $(1, 2, 0)$. Short tables can additionally contain *short* tuples, which are tuples involving the special symbol $*$ as in $(0, *, 2)$, and compressed tables can additionally contain *compressed* tuples, which are tuples involving sets of values as in $(0, \{1, 2\}, 3)$. Assuming that the tuples mentioned just above are associated with the sequence of variables $\langle x_1, x_2, x_3 \rangle$, in $(0, *, 2)$, x_2 can take any value from its domain and in $(0, \{1, 2\}, 3)$, x_2 can take the value 1 or the value 2. Smart tables are composed of *smart* tuples, which are tuples containing arithmetic expressions (column constraints). Finally, a *basic* smart table is a restricted form of smart table where column constraints are unary. In term of expressiveness, basic smart tables are equivalent to compressed tables.

3 Segmented Tables and Constraints

A segmented table contains segmented tuples that are built from so-called segments. In this section, we introduce some formal definitions before giving an illustration.

Definition 1 *A* segment*, or* tuple segment*, is a constraint γ that can take one of the three following forms:*

- $x_i = *$*, a unary tautology constraint, always holding whatever is the value assigned to x_i; it is called a* tautology *segment;*
- $x_i = a$*, a unary equality constraint, holding only when x_i is set to the value a; it is called an* equality *segment;*
- $\langle x_{i_1}, x_{i_2}, \ldots, x_{i_{r_i}} \rangle \in T$*, a table constraint, holding when the values assigned to the sequence of variables corresponds to a tuple accepted by the table T (which contains ordinary tuples of length r_i); it is called a* table *segment.*

Note that for any segment γ, $\texttt{scp}(\gamma)$ denotes the sequence of variables involved in γ; we have $\texttt{scp}(\gamma) = \langle x_i \rangle$ for equality and tautology segments, and $\texttt{scp}(\gamma) = \langle x_{i_1}, x_{i_2}, \ldots, x_{i_{r_i}} \rangle$ for table segments. The table T required for a table segment γ will be denoted by $\texttt{table}(\gamma)$. Now, we consider a sequence of r (distinct) variables $X = \langle x_1, x_2, \ldots, x_r \rangle$.

Definition 2 *A* segmented tuple Γ *over X is a sequence of segments $\langle \gamma_1, \gamma_2, \ldots, \gamma_p \rangle$ such that $X = \texttt{scp}(\gamma_1) \odot \texttt{scp}(\gamma_2) \odot \cdots \odot \texttt{scp}(\gamma_p)$. The semantics of Γ, i.e., the set of tuples represented by Γ, is given by $\texttt{sols}(P^\Gamma)$ where P^Γ is the CN defined by $\texttt{vars}(P^\Gamma) = X$ and $\texttt{ctrs}(P^\Gamma) = \{\gamma_1, \gamma_2, \ldots, \gamma_p\}$.*

In some occasions, we shall refer to the specific types of segments. This is why Γ^{tt}, Γ^{eq} and Γ^{tb} denote the respective sets of tautology, equality and table segments in Γ.

Definition 3 *A* segmented table constraint*, or* segmented constraint *for short, is a constraint c defined by a segmented table, denoted by* $\texttt{seg\_table}(c)$*, which is a set T of segmented tuples over $\texttt{scp}(c)$. The semantics of c is given by $\texttt{rel}(c) = \bigcup_{\Gamma \in T} \texttt{sols}(P^\Gamma)$.*

	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}
Γ_1	$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$	$\begin{bmatrix} b \\ a \\ b \end{bmatrix}$	$\begin{bmatrix} a \\ c \\ b \end{bmatrix}$	b	$\begin{bmatrix} a \\ c \end{bmatrix}$	$\begin{bmatrix} a \\ c \end{bmatrix}$	$*$	$\begin{bmatrix} b \\ b \\ c \end{bmatrix}$	$\begin{bmatrix} a \\ c \\ b \end{bmatrix}$	$\begin{bmatrix} a \\ c \\ a \end{bmatrix}$
Γ_2	$\begin{bmatrix} a \\ b \\ b \\ c \end{bmatrix}$	$\begin{bmatrix} b \\ a \\ a \\ b \end{bmatrix}$	$\begin{bmatrix} a \\ b \\ c \\ b \end{bmatrix}$	$\begin{bmatrix} b \\ a \\ b \\ c \end{bmatrix}$	$\begin{bmatrix} c \\ a \\ b \\ a \end{bmatrix}$	b	$*$	a	$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$	$\begin{bmatrix} b \\ c \\ a \end{bmatrix}$
Γ_3	a	c	$*$	$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$	$\begin{bmatrix} b \\ a \\ a \end{bmatrix}$	$*$	b	$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$	$\begin{bmatrix} a \\ b \\ c \end{bmatrix}$	b

Figure 1: A segmented table constraint, composed of three segmented tuples Γ_1, Γ_2 and Γ_3.

Example 1 *Let $X = \langle x_1, x_2, \ldots, x_{10} \rangle$ be a sequence of 10 variables with domains $\{a, b, c\}$. Figure 1 shows a segmented table constraint over X. Its table contains 3 segmented tuples Γ_1, Γ_2 and Γ_3. The first segmented tuple Γ_1 is defined as a sequence of five segments $\langle \gamma_{1_1}, \gamma_{1_2}, \gamma_{1_3}, \gamma_{1_4}, \gamma_{1_5} \rangle$. We have $\gamma_{1_1} : \langle x_1, x_2, x_3 \rangle \in \{(a, b, a), (b, a, c), (c, b, b)\}$, $\gamma_{1_2} : x_4 = b$, $\gamma_{1_3} : \langle x_5, x_6 \rangle \in \{(a, a), (c, c)\}$, ... As an example of tuple represented by Γ_1, we find $(a, b, a, b, a, a, a, b, a, a)$. We can observe that all segmented tuples do not overlap, i.e., do not share any tuple. Consequently, the number of ordinary tuples represented by these 3 segmented tuples is exactly $(3 \times 2 \times 3 \times 3) + (4 \times 3 \times 3) + (3 \times 3 \times 3 \times 3) = 165$. This shows the possible compression benefit of using segmented tables.*

When looking for supports of literals (in the context of a filtering procedure), one has to determine which segmented tuples are valid. Validity for a segment γ means that the constraint γ is satisfiable. Similarly, validity for a segmented tuple Γ means that Γ is satisfiable (more precisely, the set of segments/constraints in Γ is satisfiable).

Definition 4 *A segment γ is* valid *iff γ is satisfiable.*

This is the case when γ is a tautology segment, or γ is an equality segment $x = a$ with $a \in \texttt{dom}(x)$, or γ is a table segment and $\texttt{table}(\gamma) \cap \prod_{x \in \texttt{scp}(\gamma)} \texttt{dom}(x) \neq \emptyset$. Note that the intersection of $\texttt{table}(\gamma)$ with the Cartesian product of the current domains of variables in $scp(\gamma)$ is exactly the set of supports on γ, meaning that γ is satisfiable when the intersection is not empty.

Definition 5 *A segmented tuple Γ is* valid *iff Γ is satisfiable, i.e., $\texttt{sols}(P^\Gamma) \neq \emptyset$.*

Proposition 1 *A segmented tuple Γ is* valid *iff every segment in Γ is valid.*

Proof: Because, by definition, segments do not overlap (share variables), when every segment in Γ is valid, we necessarily have $\texttt{sols}(P^\Gamma) \neq \emptyset$. ■

As an illustration, let us consider again Figure 1. If b is removed from $\texttt{dom}(x_{10})$, then Γ_3 becomes clearly invalid. If a and c are respectively removed from $\texttt{dom}(x_5)$ and $\texttt{dom}(x_6)$, then Γ_1 becomes invalid because its third segment becomes invalid.

To conclude this section, do note that segmented tables are a generalization of both compressed and sliced tables. While a compressed or sliced table can be represented by a segmented table, the reverse is

not true. In particular, a compressed table constraint [18] can be seen as a particular segmented constraint where each table segment has arity 1 (whereas segmented tables allow us to use table segments of any arity). A sliced table [13] can be seen as a segmented table with exactly two segments (one for the pattern and one for the sub-table). Segemented tables can also be cast as logic programs in the 'Propria' system built over the CLP scheme [28].

4 Synthetic View of Filtering

Like many filtering algorithms developed for constraints in extensional form (i.e., using structures like tables or decision diagrams), the principle is to explore the underlying structure of the constraints so as to identify (and to delete) the literals (values) that are not supported. In this section, we present a synthetic view of an original filtering algorithm dedicated to segmented table constraints. Important implementation details (notably, the data structures) will be given in the next section.

For this high-level description, we only need to introduce the structure `gacValues`. For each variable x in the scope of the constraint c to be filtered, the filtering algorithm computes $\texttt{gacValues}[x]$, the set of values for x that are supported on c.

In Algorithm 1, Function filter() must be called (i.e., systematically triggered by the solving engine) every time a segmented table constraint c must be filtered. To start, the sets $\texttt{gacValues}[x]$ are initialized. Then, every segmented tuple of the table is iterated over: when a segmented tuple Γ is found to be valid, literals supported by Γ can be collected. After processing the segmented table, the sets $\texttt{gacValues}[x]$ represent the new domains for the variables involved in c, indirectly indicating which values must be deleted, and possibly causing a domain wipe out (i.e., an empty domain).

When a segmented tuple is valid, it remains to identify supported literals. This is the role of Lines 5-13 in Algorithm 1. For a tautology segment $x = *$, all values in $\texttt{dom}(x)$ are supported, and then can be directly collected. For an equality segment $x = a$, only the value a is supported. Finally, for a table segment, one has to identify the set `SUPs` of current supports on this segment. For each variable x involved in the segment, we can consider the projection of `SUPs` on x: $\{\tau[x] : \tau \in \texttt{SUPs}\}$ is the set of values for x occurring in one tuple of `SUPs`. These projections correspond to supported values, and then can be collected.

```
1  Function filter(c: Segmented Table Constraint)
2      gacValues[x] ← ∅, ∀x ∈ scp(c)
3      foreach Γ ∈ seg_table(c) do
4          if Γ is valid then
               // we can collect values
5              foreach x = * ∈ Γ^tt do
6                  gacValues[x] ← dom(x)
7              foreach x = a ∈ Γ^eq do
8                  add a to gacValues[x]
9              foreach γ ∈ Γ^tb do
10                 SUPs ← table(γ) ∩ ∏_{x∈scp(γ)} dom(x)
11                 foreach x ∈ scp(γ) do
12                     foreach a ∈ {τ[x] : τ ∈ SUPs} do
13                         add a to gacValues[x]
14     dom(x) ← gacValues[x], ∀x ∈ scp(x)
```

Algorithm 1: Synthetic Filtering Algorithm

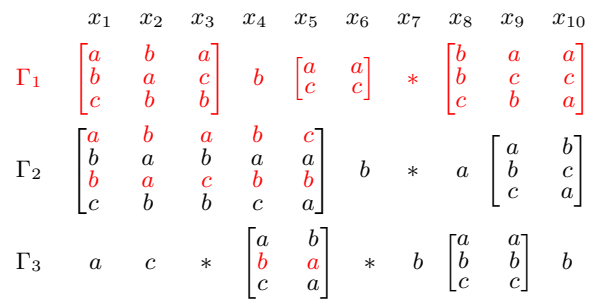


Figure 2: If b is removed from $\texttt{dom}(x_4)$, some parts of the segmented table become invalid (displayed in red color). We can then infer that $x_5 \neq c$.

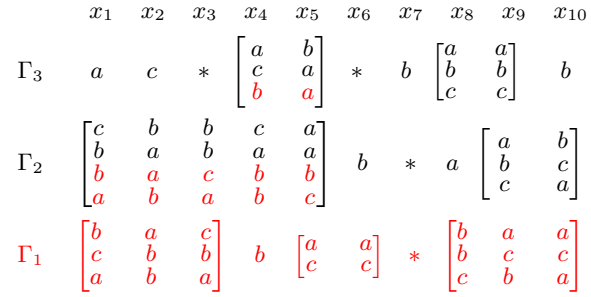


Figure 3: If b is removed from $\texttt{dom}(x_4)$, invalid parts of the segmented table are removed by swapping.

As an illustration, let us consider the first table segment of Γ_2 in Figure 1. If we suppose that b has been removed from $\texttt{dom}(x_3)$, then we have $\texttt{SUPs} = \{(a, b, a, b, c), (b, a, c, b, b)\}$. For x_1, x_2, x_3, x_4 and x_5, the supported values that can be collected are then $\{a, b\}$, $\{a, b\}$, $\{a, c\}$, $\{b\}$ and $\{b, c\}$, respectively. Now, while considering the entire segmented table, let us suppose that b has been removed from $\texttt{dom}(x_4)$, instead of $\texttt{dom}(x_3)$. We can see in Figure 2 that some parts of the segmented table become invalid: this is displayed in red color. Now, after collecting, we have $\texttt{gacValues}[x_i] = \{a, b, c\}$ for all variables x_i except for x_4 and x_5 for which we have $\texttt{gacValues}[x_4] = \{a, c\}$ and $\texttt{gacValues}[x_5] = \{a, b\}$. As $\texttt{dom}(x_4)$ was already $\{a, c\}$ (our initial assumption), after the collecting process, we can only deduce that c must be removed from $\texttt{dom}(x_5)$.

5 Detailed Description of the Algorithm

We propose now a rigorous detailed implementation. To enforce GAC on segmented table constraints, we have to deal with a main segmented table, and some secondary ordinary tables attached to table segments. Being careful about efficiency, we chose to use tabular reduction, which is a time-tested technique for dynamically maintaining tables. Indeed, based on the structure of sparse sets [7, 9], variants of Simple Tabular Reduction (STR) have been proved[2] to be quite competitive [30, 20, 21]. For the main table of segmented tuples, we maintain the set of valid segmented tuples by partitioning it in two parts. More specifically, at any time, we aim at respecting the following invariant: the segmented tuples indexed from 1 to the

[2] The state-of-the-art algorithm Compact-Table also uses tabular reduction (sparse sets) to maintain the list of non-zero words.

```
1  Method enforceGAC()
2      S^val ← {x ∈ scp : prevSizes[x] ≠ |dom(x)|}
3      S^sup ← {x ∈ scp : |dom(x)| > 1}
4      foreach x ∈ scp do
5          gacValues[x] ← ∅
6      traverseTable()
7      if tableLimit = 0 then
8          return FAILURE
9      filterDomains()
10     foreach variable x ∈ S^val ∪ S^sup do
11         prevSizes[x] ← |dom(x)|
12     return SUCCESS
13 Method traverseTable()
14     i ← 1
15     while i ≤ tableLimit do
16         if isValidSegmentedTuple(Γ_i) then
17             collectValues(Γ_i)
18             i ← i + 1
19         else                          // Γ_i must be removed
20             swap Γ_i and Γ_tableLimit
21             tableLimit ← tableLimit − 1
22 Method isValidSegmentedTuple(Γ)
23     foreach x = a ∈ Γ^eq do
24         if x ∈ S^val ∧ a ∉ dom(x) then
25             return false
26     foreach γ ∈ Γ^tb do
27         S ← S^val ∩ scp(γ)
28         if S = ∅ then
29             continue
30         i ← 1
31         while i ≤ tableLimit[γ] do
32             if isValidSubtuple(τ_i, S) then
33                 i ← i + 1
34             else                      // τ_i must be removed
35                 swap τ_i and τ_tableLimit[γ]
36                 tableLimit[γ] ← tableLimit[γ] − 1
37         if tableLimit[γ] = 0 then
38             return false
39     return true
40 Method isValidSubtuple(τ, S)
41     foreach x ∈ S do
42         if τ[x] ∉ dom(x) then
43             return false
44     return true
```

Algorithm 2: Class SegmentedConstraint

value of `tableLimit` are valid whereas segmented tuples with an index strictly greater than `tableLimit` are invalid. For simplicity, we shall denote by Γ_i the segmented tuple indexed by i in the current table at a given time. In the process of maintaining the table, if a segmented tuple Γ_i becomes invalid, it suffices to swap it with the one indexed by `tableLimit`, and then to decrement `tableLimit`; this is illustrated in Figure 3, where segmented tuples Γ_1 and Γ_3 are swapped. Similarly, for any table segment γ, the valid (ordinary) tuples are indexed from 1 to the value of $\texttt{tableLimit}[\gamma]$. In the context of a table segment γ (and so, without any ambiguity), we shall denote by τ_i the tuple indexed by i in the current table of γ.

```
1  Method collectValues(Γ)
2      foreach x = ∗ ∈ Γ^tt do
3          if x ∈ S^sup then
4              remove x from S^sup
5      foreach x = a ∈ Γ^eq do
6          add a to gacValues[x]
7          if |gacValues[x]| = |dom(x)| then
8              remove x from S^sup
9      foreach γ ∈ Γ^tb do
10         S ← S^sup ∩ scp(γ)
11         if S = ∅ then
12             continue
13         i ← 1
14         while i ≤ tableLimit[γ] do
15             foreach x ∈ S do
16                 if τ_i[x] ∉ gacValues[x] then
17                     add τ_i[x] to gacValues[x]
18                     if |gacValues[x]| = |dom(x)| then
19                         remove x from S^sup
20             i ← i + 1
21 Method filterDomains()
22     foreach variable x ∈ S^sup do
23         foreach value a ∈ dom(x) do
24             if a ∉ gacValues[x] then
25                 remove a from dom(x)
```

Algorithm 3: Class SegmentedConstraint (continued)

The class SegmentedConstraint, Algorithm 2, allows us to represent any segmented table constraint c, with the possibility of enforcing GAC at any time by simply calling Method enforceGAC(). As fields of this class we first find `scp` for representing the scope $\langle x_1, \ldots, x_r \rangle$ of c. As indicated above, for dealing with tables, we simply use `tableLimit` and $\texttt{tableLimit}[\gamma]$, while getting access to tuples with notations Γ_i and τ_i. We also have three fields $\texttt{S}^{\texttt{val}}$, $\texttt{S}^{\texttt{sup}}$ and `prevSizes` in the spirit of STR2 [20]. The set $\texttt{S}^{\texttt{val}}$ contains variables whose domains have been reduced since the previous call to Method enforceGAC() on c. To set up $\texttt{S}^{\texttt{val}}$, we need to record the domain size of each variable x right after the execution of enforceGAC() on c: this value is recorded in $\texttt{prevSizes}[x]$. The set $\texttt{S}^{\texttt{sup}}$ contains unbound variables whose domains contain each at least one value for which a support must be found. These two sets allow us to restrict loops on variables to relevant ones.

At the beginning of Method enforceGAC(), the sets $\texttt{S}^{\texttt{val}}$, $\texttt{S}^{\texttt{sup}}$ and $\texttt{gacValues}[x]$ (initially, no value has been proved to be GAC) are first initialized. Then, the main method traverseTable() is called to update tables and collect values. If after such a 'traversal', the value of `tableLimit` is 0, it means that no more segmented tuple is valid, and consequently a failure is identified. Otherwise, domains are updated by calling Method filterDomain() in order to remove the values that have not been collected in sets `gacValues`. Before returning SUCCESS (for indicating that filtering has been achieved without generating a domain wipe-out), the array `prevSizes` is modified in anticipation of the next call.

Method traverseTable() iterates over the segmented tuples from the current table (by considering indexes from 1 to `tableLimit`). When a segmented tuple Γ_i is found to be valid, Method collectValues() is called. Otherwise, Γ_i is removed from the current table.

To check whether a segmented tuple Γ is valid, Method isValidSegmentedTuple() is called. Because tautology segments are always valid, we only focus on equality and table segments. For an equality segment $x = a$, we just check if x must be tested according to $\mathtt{S}^{val}$ (although this test can be safely discarded) and if a belongs to the current domain of x. For a table segment, we start by computing the set $\mathtt{S}$ of variables occurring in both $\mathtt{S}^{val}$ and $\mathtt{scp}(\gamma)$. If ever this set $\mathtt{S}$ is empty, it means that nothing has changed for γ since the previous call to Method enforceGAC(), and consequently the table of γ is up-to-date (this is why we 'continue'). Otherwise, we iterate over the current table of the segment to only keep the tuples that are valid (tests performed by Method isValidSubtuple()). If the table becomes empty, it disqualifies the segmented tuple by returning 'false'.

Finally, Method collectValues(), Algorithm 3, allows us to collect all values that admit a support on at least a tuple. For a tautology segment $x = *$, we simply remove x from $\mathtt{S}^{sup}$ because, from now on, no more supports have to be sought for x. For an equality segment $x = a$, we indicate that a has just been found to be a support by adding a to $\mathtt{gacValues}[x]$, and we determine if x can be discarded from $\mathtt{S}^{sup}$ by comparing its size with that of $\mathtt{dom}(x)$. For a table segment, we start by computing the set $\mathtt{S}$ of variables occurring in both $\mathtt{S}^{sup}$ and $\mathtt{scp}(\gamma)$. If ever this set $\mathtt{S}$ is empty, it means that no relevant support can be found in the current table of γ (consequently, we can 'continue'). Otherwise, we iterate over the current table of γ, looking for supports of values with respect to variables in $\mathtt{S}$.

Proposition 2 *When called on a segmented table constraint c, Method* enforceGAC()*, Algorithm 2 establishes GAC on c.*

Proof: Let us suppose that the segmented table corresponds to an ordinary table, that is, every segment is an equality segment. In that case, we obtain a classical filtering STR scheme, and we know that GAC is reached. If tautology segments are also involved, the segmented table corresponds to a short table, and GAC is guaranteed [17]. Now, in case table segments are present, one can rather easily check that validity and collecting operations are correct. ■

The worst-case space complexity of representing a segmented table constraint c of arity r is as follows. First, note that the space complexity of $\mathtt{scp}$, $\mathtt{S^{val}}$, $\mathtt{S^{sup}}$ and $\mathtt{prevSizes}$ is $O(r)$. Because representing a tautology or equality segment is $O(1)$, representing all such segments is $O(rt)$ with t being the number of segmented tuples. Now, for each table segment γ, let us denote the arity and the size of the table of γ by r^γ and t^γ, and let us denote by c^{tb} the set of table segments over all segmented tuples (i.e., in the entire table). The space complexity for a segment table is $O(r^\gamma t^\gamma)$. The overall space complexity is then $O(rt + \Lambda)$ with $\Lambda = \sum_{\gamma \in c^{tb}} r^\gamma t^\gamma$. The worst-case time complexity of enforceGAC() is as follows. Without any table segment, it is $O(rt + rd)$ because in that case, handling the main table (traverseTable() is $O(rt)$ and filtering domains is $O(rd)$, where d is the size of the greatest variable domain. For any table segment γ, checking validity of tuples and collecting values is $O(r^\gamma t^\gamma)$. The overall time complexity is then $O(rt + rd + \Lambda)$.

A very useful feature of tabular reduction (more generally, sparse sets) is the possibility of restoring a table in constant time. During backtrack search, one has simply to record the table limit at each search level. When a backtrack must be performed, it suffices to change the limit in $O(1)$ by using the one recorded at the right level. For an ordinary table, backtracking is $O(1)$, but for a segmented table it is $O(k)$ where k is the number of table segments.

Finally, even when some segmented tuples overlap, the algorithm that we propose remains correct because we only consider positive tables (i.e., tuples accepted by constraints are given) in this paper.

6 Case Study: Crosswords Design

In this section, both modeling and practical benefits of using segmented table constraints are shown on a difficult optimization problem called Crosswords Design (CD). This problem was used in the COP track of the 2018 XCSP3 Competition [6, 23]. The problem is stated as follows: given a grid order n and two dictionaries, a main dictionary (containing *main* words) and an auxiliary thematic dictionary (containing *thematic* words), the objective is to fill up a grid of size $n \times n$ with words contained in these dictionaries as well as with some black points/cells (BPs, where no letter can be put). This is an optimization problem because each word w from the thematic dictionary has value $|w|$ (the length of the word), and the objective is to maximize the overall value. There is one restriction: it is not possible to have two adjacent BPs (on a row or on a column).

In what follows, we introduce and compare several models for Problem CD, using two main templates (actually, two ways of defining variables) denoted by a and b. Our aim is to compare various models so as to highlight the interest of segmented tables.

First Template. For the first template a, the variables are defined as follows:

- $x_{i,j}$, the letter put in the grid at the intersection of row i and column j, with $i \in 1..n$ and $j \in 1..n$. Possible letters are 'a', 'b', ..., 'z', and BP.
- br_i, the benefit obtained on row i according to the words formed by letters put in the ith row of x, with $i \in 1..n$. For example, if $n = 10$ and we have on row i a main word of size 3, followed by a BP and a thematic word of size 6, then $br_i = 0 + 6 = 6$.
- bc_j, the benefit obtained on column j according to the words formed by letters put in the jth column of x, with $j \in 1..n$.

The objective is to maximize $\sum_{i \in 1..n} br_i + \sum_{j \in 1..n} bc_j$. The three first models we propose for CD are called CD_a^{seg}, CD_a^{mdd}, and $\mathrm{CD}_a^{\mathrm{reg}}$ and only involve $2 \times n$ constraints because we can reason with a unique constraint per row and per column. This is rather noteworthy because do note that BPs can be put anywhere in the grid. To build constraints, we reason from valid n-patterns, where a valid n-pattern is an alternation of positive numbers and BP, summing up to n (when considering BP as being equal to 1). For example, the set of valid 4-patterns is {4, BP 3, 3 BP, BP 2 BP, 2 BP 1, 1 BP 2, 1 BP 1 BP, BP 1 BP 1}. Now, for each valid pattern, we can build several segmented tuples when considering two possibilities for each word length in the pattern: taking a word of this length from either the main dictionary or the thematic dictionary. As an illustration, let us consider that $n = 4$ and we have twenty-six 1-letter words $\{a, b, \ldots, z\}$, three 2-letter words {in, if, no}, three 3-letter words {egg, oat, tea} and four 4-letter words {cake, fish, kiwi, milk}. We assume here that the thematic words are 'kiwi' and 'tea'. Each constraint on each row involves 4 variables (since $n = 4$) and a table containing several segmented tuples built from the valid 4-patterns. This is illustrated in Figure 4 for the first row constraint, where the first pattern ('4') gives Γ_1 and Γ_2, the second pattern ('BP 3') gives Γ_3 and Γ_4 and so on. Of course, we proceed similarly with columns: there are n segmented table constraints for dealing with the n columns. Note that the constraint forbidding the presence of two adjacent BPs is directly taken into consideration by the segmented tables (tuples).

Segmented tables allow us to put different words together (i.e., in the same constraint), without memory explosion, because of the compactness of the underlying Cartesian product. At this point, it is

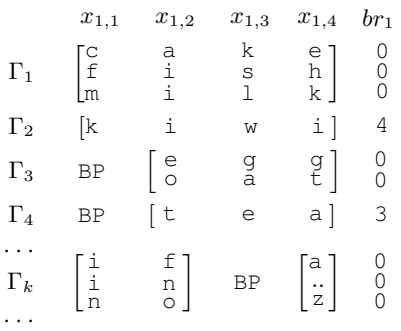


Figure 4: A Segmented Table for CD ($n = 4$).

important to note that using ordinary tables, instead of segmented tables, cannot be considered for 'large' values of n. If there are $20,000$ 7-letter words in the main dictionary, then the number of ordinary tuples generated from the pattern '7 BP 7' (when considering only words from the main dictionary) is $400,000,000$. Similarly, building an MDD while enumerating tuples is not viable. However, one can generate MDDs from segmented tables. One way to proceed is as follows. First, build an (ordered reduced) MDD from each segmented table. Second, merge all these MDDs by iteratively applying an efficient procedure [8, 27]. By replacing segmented constraints by MDD constraints (built this way), we then obtain a second model CD_a^{mdd}. Finally, if we directly consider the MDDs corresponding to the segmented tables (valid patterns) without any reduction (merging), we obtain a non-deterministic diagram for which a constraint `regular` can be used: this is model CD_a^{reg}. Although not tested in this paper, weighted automata could also be envisioned. However, we believe that required unfolding operations at propagation time might be too costly.

Second Template. For the second template b, without any loss of generality, we introduce m as being the maximal number of words put on a same row or same column. It is always possible to set m in order to avoid discarding any potential solution. For example, if $n = 5$, the maximal number of words is 3, as visible in the following pattern: 1 BP 1 BP 1 where 1 here refers to 1-letter words. So, setting $m = 3$ guarantees us that no solution can be lost.

For template b, the variables are defined as follows:

- $x_{i,j}$, defined similarly to template a.
- $br_{i,k}$, the benefit of putting the kth word on row i, from left to right, with $i \in 1..n \wedge k \in 1..m$. For example, if the second word put on row i comes from the thematic dictionary and is of length 5, then $br_{i,2} = 5$. But if instead there is just one word (of size n) put on row i, then $br_{i,2}$ is necessarily equal to 0.
- $bc_{j,k}$, the benefit of putting the kth word on column j, from top to bottom, with $j \in 1..n \wedge k \in 1..m$.

The objective is to maximize the sum of variables $br_{i,k}$ and $bc_{j,k}$.

Let us first introduce the model CD_b^{seg}. For each row i, we have exactly one segmented constraint whose scope is $\{x_{i,j} : j \in 1..n\} \cup \{br_{i,k} : k \in 1..m\}$; the arity of such constraints is then $n + m$. The segmented table contains a segmented tuple for each valid n-pattern. To build a unique segmented table per valid n-pattern, we need to relax the order imposed by Definition 2 on (variables of) segments; this is mainly a form of technical subtlety. We proceed similarly with columns. This model CD_b^{seg} is more complex than CD_a^{seg} but has the advantage of reducing the size of the segmented tables.

It is also possible to consider a more classical way of modeling, where each word is managed independently. Actually, this is the model used for generating the instances of the 2018 XCSP3 Competition. The model, called CD_b^*, of the competition, involves the three 2-dimensional arrays of variables introduced above for CD_b^{seg}, and also some additional arrays. Due to lack of space, we do not provide further details about this model (with its short tables), but the interested reader can consult the model description in [23]. It is also possible to derive MDD constraints from short table constraints: each short table is transformed into an MDD (where some arcs can be labeled with '*'). We then obtain model CD_b^{mdd*}.

Practical Evaluation. Words have been taken from some Romanian dictionaries: (i) a long *main* dictionary containing a list of $134,938$ words and (ii) a short *thematic* dictionary containing 278 words. For our experimentation, we have compared the six models described above. We have used a CP solver ($AbsCon$) that performs a classical backtrack search. For guiding search, we have used the classical heuristic `dom/wdeg` [5] and the value ordering heuristic `LastVal` that selects the last value in the domain of the selected variable. We have generated all CD instances for n ranging from 5 to 15. All experiments have been conducted on a dual socket Intel XEON X5550 quad-core 2.66 GHz with a RAM limit of 16GB.

n		seg a	mdd a	reg a	seg b	mdd* b	* b
5	bnd	***38***	*38*	*38*	*38*	*38*	*38*
	cpu	**4.9**	9.4	6.7	6.1	87	55
6	bnd	***54***	*54*	*54*	*54*	*54*	54
	cpu	**64.2**	296	159	196	461	8.2k
7	bnd	*70*	70	70	**68**	64	52
	cpu	9.5k	1.4k	1.5k	**447**	9.9k	8.3k
8	bnd	86	MO	**80**	75	76	TO
	cpu	3.6k	MO	**1.8k**	3.5k	4.9k	TO
9	bnd	91	MO	**91**	81	82	TO
	cpu	2.5k	MO	**1.6k**	1.5k	4.9k	TO
10	bnd	**110**	MO	90	92	70	TO
	cpu	9.8k	MO	7.8k	6.4k	6.8k	TO
11	bnd	**111**	MO	107	100	MO	TO
	cpu	**9.8k**	MO	2.6k	3.7k	MO	TO
12	bnd	120	MO	119	**115**	MO	TO
	cpu	6.0k	MO	3.2k	**9.5k**	MO	TO
13	bnd	113	MO	111	**138**	MO	TO
	cpu	5.4k	MO	5.6k	**4.6k**	MO	TO
14	bnd	143	MO	130	**160**	MO	TO
	cpu	7.5k	MO	9.4k	**4.1k**	MO	TO
15	bnd	156	MO	TO	**185**	MO	TO
	cpu	8.2k	MO	TO	**4.6k**	MO	TO

Table 1: Results obtained on Crosswords Design (CD) instances, for n ranging from 5 to 15. A backtrack search is run on five different models. Results are presented in terms of the best obtained bound (bnd) and the CPU time (k stands for kilo-seconds) to get it within $10,000$ seconds.

Table 1 shows the experimental results when the solver is given $10,000$ seconds to run. The best results are displayed in bold face. The best bound (bnd) found within the allowed time is indicated for each model, as well as the CPU times required to reach it. When optimality is proved, the bound is displayed in italic shape (and the CPU time indicates the total time). First, one can observe that building MDDs from segmented tables is only effective for small values of n, although we tested various orders (and chose the most rele-

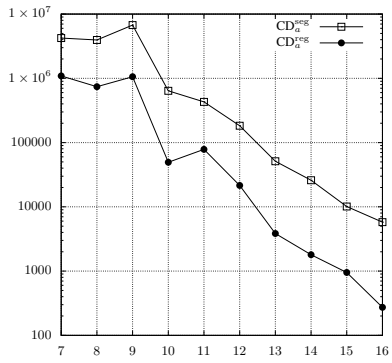


Figure 5: Number of wrong decisions (logarithmic y axis) taken in $10,000$ seconds by CD_a^{reg} and CD_a^{seg} against n (x axis).

vant one) to apply merging operations: the size of the combined reduced MDDs was quite moderate, but from $n = 8$, the size of the generated intermediate diagrams provoked a memory-out (MO). The model CD_a^{reg} does not suffer from this drawback, and even obtains quite good results for intermediate values of n (8 and 9). However, for higher values ($n \geq 10$), CD_a^{seg} and CD_b^{seg} clearly outperform the other models. Besides, we found that the good results of CD_a^{reg} for some values of n are opportunistically due to the versatility of the heuristic (different search trees are built because of minor differences in constraint weighting). Indeed, we performed an experiment with the classical heuristic `dom` [15] that guarantees that CD_a^{reg} and CD_a^{seg} perform the very same search exploration. Figure 5 shows which parts of the search trees (measured by the number of wrong decisions [4]) are explored by CD_a^{reg} and CD_a^{seg} in $10,000$ seconds, with n ranging from 7 to 16. CD_a^{seg} is between 5 and 20 times faster than CD_a^{reg}; note that we use a logarithmic scale for the y axis.

We insist that Problem CD is very challenging, and cannot be tackled by classical (ordinary) table constraints. This is why the comparison is primarily performed against MDD/regular constraints. Comparison with other forms of compressed tables is further discussed now. On the one hand, using short tables is possible, and this comparison has been made (see Model CD_b^*, rightmost column, in Table 1, and also the results obtained by all solvers at the 2018 XCSP3 competition). The algorithm behind this model is STR2 [20] which is faster than STR3 [21] and CT [12] on this particular problem (CD); actually, it appears that CT is less efficient than STR2 on Problem CD due to the huge size of domains. On the other hand, in general, compressing dictionaries with sliced tables or compressed tuples is not very effective (for example, this is shown for sliced tables in Tables 3 and 4 in [13]). Concerning Problem CD, there is absolutely no clue about how to represent rows and columns with sliced or compressed tables without any memory explosion (similarly to ordinary tables).

7 Perspectives of Segmented Tables

We have just shown how segmented tables can be the right representation choice for a specific problem. Indeed, it is rather simple and natural to express constraints of Problem CD with segmented tables. And, this modeling approach has been shown to be quite efficient in practice. However, the reader must be aware that segmented tables are not appropriate for every problem where some table constraints are involved. And it is far from being obvious how to automatically convert ordinary tables into compressed segmented ones.

Nevertheless, we think that segmented tables are really a useful modeling tool with some promising perspectives. Firstly,

v	w	x	y
a	a	a	a
a	b	a	a
a	b	a	b
b	a	a	b
b	b	a	b
b	b	b	a
a	a	b	b
b	a	b	b
b	b	b	b

x	y	z	t
a	a	a	a
a	a	a	b
a	a	b	b
a	b	b	a
b	a	a	a
b	a	a	b
b	b	a	b
b	b	b	a

$$\begin{array}{cccccc} v & w & x & y & z & t \\ \begin{bmatrix} a & a \\ a & b \end{bmatrix} & & a & a & \begin{bmatrix} a & a \\ a & b \\ b & b \end{bmatrix} & \\ \begin{bmatrix} a & b \\ b & a \\ b & b \end{bmatrix} & & a & b & \begin{bmatrix} b & a \end{bmatrix} & \\ \begin{bmatrix} b & b \end{bmatrix} & & b & a & \begin{bmatrix} a & a \\ a & b \end{bmatrix} & \\ \begin{bmatrix} a & a \\ b & a \\ b & b \end{bmatrix} & & b & b & \begin{bmatrix} a & b \\ b & a \end{bmatrix} & \end{array}$$

Figure 6: On the left, two ordinary table constraints sharing the variables (x, y). On the right, an equivalent table segmented constraint.

for certain constraints, one may identify some relevant patterns for segmentation. For example, let us consider a global constraint `allDifferent` with scope $\langle x_1, \ldots, x_r \rangle$ and $dom(x_i) = \{1, \ldots, r\}$; a permutation is enforced. This constraint can be translated into a segmented table composed of exactly $\binom{r}{r/2}$ segmented tuples. Each segmented tuple is formed by a first sub-table (all permutations of the $r/2$ selected values) followed by a second sub-table (all permutations of the $r/2$ non selected values). This means that for $r = 10$, there are 252 segmented tuples, the size of each one being equivalent to 120 tuples of arity r. Compared to the $3,628,600$ ordinary tuples, these $30,240$ equivalent "tuples" are far less memory expensive. Contrary to the constraint `allDifferent`, imposing some additional restrictions on (some of) these r variables can be envisioned by transforming further the segmented table. Note that the same kind of segmentation can be performed on other constraints (e.g., `sum`).

Secondly, segmented tables allow us to merge easily constraints having non-trivial intersections (i.e., sharing at least two variables). For example, in Figure 6, we have on the left two 4-ary ordinary table constraints whose scopes share the variables x and y. Merging these two constraints gives the segmented table constraint depicted on the right; for each instantiation of x and y, we collect two sub-tables from the two constraints. This means that enforcing GAC on this constraint is equivalent to enforcing Pairwise consistency [16] on the original pair of constraints. This opens the door to an approach for enforcing pairwise consistency (totally or partially) without any additional structures [22] or variables [25].

8 Conclusion

We have introduced the concept of segmented tables that represent a very general form of expressing constraints. Indeed, they generalize compressed tables for which subsets (sub-tables) are limited to one variable only. They also generalize sliced tables where a pattern (sub-tuple) is combined with a unique sub-table. We have presented a detailed description of a filtering algorithm. Interestingly, segmented tables could be further extended to integrate short table segments, and other arithmetic constraints (e.g., other unary constraints like in basic smart tables). This is a perspective of this work. Because of their segmented structure, developing efficient parallel filtering algorithms for segmented table constraints seems also a promising avenue. Finally, segmented tables are a new powerful modeling tool, as shown in the paper with a challenging problem, and some promising perspectives; one of them being related to pairwise consistency.

ACKNOWLEDGEMENTS

This work has been supported by the project CPER Data from the region "Hauts-de-France".

REFERENCES

[1] O. Akgun, I. Gent, C. Jefferson, I. Miguel, P. Nightingale, and A. Salamon, 'Automatic discovery and exploitation of promising subproblems for tabulation', in *Proceedings of CP'18*, (2018).
[2] K.R. Apt, *Principles of Constraint Programming*, Cambridge University Press, 2003.
[3] C. Bessiere, 'Arc consistency and arc consistency again', *Artificial Intelligence*, **65**, 179–190, (1994).
[4] C. Bessiere, B. Zanuttini, and C. Fernandez, 'Measuring search trees', in *Proceedings of ECAI'04 workshop on Modelling and Solving Problems with Constraints*, pp. 31–40, (2004).
[5] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais, 'Boosting systematic search by weighting constraints', in *Proceedings of ECAI'04*, pp. 146–150, (2004).
[6] F. Boussemart, C. Lecoutre, G. Audemard, and C. Piette, 'XCSP3: an integrated format for benchmarking combinatorial constrained problems', *CoRR*, **abs/1611.03398**, (2016).
[7] P. Briggs and L. Torczon, 'An efficient representation for sparse sets', *ACM Letters on Programming Languages and Systems*, **2**(1-4), 59–69, (1993).
[8] R. Bryant, 'Graph-based algorithms for Boolean function manipulation', *IEEE Transactions on Computers*, **35**(8), 677–691, (1986).
[9] V. Le Clément de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre, 'Sparse-sets for domain implementation', in *Proceeding of TRICS'13*, pp. 1–10, (2013).
[10] R. Dechter, *Constraint processing*, Morgan Kaufmann, 2003.
[11] J. Dekker, G. Bjordal, M. Carlsson, P. Flener, and J.-N. Monette, 'Auto-tabling for subproblem presolving in minizinc', *Constraints*, **22**(4), 512–529, (2017).
[12] J. Demeulenaere, R. Hartert, C. Lecoutre, G. Perez, L. Perron, J.-C. Régin, and P. Schaus, 'Compact-Table: efficiently filtering table constraints with reversible sparse bit-sets', in *Proceedings of CP'16*, pp. 207–223, (2016).
[13] N. Gharbi, F. Hemery, C. Lecoutre, and O. Roussel, 'Sliced table constraints: Combining compression and tabular reduction', in *Proceedings of CPAIOR'14*, pp. 120–135, (2014).
[14] T. Guns, A. Dries, S. Nijssen, G. Tack, and L. De Raedt, 'Miningzinc: A declarative framework for constraint-based mining', *Artificial Intelligence*, **244**, 6–29, (2017).
[15] R.M. Haralick and G.L. Elliott, 'Increasing tree search efficiency for constraint satisfaction problems', *Artificial Intelligence*, **14**, 263–313, (1980).
[16] P. Janssen, P. Jégou, B. Nouguier, and M.C. Vilarem, 'A filtering process for general constraint-satisfaction problems: achieving pairwise-consistency using an associated binary representation', in *Proceedings of IEEE Workshop on Tools for Artificial Intelligence*, pp. 420–427, (1989).
[17] C. Jefferson and P. Nightingale, 'Extending simple tabular reduction with short supports', in *Proceedings of IJCAI'13*, pp. 573–579, (2013).
[18] G. Katsirelos and T. Walsh, 'A compression algorithm for large arity extensional constraints', in *Proceedings of CP'07*, pp. 379–393, (2007).
[19] C. Lecoutre, *Constraint networks: techniques and algorithms*, ISTE/Wiley, 2009.
[20] C. Lecoutre, 'STR2: Optimized simple tabular reduction for table constraints', *Constraints*, **16**(4), 341–371, (2011).
[21] C. Lecoutre, C. Likitvivatanavong, and R. Yap, 'STR3: A path-optimal filtering algorithm for table constraints', *Artificial Intelligence*, **220**, 1–27, (2015).
[22] C. Lecoutre, A. Paparrizou, and K. Stergiou, 'Extending STR to a higher-order consistency', in *Proceedings of AAAI'13*, pp. 576–582, (2013).
[23] C. Lecoutre and O. Roussel, 'Proceedings of the 2018 XCSP3 competition', *CoRR*, **abs/1901.01830**, (2019).
[24] O. Lhomme, 'Arc-consistency filtering algorithms for logical combinations of constraints', in *Proceedings of CPAIOR'04*, pp. 209–224, (2004).
[25] C. Likitvivatanavong, W. Xia, and R. Yap, 'Decomposition of the factor encoding for CSPs', in *Proceedings of IJCAI'15*, pp. 353–359, (2015).
[26] J.-B. Mairy, Y. Deville, and C. Lecoutre, 'The smart table constraint', in *Proceedings of CPAIOR'15*, pp. 271–287, (2015).
[27] G. Perez and J.-C. Régin, 'Efficient operations on MDDs for building constraint programming models', in *Proceedings of IJCAI'15*, pp. 374–380, (2015).
[28] T. Le Provost and M. Wallace, 'Generalized constraint propagation over the CLP scheme', *Journal of Logic Programming*, **16**(3), 319–359, (1993).
[29] *Handbook of Constraint Programming*, eds., F. Rossi, P. van Beek, and T. Walsh, Elsevier, 2006.
[30] J. Ullmann, 'Partition search for non-binary constraint satisfaction', *Information Science*, **177**, 3639–3678, (2007).
[31] H. Verhaeghe, C. Lecoutre, Y. Deville, and P. Schaus, 'Extending Compact-Table to basic smart tables', in *Proceedings of CP'17*, pp. 297–307, (2017).
[32] H. Verhaeghe, C. Lecoutre, and P. Schaus, 'Extending Compact-Table to negative and short tables', in *Proceedings of AAAI'17*, pp. 3951–3957, (2017).
[33] R. Wang, W. Xia, R. Yap, and Z. Li, 'Optimizing Simple Tabular Reduction with a bitwise representation', in *Proceedings of IJCAI'16*, pp. 787–795, (2016).
[34] W. Xia and R. Yap, 'Optimizing STR algorithms with tuple compression', in *Proceedings of CP'13*, pp. 724–732, (2013).

Abacus : Un nouvel encodage pour SAT

Claudia Vasconcellos **Vincent Barichard** **Frédéric Lardeux**

Laboratoire d'Etude et de Recherche en Informatique d'Angers, LERIA,
EA 2645, SFR MathSTIC, Université d'Angers,
Faculté des Sciences, 2 Bd Lavoisier, 49045 Angers, FRANCE
{claudia.vasconcellos,vincent.barichard,frederic.lardeux}@univ-angers.fr

Résumé

Cet article résume le travail publié dans [7]. En général, un bon encodage doit viser à améliorer l'efficacité de la résolution (indépendamment du solveur choisi) et à produire une instance de taille raisonnable. Nous proposons l'encodage Abacus, un nouvel encodage hybride qui combine les encodages Log et Order pour offrir un bon compromis entre la taille de l'instance et l'efficacité de la résolution. Fondé sur l'idée du boulier chinois, l'Abacus décompose les valeurs entières en unités et dizaines. Les unités sont définies avec l'encodage Log et les dizaines avec l'encodage Order. Cette approche permet un bon compromis entre une représentation compacte des valeurs et l'efficacité de la résolution.

1 Introduction

Dans les dernières années, la modélisation en CSP est devenue assez intuitive grâce aux outils et langages comme [6, 5] bien que la résolution de certaines contraintes reste encore problématique. De son côté, l'approche SAT est reconnue pour la puissance de ses solveurs mais aussi pour la complexité de sa modélisation.

L'encodage d'un modèle CSP en SAT implique la modélisation de chaque variable entière et des contraintes en utilisant les variables booléennes créées sans perdre le sens de chacune des contraintes CSP dans le problème. Les encodages CSP vers SAT « standard » (Direct [3], Log [4], Support et Order [2]) se concentrent généralement sur certains aspects tels que la simplicité, la compacité ou les bonnes performances pour des contraintes spécifiques, la puissance de propagation. Les encodages « hybrides » (Direct-Support, Direct-Order, Log-Support et Compact order) combinent des encodages pour obtenir le meilleur de chacun d'entre eux (Figure 1).

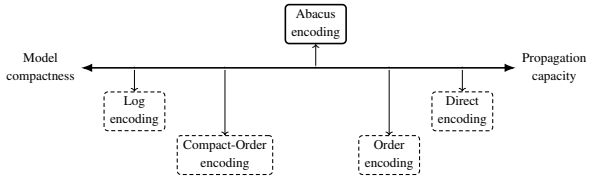


FIGURE 1 – L'encodage Abacus représente un compromis entre la taille de l'instance et la puissance de propagation

2 L'encodage Abacus

Nous proposons l'Abacus, un nouvel encodage basé sur l'idée du boulier, le célèbre outil de calcul chinois lié au système de numération décimal. Cet encodage décompose la valeur des entiers en unités et dizaines. Les unités sont traitées avec l'encodage Log et les dizaines avec l'encodage Order.

$$\underbrace{x^{b^+} \dots x^{b^-}}_{\text{Order}} \quad \underbrace{x_{(\log_2 B)-1} \dots x_1 x_0}_{\text{Log}}$$

Le nombre d'unités et de dizaines est déterminé par la *base d'encodage* B. Ce nombre correspond à une puissance de 2 afin de garder la consistance avec l'encodage Log utilisé pour les unités.

2.1 Définition de l'encodage

Pour une variable entière X définie sur le domaine $Dom(X) = [lb..ub]$, l'encodage Abacus est :

$$x^{b^+} \,..\, x^{b^-} \; x_{u-1} \,..\, x_0 \qquad (1)$$
$$b^+ = \left\lfloor \frac{ub}{B} \right\rfloor, \quad b^- = \left\lceil \frac{lb}{B} \right\rceil, \quad u = \log_2 B$$

2.2 L'offset

Pour éviter la sur-représentation des valeurs, on introduit le concept d'offset, un paramètre qui prend différentes valeurs selon le domaine de chacune des variables. Pour une variable entière X définie sur le domaine $Dom(X) = [lb..ub]$, l'offset est :

$$o^X = lb - lb \bmod B \tag{2}$$

2.3 L'encodage de l'addition

Pour une addition entière $X + Y = Z$, l'encodage se fait en deux étapes : les unités sont d'abord traitées avec des additionneurs complets et, ensuite, les dizaines sont traités selon quatre règles qui prennent en compte la retenue venant des unités. Ces règles sont :

R1 : Si les dizaines de X et Y sont vraies, alors la dizaine correspondant a Z doit être vraie aussi.

R2 : Idem que R1, mais en considérant le débordement généré au niveau de l'addition des unités.

R3 : Si les dizaines de X et Y sont fausses, alors la dizaine correspondant à Z doit être fausse aussi.

R4 : Idem que R3, mais en considérant le débordement généré au niveau de l'addition des unités.

2.4 Évaluation de l'encodage : La puissance de propagation

L'un des critères d'évaluation des codages est la puissance de la propagation unitaire. Cette métrique correspond à la quantité d'information propagée, c'est-à-dire au nombre de variables de décision affectées par chaque décision fait par le solveur.

Sur la base de l'algorithme de Brain et al. [1] pour générer automatiquement des codages complets de propagation (PCE), nous en avons développé une version simplifiée (voir l'article original). L'encodage Abacus est PCE dans le sens où il infère le maximum d'informations possible en utilisant la propagation unitaire.

3 Résultats

L'évaluation de l'encodage Abacus a été menée sur des instances des problèmes du Carré Magique et des Règles de Golomb. La taille des instances (nombre de variables) et leur puissance de propagation (nombre des variables propagées par décision) ont été les deux critères observées. Pour chaque instance différentes valeurs de base d'encodage (B) ont été testées. Nous avons donc pu répliquer le comportement des encodages Log ($B = 1$) et Order (B_{max}) et les comparer à l'encodage Abacus.

L'encodage Log fournit toujours les instances les plus petites, mais il obtient de mauvais résultats en termes de puissance de propagation. Au contraire, l'encodage Order fournit des instances de grande taille, mais avec un meilleur pouvoir de propagation.

Les meilleurs résultats correspondent aux encodages fournissant le plus grand nombre de solutions dans le temps imparti, puis, lorsque toutes les solutions sont trouvées, à ceux dont le temps d'exécution est le plus court. Ils sont principalement obtenus pour les bases qui ne correspondent pas aux encodages Log ou Order. On constate que sur les petites instances, le pouvoir de propagation contrebalance largement le nombre de solutions. Lorsque les instances deviennent plus grandes, les bases élevées limitent l'explosion du nombre de variables et de clauses.

Finalement on peut dire que les résultats obtenus pour les deux classes de problèmes montrent que l'Abacus peut établir un bon compromis entre la taille des instances générées et l'efficacité de la résolution.

Références

[1] M. Brain, L. Hadarean, D. Kroening et R. Martins : Automatic generation of propagation complete SAT encodings. *In Verification, Model Checking, and Abstract Interpretation*, pages 536–556, 2016.

[2] J. Crawford et A. Baker : Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. *In Proceedings of the 12th National Conference on Artificial Intelligence*, page 1092–1097, 1994.

[3] J. De Kleer : A comparison of ATMS and CSP techniques. *In Proceedings of the 11th IJCAI (vol. 1)*, pages 290–296, 1989.

[4] K. Iwama et S. Miyazaki : SAT-variable complexity of hard combinatorial problems. *In 13th IFIP World Computer Congress*, pages 253–258, 1994.

[5] N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck et G. Tack : Minizinc : Towards a standard CP modelling language. *In CP 2007*, pages 529–543, 2007.

[6] C. Schulte : *Programming Constraint Services : High-Level Programming of Standard and New Constraint Services*. LNCS. Springer, 2002.

[7] C. Vasconcellos-Gaete, V. Barichard et F. Lardeux : Abacus : A new hybrid encoding for sat problems. *In 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 145–152. IEEE, 2020.

Abacus: A New Hybrid Encoding for SAT Problems

Claudia Vasconcellos-Gaete, Vincent Barichard and Frédéric Lardeux
Laboratoire d'Etude et de Recherche en Informatique d'Angers, LERIA,
EA 2645, SFR MathSTIC, UNIV Angers,
Faculté des Sciences, 2 Bd Lavoisier, 49045 Angers, FRANCE
Email: {claudia.vasconcellos, vincent.barichard, frederic.lardeux}@univ-angers.fr

***Abstract*—Encoding an instance of a Constraint Satisfaction Problem (CSP) into a Propositional Satisfiability Problem (SAT) instance is usually a good way to benefit from the highly efficient SAT solvers. However, an encoding may not be suitable for all the constraints in the model because it produces a large instance, the modeling of a constraint is complicated or, it does not propagate adequately.**

In general, a good encoding should aim to improve resolution effectiveness (independently of the solver chosen) and produce an instance of reasonable size. Standard CSP-to-SAT encodings usually focus on some aspects like straightforwardness, compactness, or good performance for specific constraints. Hybrid encodings combine encodings to obtain the best from each one of them.

This article presents the Abacus Encoding, a new hybrid encoding that combines Log and Order encodings and provides a good trade-off between the instance size and the resolution effectiveness. Like the Chinese abacus, this encoding represents integer values as the addition of units and tens. Units are set with Log encoding and tens with Order encoding. This approach allows a compact representation of values, and it is easily adaptable to improve solving efficiency.

I. Introduction

Modeling a problem as a Constraint Satisfaction Problem (CSP) is usually quite intuitive, but despite the significant progress in recent years [1], [2], the resolution of some constraints is still problematic. Other paradigms, such as the Propositional Satisfiability (SAT) [3], uses highly efficient solvers [4], [5] but at the expense of the complexity, which is often transferred to the modeling. Indeed, writing the SAT model of any given problem is unnatural and typically a source of errors.

In order to exploit the simplicity of modeling in CSP and the solving power of SAT solvers, many authors have proposed model transformations between CSP and SAT [6], [7], [8]. In SAT the efficiency of solvers is well known, but the quality of models is nevertheless essential. There are several possible encodings. Some of them, as the Log encoding [9], [10], focus on compactness, but they tend to inhibit solvers capabilities as their propagation capacities. Others, like the Order [11], [12] or the Direct [13] encodings, promote the propagation during the search but to the detriment of the instance size (Figure 1).

During the last years, the need for a hybrid encoding providing instances of reasonable size and fairly efficient resolution has emerged. Several hybrid encodings have been proposed mixing Log and Support encodings [14], [15], [16], [17] or Log and Order encodings [18]. Nowadays, these hybrid encodings are either very close to Log encoding or, they propose a duplication of some parts of the model using two encodings (to favor size and an efficient resolution simultaneously).

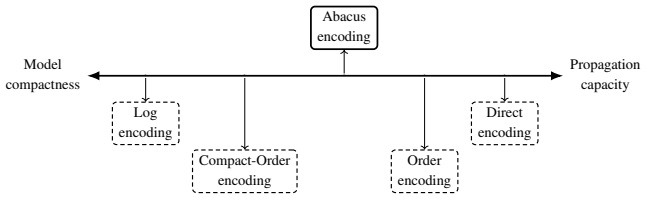


Fig. 1. Some encodings perform well on compactness and others on propagation capacity. The Abacus encoding is a good trade-off between both.

We propose a new hybrid encoding that intends to provide a good trade-off between the instance size and the solver efficiency. It is a *log-order* encoding named Abacus encoding. The name comes from the Chinese abacus, the ancient calculation tool in which the vertical rods represent digits (units, tens, or hundreds) and carries and shift are similar to the decimal number system. At first glance, the Abacus encoding might look like the Compact Order encoding [18], as both combine Order and Log encodings, but while the Compact Order encoding uses a Log-like decomposition for variables but encodes only using the Order encoding, the Abacus decomposes the variable domain, encoding one part in Log and the another in Order, so both encodings are working at the same time. At the end, the Compact Order encoding produces smaller instances, but it cannot propagate so effectively.

The article is organized as follows: Section II summarizes some of the current SAT encodings, including some hybrid encodings as well. Section III presents the definition of the Abacus encoding, its parameters, and the relation between Log, Abacus, and Order encodings. Section IV presents the rules to encode the addition using the Abacus encoding along with a preliminary experiment over equation systems. Section V reports and compares the results of our encoding for the Golomb Rulers and Magic Square problems. Finally, in Section VI, we present the main conclusions and propose some future work.

II. SAT Encodings

Encoding a CSP model into SAT involves modeling each integer variable and constraints using the created Boolean variables without losing the meaning of the CSP constraint itself.

In this section, we summarize the standard SAT encodings and we take a glance at hybrid encodings.

1) Direct encoding: Proposed by De Kleer [13], this is a straightforward SAT encoding. For each integer variable X with $Dom(X) = \{d_0 \dots d_n\}$ it creates n propositional variables x_i representing an assignment $X = d_i$. To ensure that exactly one value is chosen for each integer variable represented, the encoding adds *at-least-one* clauses ($\bigvee_0^n x_i$) and *at-most-one* clauses ($\bigwedge_{i<j}(\neg x_i \vee \neg x_j)$). In case of prohibited values ($X \neq d_i$), unary clauses of type $\neg x_i$ are posted. For two conflicting variables ($X \neq Y$), clauses of type ($\neg x_i \vee \neg y_i$) are introduced.

2) Support encoding: The Support encoding encodes variables just like in the Direct encoding but for constraints, except for *at-least-one* and *at-most-one* constraints, it encodes the support of the values (instead of the conflicts). Assigning a value to a variable in a constraint does not prohibit certain values for the other variables in the constraint (like in Direct encoding), but, on the contrary, it only allows certain values. For two conflicting variables ($X \neq Y$), clauses of type $x_i \rightarrow (\bigvee_{j \neq i} y_j)$ are introduced.

3) Log encoding: Based on the machine representation, this is a compact encoding that uses $\log_2 n$ propositional variables to represent an integer variable as a combination of power of 2 values. In this encoding, each Boolean variable x^k is true if the k-th bit of X is true. For negative values, the complement-2 representation is used. For prohibited values, it adds clauses of type $\neg(x^0 \wedge x^1 \dots \wedge x^b)$.

4) Order encoding: Proposed to model scheduling problems [11], this is an efficient encoding for handling linear constraints [19], [20], [12]. Unlike the Direct and Log encodings (where variables represent an assignment), variables in Order encoding represent a relation $X \leq d_i$. For each integer variable, the encoding uses $n - 1$ propositional variables x_i where the order relation between variables is given by a set of clauses $x_{i-1} \rightarrow x_i$. For the treatment of prohibited values $X \neq d_i$, the encoding uses the same type of clauses as in the Log encoding.

5) Hybrid encodings: The basic principle of hybrid encodings is to combine two (or more) encodings so single features can be increased; there is no single way to combine these encodings because it depends on each particular case. Some of these encodings are:

a) Direct-Support: As the Direct and Support Encodings represent integer variables in the same way, this hybrid encoding focuses on constraints, applying the encoding that provides the smaller possible output [16]. In general terms, it uses Direct encoding to encode conflicts (like in $X \neq Y$) and Support encoding to encode supports (like in $X = Y$). For inequalities like $X < Y$, the encoding considers the number of variables affected to determine the best encoding to use, always aiming to minimize the size of the output produced.

b) Direct-Order: This encoding [16] applies the Direct and Order encoding over the alldifferent() global constraint, considering its proven efficiency for certain problems. When an alldifferent() constraint is found, all the related variables are encoded in the Direct and Order encodings and then linked with translation clauses $x_i^o \leftrightarrow (\neg x_{i-1}^d \wedge x_i^d)$. For the other constraints, specialized encodings are used.

c) Log-Support encoding: Proposed by Gavanelli in [14], it uses a logarithmic number of variables, like the Log encoding. Propagation is improved by replacing conflict clauses of size $2\lceil \log_2 n \rceil$ by support clauses of size $\lceil \log_2 n \rceil + 1$.

d) Compact Order encoding: It belongs to the *log-order* encoding family [18]. Using a base $B \geq 2$, an integer variable X is represented by $\sum_{i=0}^{\lceil \log_B d \rceil} B^i x^{(i)}$ where d is the maximum value in the *Dom*(X) and $0 \leq x^{(i)} \leq B$. Each $x^{(i)}$ is then encoded in the Order encoding. This encoding reduces the size of the generated instances and propagates on the most significant digits.

III. Abacus Encoding

We propose a new encoding based on the abacus, the famous eponymous Chinese computation tool whose principle is to decompose a value into units and tens.

A. Definition

For an integer variable, the Abacus encoding decomposes its value in two parts that we call *units* and *tens*; the choice of names permits to be consistent with decimal system vocabulary. For the units, we apply the Log encoding, and for the tens, we use the Order encoding.

The choice of the combination of encodings (units Log-encoded and tens Order-encoded) intends to favor the trade-off between propagation and encoding size. The cases with units Order-encoded ("Order+Order" or "Order+Log"), produce less compact representations due to the number of variables required for units. The case "Log+Log", is compact but propagation would be affected similarly to the standard Log encoding.

To determine which part corresponds to units or tens, we add a parameter B called the *encoding base*. Considering the use of the Log encoding, this base should be a value in the power of two $B \in \{1, 2, 4, 8, 16 \dots\}$.

$$\underbrace{x^{b^+} \dots x^{b^-}}_{\text{order enc.}} \quad \underbrace{x_{(\log_2 B)-1} \dots x_1 x_0}_{\text{log enc.}}$$
Tens Units

1) Encoding: For an integer variable X with a domain $Dom(X) = [lb..ub]$, the Abacus encoding (in base B) is:

$$x^{b^+} \; .. \; x^{b^-} \; x_{u-1} \; .. \; x_0 \tag{1}$$

$$b^+ = \left\lfloor \frac{ub}{B} \right\rfloor, \quad b^- = \left\lceil \frac{lb}{B} \right\rceil, \quad u = \log_2 B$$

In this form, the encoding over-represents the domain of a variable as it considers all values between $B \times (b^- - 1)$ and $B \times (b^+ + 1) - 1$. To represent the domain exactly as it is, we introduce an *offset* parameter (Section III-A2) along with additional clauses to avoid forbidden values (Section III-A4). Note that for negative values, negative tens are generated.

2) Offset: The use of offsets limits the number of Boolean variables required to encode a variable, avoiding the over-representation of domains. Each variable can have its own offset according to its domain. The value of the offset must be a multiple of the encoding base B to ensure no gaps in the domain represented.

For a variable X with $Dom(X) = [lb..ub]$ and encoded in base B, we define the offset as:

$$o^X = lb - lb \bmod B \tag{2}$$

We notice that only the offset has to be taken into account when there are no tens. The use of offsets thus allows to save one variable (tens of index 0) and imposes that tens start at index 1 whatever the domain is; then, there will be $b^\#$ Boolean variables for the tens.

$$b^\# = \left\lfloor \frac{ub - o^X}{B} \right\rfloor \tag{3}$$

Example 1. Let $B = 8$ be the base, and variable X be a variable in the domain $[-20 \dots 33]$. The encoding with offset is as follows:

	tens							*units*			
value	56 x^7	48 x^6	40 x^5	32 x^4	24 x^3	16 x^2	8 x^1	4 x_2	2 x_1	1 x_0	offset
−20	0	0	0	0	0	0	0	1	0	0	-24
33	1	1	1	1	1	1	1	0	0	1	-24

3) Identifying the Value of the Tens: A consequence of using offsets is that variables do not always start at the same tens. To overcome this, we define the functions $val(X, i)$ to get the tens value associated to an index and, $ind(X, v)$ to return the index where a certain ten is located.

$$val(X, i) = i \times B + o^X \tag{4}$$

$$ind(X, v) = \left\lceil \frac{(v - o^X)}{B} \right\rceil \tag{5}$$

4) Forbidden Values: To avoid forbidden values because of over-representation of domains, we introduce clauses of type $\neg(x^\# \wedge \dots \wedge x^1 \wedge x_{(\log_2 B)-1} \wedge \dots \wedge x_0)$ based on the maximum domain that can be represented. For instance, in Example 1 the domain is $[-20..33]$ but it is possible to encode a domain up to $[-24..39]$ as well.

For a variable X with domain $[lb..ub]$, encoded in base B, we denote the maximum domain that can be represented as $MaxDom(X) = [L..U]$, where:

$$L = o^X \tag{6}$$

$$U = (b^\# + 1) \times B + o^X - 1 \tag{7}$$

Therefore, it is necessary to forbid all values in $[L..lb - 1]$ and $[ub + 1..U]$. Using an appropriate offset (the closest to the lower bound), the number of forbidden values should be less than $2B - 1$.

5) Ensuring consistency: We recall that Order encoding requires only one pair of consecutive variables with different values. This means that for all Boolean variables encoding the same integer variable, we impose order clauses to ensure that higher tens are always bigger or equal than the preceding tens. In Equation 8, $b_x^\#$ is the maximum tens for the variable X, $\bot$ is the logical *false* value and $\top$ is the logical *true* value.

$$\bigwedge_{i=2}^{b_x^\#} (x^i \rightarrow x^{i-1}) \qquad \forall_{i>b_x^\#}\ x^i = \bot \qquad \forall_{i\leq 0}\ x^i = \top \tag{8}$$

6) Encoding and decoding an Abacus representation: To encode an integer value $X \in [lb..ub]$ with Abacus encoding in base B, we decompose the value into offset (o^X), units (x_i) and tens (x^i).

$$\begin{aligned} &\textit{Offset} : o^X = lb - lb \bmod B \\ &\textit{Units} : \quad x_i = \frac{X}{2^i} \% B \qquad \text{with } i \in [0..(\log_2 B) - 1] \\ &\textit{Tens} : \quad x^i = i \times B + o^X \qquad \text{with } i \in [1..b^\#] \end{aligned}$$

To decode an integer value represented by the Abacus encoding, the computation is as follows[1]:

$$\begin{aligned} X = &\ max_{i\in[1..b^\#]}(x^i \times i \times B) \\ &+ o^X + \sum_{i\in[0..(\log_2 B)-1]} (x_i \times 2^i) \end{aligned} \tag{9}$$

7) Link between Log, Order and Abacus encodings: Note that with a base $B = 1$, the Abacus encoding is similar to the Order encoding with offset. Furthermore, if the base is greater than the maximum value to represent, then the Abacus encoding corresponds to the Log encoding with offset.

IV. Encoding Addition

Using standard encodings for the addition requires choosing between to generate a small instance where very few propagations are possible, or to generate an instance for which propagation is efficient at the expense of a size that is often prohibitive for solvers. As the Abacus encoding provides a good trade-off between these two options, it is therefore perfectly appropriate to this constraint.

A. Addition of Units

As unit variables are Log-encoded, this addition is handled by a series of full-adders (Figure 2), where each adder takes a pair of bits (x_i,y_i) and the incoming carry c_i (the first carry is always false $c_0 = \bot$). The full-adder was encoded in CNF using Boolean algebra and Tseitin transformations to the logical gates *and*, *or* and *xor* present in the full-adder. For the $\log_2 B$ variables corresponding to the part of units, the addition constraint is encoded with the following clauses (Equation 10):

[1] $max_{i\in I}(f(i))$ returns $f(i)$ such that $\forall (i, i') \in I^2,\ f(i) > f(i')$

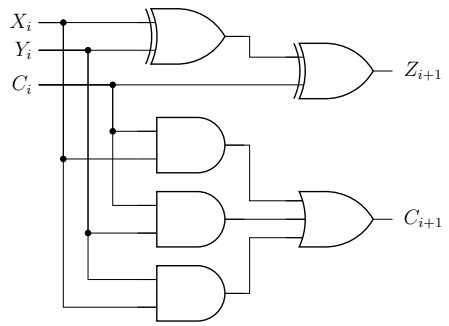


Fig. 2. Two-bit full adder schema

$$\bigwedge_{i\in[0..(\log_2 B)-1]} \begin{pmatrix} (x_i \vee y_i \vee \neg c_{i+1}) \wedge \\ (\neg x_i \vee \neg y_i \vee c_{i+1}) \wedge \\ (x_i \vee \neg z_i \vee \neg c_{i+1}) \wedge \\ (\neg x_i \vee z_i \vee c_{i+1}) \wedge \\ (y_i \vee \neg z_i \vee \neg c_{i+1}) \wedge \\ (\neg y_i \vee z_i \vee c_{i+1}) \wedge \\ (c_i \vee x_i \vee \neg c_{i+1}) \wedge \\ (\neg c_i \vee \neg x_i \vee c_{i+1}) \wedge \\ (c_i \vee y_i \vee \neg c_{i+1}) \wedge \\ (\neg c_i \vee \neg y_i \vee c_{i+1}) \wedge \\ (c_i \vee \neg z_i \vee \neg c_{i+1}) \wedge \\ (\neg c_i \vee z_i \vee c_{i+1}) \wedge \\ (c_i \vee x_i \vee y_i \vee \neg z_i) \wedge \\ (\neg c_i \vee \neg x_i \vee \neg y_i \vee z_i) \end{pmatrix} \quad (10)$$

The Abacus encoding does not use a sign bit (like in the classical full-adder), because the unit part is always positive. In case of overflow (when the sum of the units is greater than or equal to B), we store it in a variable $c = c_{\log_2 B}$ that will pass to the addition of tens.

B. Addition of Tens

To define the encoding rules for the addition of the tens, we take as example the addition $X + Y = Z$ where X, Y, and Z are three integer variables encoded with the Abacus encoding in base B and offsets o^X, o^Y and o^Z, respectively. For the sake of simplicity, we define k as the index in Z of the partial addition computed by the indices i and j (related to X and Y, respectively).

$$\begin{aligned} k &= ind(Z, val(X,i) + val(Y,j)) \\ i_{min} &= min(0, (ind(Z, o^Z) - ind(Y, o^Y))) \\ i_{max} &= max(b_X^{\#} + 1, ind(val(Z, b_Z^{\#}) - val(Y, b_Y^{\#}))) \\ j_{min} &= min(0, (ind(Z, o^Z) - ind(X, o^X))) \\ j_{max} &= max(b_Y^{\#} + 1, ind(val(Z, b_Z^{\#}) - val(X, b_X^{\#}))) \end{aligned}$$

We recall that some variables corresponding to tens out of the domain are replaced by $\top$ or $\bot$ as indicated in the *encoding consistency* part of Section III-A5.

The encoding of the tens is entirely defined by the following four rules [2]:

$$\forall\ i \in [i_{min}..i_{max}],\ j \in [j_{min}..j_{max}]$$
$$(x^i \wedge y^j) \rightarrow z^k \quad (11)$$
$$(x^i \wedge y^j \wedge c) \rightarrow z^{k+1} \quad (12)$$
$$(\neg x^i \wedge \neg y^j) \rightarrow \neg z^k \quad (13)$$
$$(\neg x^i \wedge \neg y^j \wedge \neg c) \rightarrow \neg z^{k-1} \quad (14)$$

- Rule 11: If a ten of X and a ten of Y are true, then the ten of Z corresponding to the sum of the two true tens must be true.
- Rule 12: Same as Rule 11, but considering the overflow coming from the addition of units.
- Rule 13: If a ten of X and a ten of Y are false, then the ten of Z corresponding to the sum of the two false tens must be false.
- Rule 14: The same as Rule 13, but considering the overflow which is coming from the addition of units.

Note that these rules may generate unit clauses in specific situations. The generated instance is, therefore, easily simplified using the unit propagation mechanism (UP). The unit propagation mechanism assigns the value of a literal l appearing in a clause having only one literal to make this clause true. All clauses where l appears are deleted, and those where $\neg l$ appears are simplified by removing $\neg l$ from its literals. In the following, we will present the raw instances but also their simplification by unit propagation in order to indicate the number of variables and clauses that require some work from the solvers.

C. Structural Analysis

To analyze the impact of the Abacus encoding on adder constraints, we performed a statistical study on 1000 equation systems of the form:

$$\begin{cases} X + Y = Z \\ X + Y = W \end{cases}$$

where each domain is randomly chosen in [0..100]. Of course, it may happen that some systems thus generated do not have any solution. Each equation system is generated with four different bases. Base 1 to simulate the Order encoding (all values are represented using the tens). Bases 4 and 8 to exploit different units and tens balance of the Abacus encoding. Finally, a base that we note max, which is the base simulating the Log encoding (all values are represented using units). This max bound can be different from one system to another, depending on the domains.

Table I summarizes the results obtained. These instances were directly generated and then solved by a SAT solver. Results are presented in terms of the encoding base B, the CNF structure and the propagation power. The CNF structure is expressed in number of variables (*var*) and clauses (*clauses*), before and after Unit Propagation (noted as *Raw* and *UP* respectively). For the Propagation Power, we use the *prop/dec* ratio which corresponds to the number of propagations over the number of decisions made by the solver. The columns *SAT* and *UNSAT* are the percentage of instances trivially (i.e., without decision) proved.

Results in Table I shows that the higher the base B is, the more the number of variables and clauses decreases. Contrarily, when the base simulates a Log encoding ($B = max$), many variables are not necessary (units of high degrees and carries for additions), and their number increases. This is also the case for clauses when more values have to be forbidden. Concerning the propagation power ($prop/dec$), we observe that the lower the base is, the higher the number of propagations per decision is. This propagation power allows the encoding in base $B = 1$ to trivially solve 57.9% of the equation

[2]Here, min and max correspond to the classical functions returning respectively the minimum and the maximum of the parameters.

TABLE I
ADDITION RESULTS

B	CNF Structure				Propagation Power		
	Raw		UP				
	var	clauses	var	clauses	prop/dec	SAT(%)	UNSAT(%)
1	102.5	6400.5	57.5	1457.8	28.9	1.7	56.2
4	39.1	538.7	27.5	236.0	13.6	0.0	51.0
8	32.5	275.9	26.4	177.2	9.8	0.0	44.2
max	42.4	614.5	38.8	589.6	7.4	0.0	0.0

systems. On the contrary, when base $B = max$, no equation system is trivially solved. This result is consistent with the fact that the Log encoding can only propagate the upper bounds of the domains, whereas other encodings, like the Direct encoding can propagate both the lower and upper bounds of the domains.

We rediscover the well-known properties of the Order and Log encodings: propagation power versus instance size. We see here that Abacus encoding using bases between 1 and max provides a trade-off between these two properties.

Variables and clauses removed by unit propagation increase as the base decreases. For example, for base $B = 1$, the number of variables is divided by 2 and the number of clauses by 4, whereas for high bases, these numbers only slightly decrease. Nevertheless, the ranking observed on the raw instances remains the same on those instances reduced by unit propagation. The simplified version by unit propagation corresponds to the real encoded instance where unit clauses have been removed (actually, this simplification is automatically done by all SAT solvers).

D. Complete Propagation

One of the criteria to evaluate encodings is the power of unit propagation. This metric corresponds to the quantity of information propagated, i.e., the number of decision variables affected as a result of the application of unit propagation.

Based on the algorithm of Brain et al. [21] to generate automatically propagation complete encodings (PCE), we developed a simplified version of it (see Algorithm 1). We show that our Abacus encoding is PCE in the sense that it infers the maximum possible amount of information by using unit propagation.

Our algorithm browses through all partial assignments, applies unit propagation, and checks that no new information can be inferred, stopping as soon as a counter-example is found. As inputs, it takes an encoding E (the CNF formula with the Abacus encoding of a given problem) and a set of Boolean variables Σ (the set of Boolean variables from E). It returns *true* if the encoding is PCE and *false* otherwise.

For each partial assignment, we examine the variables v that cannot be inferred by unit propagation. We test if v or $\neg v$ can be added to the partial assignment using an oracle (the SAT solver) that tells us whether or not the problem is still satisfiable.

We tested the case $x+y = z$, where x, y and z are integer variables encoded with the Abacus encoding. Algorithm 1 only proves the propagation completeness of encoding for a given number of bits. To cover the majority of the basic cases, we vary on the number of bits, offsets and bases. To build a test case, we fix the number of bits, a base (the same for all variables), and an offset for each variable. Then, we post the Abacus addition constraint $x + y = z$. Table II summarizes the basic cases tested here.

Algorithm 1 proves the completeness of each tested case. It is possible to do the calculation for a different number of bits and different base and offset values, but the runtime quickly becomes prohibitive.

Algorithm 1 Testing a Propagation Complete Encoding

Input: Σ, E
Output: *true/false*
  $\{pa$: partial assignment$\}$
  for all $pa \in \Sigma$ **do**
    $\{pa'$ is empty iff UP(E)(pa) = UNSAT$\}$
    $pa' \leftarrow$ UP(E)(pa)
    if $pa' \neq \emptyset$ **then**
      for all $v \in \{x | x \in \Sigma \; and \; x \notin pa'\}$ **do**
        for all $l \in \{v, \neg v\}$ **do**
          $pa'' \leftarrow pa' \sqcap assign(l)$
          if SATSolver(E, pa') = UNSAT **then**
            return FALSE
          end if
        end for
      end for
    end if
  end for
  return TRUE

TABLE II
PROPAGATION COMPLETE ENCODING TEST CASES

Bits	Bases	Offsets
4	$b = \{1, 2, 4, 8, 16\}$	$o^x = \{-b, 0, b\}$ $o^y = \{-b, 0, b\}$ $o^z = \{-b, 0, b\}$
5	$b = \{1, 2, 4, 8, 16, 32\}$	$o^x = \{-b, 0, b\}$ $o^y = \{-b, 0, b\}$ $o^z = \{-b, 0, b\}$

V. EXPERIMENTAL RESULTS

To verify our assumptions, we tested the Abacus encoding on two problems, the Magic Square and the Golomb Rulers; for each of them, we provide a results table and a discussion on the experiments.

For solving the SAT instances, we use `bc_minisat_all 1.1.2` [22], a blocking AllSAT solver based on Minisat [4]. The solver was compiled with the `continuous` macro option in order to force it to search from the point where the last solution was found. This makes the study of the proposed encoding less dependent on the heuristics of the solver.

All the experiments were carried out on a computing cluster with Intel-E5-2695 CPUs and 128 GB of memory. Each run had a dedicated processor, and the execution time was limited to 2 hours.

A. Magic Square problem

The *Magic Square* (MS) is a mathematical puzzle that aims to find an assignment of different natural values $x \in [1 \dots N^2]$ disposed

in a $N \times N$ matrix so that the sum across the rows, columns and diagonals always results in the constant value M called the *magic number* ($M = N(N^2+1)/2$) [23].

A basic CSP model for the Magic Square is detailed in Equations 15, 16, 17, and 18. Each x_{ij} is an integer CSP variable in the domain=[0..N^2] that represents the cell at row i and column j.

$$\forall\, i \in [1..N] \quad \sum_{j=1}^{N} x_{ij} = M \tag{15}$$

$$\forall\, j \in [1..N] \quad \sum_{i=1}^{N} x_{ij} = M \tag{16}$$

$$\sum_{i=1}^{N} x_{ii} = \sum_{i=1}^{N} x_{i(N-i+1)} = M \tag{17}$$

$$\forall\, i \in [1..N^2] \quad \texttt{AllDifferent}(x_i) \tag{18}$$

The rotations of the values in the grid produce symmetries. Our model considers four symmetry breaking constraints to establish an order relationship between the values at the corners of the square: $x_{11} < x_{N1}$, $x_{11} < x_{1N}$, $x_{11} < x_{NN}$ and $x_{1N} < x_{N1}$.

B. Golomb Rulers

The *Golomb Rulers problem* of order N and length L consists of finding a sequence of N integer values such that no two pairs of integers are the same distance apart and the largest difference between two of these values is L.

A basic CSP model for the Golomb Rulers problem is detailed in Equations 19 and 20. Each x_i is an integer CSP variable with a domain $[0..L]$.

$$x_i \in [0..L] \quad i \in [1..N]$$

$$\forall\, i,j \in [1..N], i < j, \quad x_i < x_j \tag{19}$$

$$\forall\, i \neq j \quad \texttt{AllDifferent}(x_j - x_i) \tag{20}$$

Some couples (N,L) cannot produce a correct sequence. On the contrary, when there is a sequence solution, there are always symmetrical solutions. To avoid these symmetries, the first value of the sequence is fixed at 0 ($x_1 = 0$), and the distance between the first two integers of the sequence must be less than the distance between the last two ones ($x_1 - x_0 < x_L - x_{L-1}$).

C. Results

Tables III and IV summarize the results obtained after applying the Abacus encoding to several instances of the Magic Square and Golomb Rulers. Instances are named ms$X$ and gX respectively; in both cases, the "X" denotes the order of the instance.

For each instance, we test the Log, Order and Abacus encodings. For the Abacus, we use different encoding bases (noted as *Abacus*B, with "B" indicating the base). From a structural point of view, the number of variables (*var*) and the number of clauses (*clauses*) is provided. From a resolution point of view (running `bc_minisat_all`), the number of solutions found during the run (*sol*), the propagation power (*prop*/*dec*) as well as the execution time in seconds (*time*) are provided; a "-" means that resolution was not completed after the 2 hours authorized time. These results are proposed for raw instances (*Raw*) but also for simplified instances (application of unit propagation until a fixed point is reached) noted *UP*. Best results in terms of the number of solutions (*sol*) and then execution times (*time*) are highlighted in bold for Raw and UP instances. Note that for the Golomb Rulers results (Table IV).

We observe in Tables III and IV that the instance sizes, as well as the propagation power, differ a lot from one base to another. These results are in line with those obtained in Section IV-C, despite the presence of other constraints different from the addition constraint.

The Log encoding always provides the smallest instances, but it obtains poor results in terms of propagation power. On the contrary, the Order encoding provides large instances, but with a better propagation power.

Best results correspond to encodings providing the highest number of solutions within the given time limit and then, when all solutions are found, to those with the shortest running time. They are mainly obtained for bases that do not correspond to the Log or the Order encodings. Note that on small instances (ms4 and g7), the propagation power widely counterbalances the number of variables and clauses for small bases. When the instances become larger (ms7, ms8, and g10), the high bases limit the explosion of the number of variables and clauses.

We can observe that an encoding base bigger than this corresponding to the Log encoding (instance g7 in Table IV with Abacus128) increases the number of forbidden values and then the number of variables and clauses.

Overall, the results show a direct relationship between the instance size and the encoding base. We notice that an intermediate base between those that simulate Order encoding and Log encoding, provides a good trade-off between instance size and propagation power, so as the best results.

VI. Conclusion and Future Work

In this article, we presented the Abacus encoding, a new hybrid encoding to model a CSP into a SAT problem. It combines the Log and Order encodings and decomposes CSP variables into units and tens; units are Log-encoded and tens are Order-encoded. An in-depth study has been carried out on the adder constraint for this new encoding. Extensive experiments shows that this encoding is a good trade-off between the propagation power and the instance size.

The critical point in the Abacus encoding is the choice of the base, as it allows to favor either the instance size or the propagation power. Our experiments show that using an intermediate base (between those those that simulate Order encoding and Log encoding) allows to obtain good results. In this article, we propose to use a common base for all the variables, as the treated problems have CSP variables with quite similar domains. In our future work, we intend to study the possibility of using different bases depending on the CSP variables and constraints. This could be suitable for problems where variable domains are quite different. In terms of constraints, in this article we focused on the addition constraint, but more constraints are required to model more varied CSP problems. The ongoing work is about the proposal of new Abacus encoded CSP constraints. We aim to gather a set of core constraints that will be sufficient to model more CSP problems.

References

[1] C. Schulte, *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, ser. Lecture Notes in Computer Science. Springer, 2002.

[2] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack, "Minizinc: Towards a standard CP modelling language," in *Principles and Practice of Constraint Programming – CP 2007*, 2007, pp. 529–543.

[3] M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W.H. Freeman & Company, 1979.

[4] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, 2004, pp. 502–518.

[5] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern SAT solvers," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 2009, p. 399–404.

[6] C. Bessière, E. Hebrard, and T. Walsh, "Local consistencies in SAT," in *Theory and Applications of Satisfiability Testing*, 2004, pp. 299–314.

[7] F. Bacchus, "GAC via unit propagation," in *Principles and Practice of Constraint Programming – CP 2007*. Springer Berlin Heidelberg, 2007, pp. 133–147.

TABLE III
MAGIC SQUARE RESULTS

Inst.	Encoding	CNF Structure				Propagation Power		
		Raw		UP				
		var	clauses	var	clauses	sol	prop/dec	time
ms4	**Order**	8 322	135 725	6 680	73 550	880	606	**12,14**
	Abacus4	2 670	16 315	2 300	13 984	880	246	19,23
	Abacus8	1 986	9 681	1 670	8 138	880	167	17,25
	Log	1 838	8 688	1 470	6 966	880	142	13,47
ms5	Order	31 599	730 328	26 936	425 620	18 704	1 139	-
	Abacus4	9 182	72 762	8 203	65 291	29 164	429	-
	Abacus8	5 877	32 786	5 137	28 949	19 780	298	-
	Log	4 153	19 008	3 465	16 079	**29 872**	212	-
ms6	Order	98 984	2972 594	87 730	1806 558	4 972	2 196	-
	Abacus4	26 636	257 130	24 402	231 472	**20 811**	725	-
	Abacus8	15 214	95 642	13 640	85 580	11 220	513	-
	Log	7 538	33 258	6 322	28 618	20 249	339	-
ms7	Order	267 309	9866 264	243 308	6152 704	2 630	3 012	-
	Abacus4	70 031	806 688	65 444	741 341	3 851	1 148	-
	Abacus8	38 228	277 388	35 186	254 501	**11 187**	761	-
	Log	14 404	62 148	12 401	55 379	7 645	452	-
ms8	Order	642 534	28078 259	596 000	17806 396	1 830	4 562	-
	Abacus4	167 620	2255 276	158 808	2141 176	**8 959**	1 495	-
	Abacus8	88 962	736 901	83 350	697 130	7 476	796	-
	Log	25 706	108 990	22 508	99 772	2 570	604	-

[8] F. Lardeux, E. Monfroy, E. Rodriguez-Tello, B. Crawford, and R. Soto, "Solving complex problems using model transformations: From set constraint modeling to SAT instance solving," *Expert Syst. Appl.*, vol. 149, 2020.

[9] K. Iwama and S. Miyazaki, "SAT-variable complexity of hard combinatorial problems," in *Proceedings of the 13th IFIP World Computer Congress*, 1994, pp. 253–258.

[10] N.-F. Zhou and H. Kjellerstrand, "Optimizing SAT encodings for arithmetic constraints," in *Principles and Practice of Constraint Programming*, 2017, pp. 671–686.

[11] J. M. Crawford and A. B. Baker, "Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems," in *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 2)*, 1994, p. 1092–1097.

[12] N. Tamura, M. Banbara, and T. Soh, "Compiling pseudo-boolean constraints to SAT with order encoding," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, 2013, pp. 1020–1027.

[13] J. De Kleer, "A comparison of ATMS and CSP techniques," in *Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI), Vol. 1*, 1989, p. 290–296.

[14] M. Gavanelli, "The Log-Support Encoding of CSP into SAT," in *Principles and Practice of Constraint Programming – CP 2007*, 2007, pp. 815–822.

[15] V. Nguyen, M. N. Velev, and P. Barahona, "Application of hierarchical hybrid encodings to efficient translation of CSPs to SAT," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, 2013, pp. 1028–1035.

[16] M. Stojadinović and F. Marić, "meSAT: Multiple encodings of CSP to SAT," *Constraints*, vol. 19, pp. 380–403, 10 2014.

[17] T. Soh, M. Banbara, and N. Tamura, "A hybrid encoding of CSP to SAT integrating order and log encodings," in *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, 2015, pp. 421–428.

[18] T. Tanjo, N. Tamura, and M. Banbara, "Azucar: A SAT-based CSP solver using compact order encoding," in *Theory and Applications of Satisfiability Testing – SAT 2012*, 2012, pp. 456–462.

[19] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara, "Compiling finite linear csp into SAT," *Constraints*, p. 254–272, 2009.

[20] T. Tanjo, N. Tamura, and M. Banbara, "Proposal of a compact and efficient sat encoding using a numeral system of any base," in *Proceedings of the 1st International Workshop on the Cross-Fertilization Between CSP and SAT (CSPSAT)*, 2011.

[21] M. Brain, L. Hadarean, D. Kroening, and R. Martins, "Automatic generation of propagation complete SAT encodings," in *Verification, Model Checking, and Abstract Interpretation*, 2016, pp. 536–556.

[22] T. Toda and T. Soh, "Implementing efficient all solutions SAT solvers," *J. Exp. Algorithmics*, 2016.

[23] H. Derksen, C. Eggermont, and A. Van Den Essen, "Multimagic squares," *American Mathematical Monthly*, no. 8, pp. 703–713, 2007.

TABLE IV
GOLOMB RULERS RESULTS

Inst.	Encoding	CNF Structure				Propagation Power		
		Raw		UP				
		var	clauses	var	clauses	sol	prop/dec	time
g7	Order	15 343	215 715	14 476	142 605	5	439,42	1,31
	Abacus2	8 140	70 247	7 555	60 288	5	463,14	0,74
	Abacus4	4 549	26 437	4 108	23 097	5	428,53	0,58
	Abacus8	3 065	14 197	2 678	12 450	5	256,27	0,39
	Abacus16	2 484	10 519	2 115	9 165	5	159,74	**0,22**
	Abacus32	2 204	9 191	1 841	7 926	5	244,76	0,30
	Log	2 225	9 464	1 821	7 975	5	221,00	0,33
	Abacus128	2 547	10 946	2 119	9 280	5	192,54	0,20
g8	Order	33 525	517 594	32 194	346 618	1	1075,46	22,50
	Abacus2	17 515	164 484	16 606	144 602	1	851,41	7,97
	Abacus4	9 767	61 920	9 059	55 439	1	748,26	6,28
	Abacus8	5 907	28 372	5 303	25 645	1	542,14	4,06
	Abacus16	4 477	19 122	3 905	17 225	1	379,57	2,82
	Abacus32	4 019	16 668	3 453	14 927	1	389,49	**2,28**
	Abacus64	4 047	16 856	3 467	15 039	1	397,91	3,23
	Log	4 075	17 220	3 444	15 118	1	370,83	3,22
g9	Order	67 274	1098 209	65 330	743 497	1	1827,05	262,03
	Abacus2	34 802	344 275	33 452	308 014	1	1396,04	202,00
	Abacus4	18 962	125 887	17 898	114 721	1	991,63	59,95
	Abacus8	11 438	57 660	10 506	53 131	1	788,06	82,73
	Abacus16	7 694	33 064	6 832	30 454	1	572,00	**28,66**
	Abacus32	6 218	25 445	5 384	23 272	1	570,78	36,96
	Abacus64	6 254	25 688	5 404	23 428	1	565,54	37,47
	Log	6 290	26 156	5 378	23 543	1	571,65	41,83
g10	Order	125 974	2127 198	123 238	1453 386	1	3421,94	1820,52
	Abacus2	64 729	661 511	62 794	599 790	1	2331,40	2245,02
	Abacus4	34 129	231 625	32 599	213 915	1	1516,38	1314,80
	Abacus8	19 419	98 651	18 084	92 262	1	1176,07	785,63
	Abacus16	12 654	54 598	11 409	51 123	1	874,27	559,01
	Abacus32	10 429	42 727	9 205	39 819	1	780,44	**481,65**
	Abacus64	9 339	37 889	8 121	35 103	1	754,94	601,44
	Log	9 384	38 474	8 092	35 260	1	771,08	596,04
g11	Order	239 238	4363 707	235 338	2985 094	0	5402,87	7150,44
	Abacus2	122 143	1335 505	119 369	1221 555	0	3410,03	7138,35
	Abacus4	64 448	466 619	62 224	434 210	1	2488,76	7162,35
	Abacus8	36 453	196 251	34 491	185 047	**2**	1701,71	7156,77
	Abacus16	23 308	104 857	21 464	99 270	1	1273,60	7162,90
	Abacus32	16 763	68 917	14 983	64 969	0	1067,04	7153,32
	Abacus64	15 168	61 285	13 394	57 550	**2**	1056,70	7156,99
	Log	15 223	61 659	13 429	57 815	1	994,12	7160,94

Actes JFPC 2021

Parallélisme en acquisition de contraintes

Nadjib Lazaar

LIRMM, Université de Montpellier, CNRS, Montpellier, France
nadjib.lazaar@lirmm.fr

Résumé

La programmation par contraintes a connu des progrès considérables au cours des quarante dernières années devenant un puissant paradigme pour la modélisation et la résolution de problèmes combinatoires. Plusieurs algorithmes parallèles ont été proposés pour résoudre un problème représenté par un réseau de contraintes. Ces derniers sont classés en catégories : CSP distribués ; propagation parallèle ; recherche parallèle ; méthode du portefeuille ; décomposition de l'arbre de recherche (Régin and Malapert 2018).

Cependant, la modélisation sous forme de contraintes nécessite une certaine expertise en programmation par contraintes. Cela empêche l'utilisation de cette technologie par un novice, ce qui constitue un frein à l'adoption d'une telle technologie hors des entreprises disposant d'ingénieurs spécialisés.

Plusieurs systèmes d'acquisition de contraintes ont été proposées pour aider l'utilisateur dans la tâche de la modélisation. Freuder et Wallace ont proposé MATCHMAKER AGENT (Freuder and Wallace 1998). Cet agent interagit avec l'utilisateur durant la résolution. L'utilisateur explique pourquoi une solution proposée n'est pas bonne. Lallouet et al. ont proposé un système basé sur la programmation logique inductive avec des connaissances de base sur la structure du problème (Lallouet et al. 2010). Beldiceanu et Simonis ont proposé MODELSEEKER, un système dédié aux problèmes structurés et basé sur le catalogue de contraintes globales (Beldiceanu and Simonis 2012). Bessiere et al. ont proposé CONACQ, un système qui génère des requêtes complètes (instanciations complètes) à classer par l'utilisateur (Bessiere et al. 2017, 2005). Shchekotykhin et Friedrich ont étendu la toute première version de CONACQ pour permettre à l'utilisateur de fournir des arguments sous la forme de contraintes pour accélérer la convergence (Shchekotykhin and Friedrich 2009).

Bessiere et al. ont proposé QUACQ (pour *Quick Acquisition – Acquisition Rapide*), un système d'apprentissage actif capable de soumettre à l'utilisateur des requêtes partielles (instanciations partielles) (Bessiere et al. 2020, 2013). QUACQ génère itérativement des requêtes complètes. Si la réponse de l'utilisateur est positive, QUACQ réduit l'espace de recherche en supprimant toutes les contraintes violées par l'exemple positif. Si maintenant la réponse est négative, QUACQ se lance dans un processus qui lui permet de trouver la portée de l'une des contraintes violées du réseau en posant un nombre logarithmique de requêtes sur la taille de l'exemple. Cette composante clé de QUACQ lui permet de toujours converger vers l'ensemble cible de contraintes dans un nombre polynomial de requêtes. Cependant, même avec une telle borne théorique, il est difficile de mettre en pratique un tel système (nombre important d'exemples à classer). De plus, la génération d'un exemple complet est NP-difficile. Par exemple, QUACQ invite l'utilisateur à classer plus de $9K$ exemples et peut prendre plus de 20 minutes pour générer un exemple complet dans le cadre d'une acquisition du réseau de contraintes du problème du Sudoku.

Dans cet article, nous présentons une toute première approche pour introduire le parallélisme dans une modélisation PPC basée sur l'acquisition de contraintes. Nous présentons PACQ, un système d'acquisition de contraintes parallèle basé sur la méthode du portefeuille. PACQ apprend un réseau de contraintes en échangeant avec plusieurs utilisateurs en ouvrant plusieurs sessions. Les utilisateurs partagent la connaissance du problème cible sans savoir comment le modéliser en un réseau de contraintes. PACQ est une version parallèle de QUACQ qui préserve la correction, la complétude et la terminaison du processus d'acquisition. PACQ permet de donner une réponse aux principales limitations du système séquentiel QUACQ, à savoir : (i) grand nombre d'exemples à classer par l'utilisateur ; et (ii) un temps d'attente important entre deux requêtes.

PACQ prend comme entrée un biais de contraintes B sur un vocabulaire (X, D) (X : variables ; D :domaines) et un langage de relations Γ. Le vocabulaire est partagé avec les N. Le système soumis des requêtes (partielles/complètes) aux N utilisateurs jusqu'à ce qu'il ait convergé sur un réseau de contraintes L équivalent au réseau cible T. Le réseau cible T étant une représentation possible du concept à apprendre f_T qui est partagé par les N utilisateurs. L'idée de base de PACQ

est d'apprendre un réseau de contraintes en ouvrant en parallèle N sessions d'acquisition. Autrement dit, nous avons un portefeuille de sessions d'acquisition visant à acquérir simultanément différentes parties du problème avec un modèle à mémoire partagée.

Nous avons évalué expérimentalement l'intérêt d'utiliser le parallélisme dans l'acquisition de contraintes sur plusieurs problèmes. Les résultats montrent que (i) PACQ assure un excellent niveau d'équilibrage des charges entre les sessions ; (ii) le nombre total de requêtes augmente sous une borne théorique ; (iii) lorsque le nombre d'utilisateurs augmente, le nombre de requêtes soumises à un utilisateur se rapproche de plus en plus de zéro.

Parallel Constraint Acquisition

Nadjib Lazaar
LIRMM, University of Montpellier, CNRS, Montpellier, France
lazaar@lirmm.fr

Abstract

Constraint acquisition systems assist the non-expert user in modelling her problem as a constraint network. QUACQ is a sequential constraint acquisition algorithm that generates queries as (partial) examples to be classified as positive or negative. The drawbacks are that the user may need to answer a great number of such examples, within a significant waiting time between two examples, to learn all the constraints. In this paper, we propose PACQ, a portfolio-based parallel constraint acquisition system. The design of PACQ benefits from having several users sharing the same target problem. Moreover, each user is involved in a particular acquisition session, opened in parallel to improve the overall performance of the whole system. We prove the correctness of PACQ and we give an experimental evaluation that shows that our approach improves on QUACQ.

Introduction

Constraint programming (CP) has made considerable progress over the last forty years, becoming a powerful paradigm for modelling and solving combinatorial problems. Several parallel algorithms have been proposed to solve a problem as a constraint network and they are grouped under categories: distributed CSPs; parallel propagation; parallel search; portfolio algorithms; problem decomposition (Régin and Malapert 2018). However, modelling a problem as a constraint network still remains a challenging task that requires some expertise in the field. Several constraint acquisition systems have been introduced to support the uptake of constraint technology by non-experts. Freuder and Wallace proposed the matchmaker agent (Freuder and Wallace 1998). This agent interacts with the user while solving her problem. The user explains why she considers a proposed solution as a wrong one. Lallouet et al. proposed a system based on inductive logic programming with the use of the structure of the problem as a background knowledge (Lallouet et al. 2010). Beldiceanu and Simonis proposed MODELSEEKER, a system devoted to problems with regular structures and based on the global constraint catalog (Beldiceanu and Simonis 2012). Bessiere et al. proposed CONACQ, which generates membership queries (i.e., complete examples) to be classified by the user (Bessiere et al. 2017). Shchekotykhin and Friedrich extended CONACQ to allow the user to provide *arguments* as constraints to speed-up the convergence (Shchekotykhin and Friedrich 2009).

Bessiere et al. proposed QUACQ (for Quick Acquisition), an active learning system that is able to ask the user to classify partial queries (Bessiere et al. 2020, 2013). QUACQ iteratively computes membership queries. If the user says *yes*, QUACQ reduces the search space by discarding all constraints violated by the positive example. When the answer is *no*, QUACQ finds the scope of one of the violated constraints of the target network in a number of queries logarithmic in the size of the example. This key component of QUACQ allows it to always converge on the target set of constraints in a polynomial number of queries. However, even that good theoretical bound can be hard to put in practice. Generating a complete example is NP-hard and the total number of examples to classify can be large. For instance, QUACQ can take more than 20 minutes to generate a complete example during the acquisition process of the Sudoku constraint network and it requires the user to classify more than $9K$ examples.

In this paper, we introduce a first ever approach to combine CP modelling through constraint acquisition with parallelism. We present PACQ, a portfolio-based parallel constraint acquisition system. PACQ learns constraint network by exchanging with several users in different acquisition sessions. Users have in mind the same target problem without knowing how to model it as a constraint network. PACQ benefits from the main limitations of the sequential QUACQ system: (i) large number of examples to classify by the user; and (ii) significant waiting time between two queries. We experimentally evaluate the benefit of using parallelism in constraint acquisition on several problems. The results show that the number of queries increases under an upper-bound using PACQ and that PACQ dramatically improves the sequential version of QUACQ in terms of queries asked per user and in terms of CPU time needed to generate queries.

Background

The constraint acquisition process can be seen as an interplay between the user and the learner. In our context, we have N users (U_1 to U_N) involved in N different acquisition sessions (A_1 to A_N) with one user per session and under a memory-shared model. The learner and the users need to share some common knowledge to communicate.

We suppose this common knowledge, called the *vocabulary*, is a tuple of n variables $X = (x_1, \ldots, x_n)$ and a domain $D = \{D(x_1), \ldots, D(x_n)\}$, where $D(x_i) \subset \mathbb{Z}$ is the finite set of values for x_i. A constraint c_S is defined by the sequence of variables S, a sub-sequence of X (i.e. $S \preceq X$), called *the constraint scope*, and the relation c over $\mathbb{Z}$ specifying which sequences of $|S|$ values are allowed for the variables S. A *constraint network* is a set C of constraints on the vocabulary (X, D). An assignment $e_Y \in D^Y$, where $D^Y = \Pi_{x_i \in Y} D(x_i)$, is called a partial assignment when $Y \prec X$ and a complete assignment when $Y = X$. An assignment e_Y on a set of variables $Y \preceq X$ *is rejected by* a constraint c_S (or e_Y *violates* c_S) if $S \preceq Y$ and the projection $e_Y[S]$ of e_Y on the variables in S is not in c. If e_Y does not violate c_S, it *satisfies* it. An assignment e_Y on Y *is accepted by* C if and only if it satisfies all constraint in C. An assignment on X that is accepted by C is a *solution* of C. We write $sol(C)$ for the set of solutions of C, and $C[Y]$ for the set of constraints from C whose scope is included in Y.

In addition to the vocabulary, the learner owns a *language* Γ of bounded arity relations from which it can build constraints on specified sets of variables. Adapting terms from machine learning, the *constraint basis*, denoted by B, is a set of constraints built from the language Γ on the vocabulary (X, D) from which the learner builds a constraint network.

Given a prefixed vocabulary (X, D), a *concept* is a Boolean function f over D^X, that is, a map that assigns to each assignement e a value in $\{0, 1\}$. A *representation* of a concept f is a constraint network C for which $f^{-1}(1) = sol(C)$. A *target concept* is a concept f_T that returns 1 for e if and only if e is a solution of the problem that the N users share and have in mind. The *target network* is a network T such that $T \subseteq B$ and T is a representation of f_T. Then we say that the target concept f_T is representable by B.
A *membership query* $ASK_{U_i}(e)$ takes as input a *complete* assignment e in D^X and asks the user U_i to classify it. The answer to $ASK_{U_i}(e)$ is *yes* if and only if $e \in sol(T)$. A *partial query* $ASK_{U_i}(e_Y)$, with $Y \subseteq X$, takes as input a *partial* assignment e_Y in D^Y and asks the user U_i to classify it. The answer to $ASK_{U_i}(e_Y)$ is *yes* if and only if e_Y does not violate any constraint in T. It is important to observe that "$ASK_{U_i}(e_Y)$=*yes* " does not mean that e_Y extends to a solution of T, which would put an NP-complete problem on the shoulders of the user. For any assignment e_Y on Y, $\kappa_B(e_Y)$ denotes the set of all constraints in B rejecting e_Y. A classified assignment e_Y is called positive or negative *example* depending on whether $ASK_{U_i}(e_Y)$ is *yes* or *no*. Knowing that (i) any extension of a negative example is a negative example and any shortening of a positive example is a positive example; and (ii) under a memory-shared model: the ASK function checks first if the query is not a redundant one w.r.t. another acquisition session where the classification can be deduced.

We now define *convergence*, which is the constraint acquisition problem we are interested in. Given a set E of (partial) examples labelled by the user *yes* or *no*, we say that a network C agrees with E if C accepts all examples labelled *yes* in E and does not accept those labelled *no*. The learning process has *converged* on the network $L \subseteq B$ if **(i)** L agrees with E and **(ii)** for every other network $L' \subseteq B$ agreeing with E, we have $sol(L') = sol(L)$. We are thus guaranteed that $sol(L) = sol(T)$. We say that the learning process reached a *premature convergence* if only **(i)** is guaranteed.

PACQ: Portfolio-Based Parallel Constraint Acquisition

We propose PACQ, a portfolio-based parallel constraint acquisition system. PACQ takes as input a basis B on a vocabulary (X, D) shared with N users. It asks (partial) queries of the N users until it has converged on a constraint network L equivalent to the target network T. The rationale behind PACQ is to learn a constraint network using N parallel acquisition sessions. That is, we have a portfolio of acquisition sessions aiming to acquire simultaneously different parts of the problem using a shared-memory model.

Description of PACQ

PACQ (see Algorithm 1) shares between acquisition sessions the basis B and the network L (line 2). PACQ initializes the network L it will learn to empty set (line 3). At line 4, PACQ makes a set-partition of the basis B into N subsets $(B_1 \ldots B_N)$ using `split` function (i.e., $B_i \neq \varnothing$, $B_i \cap B_j = \varnothing, \forall i, j$ and $\bigcup_{i=1}^{N} B_i = B$). We will see later that there are multiple ways to design the `split` function. Then, PACQ opens in parallel N `Acq_session` (line 6) and it converges on L once all sessions closed (line 7).

`Acq_session` starts by calling `GenerateExample` function that generates an example e on X satisfying the constraints of L, but violating at least one constraint from B_i (line 2). Bear in mind that, from a session to another, generating an example on a different B_i allows us to have sessions with different and complementary viewpoints on the acquisition process as a whole. If there does not exist any example e accepted by L and rejected by B_i, then all constraints in B_i are implied by L and we can safely remove them from B (line 3). Then, the current session is closed (line 3). If an example e is returned by `GenerateExample`, e is classified as positive or negative by the user U_i (line 4). If the answer is *yes*, we can remove from B the set $\kappa_B(e)$ (line 5). If the answer is *no*, we are sure that e violates at least one constraint of the target network T. We then call the function `FindScope` to discover the scope scp of these violated constraints (line 7), and the procedure `FindC` will learn (that is, put in L) at least one constraint of T whose scope is in scp (line 9). Function `FindScope` and procedure `FindC` ask queries to the corresponding user U_i and they are used exactly as they appear in, respectively, (Bessiere et al. 2013) and (Bessiere et al. 2020). The unique portion of the algorithm that cannot be parallelized is the call of `FindC` within connected scopes. Here, the procedure `FindC` has a unique permit access. For instance, if two `Acq_session` A_i and A_j return simultaneously at line 7 the scopes scp_1 and scp_2, such that $scp_1 \cap scp_2 \neq \varnothing$ (i.e., connected scopes), only one session acquires an access to `FindC` on scp_1 or scp_2 (line 8) and the second one must wait for its release at line 10. In case we have two sessions looking simultaneously for constraints on scp, only one session will have access to `FindC`

whereafter $\kappa_B(e[scp]) = \varnothing$.

Algorithm 1: PACQ

```
   In  : A basis B, Number of Users N
   Out : A learned network L
1  begin
2      shared B; L;
3      L ← ∅;
4      split (B, N);        // split B into N parts
5      foreach i ∈ 1..N do in parallel
6          Acq_session (U_i);
7      return "convergence on L"
```

Theoretical Analysis

We first show that a parallel acquisition using PACQ (algorithm 1) is a correct algorithm to learn a constraint network representing the target problem over B. We prove that PACQ is sound, complete, and it terminates.

Proposition 1 (Soundness) *Given a basis B, a target network $T \subseteq B$ and N users, the network L returned by* PACQ *is such that $sol(T) \subseteq sol(L)$.*

Proof. Suppose there exists $e \in sol(T) \setminus sol(L)$. Hence, there exists at least a constraint $c_Y \in L$ rejecting e and learned by PACQ within a `Acq_session` A_i of user U_i. The only place where we can add c_Y to L is (algo:2-line:9) with `FindC` on Y scope that is returned by `FindScope` at (algo:2-line:7). `FindC` represents the portion of PACQ that is not parallelized and the access is conditioned by the fact that no previous call occurred on Y (i.e., $\kappa_B(e[Y]) \neq \varnothing$). We know from (Bessiere et al. 2013, 2020) that `FindScope` and `FindC` functions are sound. The learned constraint c_Y is one of the target network T. Therefore, adding a constraint to L cannot reject a tuple accepted by T. □

Proposition 2 (Completeness) *Given a basis B, a target network $T \subseteq B$ and N users, the network L returned by* PACQ *is such that $sol(L) \subseteq sol(T)$.*

Proof. Suppose there exists $e \in sol(L) \setminus sol(T)$ when PACQ terminates. Hence, there exists a constraint c_Y from B that rejects e. Knowing that at (algo:1-line:4) we have a set-partition of B, it exists a `Acq_session` A_i where $c_Y \in B_i$. The only way for PACQ to terminate is to have all `Acq_session` closed. This means that within A_i session and at (algo:2-line:2), `GenerateExample` was not able to generate an example e' accepted by L and rejected by B_i. c_Y was in B_i before starting PACQ ($c_Y \in T$) and it is not in B_i when PACQ terminates. Constraints can be removed in `FindC`/`FindScope` functions and at (algo:2-line:3 and 5). We know from (Bessiere et al. 2013, 2020) that `FindC`/`FindScope` cannot remove a constraint that rejects an example accepted by L. A constraint c removed from (algo:2-line:3 and 5) cannot be c_Y because e violtates c_Y and is accepted by L. Therefore, c_Y cannot reject an example accepted by L, which proves that $sol(L) \subseteq sol(T)$. □

Algorithm 2: `Acq_session` (U_i)

```
1  while true do
2      e ←GenerateExample (L, B_i);
3      if e = nil then B ← B \ B_i;   break ;
4      if ASK_{U_i}(e) = yes then
5          B ← B \ κ_B(e);
6      else
7          scp ←FindScope_{U_i}(e, ∅, X);
8          acquire(scp);
9          if κ_B(e[scp]) ≠ ∅ then FindC_{U_i}(e, scp, L);
10         release(scp);
```

Proposition 3 (Termination) *Given a basis B, a target network $T \subseteq B$ and N users,* PACQ *terminates.*

Proof. The termination of PACQ immediately follows the closure of the N `Acq_session`. Let us consider a given A_i session. An example is generated such that it satisfies L and violates B_i. If no such example exists, B_i is reduced to empty and A_i session is closed. Otherwise, `GenerateExample` at (algo:2-line:2) returns an example e to submit to the user U_i. B decreases in size by removing $\kappa_B(e)$ when user U_i says *yes* (algo:2-line:5). If the user U_i says *no*, B also decreases in size by learning at least one constraint from B (algo:2-line:9). Let us suppose now that the user U_i says *no* on e because of a unique constraint to learn c_Y that is rejected by e. Suppose that the same example is generated at the same time in $k-1$ sessions. Then Y scope is returned by the `FindScope` calls in the k `Acq_session`. We know that for connected scopes, `FindC` has a single permit access. Thus, we have only one user U_j calling `FindC`$_{U_j}$, adding c_Y to L and removing c_Y from B. Afterwards, the other $k-1$ sessions will not have access to `FindC` on Y as $\kappa_B(e[Y])$ is reduced to empty after the first call. Therefore, at each execution of the loop, we have at least one B_i that strictly decreases in size. As B_i represent finite-size subsets coming from a set-partition of B, we have termination. □

Theorem 1 (Correctness) *Given a basis B, a target network $T \subseteq B$ and N users,* PACQ *returns a network L such that $sol(L) = sol(T)$.*

Proof. Correctness immediately follows from Propositions 1, 2, and 3. □

Strategies and Settings

PACQ can be improved by making the use of `GenerateExample` and `split` functions less brute-force, and by adapting it to a particular context (e.g., distributed CSPs).

`GenerateExample`. We can speed up the example generation by using well-known variable heuristic selectors (e.g., `minDom`, `domOverWdeg`, `impact`,...), or by using a dedicated one like `bdeg` heuristic (Tsouros and Stergiou 2020). `bdeg` selects the variable involved in a maximum number of constraints present in $B_i \setminus L$. Knowing that each session is reasoning on a particular B_i and based on preliminary comparisons, `bdeg` heuristic provides a good diversification.

`split`. In our study, we have investigated five set-partitions of B based on different background knowledge:
- `Scope`: put in the same B_i all constraints of a given scope;
- `Negation`: put in B_i a constraint and its negation;
- `Language`: put in B_i constraints of the same relation;
- `Graph`: B_i's are connected components;
- `Rule`: put in B_i constraints satisfying a set of rules.

The preliminary tests show that a B partition using background knowledge boosts the acquisition process and that the same findings are observed with the five set-partitions. We focus our analysis on the `Rule` based set-partition.

The `split` function based on `Rule` groups in the same B_i's constraints satisfying a set of rules adapted to constraint acquisition. For instance, if we know that $c_1 \wedge c_2 \rightarrow c_3$, then putting the three constraints c_1, c_2, c_3 in the same B_i can speed up the generation of examples. Building a complete set of rules is often too expensive, both in time and space as it requires generating a set of rules potentially exponential in space (all combinations of constraints that imply another one). However, it is possible to compute approximations by bounding the number of constraints in the body of a rule. Here, we only considered the rule that contain two constraints in the body `rule` : $c_i \wedge c_j \rightarrow c_k$. That is, `split` performs a random partition of triplets $(c_i, c_j, c_k) \in B^3$ where (c_i, c_j, c_k) satisfies `rule`. The rationale behind `Rule` partition is to group in the same session a certain percentage of redundancies that B contains. " Doing so, (i) we facilitate the task of `GenerateExample` to find an assignment satisfying L and rejecting at least one constraint in B, and (ii) avoiding parallel sessions to learn redundancies.

PACQ for Distributed CSPs. In some cases, parallel acquisition can be subject to privacy and/or security requirements with information that should not be shared between sessions. PACQ can easily be adapted to act in a distributed context by (i) taking into account the visibility of each agent (i.e., set of variables) in the `split` function (algo:1-line:4); and (ii) for a given session, generating examples on its own learned network L_i (algo:2-line:2).

PACQ versions. With the different strategies and settings in hand, we evaluate the four following versions of PACQ:
- **PACQ.0** using a *random* variable ordering and a *random* set-partition of B into N subsets.
- **PACQ.1** using `bdeg` heuristic;
- **PACQ.2** using `bdeg` heuristic and `Rule` based `split`;
- **PACQ.3** a revised version of PACQ.2 for distributed CSPs.

Experimental Evaluation

In this section, we experimentally evaluate our portfolio-based parallel constraint acquisition system. As finding an assignment satisfying the constraints of L and violating at least one constraint from B is an NP-complete problem, we use a time limit, denoted by `TL`, once reached, the acquisition process returns a *premature convergence* on L. The only parameter we will keep fixed in all our experiments is `TL`, that we set to 5 seconds as it corresponds to an acceptable waiting time for a human user (Lallemand and Gronier 2012). The implementation of PACQ were carried out in Java using `Choco solver 4.10.2`.[1] The code is publicly available at (gite.lirmm.fr/constraint-acquisition-team). All tests were conducted on an HPC node of 28 CPU cores and 128Gb of RAM. Each core is an Intel(R) Xeon(R) CPU E5-2640 v4 @2.40GHz. Our evaluation aims to answer the following five research questions:

- **RQ1**: *How effective is an acquisition in a parallel configuration?*
- **RQ2**: *How to make* PACQ *more effective?*
- **RQ3**: *Is* PACQ *achieving a good level of load balancing between sessions?*
- **RQ4**: *How does* PACQ *scale with the number of sessions?*
- **RQ5**: *How effective is* PACQ *on distributed CSPs?*

Benchmark Problems

Random. We generated binary random target networks with 50 variables, domains of size 10, and 122 binary arithmetic constraints, denoted by `rand_122`. PACQ is initialized with the basis B containing the complete graph of $12,250$ binary arithmetic constraints.

Purdey. The problem has a single solution. Four families have stopped by Purdey's general store, each to buy a different item and paying differently. The problem is how can we match each family with the item they bought and how they paid for it. The target network has 12 variables with domains of size 4 and 27 binary arithmetic constraints. We initialized PACQ with a basis of 950 binary constraints.

Zebra. Lewis Carroll's zebra problem has a single solution. The target network is formulated using 25 variables of 5 values with 5 cliques of $\neq$ constraints and 14 additional constraints given in the description of the problem. PACQ is initialized with a basis B of $4,950$ unary and binary (arithmetic and distance) constraints.

Queens. (prob054 in CSPLib[2]) The problem is to place n queens on an $n \times n$ chessboard such that the placement of no queen constitutes an attack on any other. The target network is formulated using n variables of n values and $3n*(n-1)/2$ binary constraints with 3 constraints between each pair of variables ($\neq$, `out_diag1` and `out_diag2`). We take the instance of 30 queens. PACQ is initialized with a basis B of $4,350$ binary constraints.

Sudoku. The Sudoku logic puzzle is a 9×9 grid. It must be filled in such a way that all the rows, all the columns and the 9 non overlapping 3×3 squares contain the numbers 1 to 9. We run experiments also on a variant of Sudoku problem, the Jigsaw Sudoku (`jsudoku`) displayed in figure 1. Instead of having 3×3 squares, we have irregular shapes. The two target networks of `sudoku` and `jsudoku` have 81 variables of 9 values and, respectively, 810 and 811

[1] github.com/chocoteam/choco-solver
[2] www.csplib.org/Problems/prob054/

binary $\neq$ constraints on rows, columns and shapes. PACQ is initialized with B of $19,440$ binary arithmetic constraints.

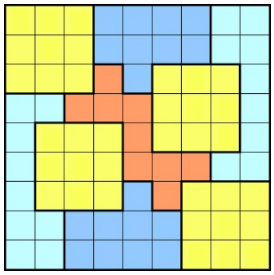


Figure 1: Jigsaw Sudoku logic puzzle instance.

Latin Square. The Latin square problem consists of an $n \times n$ table in which each element occurs once in every row and column. For this problem, we use 100 variables with domains of size 10 and 900 binary $\neq$ constraints on rows and columns. PACQ is initialized with $|B| = 29,700$ constraints.

Meeting Scheduling. (prob046 in CSPLib[3]) The Meeting Scheduling problem (`msp`) consists of n meetings and m attendees. Each meeting is of a given duration and location, and has a set of attendees. Thus, each attendee has a set of meetings that must attend. We take the instances of 40 meetings, presented in the CSPLib (`msp_19` to `msp_27`). The target networks contain from 75 to 125 constraints (attendee and time-arrival constraints). PACQ is initialized with $|B| = 4,680$ constraints.

Results

Table 1 reports the performance of PACQ.0 averaged over ten runs on each instance. We report the number of users $\#\mathtt{U}$; the size of the learned network $|L|$; the total number of asked queries $\#\mathtt{Q}$ in all sessions; the averaged, min and max number of queries asked per session $(\#\mathtt{q}, \mathtt{min}, \mathtt{max})$; time $\mathtt{T_A}$ needed to learn `L`; time of the learning process until convergence $\mathtt{T_C}$; the acquisition rate $\%\mathtt{A} = |T| - |L'|/|T|$, where L' are the constraints that have to be added to L to make it equivalent to T; and the convergence rate $\%\mathtt{C} = (\mathtt{B_{init}} - \mathtt{B_{final}})/\mathtt{B_{init}}$, where $\mathtt{B_{init}}$ and $\mathtt{B_{final}}$ are, respectively, the initial and the final size of the the basis B. $\mathtt{T_A}$ and $\mathtt{T_C}$ are CPU times of the last closed session. That is, we report the maximum $(\mathtt{T_A}, \mathtt{T_C})$ times needed for a given session. We report results with 1, $n/10$ and n users, where n corresponds to the number of variables of the given instance. We denote by PACQ.x$_{[n]}$ the call of PACQ with n users. Note that PACQ.0$_{[1]}$ is equivalent to a sequential acquisition under QUACQ.

[RQ1]: PACQ.0 effectiveness. From table 1, we observe that parallel acquisition under PACQ reduces the number of asked queries per user $\#\mathtt{q}$ and increases the total number of queries $\#\mathtt{Q}$. The number of queries per user $\#\mathtt{q}$ is reduced by a factor ranging between 2 and 60. However, the total number of queries $\#\mathtt{Q}$ is increased by a factor ranging between 2 and 4, which is far below the theoretical bound represented by the factor $\#\mathtt{U}$ (Amdahl's Law (Amdahl 1967)). For instance, on `latin` QUACQ asks more than 11K queries to a unique user, where PACQ.0$_{[100]}$ asks 192 queries per user (reduction factor of 60) with a total number of queries of 19K (growth factor of 2). Two reasons explain the growth of $\#\mathtt{Q}$. The first reason is related to the cost of learning a constraint (i.e., learning-ratio) that increases in a parallel configuration. For instance, we need 5 queries on average to learn a constraint of `queens` using QUACQ, where we need 12 queries under PACQ.0$_{[30]}$. The increase of the learning-ratio is due to the fact that several sessions can visit simultaneously the same part of the search space looking for the same constraint to learn where only one session succeeds at the end. The second reason that explains the growth of $\#\mathtt{Q}$ is learning implied constraints. If $c_1 \wedge c_2 \Rightarrow c_3$, parallel sessions can learn c_1, c_2 and the implied constraint c_3, where in a sequential configuration, c_3 can be removed. For instance, QUACQ converges on `zebra` with a constraint network of size 50. Within 25 parallel sessions, we learn a constraint network of size 70 including 20 implied constraints.

The second observation that we can draw is that a parallel acquisition speeds up the convergence. For instance, QUACQ is not able to learn the whole constraint network of the `queens` instance and it returns a premature convergence state. Here, generating $e \in sol(L \wedge \neg B)$ is hard enough that it requires more than a `TL` of $5s$. Whereas, PACQ.0$_{[3]}$ converges in $6.15s$ with B split between 3 sessions, which makes example generation much easier. Let us take another example with `sudoku` instance. QUACQ asks more than $8K$ queries to learn 76% of the target network. Than it reaches a state where finding an example in $sol(L \wedge \neg B)$ is too hard to be returned in less than $5s$. Within 8 sessions, PACQ.0 asks $1,378$ queries per user to reach an acquisition rate of 94% (*min, max* sessions of, resp., $1,123$ and $2,043$ queries). Whereas, PACQ.0$_{[81]}$ asks less than 200 queries per user to learn the whole target network and reaches a convergence rate of 99% ($[min, max] = [107, 346]$).

[RQ2]: Strategies and Settings. Table 2 reports the performance of PACQ.0, 1 and 2 averaged on ten runs on each instance. As table 1, we report the size of the learned network $|L|$; total number of queries $\#\mathtt{Q}$; queries per user, min and max $(\#\mathtt{q}, \mathtt{min}, \mathtt{max})$; acquisition time $\mathtt{T_A}$; convergence time $\mathtt{T_C}$; acquisition rate $\%\mathtt{A}$; convergence rate $\%\mathtt{C}$. We report results of $10n$ users, where n is the number of variables of the given instance.
In terms of queries, we observe a slight difference between the three versions. However, PACQ.1 and 2 outperform the basic setting of PACQ in terms of time and convergence with the use of a dedicated heuristic `bdeg` and the `Rule` split. For instance, we need 11 seconds to acquire the `jsudoku` instance, where PACQ.1 and 2 acquire it in one second. On the same instance, PACQ.0 reaches $(\%\mathtt{C} = 98\%)$, where PACQ.1 and PACQ.2 slightly improve it by both reaching 99%.

[RQ3]: Workload Balancing. From table 1 and 2, we observe that PACQ.2 provides well-balanced sessions in terms of queries with values close to the mean comparing

[3] www.csplib.org/Problems/prob046/

#U	$\|L\|$	#Q	(#q, min, max)	(T_A, T_C)	%A	%C
rand_122:						
1	116	1K	1K	(4, 9)	100	100
5	116	4K	(898, 781, 973)	(4, 5)	100	100
50	117	4K	(90, 18, 475)	(0, 2)	100	100
purdey:						
1	17	159	159	(0, 0)	100	100
2	15	154	(77, 70, 84)	(0, 0)	100	100
12	30	372	(31, 20, 44)	(0, 0)	100	100
zebra:						
1	50	601	601	(1, 1)	100	100
3	61	852	(284,276 , 340)	(0, 0)	100	100
25	70	1K	(53, 41, 108)	(0, 0)	100	100
queens:						
1	1295	6K	6K	(-, -)	99	99
3	1305	8K	(2K, 2K, 2K)	(4, 6)	100	100
30	1305	16K	(534, 470, 577)	(2, 5)	100	100
sudoku:						
1	621	8K	8K	(-, -)	76	96
8	769	11K	(1K, 1K, 2K)	(-, -)	94	99
81	801	15K	(195,100 , 346)	(2, -)	100	99
jsudoku:						
1	586	7K	7K	(-, -)	72	95
8	746	9K	(1K, 1K, 1K)	(-, -)	92	98
81	791	12K	(155, 107 , 188)	(10, -)	100	99
latin:						
1	818	11K	11K	(-, -)	90	98
10	879	13K	(1K, 820, 1K)	(-, -)	97	99
100	898	19K	(192, 65, 1K)	(10, -)	100	99
msp_27:						
1	92	1K	1K	(-, -)	36	63
4	134	1K	(390, 202, 410)	(-, -)	53	77
40	223	2K	(71, 20, 79)	(-, -)	89	89

Table 1: PACQ.0 results

P.	$\|L\|$	#Q	(#q, min, max)	(T_A, T_C)	%A	%C
rand_122:						
0	116	9K	(18, 8, 115)	(0, 1)	100	100
1	116	8K	(17, 9, 117)	(0, 0)	100	100
2	115	9K	(18, 8, 114)	(0, 0)	100	100
purdey:						
0	41	1K	(9,4, 25)	(0, 0)	100	100
1	48	1K	(10, 4, 21)	(0, 0)	100	100
2	47	1K	(11, 4, 20)	(0, 0)	100	100
zebra:						
0	94	3K	(14, 5, 41)	(0, 2)	100	100
1	88	3K	(14, 5, 27)	(0, 0)	100	100
2	88	3K	(14, 5, 27)	(0, 0)	100	100
queens:						
0	1305	61K	(204, 23, 275)	(9, 9)	100	100
1	1305	63K	(210, 144, 272)	(7, 8)	100	100
2	1305	61K	(206, 146, 294)	(5, 8)	100	100
sudoku:						
0	798	18K	(23, 6, 143)	(10, -)	100	98
1	805	22K	(28, 6, 142)	(5, -)	100	99
2	806	21K	(27, 6, 67)	(3, -)	100	99
jsudoku:						
0	794	15K	(19, 9, 194)	(11, -)	100	98
1	817	22K	(28, 9, 182)	(1, -)	100	99
2	816	22K	(28, 9, 62)	(1, -)	100	99
latin:						
0	899	23K	(23, 7, 326)	(4, -)	100	98
1	900	27K	(27, 7, 254)	(1, -)	100	99
2	900	29K	(29, 7, 66)	(1, -)	100	99
msp_27:						
0	197	5K	(13, 5, 76)	(0, -)	78	97
1	168	10K	(27, 5, 69)	(0, -)	67	99
2	170	11K	(29, 5, 43)	(0, -)	67	99

Table 2: PACQ (P.) 0, 1 and 2 results with $10n$ users.

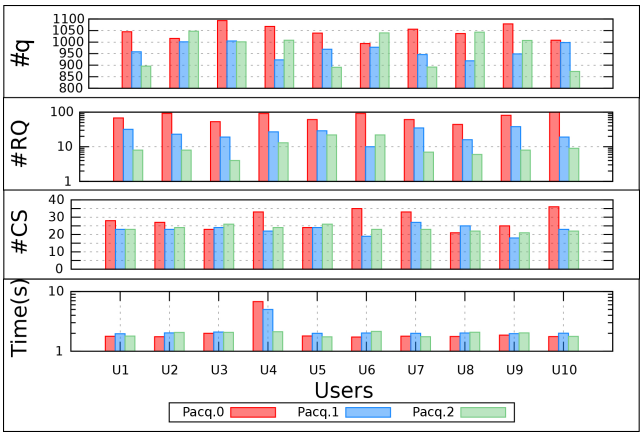


Figure 2: PACQ acting on 30-queens with 10 users

to PACQ.0 and 1. Thanks to the dedicated background knowledge partition based on Rule making sessions homogeneous comparing to a random partition used in PACQ.0 and 1. Let us take a closer look at latin instance. PACQ.0 and 1 with 1K users asks less than 30 queries per user with a minimum session of 7 queries and $\approx$ 10 sessions exceeding 200 queries. However, PACQ.2 asks queries per user in a tight interval of $[7, 66]$ with a standard deviation of 13.

In order to strengthen our previous observations, we run PACQ.0, 1 and 2 on queens with 10 sessions A_1 to A_{10}. For each session, we report in figure 2 the number of queries (#q), the number of redundant queries (#RQ), the number of connected scopes (#CS) and time in seconds of each session A_i. Note that RQ and CS allow us to estimate the degree of diversification between the different sessions.

The instance of queens has a target network of $1,305$ binary constraints. For the three versions, the averaged number of constraints acquired by each session is 130 with a standard deviation of 15 constraints for PACQ.0 and less than 8 constraints for PACQ.1 and 2. That is, PACQ provides a well-balanced workload of sessions. It follows that the same observation can be made on #Q, #RQ and #CS. Comparing the three versions, we observe that PACQ.1 (using bdeg heuristic) outperforms PACQ.0 by reducing the number of redundant queries to 50% and the number of connected scopes to 90%. We also observe that using Rule based split in PACQ.2 further improves the performance. In terms of CPU time, we observe an overload for A_4 session under PACQ.0 and PACQ.1. That is, the acquisition process terminates after

the close of session A_4 ($7s$ for PACQ.0 and $7s$ for PACQ.1). Whereas, PACQ.2 is ensuring an excellent level of load balancing with sessions of $\approx 1s$.

[RQ4]: Scalability. From table 1, we selected the three instances where QUACQ needs to ask more than $7K$ queries to acquire the corresponding target network (i.e., `sudoku`, `jsudoku` and `latin`). We run PACQ.2 on the three instances by varying the number of users up to $1K$. Figure 3 reports the total number of queries (#Q), the number of queries asked per user (#q), the learning-ratio R (i.e., the number of queries needed to learn a constraint $\mathtt{R} = \#\mathtt{Q}/|\mathtt{T}|$), the number of redundant queries (#RQ) and the number of connected scopes (#CS). Figure 3 shows that when the number of users grows, #Q follows an $(a - bx^{-c})$ scale with $a = 25K$, $b = 18K$ and $c = 0.74$, which means that when we get more and more users, the total number of queries gets closer and closer to the bound $a = 25K$, which is far below the Amdahl's Law theoretical bound (Amdahl 1967) represented by number of queries asked by QUACQ ($> 8K$) multiplied by #U. The second observation is that when the number of users grows, the number of queries asked per user #q follows a negative power function scale ($a\ x^{-b}$) with $a = 11K$ and $b = 0.85$, which means that when we get more and more users, number of queries asked per user gets closer and closer to 0 (horizontal asymptotes of negative power functions). This is very good news as it means that learning problems in parallel will scale well. For instance, QUACQ asks more than $8K$ to one user to learn `sudoku` instance. PACQ with, respectively, 2, 10, 100 and 1000 parallel sessions, asks to the same user, respectively, $3K$, $1K$, 163 and 13. Also, the learning-ratio R scales well when the number of users grows. The number of queries asked to learn one constraint follows a logarithmic scale bounded above by 35 queries per constraint ($c\ log(x) + d$ with $c = 3$ and $d = 8$). Following the conclusions drawn in **RQ3** and thanks to `bdeg` heuristic and `Rule` based `split`, we observe that #RQ and #CS per user decreases when the number of users grows. That is, PACQ.2 ensures an excellent level of workload balancing between sessions up to $1K$.

[RQ5]: PACQ for distributed CSP. For our last experiment, we illustrate the use of PACQ on distributed CSP with `msp` problem. In `msp`, each attendee comes with her own constraints and shares meetings (i.e., variables) with the other attendees. This means that the problem can be acquired in a fully-distributed scheme by having a session per attendee. Figure 4 reports %A and %C rates performed by QUACQ, PACQ.2$_{[v]}$ and PACQ.3$_{[v]}$ (where $v \in \{9, 13, 14, 17\}$ is the number of attendees) on the 9 `msp` instances. Darker color indicates higher convergence rate. The number in each cell indicates the acquisition rate. The sequential version using QUACQ is not able to learn and to converge on the 9 instances. PACQ.2 is not converging. However, PACQ.3 learns and converges on the 9 instances. For instance, on `msp_21` QUACQ learns 58% of the target network and returns a premature convergence of 62% in 10 seconds. PACQ.2 learns the instance without converging (%C = 84%) in $8s$. Then, the distributed version with PACQ.3 converges

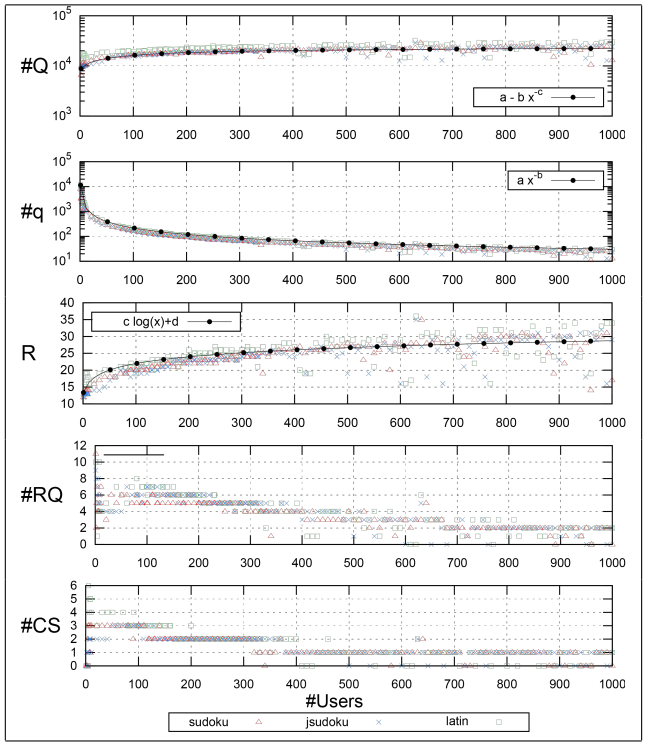


Figure 3: Scalability of PACQ.2 up to $1K$ users

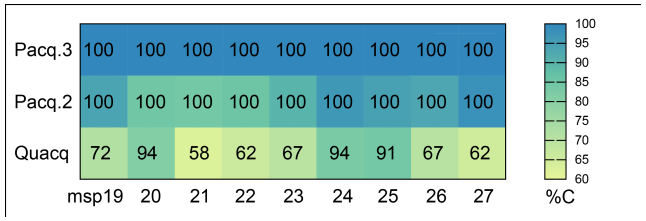


Figure 4: %A and %C comparison (QUACQ, PACQ.2 and 3)

in $0.38s$. This is explained by the fact that in PACQ.3 we have sessions of small size in terms of variables. Again on `msp_21`, PACQ.3 opens 13 sessions of 5 variables, where PACQ.2 opens 13 sessions of 40 variables. That is, generating examples on 5 variables is easier than generating examples on 40 variables and thus, avoid premature convergence.

Conclusion

In this paper we proposed a parallel constraint acquisition system PACQ, where numerous users answer in parallel queries in order to learn all the constraints. PACQ is a parallel extension of QUACQ and preserves the fundamentals of its active learning system and its soudness, correctness and completeness properties. Performed experiments showed that (i) PACQ ensures an excellent level of load balancing; (ii) the total number of queries increases under an upper-bound; (iii) when the number of users grows, the number of queries asked per user gets closer and closer to zero.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 952215. This work also received support from University of Montpellier and I-Site MUSE under CAR-UM2020/2021 project. We thank Nassim Belmecheri and Yahia Lebbah who provided insight and expertise that greatly assisted the work. We would also like to show our gratitude to the "anonymous" reviewers for their insightful suggestions and careful reading.

References

Amdahl, G. M. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, volume 30 of *AFIPS Conference Proceedings*, 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C. doi:10.1145/1465482.1465560. URL https://doi.org/10.1145/1465482.1465560.

Beldiceanu, N.; and Simonis, H. 2012. A Model Seeker: Extracting Global Constraint Models from Positive Examples. In Milano, M., ed., *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, 141–157. Springer. doi:10.1007/978-3-642-33558-7_13. URL https://doi.org/10.1007/978-3-642-33558-7_13.

Bessiere, C.; Carbonnel, C.; Dries, A.; Hebrard, E.; Katsirelos, G.; Lazaar, N.; Narodytska, N.; Quimper, C.; Stergiou, K.; Tsouros, D. C.; and Walsh, T. 2020. Partial Queries for Constraint Acquisition. *CoRR* abs/2003.06649. URL https://arxiv.org/abs/2003.06649.

Bessiere, C.; Coletta, R.; Hebrard, E.; Katsirelos, G.; Lazaar, N.; Narodytska, N.; Quimper, C.; and Walsh, T. 2013. Constraint Acquisition via Partial Queries. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 475–481.

Bessiere, C.; Koriche, F.; Lazaar, N.; and O'Sullivan, B. 2017. Constraint acquisition. *Artif. Intell.* 244: 315–342. doi:10.1016/j.artint.2015.08.001. URL https://doi.org/10.1016/j.artint.2015.08.001.

Freuder, E. C.; and Wallace, R. J. 1998. Suggestion Strategies for Constraint-Based Matchmaker Agents. In Maher, M. J.; and Puget, J., eds., *Principles and Practice of Constraint Programming - CP98, 4th International Conference, Pisa, Italy, October 26-30, 1998, Proceedings*, volume 1520 of *Lecture Notes in Computer Science*, 192–204. Springer. doi:10.1007/3-540-49481-2_15. URL https://doi.org/10.1007/3-540-49481-2_15.

Lallemand, C.; and Gronier, G. 2012. Enhancing User eXperience During Waiting Time in HCI: Contributions of Cognitive Psychology. In *Proceedings of the Designing Interactive Systems Conference*, DIS '12, 751–760. New York, NY, USA: ACM. ISBN 978-1-4503-1210-3. doi:10.1145/2317956.2318069. URL http://doi.acm.org/10.1145/2317956.2318069.

Lallouet, A.; Lopez, M.; Martin, L.; and Vrain, C. 2010. On Learning Constraint Problems. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, 45–52. IEEE Computer Society. doi:10.1109/ICTAI.2010.16. URL https://doi.org/10.1109/ICTAI.2010.16.

Régin, J.; and Malapert, A. 2018. Parallel Constraint Programming. In Hamadi, Y.; and Sais, L., eds., *Handbook of Parallel Constraint Reasoning*, 337–379. Springer. doi:10.1007/978-3-319-63516-3_9. URL https://doi.org/10.1007/978-3-319-63516-3_9.

Shchekotykhin, K. M.; and Friedrich, G. 2009. Argumentation Based Constraint Acquisition. In Wang, W.; Kargupta, H.; Ranka, S.; Yu, P. S.; and Wu, X., eds., *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*, 476–482. IEEE Computer Society. doi:10.1109/ICDM.2009.62. URL https://doi.org/10.1109/ICDM.2009.62.

Tsouros, D. C.; and Stergiou, K. 2020. Efficient multiple constraint acquisition. *Constraints* doi:10.1007/s10601-020-09311-4. URL https://doi.org/10.1007/s10601-020-09311-4.

Actes JFPC 2021

Adaptation des stratégies des solveurs SAT CDCL aux solveurs PB natifs

Daniel Le Berre[1] **Romain Wallon**[2*]

[1] CRIL, Univ Artois & CNRS

[2] LIX, Laboratoire d'Informatique de l'X, École Polytechnique, Chaire X-Uber

leberre@cril.fr wallon@lix.polytechnique.fr

Résumé

Les solveurs pseudo-booléens (PB) travaillant nativement sur des contraintes PB sont fondés sur l'architecture CDCL à l'origine des hautes performances des solveurs SAT modernes. En particulier, ces solveurs PB utilisent non seulement une procédure d'analyse de conflits (utilisant les plans-coupes), mais également des stratégies complémentaires qui sont cruciales pour l'efficacité du solveur, comme les heuristiques de choix de variable, de suppression des contraintes apprises et de redémarrage. Cependant, ces stratégies sont le plus souvent réutilisées par les solveurs PB sans tenir compte de la forme particulière des contraintes PB qu'ils considèrent. Dans cet article, nous présentons et évaluons différentes manières d'adapter ces stratégies pour tenir compte des spécificités des contraintes PB tout en préservant le même comportement que dans le cadre clausal. Nous avons implanté ces stratégies dans deux solveurs différents, à savoir *Sat4j* (pour lequel nous considérons trois configurations) et *RoundingSat*. Nos expérimentations montrent que ces stratégies dédiées permettent d'améliorer, parfois significativement, les performances de ces solveurs, à la fois sur des problèmes de décision et d'optimisation.

Abstract

Current implementations of pseudo-Boolean (PB) solvers working on native PB constraints are based on the CDCL architecture which empowers highly efficient modern SAT solvers. In particular, such PB solvers not only implement a (cutting-planes-based) conflict analysis procedure, but also complementary strategies for components that are crucial for the efficiency of CDCL, namely branching heuristics, learned constraint deletion and restarts. However, these strategies are mostly reused by PB solvers without considering the particular form of the PB constraints they deal with. In this paper, we present and evaluate different ways of adapting CDCL strategies to take the specificities of PB constraints into account while preserving the behavior they have in the clausal setting. We implemented these strategies in two different solvers, namely *Sat4j* (for which we consider three configurations) and *RoundingSat*. Our experiments show that these dedicated strategies allow to improve, sometimes significantly, the performance of these solvers, both on decision and optimization problems.

1 Introduction

Le succès des solveurs SAT dits *modernes* a motivé la généralisation de l'architecture CDCL (*Conflict-Driven Clause Learning*, ou apprentissage de clauses guidé par les conflits) [30, 31, 13] à la résolution de problèmes pseudo-booléens (PB) [35]. La principale motivation derrière le développement de solveurs PB est que les solveurs SAT classiques sont fondés sur le système de preuve par *résolution*, qui est relativement *faible* : les instances difficiles pour ce système (tels que ceux nécessitant de « savoir compter », comme les formules du *principe du pigeonnier*, ou *principe des tiroirs* [20]) sont difficiles pour les solveurs SAT. Le système de preuve des plans-coupes [19, 21, 32] constitue une alternative plus puissante, permettant par exemple de résoudre des formules du principe du pigeonnier en un nombre linéaire d'étapes d'inférence. Plus généralement, ce système de preuve *p-simule* la résolution : toute preuve par résolution peut être simulée par une preuve du système des plans-coupes de taille polynomiale [10]. En théorie, les solveurs PB devraient donc être capables de trouver des preuves d'incohérence plus courtes, et donc d'être plus efficaces que les solveurs SAT classiques. En pratique cependant, les solveurs PB actuels ne parviennent pas à tenir les promesses de la théorie. En particulier, la plupart des solveurs PB [12, 9, 36, 25] im-

*Une grande partie des travaux présentés dans cet article ont été réalisés lorsque cet auteur était doctorant au CRIL

plantent un sous-ensemble du système des plans-coupes appelé *résolution généralisée* [21], qui permet d'étendre l'algorithme CDCL aux contraintes PB. Lorsqu'une contrainte devient conflictuelle, la règle de résolution généralisée est appliquée entre cette contrainte et la raison de la propagation de l'un de ses littéraux pour inférer une nouvelle contrainte conflictuelle. Cette opération est répétée jusqu'à ce qu'une contrainte assertive soit finalement produite. Cependant, les solveurs implantant cette procédure n'exploitent pas toute la puissance du système des plans-coupes [37], et sont moins performants que les solveurs fondés sur la résolution dans les compétitions PB [34].

Malgré les améliorations récemment apportées par *RoundingSat* [16] avec l'utilisation de la règle de *division* pendant l'analyse de conflit, les implantations actuelles du système des plans-coupes ont toujours un point faible majeur : elles sont équivalentes au système par résolution lorsqu'elles reçoivent une CNF en entrée. De plus, ces implantations sont plus complexes que le simple remplacement de la résolution dans l'analyse de conflit par la résolution généralisée : trouver *quelles* règles appliquer et *quand* n'est pas si évident que cela [17, 24]. En particulier, les solveurs PB doivent tenir compte de propriétés spécifiques aux contraintes PB et au système des plans-coupes pour les adapter à l'architecture CDCL. De plus, de nombreuses autres fonctionnalités des solveurs SAT CDCL sont requises pour garantir l'efficacité pratique de ces derniers (voir, par exemple, [15]). Dans cet article, nous nous concentrons sur ces fonctionnalités dans le cadre PB. A notre connaissance, peu de travaux étudient l'extension de ces composants aux solveurs PB : ils sont le plus souvent réutilisés tels que définis dans les solveurs SAT classiques, et adaptés juste assez pour fonctionner dans le solveur, sans considérer leur impact dans le contexte de la résolution de problèmes PB.

Dans la Section 3, nous introduisons de nouvelles variantes de l'*heuristique de choix de variable VSIDS* (*variable state independent decaying sum*) [31]. Ces variantes généralisent les propriétés de cette heuristique aux solveurs PB, en considérant les affectations des littéraux rencontrés. Dans la Section 4, nous proposons diverses stratégies de *suppression des contraintes apprises*, en définissant plusieurs nouvelles mesures visant à évaluer la qualité des contraintes apprises par le solveur. En particulier, nous considérons de nouvelles définitions de la mesure du *LBD* (*Literal Block Distance*) [2] qui, comme nous le montrons, n'est pas bien défini pour les contraintes PB. Dans la Section 5, nous utilisons ensuite ces nouvelles mesures pour détecter *quand* déclencher un *redémarrage*, en les utilisant dans des politiques *dynamiques* [3]. Enfin, dans la Section 6, nous évaluons empiriquement l'impact de ces différentes stratégies dans plusieurs solveurs PB.

2 Préliminaires

Nous considérons un cadre propositionnel classique défini sur un ensemble fini de variables propositionnelles $\mathcal{V}$. Un *littéral* ℓ est une variable $v \in \mathcal{V}$ ou sa négation $\bar{v}$. Les valeurs booléennes sont représentées par les entiers 1 (vrai) et 0 (faux), de sorte que $\bar{v} = 1-v$.

Une *contrainte pseudo-booléenne (PB)* est une équation ou inéquation de la forme $\sum_{i=1}^{n} \alpha_i \ell_i \ \triangle \ \delta$, dans laquelle les *coefficients* α_i et le *degré* δ sont des entiers, les ℓ_i sont des littéraux et $\triangle \in \{<, \leq, =, \geq, >\}$. Une telle contrainte peut être *normalisée* en une conjonction de contraintes de la forme $\sum_{i=1}^{n} \alpha_i \ell_i \geq \delta$ dans laquelle les coefficients et le degré sont des entiers positifs. Dans la suite, nous supposons donc que toutes les contraintes sont normalisées. Une *contrainte de cardinalité* est une contrainte PB dont tous les coefficients sont égaux à 1 et une clause est une contrainte de cardinalité de degré 1. Cette définition montre que les contraintes PB généralisent les clauses, et que le raisonnement clausal est donc un cas particulier du raisonnement PB.

Les solveurs PB ont donc été conçus pour étendre l'algorithme CDCL des solveurs SAT classiques. En particulier, pendant la phase d'exploration, les solveurs PB *affectent* des variables. Dans la suite, nous notons $\ell(V@D)$ le fait que la valeur V est affecté au littéral ℓ au niveau de décision D, et $\ell(?@?)$ le fait que ℓ n'est pas affecté. L'affectation des variables se fait soit en *prenant une décision*, soit en *propageant* une valeur de vérité pour une variable. Dans ce contexte, la forme normalisée des contraintes PB est particulièrement commode pour détecter des propagations : comme pour les clauses, les propagations sont déclenchées par la falsification de littéraux dans une contrainte. Cependant, contrairement aux clauses, une contrainte PB peut propager des littéraux même si d'autres littéraux sont satisfaits ou non-affectés, comme dans cet exemple.

Exemple 1. *La contrainte* $5a(0@3) + 5b(?@?) + c(?@?) + d(?@?) + e(0@1) + f(1@2) \geq 6$ *propage le littéral* b *à* 1 *(vrai) sous l'affectation courante. Si* b *est affecté à* 0*, la contrainte* $5a(0@3) + 5b(0@3) + c(?@?) + d(?@?) + e(0@1) + f(1@2) \geq 6$ *devient conflictuelle. Dans les deux cas, observons que* f *est satisfait et que* c *et* d *ne sont pas affectés.*

Les solveurs PB implantent par ailleurs une procédure d'analyse de conflit permettant, grâce au système des plans-coupes, d'apprendre de nouvelles contraintes et de réaliser des retours-arrière non-chronologiques. Nous omettons délibérément sa description qui sort du cadre de cet article, et renvoyons le lecteur intéressé vers [35, 16, 24] pour plus de détails sur le sujet.

3 Heuristique de choix de variable

Un composant important d'un solveur SAT est son *heuristique de choix de variable* : pour trouver efficacement une solution ou une preuve d'incohérence, le solveur doit choisir les *bonnes* variables sur lesquelles prendre des décisions. La plupart des solveurs SAT actuels utilisent VSIDS [31] ou l'une de ses variantes [7], ou encore l'heuristique plus récente LRB [27]. Nous nous concentrons ici sur VSIDS, qui est celle adoptée par les solveurs que nous étudions dans la suite.

3.1 VSIDS et ses variantes dans les solveurs SAT

Dans l'implantation originale de VSIDS, chaque variable possède un *score* qui est incrémenté chaque fois qu'une nouvelle clause contenant cette variable est apprise. De plus, ces scores sont régulièrement divisés par 2 (en pratique, tous les 256 conflits), de manière à favoriser les variables apparaissant dans les clauses apprises les plus récentes. Au moment de choisir une variable, c'est celle ayant le plus haut score qui est sélectionnée.

La variante la plus commune de VSIDS est *exponential VSIDS* (EVSIDS), introduite par *MiniSat* [13]. Dans cette heuristique, une valeur g est choisie entre 1.01 et 1.2 au début de l'exécution du solveur. Quand une variable est rencontrée pendant l'analyse du i-ème conflit (soit dans la clause apprise, soit dans celles ayant servi à la produire), son score est mis à jour en lui ajoutant g^i. Cette mise à jour est souvent appelée *bumping*, ou *incrémentation*. Elle préserve le fait que les variables favorisées sont celles *impliquées* dans les conflits récents, tout en évitant le coût d'une division fréquente. Notons toutefois qu'une division régulière reste nécessaire pour éviter des dépassements de capacité, mais cette division reste moins fréquente que celle utilisée dans la version originale de VSIDS.

3.2 VSIDS dans les solveurs PB

Les solveurs PB actuels utilisent l'heuristique VSIDS (ou l'une de ses variantes) pour décider quelle sera la prochaine variable à affecter. En pratique, cette heuristique peut être utilisée en l'état par les solveurs PB, mais cela ne permet pas de prendre en compte toutes les informations contenues dans la contrainte, comme cela a été observé dans [9] (qui, cependant, ne propose pas d'heuristique plus adaptée). C'est pourquoi différentes variantes de cette heuristique ont été proposée.

Dans [11, Section 4.5], il est ainsi proposé d'ajouter, pour chaque variable apparaissant dans les *contraintes de cardinalité du problème original*, le degré de la contrainte au score *initial* de ces variables. Cette approche permet de compter le nombre d'occurrences des variables dans les clauses représentées par cette contrainte de cardinalité, qui correspond exactement à la valeur de son degré. Le score des variables des contraintes apprises n'est cependant incrémenté que de 1 (une seule des clauses sous-jacentes étant en général responsable du conflit analysé).

Exemple 2 (Tiré de [11, Section 4.5])**.** *Si la contrainte de cardinalité* $a+b+c \geq 2$ *est présente dans le problème original, le score de ses variables est incrémenté de* 2*. En effet, cette contrainte est équivalente à la conjonction des clauses* $a+b \geq 1$*,* $a+c \geq 1$ *et* $b+c \geq 1$*. Si cette contrainte est apprise, les scores ne sont incrémentés que de* 1*.*

Bien que cette heuristique soit plus spécifique que la version originale de VSIDS pour les problèmes PB, elle n'est cependant pas satisfaisante, car elle ne s'adapte pas bien aux implantations modernes de VSIDS (et en particulier, EVSIDS). Tout d'abord, comme seules les contraintes originales sont concernées, l'heuristique n'a aucune influence sur l'incrémentation du score des variables impliquées dans les conflits récents. Par ailleurs, la forme particulière des contraintes PB générales n'est pas prise en compte par cette heuristique. La principale raison du choix de ne considérer que les contraintes de cardinalité est ici que déterminer le nombre de clauses dans lesquelles un littéral d'une contrainte PB apparaît est difficile en général.

Une autre alternative, implantée dans *Pueblo* [36], est d'estimer l'importance relative d'un littéral dans la contrainte, en calculant le rapport de son coefficient par le degré de la contrainte. Cette valeur est ensuite ajouté au score de la variable.

Exemple 3. *Si le score de* a *est incrémenté dans la contrainte* $5a + 5b + c + d + e + f \geq 6$*, l'incrément est multiplié par* $5/6$*.*

Concernant *Sat4j* [25] et *RoundingSat* [16], ces deux solveurs implantent une approche plus classique d'EVSIDS, en incrémentant le score des variables rencontrées pendant l'analyse de conflit. Cependant, certains détails d'implantation diffèrent entre ces deux solveurs. En particulier, dans *Sat4j* le score d'une variable est incrémenté *chaque fois* qu'elle apparaît dans une clause rencontrée pendant l'analyse de conflit, tandis que *RoundingSat* ne l'incrémente qu'une seule fois (comme dans *MiniSat* [13]), et deux fois dans le cas des variables *éliminées* au cours de l'analyse de conflit.

3.3 Vers un meilleur VSIDS pour les solveurs PB

Comme mentionné plus haut, les implantations actuelles de VSIDS, et en particulier d'EVSIDS, sont conçues pour favoriser la sélection de variables impliquées dans les conflits récents. Lorsque seules des

clauses sont considérées, identifier ces littéraux est évident : les littéraux impliqués dans le conflit sont exactement ceux apparaissant dans la clause. Cependant, ce n'est plus le cas lorsque des contraintes PB sont considérés. En effet, étant donnée une contrainte PB, ses littéraux ne jouent pas le même rôle dans la contrainte de par la présence de coefficients, et n'ont donc pas nécessairement le même impact sur le conflit.

Une observation cruciale pour détecter les littéraux qui sont réellement *impliqués* dans un conflit est l'impact de l'affectation courante. En effet, dans les solveurs SAT classiques, tous les littéraux apparaissant dans les clauses rencontrées pendant l'analyse de conflit sont toujours *affectés*, et tous sauf un sont même *falsifiés*. Cependant, dans les contraintes PB, ce n'est pas toujours le cas (voir Exemple 1), et même des littéraux falsifiés peuvent être *ineffectifs* [24, Section 3.1].

Définition 1 (Littéral effectif)**.** *Étant donnée une contrainte PB conflictuelle (resp. assertive) χ, un littéral ℓ de χ est dit* effectif *dans χ s'il est falsifié et si le satisfaire ne préserve pas le conflit (resp. la propagation). ℓ est dit* ineffectif *s'il n'est pas effectif.*

Même si de tels littéraux apparaissent pendant l'analyse de conflit, ils ne jouent aucun rôle dans celui-ci, tout comme les variables associées. Nous proposons donc de prendre en compte l'affectation courante lors de la mise à jour des scores des variables à l'aide de trois nouvelles stratégies, à savoir *bump-assigned*, qui incrémente uniquement le score des variables affectées rencontrées pendant l'analyse de conflit, *bump-falsified*, qui incrémente uniquement le score des variables pour lesquelles un littéral apparaît falsifié pendant l'analyse de conflit, et *bump-effective*, qui incrémente uniquement le score des variables pour lesquelles un littéral apparaît effectif pendant l'analyse de conflit.

Exemple 4. *Lors de la mise à jour du score des variables de la contrainte* $5a(0@3) + 5b(1@3) + c(?@?) + d(?@?) + e(0@1) + f(1@2) \geq 6$*, la stratégie* bump-assigned *incrémente le score de a, b, e et f, la stratégie* bump-falsified *incrémente le score de a et e et la stratégie* bump-effective *incrémente le score de a.*

4 Suppression des contraintes apprises

Les solveurs PB, tout comme les solveurs SAT, doivent régulièrement *supprimer* les contraintes apprises pendant leur exécution. En effet, stocker ces contraintes peut non seulement accroître l'espace mémoire requis par le solveur, mais également ralentir la propagation unitaire. Il est alors crucial de choisir judicieusement *quelles* contraintes sont à supprimer.

Cette fonctionnalité est le plus souvent directement héritée des solveurs SAT dans les solveurs PB. Par exemple, *Pueblo* [36] utilise l'approche de *MiniSat* [13] fondée sur l'activité des contraintes apprises (les moins actives sont supprimées les premières), *Sat4j* [25] utilise également uns stratégie fondée sur l'activité, mais plus agressive à l'instar de *Glucose* [2], et *RoundingSat* [16] utilise une approche hybride qui lui est propre, fondée à la fois sur le LBD et l'activité (le second est utilisé pour départager des contraintes lorsqu'elles ont le même LBD). Dans d'autres solveurs PB tels que *pbChaff* [12] et *Galena* [9], la suppression (éventuelle) des contraintes apprises n'est pas documentée. Dans [9], une perspective d'affaiblissement des contraintes apprises est toutefois suggérée comme une alternative à leur suppression.

Cependant, bien que des mesures comme celle de l'activité peuvent être réutilisées telles quelles par les solveurs PB (elles ne tiennent pas compte de la représentation ni de la sémantique des contraintes qu'elles évaluent), pour d'autres mesures, il peut être pertinent de prêter attention à la forme particulière des contraintes PB. Cette section étudie deux principales approches dans cette direction.

4.1 Mesures fondées sur la taille

Dans les solveurs SAT classiques, les mesures fondées sur la taille suppriment les clauses les plus longues, c'est-à-dire, celles contenant de nombreux littéraux. Ces clauses sont faibles, notamment du point de vue des propagations : une propagation ne peut être déclenchée qu'après que de nombreux littéraux ont été falsifiés. Lorsque l'on considère des contraintes PB, ce n'est plus le cas. En effet, rappelons qu'une contrainte PB peut propager des littéraux alors que d'autres littéraux ne sont pas encore affectés, et que le nombre de littéraux d'une telle contrainte ne présume en rien de sa force.

Une autre raison qui a motivé l'utilisation de mesures fondées sur la taille est que les longues clauses sont coûteuses à maintenir, ce qui est aussi vrai dans le cas des contraintes PB. En particulier, la taille de ces contraintes prend aussi en compte la taille des coefficients, qui n'est pas négligeable : comme les coefficients peuvent devenir très grands pendant l'analyse de conflit, l'utilisation de la précision arbitraire est requise pour les représenter. Cette représentation peut ralentir les opérations arithmétiques, et donc l'analyse de conflit réalisée par le solveur. Différentes approches ont été étudiées pour limiter l'accroissement des coefficients, comme celles fondées sur les règles de division [16] ou d'affaiblissement [24]. Cependant, ces approches conduisent à l'inférence de contraintes plus faibles. En utilisant des mesures de qualité prenant en compte la taille des coefficients, il est possible de favoriser l'apprentissage de contraintes ayant des « petits » coefficients. Dans ce but, nous définissons ci-dessous des

mesures fondées sur le degré des contraintes apprises : *degree*, qui évalue la qualité d'une contrainte apprise par la *valeur* de son degré, et *degree-bits*, qui évalue la qualité d'une contrainte apprise par la *taille* de son degré, mesurée par le nombre minimum de bits nécessaires pour le représenter. L'intérêt de cette dernière mesure est d'évaluer plus finement l'espace mémoire utilisé pour la représentation du degré (qui permet d'estimer le coût des opérations arithmétiques), mais également de pouvoir représenter les valeurs fournies par cette mesure en utilisant une précision fixe.

Dans les deux cas, plus le degré est petit, meilleure est la contrainte. En effet, il est bien connu que le degré d'une contrainte PB est une borne supérieure des coefficients de cette contrainte (par la règle dite de *saturation*), de sorte que ne considérer que le degré est suffisant pour cette mesure, qui vise à privilégier des contraintes ayant des petits coefficients pour préserver l'efficacité des opérations arithmétiques réalisées.

Exemple 5. *Les mesures de qualités fondées sur le degré de la contrainte* $5a+5b+c+d+e+f \geq 6$ *sont* 6 *pour la mesure* degree*, et* 3 *pour la mesure* degree-bits *(comme la représentation binaire de* 6*, à savoir* 110*, requiert* 3 *bits).*

4.2 Mesures fondées sur le LBD

Une autre mesure permettant d'évaluer les clauses apprises par un solveur SAT est le LBD [2].

Définition 2 (LBD)**.** *Considérons une* clause γ *et l'affectation de ses littéraux. Soit* π *une partition des littéraux, calculée à partir du niveau de décision des littéraux. Le* LBD *de* γ *est le nombre d'éléments de* π*.*

Le LBD d'une clause est calculé la première fois lors de l'apprentissage de cette clause, et est ensuite mis à jour chaque fois que celle-ci est utilisée comme raison. Dans ce contexte, les meilleures clauses sont celles de plus petit LBD. Cette approche tire parti du fait que tous les littéraux d'une clause conflictuelle sont falsifiés, et que lorsque la clause est utilisée comme raison, seul le littéral propagé n'est pas falsifié, mais son niveau de décision est aussi celui d'un autre littéral, qui lui est falsifié et est à l'origine de la propagation. Lorsque l'on considère des contraintes PB, ce n'est plus le cas. Le LBD n'est donc pas bien défini pour ces contraintes. Pour pouvoir l'utiliser comme une mesure de la qualité des contraintes apprises, nous devons donc prendre en compte les littéraux non falsifiés apparaissant dans ces contraintes. Dans ce but, nous introduisons cinq nouvelles définitions du LBD. Pour commencer, nous considérons d'abord une sorte de définition par défaut du LBD, qui ne prend en compte que les littéraux affectés. Cette définition était utilisée dans la première version de *RoundingSat* [16].

Définition 3 (LBD_a)**.** *Considérons une* contrainte χ *et l'affectation de ses littéraux. Soit* π *une partition des littéraux* affectés*, calculée à partir du niveau de décision des littéraux. Le* LBD_a *de* χ *(« a » pour « affecté ») est le nombre d'éléments de* π*..*

Nous pouvons également supposer que les littéraux non affectés sont en fait affectés à un niveau de décision « imaginaire ». Ce niveau de décision peut être le même pour tous les littéraux, ou pas.

Définition 4 (LBD_s)**.** *Considérons une* contrainte χ *et l'affectation de ses littéraux. Soit* π *une partition des littéraux* affectés*, calculée à partir du niveau de décision des littéraux. Soit* n *le nombre d'éléments de* π*. Le* LBD_s *de* χ *(« s » pour « similaire ») est* n *si tous les littéraux de* χ *sont affectés, et* $n+1$ *sinon.*

Définition 5 (LBD_d)**.** *Considérons une* contrainte χ *et l'affectation de ses littéraux. Soit* π *une partition des littéraux* affectés*, calculée à partir du niveau de décision des littéraux. Soit* n *le nombre d'éléments de* π*. Le* LBD_d *de* χ *(« d » pour « différent ») est* $n+u$*, où* u *est le nombre de littéraux non affectés dans* χ*.*

Une autre extension possible du LBD est d'uniquement considérer les littéraux falsifiés, comme dans la version actuelle de *RoundingSat*.

Définition 6 (LBD_f)**.** *Considérons une* contrainte χ *et l'affectation de ses littéraux. Soit* π *une partition des littéraux* falsifiés*, calculée à partir du niveau de décision des littéraux. Le* LBD_f *de* χ *(« f » pour « falsifié ») est le nombre d'éléments de* π*.*

La définition ci-dessus part de l'observation que, lorsqu'une clause est apprise, tous les littéraux de cette clause sont falsifiés. Ils sont par ailleurs également effectifs, de sorte que nous pouvons également restreindre le calcul du LBD à ces littéraux.

Définition 7 (LBD_e)**.** *Considérons une* contrainte χ *et l'affectation de ses littéraux. Soit* π *une partition des littéraux* effectifs*, calculée à partir du niveau de décision des littéraux. Le* LBD_e *de* χ *(« e » pour « effectif ») est le nombre d'éléments de* π*.*

Exemple 6. *Les différentes mesures de* LBD*de la contrainte* χ *donnée par* $5a(0@3)+5b(1@3)+c(?@?)+d(?@?)+e(0@1)+f(1@2) \geq 6$ *sont :*

- $LBD_a(\chi) = |\{\{a,b\},\{e\},\{f\}\}| = 3$
- $LBD_s(\chi) = |\{\{a,b\},\{c,d\},\{e\},\{f\}\}| = 4$
- $LBD_d(\chi) = |\{\{a,b\},\{c\},\{d\},\{e\},\{f\}\}| = 5$
- $LBD_f(\chi) = |\{\{a\},\{e\}\}| = 2$
- $LBD_e(\chi) = |\{\{a\}\}| = 1$

Remarquons que les mesures de LBD présentées dans cette section sont des *extensions* de la définition originale du LBD (Définition 2), dans le sens où elles coïncident toutes lorsqu'elles sont calculées sur des clauses.

Nous proposons donc d'utiliser des stratégies de suppression fondées sur les mesures définies dans cette section (fondées sur le degré ou le LBD).

5 Redémarrages

Les redémarrages sont une fonctionnalité très puissante des solveurs SAT [18]. Bien qu'elle ne soit pas complètement comprise, elle semble requise pour mieux exploiter le pouvoir d'inférence de la résolution [33, 14, 1]. Redémarrer revient alors à oublier toutes les décisions faites par le solveur. Le principal avantage de cette approche est que les mauvaises décisions prises au début de l'exécution peuvent être annulées pour éviter au solveur d'être « bloqué » dans une sous-partie de l'espace de recherche. Dans ce but, plusieurs stratégies de redémarrage ont été proposées [8], soit statiques comme celles utilisant la suite de Luby [28, 22] soit dynamiques comme dans *PicoSAT* [5] ou *Glucose* [3]. Dans cette section, nous considérons ces dernières, en utilisant les mesures définies dans la Section 4.

De telles stratégies ne sont pas exploitées dans les solveurs PB actuels. Dans *pbChaff* [12] ou *Galena* [9], il n'est pas fait mention d'une implantation des redémarrages. Concernant *Pueblo* [36], qui est fondé sur *MiniSat* [13], il est très probable qu'il en hérite sa stratégie de redémarrage, même s'il n'en est pas fait mention dans [36] non plus. Concernant les solveurs plus récents, *Sat4j* [25] implante la politique de redémarrage statique et agressive de *PicoSAT* [6], tandis que *RoundingSat* [16] utilise la suite de Luby [28, 22]. Notons que ces deux stratégies sont indépendantes du type des contraintes apprises, (comme elles sont statiques), et peuvent donc être réutilisées en l'état.

Dans cette section, nous proposons au contraire d'utiliser l'approche de *Glucose* [3]. Dans ce solveur, un redémarrage est déclenché en fonction de l'évolution de la qualité des contraintes apprises : si celle-ci diminue, le solveur est vraisemblablement en train d'explorer la mauvaise partie de l'espace de recherche. En pratique, *Glucose* mesure cette qualité à l'aide du LBD (voir Définition 2). Pour mesurer son évolution, le LBD moyen est calculé sur les (100) dernières clauses apprises. Si cette moyenne est supérieure de 70% à la moyenne des LBD de toutes les clauses apprises, un redémarrage doit être réalisé.

Nous exploitons donc les mesures définies dans la section précédente pour définir des stratégies de redémarrage fondées sur ces mesures, en suivant la politique de *Glucose* décrite ci-dessus.

6 Résultats expérimentaux

Cette section présente des résultats expérimentaux des différentes stratégies introduites dans cet article et implantées dans deux solveurs PB, en l'occurrence *Sat4j* [25] et *RoundingSat* [16]. Nos expérimentations ont été exécutées sur un cluster équipé de processeurs quadricœurs Intel XEON X5550 (2.66 GHz, 8 Mo de cache). Le temps d'exécution était limité à 1200 secondes et la mémoire était limitée à 32 Go.

Par souci d'espace, cette section se limite aux performances des stratégies permettant d'améliorer le plus les solveurs considérés. Le lecteur intéressé pourra se référer aux résultats détaillés de nos expérimentations qui sont publiquement accessibles [26].

6.1 Configuration des solveurs

Commençons par décrire nos implantations des différentes stratégies dans *Sat4j* [25], disponibles sur son dépôt [1]. Pour ce solveur, nous considérons trois configurations, à savoir *Sat4j-GeneralizedResolution*, *Sat4j-RoundingSat* et *Sat4j-PartialRoundingSat* [24]. Pour ces trois configurations, les stratégies par défaut sont une heuristique de choix de variable qui incrémente le score de toutes les variables apparaissant dans les contraintes rencontrées pendant l'analyse de conflit (chaque fois que ces variables sont rencontrées), une stratégie de suppression des contraintes apprises considérant leur activité [13] (les contraintes supprimées sont celles moins impliquées dans les conflits récents), et la politique de redémarrage statique de *PicoSAT* [6].

Nos expérimentations ont montré que les meilleures stratégies pour *Sat4j-GeneralizedResolution* sont l'heuristique *bump-effective*, la stratégie de suppression des contraintes apprises utilisant le LBD_s, et la politique de redémarrage dynamique fondée sur la mesure *degree*. Pour *Sat4j-RoundingSat* et *Sat4j-PartialRoundingSat*, il s'agit de l'heuristique *bump-assigned*, la stratégie de suppression des contraintes apprises utilisant la mesure *degree-bits* et la politique statique de redémarrage de *PicoSAT* [6]. Dans la suite, nous considérons les combinaisons de ces stratégies comme la configuration `best` des différents solveurs mentionnés. Notons toutefois que toutes les combinaisons des stratégies n'ont pas été testées, compte-tenu des ressources de temps de calcul

1. https://gitlab.ow2.org/sat4j/sat4j/tree/cdcl-strategies

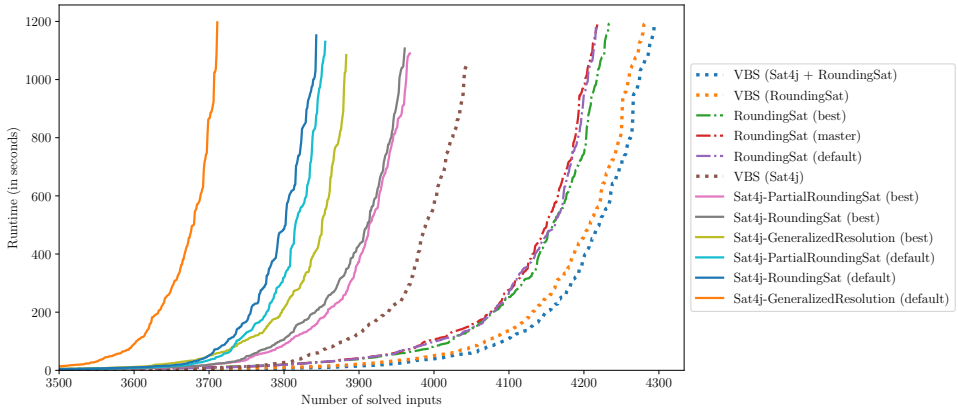


FIGURE 1 – « Cactus plot » de différentes configurations de *Sat4j* et de *RoundingSat* sur des problèmes de décision. Pour plus de lisibilité, les premières 3500 instances (faciles) sont ignorées.

nécessaires, et qu'il est possible que cette configuration ne soit pas la meilleure possible.

Pour *RoundingSat* [16], notre implantation est disponible dans un dépôt dédié [2]. Nous avons modifié ce solveur à partir du commit `a17b7d0e` (ci-après dénommé `master`) pour permettre l'utilisation des stratégies présentées dans cet article. La configuration `default` de ce solveur correspond à cette version modifiée de *RoundingSat* configurée avec les stratégies par défaut de sa version originale. Plus précisément, l'heuristique de choix de variable incrémente le score de toutes les variables apparaissant dans les contraintes rencontrées pendant l'analyse de conflit, (une seule fois, et deux fois si elles sont éliminées), la stratégie de suppression des contraintes apprises combinant leur LBD_f et leur activité, ainsi que la politique de redémarrage fondée sur la suite de Luby (avec un facteur 100) [22].

La combinaison des meilleures stratégies pour ce solveur (ci-après dénommée `best`) se compose, d'après nos expérimentations, de l'heuristique *bump-assigned* (avec une incrémentation du score des variables à chaque fois qu'elles sont rencontrées), et des stratégies de suppression des contraintes et de redémarrage fondées sur le LBD_e.

6.2 Problèmes de décision

Considérons dans un premier temps les performances des différents solveurs sur des problèmes de décision. Dans ce but, nous les avons exécutés sur la totalité des problèmes de décision utilisés dans les compétitions PB depuis la première édition [29, 34] et contenant des « petits » entiers, pour un total de 5582 instances. La Figure 1 donne les résultats obtenus pour les différents solveurs, dans leurs configurations `default` et `best`.

Le « cactus plot » montre que les différentes configurations de *Sat4j* sont significativement améliorées par nos stratégies dédiées. Notons de plus que la meilleure configuration de *Sat4j-GeneralizedResolution* est plus efficace que les implantations par défaut de *RoundingSat* dans *Sat4j*. Nous pouvons également noter une légère amélioration de *RoundingSat* par rapport à sa configuration par défaut, qui n'est cependant pas aussi importante que dans *Sat4j*.

Remarquons que combiner les meilleures stratégies n'est pas suffisant pour obtenir le meilleur de toutes les stratégies étudiées. En particulier, pour chaque fonctionnalité considérée, le *VBS* (pour *Virtual Best Solver*) des différentes stratégies, c'est-à-dire, le solveur obtenu en sélectionnant la meilleure stratégie pour chaque instance, a de bien meilleures performances que les stratégies considérées individuellement. Ce constat s'applique par ailleurs à toutes les configurations de *Sat4j* et *RoundingSat*. Ceci suggère qu'aucune stratégie n'est meilleure que les autres sur toutes les instances, et qu'elles sont en fait complémentaires.

6.3 Problèmes d'optimisation

Considérons maintenant les performances des différents solveurs sur des problèmes d'optimisation. Dans ce but, nous avons exécuté ces solveurs sur la totalité des problèmes d'optimisation utilisés dans les compétitions PB depuis la première édition [29, 34] et contenant des « petits » entiers, pour un total de 4374 instances. Au vu du temps de calcul considérable nécessaire pour réaliser nos expérimentations exhaustives sur les problèmes de décision (plus de 8 ans CPU), nous considérons ici uniquement les meilleures configurations des solveurs identifiées sur les problèmes de

2. https://gitlab.com/pb-cdcl-strategies/roundingsat/-/tree/cdcl-strategies

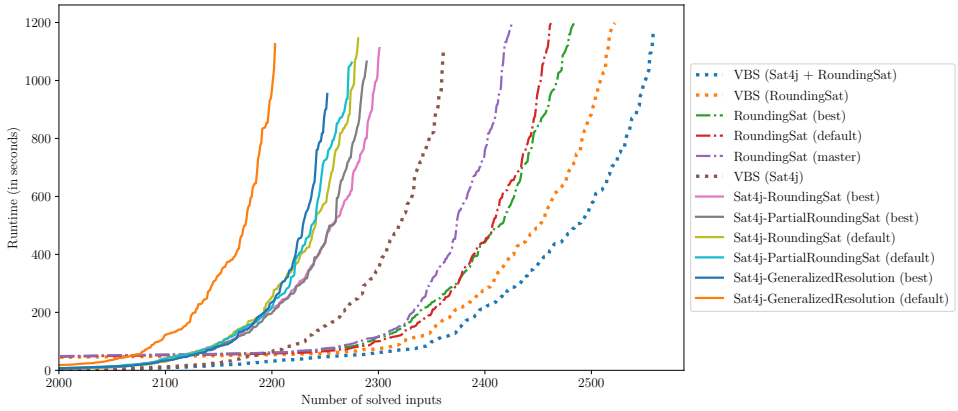


FIGURE 2 – « Cactus plot » de différentes configurations de *Sat4j* et de *RoundingSat* sur des problèmes d'optimisation. Pour plus de lisibilité, les premières 2000 instances (faciles) sont ignorées.

décision (ce qui a demandé malgré tout 9 mois CPU de calcul). La Figure 2 donnent les résultats obtenus.

Comme pour les problèmes de décision, nous pouvons observer sur le cactus plot que tous les solveurs sont améliorés par les stratégies dédiées sur les problèmes de décision, avec une amélioration particulièrement importante pour *Sat4j-GeneralizedResolution*.

6.4 Discussion

Sans surprise, la stratégie ayant le plus d'impact, en particulier sur *Sat4j*, est l'heuristique de choix de variable. En pratique, il s'agit de la stratégie individuelle qui améliore le plus les performances de *Sat4j*. Par exemple, nous avons constaté que *Sat4j-GeneralizedResolution* résout les problèmes (d'optimisation) de la famille `factor` beaucoup plus efficacement grâce à la stratégie *bump-effective* (modifier les stratégies de suppression et de redémarrage ne change presque rien pour cette famille). Nous avons étudié plus en détail le comportement du solveur sur ces instances pour mieux comprendre ce gain, et nous nous sommes aperçus que la production de littéraux non-pertinents (c'est-à-dire, des littéraux apparaissant dans la contrainte, mais n'affectant jamais sa valeur de vérité, quelque soit leur affectation) pénalise fortement le solveur sur cette famille. Il est connu que ces littéraux peuvent impacter la taille des preuves produites par les solveurs PB [23]. Nos expérimentations montrent ici qu'ils peuvent aussi polluer l'heuristique du solveur. Notons en effet que l'heuristique *bump-effective* n'incrémente jamais le score de ces littéraux, qui sont toujours ineffectifs. Cette heuristique propose donc une nouvelle approche pour composer avec ces littéraux.

L'impact important des variantes d'EVSIDS dans *Sat4j* peut aussi expliquer pourquoi le gain obtenu avec *RoundingSat* est plus limité. En effet, les affaiblissements agressifs appliqués par *RoundingSat* (qui sont par ailleurs plus agressifs que ceux pratiqués dans *Sat4j-RoundingSat* et *Sat4j-PartialRoundingSat*) ont tendance à identifier les littéraux impliqués dans le conflit. Ceci est particulièrement visible si l'on regarde le comportement des différentes stratégies de *bumping* dans *RoundingSat* : il n'y a presque aucune différence entre elles. Ceci suggère que le gain sur ce solveur est en grande partie obtenu par les stratégies de suppression des contraintes apprises et de redémarrage, dont l'impact sur les performances du solveur est plus limité que celui de l'heuristique de choix de variable en général.

Nous avons notamment pu observer que, dans *Sat4j*, ne jamais supprimer les contraintes apprises est en fait préférable à la stratégie (par défaut) supprimant les contraintes de faible activité. Cela peut s'expliquer par le fait que les solveurs PB sont souvent plus lents en pratique que le solveurs SAT, car les opérations qu'ils doivent réaliser, comme la détection des propagations ou l'application de la résolution généralisée, sont plus complexes que leur pendant dans les solveurs SAT. Cela signifie que le nombre de conflits par seconde dans un solveur PB est plus faible que dans un solveur SAT, tout comme le nombre de contraintes apprises. En conséquence, les solveurs PB n'ont pas besoin d'effacer des contraintes aussi souvent qu'un solveur SAT.

De plus, il est important de noter que les différentes stratégies considérées sont souvent très liées au sein du solveur, et peuvent donc interagir entre elles. C'est particulièrement vrai pour les stratégies de suppression des contraintes apprises et de redémarrage, qui utilisent les mêmes mesures de qualité. Par exemple, alors que les meilleures stratégies individuelles dans *RoundingSat* sont la politique de redémarrage de *PicoSAT* et la suppression utilisant le LBD_s, le meilleur gain est obtenu

en utilisant le LBD_e pour ces deux fonctionnalités.

Une autre conséquence des liens étroits entre les différentes stratégies dans le solveur est que des détails d'implantation peuvent avoir des effets de bord inattendus sur les performances du solveur. Par exemple, pour implanter les nouvelles stratégies dans *RoundingSat*, nous avons dû adapter son code et changer certaines structures de données dans l'heuristique de choix de variable (en remplaçant un ensemble ordonné par une structure associative ne préservant pas l'ordre), faisant que les scores des *mêmes* variables sont incrémentés, mais dans un *ordre différent*. Comme l'ordre d'insertion et de mise à jour des variables est utilisé comme départage par EVSIDS, l'ordre dans lequel les variables sont sélectionnées varie entre les configurations `master` et `default` du solveur, ce qui rend plus difficile l'interprétation des résultats de *RoundingSat*, notamment dans le cas des problèmes d'optimisation.

7 Conclusion

Dans cette article, nous avons présenté différentes heuristiques, stratégies de suppression des contraintes apprises et politiques de redémarrage dédiées à la résolution native de problèmes PB. Ces stratégies généralisent celles implantées dans les solveurs SAT, et sont conçues pour tenir compte des propriétés des contraintes PB pour s'adapter au contexte CDCL. Nos expérimentations ont montré que l'un des aspects essentiels des contraintes PB à considérer est l'impact de l'affectation courante sur leurs littéraux. C'est en particulier vrai pour les heuristiques fondées sur EVSIDS, mais aussi pour les stratégies de suppression des contraintes apprises et de redémarrage, au travers de nouvelles mesures de LBD. Combinées, ces stratégies permettent d'améliorer les performances de *RoundingSat* et *Sat4j*, avec des améliorations significatives pour ce dernier, à la fois sur des problèmes de décision et d'optimisation.

Néanmoins, aucune des ces stratégies n'est meilleure que les autres sur toutes les instances : leur *VBS* est clairement meilleur que chaque stratégie individuelle, même en considérant leur association. Cet article a cependant montré qu'une meilleure adaptation de ces stratégies aux solveurs PB peut améliorer leurs performances. Une perspective dans ce domaine serait de trouver de meilleures manières d'adapter les différentes stratégies considérées, voire d'en définir de nouvelles conçues *spécifiquement* pour les solveurs PB. Une autre piste à explorer est de trouver comment combiner au mieux ces stratégies pour en obtenir le meilleur, tout en considérant les interactions qu'elles peuvent avoir, par exemple en utilisant des techniques de configuration dynamique d'algorithmes pour sélectionner les stratégies les plus appropriées en fonction de l'état du solveur [4].

Remerciements

Les auteurs remercient les relecteurs pour leurs nombreux commentaires ayant permis d'améliorer cet article. Ces travaux ont partiellement bénéficié du soutien du Ministère de l'Enseignement Supérieur et de la Recherche et du Conseil Régional des Hauts-de-France au travers du « Contrat de Plan État Région (CPER) DATA ». Cet article bénéficie du soutien de la Chaire « Integrated Urban Mobility » portée par l'X - École Polytechnique et La Fondation de l'École Polytechnique, et soutenue par Uber. La responsabilité des Partenaires de la Chaire ne peut en aucun cas être mise en cause en raison du contenu de la présente communication, qui n'engage que son auteur.

Références

[1] Albert Atserias, Johannes Klaus Fichte et Marc Thurley : Clause-Learning Algorithms with Many Restarts and Bounded-Width Resolution. *JAIR*, 40:353–373, 2011.

[2] Gilles Audemard et Laurent Simon : Predicting Learnt Clauses Quality in Modern SAT Solvers. *In Proceedings of IJCAI'09*, pages 399–404, 2009.

[3] Gilles Audemard et Laurent Simon : Refining Restarts Strategies for SAT and UNSAT. *In* Michela Milano, éditeur : *Proceedings of CP'12*, pages 118–126, 2012.

[4] André Biedenkapp, H. Furkan Bozkurt, Theresa Eimer, Frank Hutter et Marius Lindauer : Dynamic Algorithm Configuration : Foundation of a New Meta-Algorithmic Framework. *In Proceedings of ECAI'20*, 2020.

[5] Armin Biere : Adaptive Restart Strategies for Conflict Driven SAT Solvers. *In Proceedings of SAT 2008*, pages 28–33, 2008.

[6] Armin Biere : PicoSAT Essentials. *JSAT*, 4(2-4):75–97, 2008.

[7] Armin Biere et Andreas Fröhlich : Evaluating CDCL variable scoring schemes. *In Proceedings of SAT'15*, pages 405–422, 2015.

[8] Armin Biere et Andreas Fröhlich : Evaluating CDCL Restart Schemes. *In Proceedings of Pragmatics of SAT 2015 and 2018*, volume 59 de *EPiC Series in Computing*, pages 1–17, 2019.

[9] Donald Chai et Andreas Kuehlmann : A fast pseudo-Boolean constraint solver. *IEEE Trans. on CAD of Integrated Circuits and Systems*, pages 305–317, 2005.

[10] William Cook, Collette R. Coullard et György Turán : On the Complexity of Cutting-plane Proofs. *Discrete Appl. Math.*, pages 25–38, 1987.

[11] Heidi Dixon : *Automating Pseudo-Boolean Inference Within a DPLL Framework*. Thèse de doctorat, University of Oregon, 2004.

[12] Heidi E. Dixon et Matthew L. Ginsberg : Inference Methods for a Pseudo-Boolean Satisfiability Solver. *In Proceedings of AAAI'02*, pages 635–640, 2002.

[13] Niklas Eén et Niklas Sörensson : An Extensible SAT-solver. *In Proceedings of SAT'04*, pages 502–518, 2004.

[14] Jan Elffers, Jesús Giráldez-Crú, Jakob Nordström et Marc Vinyals : Using Combinatorial Benchmarks to Probe the Reasoning Power of Pseudo-Boolean Solvers. *In Proceedings of SAT'18*, pages 75–93, 2018.

[15] Jan Elffers, Jesús Giráldez-Cru, Stephan Gocht, Jakob Nordström et Laurent Simon : Seeking Practical CDCL Insights from Theoretical SAT Benchmarks. *In Proceedings of IJCAI'18*, pages 1300–1308, 2018.

[16] Jan Elffers et Jakob Nordström : Divide and Conquer : Towards Faster Pseudo-Boolean Solving. *In Proceedings of IJCAI'18*, pages 1291–1299, 2018.

[17] Stephan Gocht, Jakob Nordström et Amir Yehudayoff : On Division Versus Saturation in Pseudo-Boolean Solving. *In Proceedings of IJCAI'19*, pages 1711–1718, 2019.

[18] Carla P. Gomes, Bart Selman et Henry Kautz : Boosting Combinatorial Search through Randomization. *In Proceedings of AAAI'98*, page 431–437, 1998.

[19] Ralph E. Gomory : Outline of an algorithm for integer solutions to linear programs. *Bulletin of the American Mathematical Society*, pages 275–278, 1958.

[20] Armin Haken : The intractability of resolution. *Theoretical Computer Science*, pages 297–308, 1985.

[21] John N. Hooker : Generalized resolution and cutting planes. *Annals of Operations Research*, pages 217–239, 1988.

[22] Jinbo Huang : The Effect of Restarts on the Efficiency of Clause Learning. *In Proceedings of IJCAI'07*, pages 2318–2323, 2007.

[23] Daniel Le Berre, Pierre Marquis, Stefan Mengel et Romain Wallon : On Irrelevant Literals in Pseudo-Boolean Constraint Learning. *In Proceedings of IJCAI'20*, pages 1148–1154, 2020.

[24] Daniel Le Berre, Pierre Marquis et Romain Wallon : On Weakening Strategies for PB Solvers. *In Proceedings of SAT'20*, pages 322–331, 2020.

[25] Daniel Le Berre et Anne Parrain : The SAT4J library, Release 2.2, System Description. *JSAT*, pages 59–64, 2010.

[26] Daniel Le Berre et Romain Wallon : On Dedicated CDCL Strategies for PB Solvers Companion Artifact, mai 2021.

[27] Jia Hui Liang, Vijay Ganesh, Pascal Poupart et Krzysztof Czarnecki : Learning Rate Based Branching Heuristic for SAT Solvers. *In Proceedings of SAT'16*, pages 123–140, 2016.

[28] Michael Luby, Alistair Sinclair et David Zuckerman : Optimal speedup of Las Vegas algorithms. *In Information Processing Letters*, pages 173–180, 1993.

[29] Vasco Manquinho et Olivier Roussel : The First Evaluation of Pseudo-Boolean Solvers (PB'05). *JSAT*, pages 103–143, 2006.

[30] Joao Marques-Silva et Karem A. Sakallah : GRASP : A Search Algorithm for Propositional Satisfiability. *IEEE Trans. Computers*, pages 220–227, 1999.

[31] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang et Sharad Malik : Chaff : Engineering an Efficient SAT Solver. *In Proceedings of DAC'01*, pages 530–535, 2001.

[32] Jakob Nordström : On the Interplay Between Proof Complexity and SAT Solving. *ACM SIGLOG News*, pages 19–44, 2015.

[33] Knot Pipatsrisawat et Adnan Darwiche : On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011.

[34] Olivier Roussel : Pseudo-Boolean Competition 2016. http://www.cril.fr/PB16/, 2016 (accessed May 20, 2020).

[35] Olivier Roussel et Vasco M. Manquinho : Pseudo-Boolean and Cardinality Constraints. *In* Armin Biere, Marijn Heule, Hans van Maaren et Toby Walsh, éditeurs : *Handbook of Satisfiability*, volume 185 de *Frontiers in Artificial Intelligence and Applications*, pages 695–733. IOS Press, 2009.

[36] Hossein M. Sheini et Karem A. Sakallah : Pueblo : A Hybrid Pseudo-Boolean SAT Solver. *JSAT*, pages 165–189, 2006.

[37] Marc Vinyals, Jan Elffers, Jesús Giráldez-Crú, Stephan Gocht et Jakob Nordström : In Between Resolution and Cutting Planes : A Study of Proof Systems for Pseudo-Boolean SAT Solving. *In Proceedings of SAT'18*, pages 292–310, 2018.

Génération d'ensembles de modèles explorables par couplage de contraintes et de transformation de modèles

Théo Le Calvar[1,2,3] **Fabien Chhel**[2]
Frédéric Jouault[2] **Frédéric Saubion**[1] **Eugene Syriani**[3]
[1] Univ Angers, LERIA, SFR MATHSTIC, 49000 Angers,France
[2] ERIS Team, ESEO Group, Angers, France
[3] DIRO, Université de Montréal, Montréal, Canada
prénom.nom@{univ-angers.fr,eseo.fr}, nom@iro.umontreal.ca

Résumé

Cet article présente les travaux réalisés durant les projets COMOG et Optimodon. Ces travaux s'inscrivent dans les thématiques de l'ingénierie dirigée par les modèles (IDM) et y intègrent des notions de programmation par contraintes. Ces travaux se sont concrétisés par le développement d'ATLc, une extension d'ATL, un langage de transformation de modèles, intégrant des contraintes.

ATLc repose sur la notion d'exploration d'ensembles de modèles. L'exploration d'ensemble de modèles permet à l'utilisateur, grâce à l'utilisation d'ATLc, de générer un ensemble de modèles valides défini en intention par des contraintes.

Dans cet article nous présentons les changements apportés à ATLc au travers de plusieurs cas d'études. Ces changements se concentrent sur deux axes majeurs. Ajouts de sucres syntaxiques au langage pour faciliter l'utilisation. Ajout d'une seconde cible d'exécution, les navigateurs web, par l'utilisation de SVG et de JavaScript. Cette seconde plateforme d'exécution ajoute de nombreuses petites améliorations et est spécialisée dans les diagrammes interactifs.

Abstract

This article presents works realised during COMOG and Optimodon projects. These works integrate constraint solving into model driven ingineering (MDE) and more precisely into model transformation. They materialized with the developement of ATLc, an extension of ATL, a model transformation langage, which integrate constraints.

ATLc uses the model set exploration concept. Model set exploration allows user to generate set of valid models and navigate into the set thank to the use of constraints solver.

In this article we present recent changes to ATLc thourh several case studies. These changes focus on two key points. First, improvements to the langage itself, making it easier to use. Second, adding another execution target to ATLc, the first versions were only running on JavaFX and JVM. This new version allows user to compile their transformations to SVG and JavaScript. This new ATLc compiler is specialized to generate interactive diagrams and feature many quality of life improvements other the older compiler.

1 Introduction

La transformation de modèles s'est imposée comme une technique clef de l'ingénierie dirigée par les modèles (IDM). L'IDM est une branche du génie logiciel prônant l'utilisation de modèles tout au long du cycle de production de systèmes informatiques. L'utilisation de modèles permet d'adapter les concepts manipulés par l'utilisateur à la tâche qu'il doit effectuer, réduisant ainsi la complexité de cette tâche. De plus, l'IDM recommande l'automatisation des traitements des modèles lorsque cela est possible.

La transformation de modèles permet l'automatisation du traitement des modèles et il est ainsi possible d'automatiser le raffinement de modèles abstraits vers de nouveaux modèles plus concrets. Ceci offre la possibilité de travailler sur des modèles de haut niveau d'abstraction qui seront, par l'application d'une suite de transformations, automatiquement raffinés en modèles concrets, pouvant aller jusqu'à la production du code spécifique à une plateforme d'exécution donnée.

Il existe de nombreuses approches de transformation de modèles offrant différents avantages en fonction des utilisations visées (comme l'incrémentalité, la bidirectionalité, ou la *lazyness*). Cependant, la majorité de ces approches se limitent à la génération d'un unique modèle cible pour un modèle source donné. Lorsque plusieurs modèles cibles peuvent correspondre à unique modèle source, il peut être pertinent d'avoir plutôt accès à cet ensemble de modèles.

Avec ATLc et l'exploration de modèles, nous proposons une approche permettant à une transformation de modèles de générer des ensembles de modèles explorables grâce à l'application de contraintes à un modèle partiellement instancié. Ces contraintes permettent de définir en intention l'ensemble des modèles valides et un solveur de contraintes permet de calculer ces solutions. De plus, nous permettons à l'utilisateur de suggérer des changements au modèle qui lui est présenté. Ces changements sont validés par le solveur de contraintes qui pourra, au besoin, réparer le modèle pour assurer la validité des contraintes.

Nous illustrerons différents aspects d'ATLc/web, une nouvelle implémentation d'ATLc permettant l'exécution sur des plateformes web grâce à l'utilisation de SVG et JavaScript, au travers de plusieurs exemples de transformations. Plus de détails sur l'approche et l'implémentation en Java sont disponibles dans [6].

La résolution des contraintes générées est assurée par Cassowary.js [1], une implémentation en JavaScript du solveur linéaire incrémental Cassowary [5]. Cassowary repose principalement sur l'algorithme du simplexe et offre en pratique de très bonnes performances.

Dans les sections 2 et 3 nous illustrons les capacités d'ATLc au travers de deux exemples de diagrammes (respectivement le diagramme de features et le diagramme de séquences) pouvant s'exécuter sur le web. Nous concluons dans la section 4.

Tous les outils présentés dans cet article ainsi que les exemples sont disponibles à [2]. De plus, les exemples présentés dans cet article sont disponibles directement à [3].

2 Diagramme de features

La Figure 1 montre un diagramme de features généré par la transformation. En plus de la version statique visible via la démonstration [3], une version éditable en ligne est disponible [4].

La transformation de modèles avec ATLc reprend les mêmes bases qu'en ATL classique. Une transformation est composée d'un ensemble de règles spécifiant un ou plusieurs éléments à transformer (section `from`) et un ou plusieurs éléments à créer (section `to`). Pour chaque élément créé, il est possible de spécifier les valeurs qui devront être affectées à ses différentes propriétés en utilisant des *bindings* (notés avec l'opérateur `<-`). ATLc reprend la même syntaxe, mais interprète les expressions des contraintes en sorties comme des contraintes et non comme des expressions booléennes.

Avec cette nouvelle version d'ATLc nous permettons l'exécution des transformations dans un navigateur web récent. Pour cela nous compilons les règles ATLc en éléments SVG (`<defs>`) et en code JavaScript servant à créer les contraintes. Ces patrons peuvent ensuite être instanciés par l'ajout d'élément SVG (`<use>`) paramétrables. Grâce à cela il est possible d'intégrer les diagrammes générés avec ATLc dans n'importe quelle page web.

Une des améliorations majeures de cette nouvelle version d'ATLc est la possibilité d'intégrer directement du SVG dans la transformation. Par exemple, dans le Listing 1 il n'y a que deux éléments de sortie (`t` l.5-12 et `cstr` l.13-25). Cependant, le groupe `t` utilise du SVG pour ajouter des éléments de sortie (l. 7-11) sans la lourdeur de l'ATL classique. Il est possible de spécifier des valeurs pour les attributs ou contenus des éléments spécifiés en SVG grâce à l'annotation `${propSource}`.

```
1  unique lazy rule Feature {
2    from
3      s : FeatureDiagram!Feature
4    to
5      t: SVG!Group (
6        class <- 'feature',
7        content <- '
8          <rect id=".box" />
9          <rect id=".outline" />
10         <text id=".label">${s.label}</text>
11       '
12     ),
13     cstr: Constraints!ExpressionGroup (
14       constraints <-
15         outline.width = label.width + thisModule
               .MARGIN
16     and outline.height = label.height +
           thisModule.MARGIN
17     and outline.center = label.center
18     and outline.topLeft.stay('medium')
19     and box.mustContain(outline)
20     and box.center.x = outline.center.x
21     and box.width = 0 -- strong
22     and box.height = 0  -- strong
23     and box.x >= 0
24     and box.y >= 0
25     )
26 }
```

Listing 1: Règle de transformation ATLc pour une `Feature`

La seconde règle de cet exemple, visible dans le Listing 2, illustre quant à elle les améliorations apportées au dialecte de contraintes. Ce nouveau compilateur ATLc impose aux transformations un typage bien plus strict que les versions précédentes. La notation des types en ATL peut être fastidieuse, notamment lors du typage du résultat de l'application d'une règle. Pour

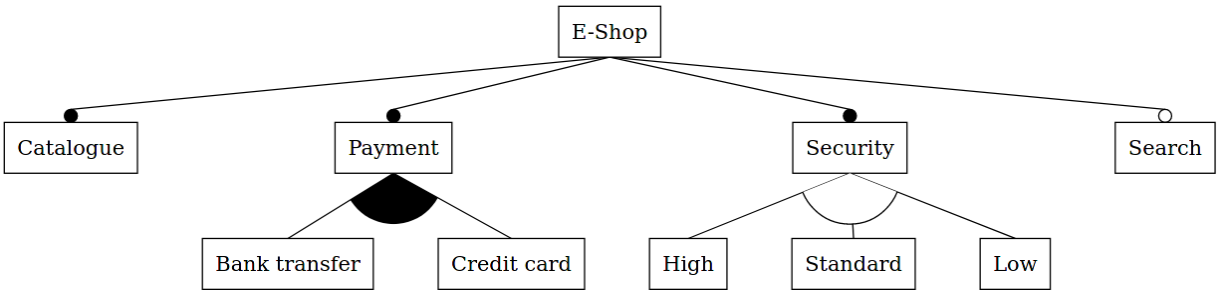


FIGURE 1 – Capture d'écran d'un diagramme de feature généré par ATLc

faciliter ceci, nous avons ajouté un type spécial, marqué `INFER!TYPE` (l.16), indiquant au compilateur d'inférer le type par rapport au contexte.

Dernière particularité de cet exemple. Les lignes 26 à 38 montrent un exemple d'expressions OCL complexes gérées par cette nouvelle version d'ATLc. Cette expression utilise de nombreuses opérations non supportées dans les anciennes versions. On notera l'utilisation des opérations `zipWith` et `forAll` non standard, mais ajoutée ici par soucis pratique.

3 Diagramme de séquences

La Figure 2 montre un exemple de digramme de séquence généré par ATLc. Comme pour le diagramme de *feature*, la transformation complète est accessible [2], ainsi que diagramme de la Figure 2 [3].

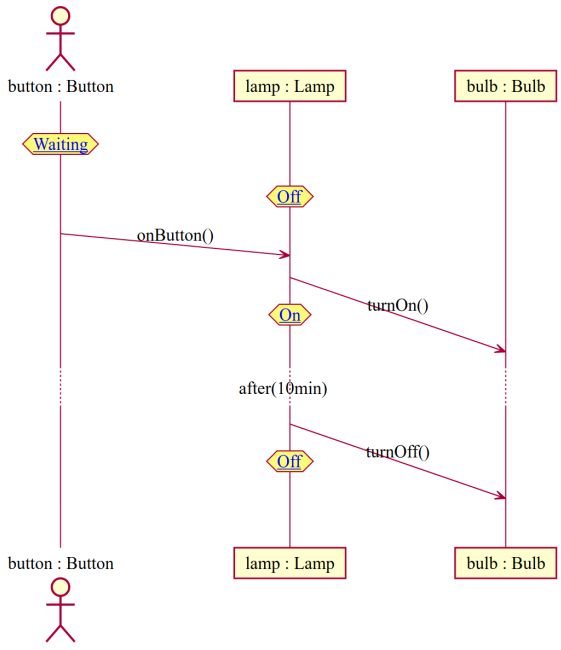


FIGURE 2 – Capture d'écran d'un diagramme de séquence généré avec ATLc

Cette nouvelle version d'ATLc permet l'utilisation de contraintes dans des *helpers* ATL. Un *helper* ATL peut être vu comme une fonction pouvant être appelée dans les règles de transformation. Le Listing 3 montre un *helper* ATL générant des contraintes. Ceci se fait par l'ajout d'une annotation en commentaire avant le *helper* (l.1). L'utilisation de *helpers* permet de factoriser le code de contraintes tout au long de la transformation. Il est possible d'appeler des *helper* depuis un *helper* (l.18).

Ce *helper* illustre aussi le support étendu d'OCL dans les expressions de contraintes. En effet, cette nouvelle version d'ATLc permet l'utilisation de structures conditionnelles dans les contraintes. Nous avons deux structures conditionnelles, le `if` (non utilisé dans ce *helper*) et le `implies` (l.6).

L'ajout de `if` permet de conditionner l'ajout de contraintes par une condition sur le modèle source. Cependant, cette condition n'est pas envoyée au solveur de contraintes, mais est gérée avant, par le code JavaScript postant les contraintes. Ainsi il est possible de contrôler finement l'activation de contraintes sur des solveurs ne le supportant pas.

Le `implies` est un sucre syntaxique pour les `if` sans `else` (ATL impose un `else` aux `if`).

4 Conclusion

Dans cet article nous avons présenté une version d'ATLc pouvant générer des diagrammes interactifs sur le web. Ces diagrammes utilisent SVG et JavaScript et sont donc facilement intégrables dans une page web. La résolution des contraintes est assurée par Cassowary. Nous travaillons actuellement à l'intégration de diagrammes spécifiés avec ATLc dans l'éditeur Gentleman [7] comme primitives de projection et ainsi permettre une plus grand liberté que ce qui serait uniquement faisable en HTML.

La nature déclarative d'ATLc nous a permis de facilement développer ce second environnement d'exécution adapté au web. De plus, nous avons profité de

```
unique lazy rule Alternative {
  from
    s: FeatureDiagram!Alternative
  to
    t: SVG!Group (
      class <- 'arc',
      content <- '
        <clipPath id=".clipPath">
          <polygon id=".poly" />
        </clipPath>
        <circle id=".outline" fill="${s.fill}"
            stroke="${s.stroke}" clip-path="url
            (#%this%.clipPath)"/>
      '
    ),
    cstr: Constraints!ExpressionGroup (
      constraints <-
        let parent : INFER!TYPE =
          thisModule.Feature(s.arcs->first().
              parent) in
        let firstChild : INFER!TYPE =
          thisModule.Feature(s.arcs->first().
              child) in
        let lastChild : INFER!TYPE =
          thisModule.Feature(s.arcs->last().
              child) in

          outline.center = parent.outline.
              bottom.center
        and ...
        and (
          let childArcs : INFER!TYPE = s.arcs->
              collect(arc|
            thisModule.Feature(arc.child)
          )
          in
          Tuple {
            left = childArcs,
            right = childArcs->subSequence(2,
                childArcs->size())
          }->zipWith(cur, next |
              cur.box.topRight.x
              + thisModule.HORIZONTAL_SEPARATION
            =
              next.box.x
          )->forAll(e | e)
        )
    )
}
```

Listing 2: Extrait d'une règle de transformation ATLc pour une `Alternative`

la flexibilité de cette nouvelle plateforme d'exécution pour ajouter de nouvelles fonctionnalités à ATLc. Ces nouveautés, comme la possibilité d'intégrer du SVG ou un meilleur support d'OCL dans la spécification des contraintes, facilitent l'utilisation du langage en réduisant la complexité globale de la transformation.

Tous ces outils, ainsi que les exemples de cet article, sont libres et disponibles [2].

Remerciements

Ces travaux sont financés par le projet de la région des Pays de la Loire RFI Atlanstic 2020.

```
-- @constraints (
helper def: participantConstraints(part : Seq!
    Participant) : Boolean =
  let tgt : INFER!TYPE = thisModule.Participant(
      part) in
  let afterTgt : INFER!TYPE = thisModule.
      Participant(part.after)
  in
  ((not part.x.oclIsUndefined()) implies (
      tgt.outline.x = part.x
    and tgt.outline.y = part.y
  ))
  and ((not part.after.oclIsUndefined()) implies
      (
    tgt.outline.bottom.p1.y = afterTgt.outline.
        bottom.p1.y
    and tgt.outline.center.x >= afterTgt.outline
        .center.x
      + (tgt.outline.width + afterTgt.outline.
          width) / 2
      + thisModule.INTER_OBJECT_MARGIN
    and tgt.line.p2.y = afterTgt.line.p2.y
  ))
  and ...
  and thisModule.textOutline(tgt.outline, tgt.
      name)
  and ...;
```

Listing 3: Extrait d'un *helper* contenant des contraintes

Références

[1] Cassowary.js project page. https://github.com/slightlyoff/cassowary.js. 2021-05-14.

[2] Code source des compilateur atlc et autres outils liés à atl. https://github.com/ESEO-Tech/ATL-Tools-Library. 2021-06-04.

[3] Démo en ligne de diagrammes générés avec atlc. https://eseo-tech.github.io/ATL-Tools-Library/atlc/. 2021-06-04.

[4] Éditeur de feature diagram en ligne utilisant une transformation atlc. https://aof.page.kher.nl/feature-diagram/. 2021-06-04.

[5] Greg J Badros, Alan Borning et Peter J Stuckey : The Cassowary linear arithmetic constraint solving algorithm. *TOCHI*, 8(4):267–306, 2001.

[6] Théo Le Calvar, Fabien Chhel, Frédéric Jouault et Frédéric Saubion : Coupling solvers with model transformations to generate explorable model sets. *Software and Systems Modeling*, feb 2021.

[7] Louis-Edouard Lafontant et Eugene Syriani : Gentleman. *In Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems : Companion Proceedings*. ACM, oct 2020.

Actes JFPC 2021

Génération de donjons à l'aide de la programmation par contraintes *

Gaël Glorian[1,2] **Adrien Debesson**[2] **Sylvain Yvon-Paliot**[2] **Laurent Simon**[1]

[1] LaBRI – CNRS UMR 5800, Université de Bordeaux, Talence, Nouvelle-Aquitaine, France
[2]Ubisoft, Bordeaux, Nouvelle-Aquitaine, France

{prenom.nom}@labri.fr {prenom.nom}@ubisoft.com

Résumé

L'industrie du jeu vidéo est l'une des plus importantes industries du secteur du loisir, générant des milliards de dollars de chiffre d'affaire chaque année. Les jeux vidéos se doivent donc d'offrir des expériences de jeu de plus en plus complexes, impliquant des équipes de développeurs et d'artistes de plus en plus importantes. Dans cet article, nous proposons une approche basée sur la programmation par contraintes pour la génération procédurale de donjons dans un contexte de monde/univers ouvert, dans le but d'offrir aux joueurs des mondes ouverts mais offrant une excellente cohérence et une narration de qualité. Grâce à une description globale capturant toutes les salles et situations possibles d'un donjon donné, notre approche propose d'énumérer des variations de ce schéma global, pouvant alors être présentées au joueur pour plus de diversité. Nous formalisons ce problème à l'aide de la programmation par contraintes en exploitant une abstraction sous forme de graphe de la structure du donjon, sur laquelle chaque chemin intéressant représente une variation possible correspondant aux contraintes désirées. Pour ce faire, nous introduisons un nouveau propagateur étendant la contrainte de graphe *connected*, qui permet de considérer des graphes dirigés avec cycles. Nous montrons que, grâce à cette modélisation et le nouveau propagateur proposé, il est possible de prendre en compte des scénarios réalistes pouvant être utilisés dans les jeux AAA. Nous démontrons son efficacité en la comparant à une solution de base consistant à filtrer uniquement les solutions pertinentes a posteriori.

Abstract

The video games industry generates billions of dollars in sales every year. However, videos games are more and more complex, involving larger and larger teams of developers and artists to offer gigantic (but consistent) open worlds to players. In this paper, we propose a constraint-based approach to the procedural content generation of dungeons in an open world context. Thanks to a global description capturing all the possible rooms and items of a given dungeon, our approach allows enumerating variations of this global pattern that can be presented to the player for more diversity. We formalize this problem in constraint programming by exploiting a graph abstraction of the dungeon pattern structure on which each interesting path represents a possible variation matching a given set of constraints. We introduce a new propagator extending the "connected" graph constraint, which allows considering directed graphs with cycles. We show that, thanks to this new propagator, it is possible to handle realistic scenarios used in the game industry. We also demonstrate its efficiency by comparing it with a more basic solution consisting of only filtering relevant solution a posteriori. We then conclude and offer several interesting perspectives raised by this approach to the *Dungeon Variations Problem*.

*Ce travail a été soutenu par le projet « KIWI » de la région Nouvelle-Aquitaine.

1 Introduction

L'industrie du jeu vidéo est l'une des plus importantes industries du loisir, toujours à la pointe des innovations, générant des milliards de dollars de chiffre d'affaire chaque année. Depuis l'apparition des premiers jeux dans les années 1970, le paysage de cette industrie a radicalement changé. Les jeux vidéos sont désormais produits par de grandes équipes d'artistes et de développeurs, offrant des graphismes photo-réalistes et un degré de simulation constamment amélioré.

L'un des plus grands défis de l'industrie du jeu est de pouvoir construire des mondes ouverts, dans lesquels les utilisateurs doivent se sentir libres et où un nombre colossal d'actions leur sont possibles. Générer

un tel monde sans intervention finale d'un Level Designer (*LD*) pour valider le niveau est toujours un rêve : même si le jeu est très bien conçu, un seul niveau de jeu incohérent peut rapidement briser la réputation d'un jeu, même ayant coûté des millions de dollars. La vérification formelle des niveaux générés peut jouer un rôle essentiel dans l'avenir de l'industrie du jeu. Cependant, s'il est encore trop difficile de vérifier formellement les propriétés des niveaux générés *a posteriori*, nous proposons, dans cet article, de prouver les propriétés des niveaux générés par construction. Intuitivement, dans l'approche proposée, un *LD* produit ce que nous appelons un *donjon source* qui contient un ensemble de salles reliées par des couloirs. Chaque salle a ses propriétés et ses zones identifiées pour des situations possibles (combats, trésors, ...). Le *problème des variantes de donjons* peut être formulé comme le problème de générer une variation appropriée du *donjon source* [1] en désactivant un sous-ensemble de salles et couloirs initiaux de telle sorte que l'ensemble de salles restant corresponde à un ensemble donné de contraintes (par exemple, au moins une entrée, au moins un nombre donné de monstres sur tout chemin).

Si cette approche ne répond pas encore totalement au problème global de la génération procédurale [14], elle ouvre un nouveau champ passionnant pour la PPC. De plus, elle permet aux concepteurs de niveaux de vérifier la cohérence et la qualité des variations de donjons avant de livrer le jeu. Nous spécifions formellement le *problème des variantes de donjons* dans la section 2 et le décrivons comme un problème de programmation par contraintes à l'aide de la contrainte de graphe *connected* présentée dans la section 3. Nous introduisons ensuite, dans la section 4, un nouveau propagateur étendant cette contrainte (*connected+*) qui prend en compte les graphes orientés avec cycles. Dans la partie expérimentale (Section 5), nous montrons que ce propagateur offre une amélioration spectaculaire par rapport à une approche plus naïve de *filtrage*. Avant de conclure, nous énumérons un certain nombre de nouveaux et passionnants travaux supplémentaires rendus possibles par notre approche, offrant de nouveaux défis à la fois pour la programmation par contraintes et l'industrie du jeu.

FIGURE 1 – Un donjon source.

FIGURE 2 – Quelques variations du donjon source (en vert, les salles et couloirs conservés dans la variations).

2 Définition du problème

Notre objectif est de proposer un assistant de création de niveaux de jeux pour l'aide à la génération d'ensembles de variations de niveaux. Il s'agit d'une vision restreinte de la problématique de génération procédurale de jeux, dont le but est de garantir des générations automatiques de grande qualité. Dans les jeux que nous considérons, chaque niveau doit être cohérent avec la narration, avec la progression de l'utilisateur et surtout validé par un *LD*. Livrer un jeu AAA ayant un niveau injouable peut en effet avoir des effets dramatiques en termes d'images et de coûts pour le studio de jeux responsable. Pour atteindre la meilleure qualité possible, chaque niveau est donc fabriqué à la

1. La cohérence du donjon source peut aussi être validée en générant une variation prenant en compte toutes les salles ainsi que les couloirs.

main par un designer et sert un objectif précis dans l'histoire globale. Par conséquent, notre approche n'est pas de générer des niveaux à partir de zéro mais de proposer des variations d'un donjon source, conçu par un artiste.

Nous appelons *donjon source* le *design* original (Fig. 1). Ce donjon contient des *salles* avec des propriétés (en pratique, le concept de *salles* peut être trompeur : une salle peut être composée, dans le jeu, par un ensemble de zones qui seront considérées comme un bloc entier). Une salle peut être une *entrée* (une connexion depuis l'extérieur), une *sortie* (une connexion vers l'extérieur). Les salles peuvent également avoir un ensemble d'*étiquettes* (pour gérer les situations de jeu, sa conception, ...). L'ensemble des salles que l'utilisateur pourra explorer et les situations (récompenses, combats, ...) rencontrées s'appelle le *flow*. Il est fréquent d'ajouter des salles *terminales* (*i.e.* des impasses) aux donjons pour permettre des quêtes secondaires ou des explorations facultatives. Ces dernières, qui sont des salles *sans issues* (autre que le couloir qui y mène), contiennent généralement des trésors, des clés, des objets spéciaux ou des monstres, mais ne bloquent pas le joueur. Il faut noter que les *connections* entre les pièces sont dirigées (une porte peut être un accès à sens unique, le joueur peut sauter d'une pièce à une autre, ...).

Le but du *problème des variantes de donjons* est de générer un sous-ensemble cohérent de salles à partir du donjon source satisfaisant certaines contraintes (Fig. 2). Le premier ensemble de contraintes est structurel : chaque ensemble de pièces doit être jouable (toutes les pièces sont connectées, il y a au moins une entrée et une sortie), et les pièces terminales doivent être étiquetées ainsi (pour que le *LD* vérifie leur intérêt). Les contraintes du deuxième ensemble sont des contraintes de *configuration* qui sont facultatives et peuvent être définies par l'utilisateur : certaines salles spéciales peuvent être forcées d'être *active*, *entrée*, *sortie* ou *terminale* dans la solution. Notre outil devra proposer, dans un délai raisonnable (quelques secondes), des variantes intéressantes au *LD*.

La génération procédurale de niveaux et de *flow* n'est pas nouvelle [5,12,13]. Cependant, comme indiqué dans ces références, les systèmes ne sont pas encore mûrs pour être autonomes. De plus, aucun de ces travaux n'a le potentiel d'atteindre le niveau de qualité attendu dans notre contexte. De plus, à notre connaissance, aucun travail précédent ne traite à la fois de la génération des donjons et de l'histoire elle-même (qui est une pierre angulaire de notre travail, assurée par la définition du donjon source). Comme décrit dans la section suivante, nous proposons de traiter le problème comme un problème de graphe. La propriété de connexité sera garantie par une nouvelle extension de la contrainte de graphe *connected* [4,9] adaptée à notre problème.

3 Modélisation du *problème des variantes de donjons* en PPC

Dans cette section, nous formulons le problème en problème de satisfaction de contraintes. Les contraintes sont présentées en langage naturel dans un premier temps puis sous forme clausale et ensembliste (Tableau 3.1).

3.1 Variables du modèle

Soit n le nombre de nœuds du donjon source (numéroté de 0 à $n-1$). Les connexions (e.g. *couloirs*) entre les nœuds sont identifiés par les numéros de nœuds (C_{ij} est une connexion du nœud i au nœud j). Nous avons besoin de six tableaux de taille n pour modéliser le problème : Quatre tableaux de *booléens* `entries`, `exits`, `actives` et `finals` qui indiquent, respectivement, si un nœud est une entrée, une sortie, s'il est actif et s'il est terminal. Deux tableaux `sumToNode` et `sumFromNode` qui comptent respectivement le nombre de connexions vers chaque nœud et depuis chaque nœud. Pour encoder les connexions entre les salles, nous utilisons une matrice d'adjacence C de taille $n \times n$ qui indique les arêtes actives (connexions) du graphe (les nœuds peuvent avoir plusieurs successeurs). Les tableaux `sumToNode` et `sumFromNode` sont définis à l'aide de contraintes de somme. Nous évitons également, à ce niveau, de multiples couloirs entre les mêmes salles. Une telle variation, si nécessaire, peut être gérée à un autre niveau. Pour éviter les boucles triviales, notre modèle doit également bloquer la diagonale de la matrice d'adjacence C.

Formellement, le problème peut s'exprimer sous forme de graphe de manière assez simple : soit $G(V, E)$ un graphe où $V \subset \mathbb{N}$ représente les salles et $E \subset \{(i, j) \mid i, j \in V \wedge i \neq j\}$ les couloirs. Le but du *problème des variantes de donjons* est de générer un graphe $G'(V', E')$ avec $V' \subset V$ et $E' \subset E$. Il est à remarquer que `actives` $\equiv V'$. Les ensembles `entries`, `exists` et `finals` sont des sous ensembles de V'.

3.2 Contraintes du modèle

Les contraintes du modèle assurent la cohérence de chaque variation, donnée en langage naturel dans la liste suivante. Ces contraintes sont exprimées comme contraintes d'intention dans le modèle XCSP3 [1] :

1. Si un nœud est une entrée, il doit être actif
2. Si un nœud est une sortie, il doit être actif
3. Si un nœud est terminal, il doit être actif
4. Si un nœud est une entrée, il ne peut pas être terminal

	Forme Clausale	Forme ensembliste
1	$entries_i = 0 \vee actives_i = 1$	$i \in entries \Rightarrow i \in V'$
2	$exits_i = 0 \vee actives_i = 1$	$i \in exits \Rightarrow i \in V'$
3	$finals_i = 0 \vee actives_i = 1$	$i \in finals \Rightarrow i \in V'$
4	$entries_i = 0 \vee finals_i = 0$	$entries \cap finals = \emptyset$
5	$exits_i = 0 \vee finals_i = 0$	$exits \cap finals = \emptyset$
6	$C_{ij} = 0 \vee (actives_i = 1 \wedge actives_j = 1)$	$(i,j) \in E' \Rightarrow \{i,j\} \in V'$
7	$actives_i = 0 \vee sumFromNode_i > 0$ $\vee sumToNode_i > 0$	$i \in V' \Rightarrow \exists j \mid ((i,j) \in E' \vee (j,i) \in E')$
8	$finals_i = 0 \vee sumToNode_i = 1$	$i \in finals \Leftrightarrow \exists! j \mid (\{(i,j),(j,i)\} \subset E')$
9	$finals_i = 0 \vee sumFromNode_i = 1$	Couvert par 8.
10	$finals_i = 0 \vee C_{ij} = 0 \vee C_{ji} = 1$	Couvert par 8.
11	$finals_i = 0 \vee C_{ij} = 1 \vee C_{ji} = 0$	Couvert par 8.
12	$C_{ij} = 0 \vee C_{ji} = 0 \vee sumToNode_i \neq 1$ $\vee sumFromNode_i \neq 1 \vee finals_i = 1$	$\{(i,j),(j,i)\} \subset E' \wedge \ \forall k \ \lvert\{(i,k)\} \in E'\rvert = 1$ $\wedge \ \forall k \ \lvert\{(k,i)\} \in E'\rvert = 1 \Rightarrow i \in finals$

TABLE 1 – Expression des contraintes sous forme clausale et ensembliste.

5. Si un nœud est une sortie, alors il ne peut pas être terminal
6. Si une connexion est utilisée, alors les nœuds associés doivent être actifs
7. Si un nœud est actif, au moins une connexion doit aller vers ou partir de ce nœud
8. Si un nœud est terminal, alors une et une seule connexion doit aller vers ce nœud
9. Si un nœud est terminal, alors une et une seule connexion doit partir de ce nœud
10. Si un nœud i est terminal et qu'une connexion va de i à j alors une connexion doit aller de j à i
11. Si un nœud i est terminal et qu'une connexion va de j à i alors une connexion doit aller de i à j
12. S'il y a un aller-retour entre deux nœuds i et j et il n'y a qu'une seule connexion allant et partant de i, alors le nœud i doit être terminal

L'ensemble de contraintes ci-dessus permet de spécifier la structure des variations souhaitées (le problème des solutions non connectées sera abordé dans la section suivante). On notera que la gestion des salles *terminales* (appelées *finals* dans le tableau), est couverte par les lignes 8 à 11. Il s'agit de s'assurer que le joueur pourra revenir dans le chemin principal depuis une impasse correspondant éventuellement à une quête secondaire. Certaines autres contraintes, appelées contraintes de *configuration*, sont également autorisées dans l'outil que nous proposons : bornes sur le nombre d'entrées, de sorties, de salles actives et terminales. Il faut également que notre modèle permette à l'utilisateur de forcer des pièces spécifiques à être active (ou non) dans toutes les variantes générées. L'utilisateur peut également désactiver un sous-ensemble de couloirs pour explorer les variations correspondantes. De même, il est possible de donner le contrôle sur les situations (et par conséquent sur le *flow*) en bornant le nombre de salles ayant des étiquettes données. Par exemple, si nous avons 8 salles marquées comme salle de combat dans un donjon source, un *LD* peut vouloir générer une variation avec seulement 3 d'entre elles. Pour gérer ces limites facilement, nous pouvons créer une contrainte de somme sur la propriété *actives* des salles étiquetées. Nous ne décrivons pas plus en détail cette partie du modèle, puisqu'elle est couverte par une approche classique par contraintes.

4 Évolution de la contrainte *connected*

Comme mentionné précédemment, nous devons nous assurer que nos variantes soient connectées pour répondre à toutes les contraintes. Nous avons deux choix pour cela. (1) vérifier chaque solution avec un algorithme *ad-hoc* (c'est une approche classique dans les jeux vidéos : les niveaux générés peuvent être testés par la suite par des algorithmes, des bots et/ou des humains) ou (2) forcer toute variation proposée à être correcte par construction, en filtrant les solutions partielles dès que possible lors de la recherche de variations. Il faut garder à l'esprit ici que le but de notre approche est d'aider le *LD* à naviguer dans les alternatives possibles le plus aisément possible. Arriver à garantir que chaque solution proposée aient certaines propriétés par construction est donc un élément déterminant.

La plupart des travaux sur les contraintes et les graphes (par exemple [3,6,10]) considèrent en général un nœud source et un nœud objectif, au travers du problème de chemin/circuit. On ne peut pas malheureusement pas s'appuyer directement sur ces travaux. Ils sont soit non applicables, soit trop contraints. De plus, nous devons considérer le fait que les graphes

Algorithme 1 : connected() : Boolean

```
1  seen ← {∅};
2  stack ← selectActiveNode();
3  if stack = ∅ then
4      if {node | dom(actives_node) = 2} = ∅ then
5          return false ;    // conflict detected
6      else
7          return true;
8  while stack ≠ ∅ do
9      current ← pop(stack);
10     if current ∉ seen then
11         seen ← seen ∪ current;
12         stack ← stack ∪ {x | actives_x ≠
               0 ∧ (C_current,x ≠ 0 ∨ C_x,current ≠ 0)} ;
13 return filterNodes(seen);
```

Algorithme 2 : filterNodes(*seen* : Set of nodes) : Boolean

```
1  foreach node ∉ seen do
2      if actives_node = 1 then
3          return false;    // conflict detected
4      actives_node ← 0;
5  return true;
```

Algorithme 3 : handleNode(*node* : a node, *seen* : a set, *possible* : a set, *kept* : a set) : Boolean

```
1  valid ← false;
2  seen ← seen ∪ {node};
3  possible ← possible ∪ {node};
4  if node ∈ kept ∨ exits_node ≠ 0 then
5      valid ← true;
6      kept ← kept ∪ possible;
7  foreach a |actives_node ≠ 0 ∧ C_node,a ≠ 0 do
8      if a ∈ kept then
9          valid ← true;
10         kept ← kept ∪ possible;
11     else if a ∉ seen then
12         if handleNode(a, seen, possible, kept)
               then
13             valid ← true;
14 possible ← possible \ {node};
15 return valid;
```

sont dirigés, et chaque variation peut avoir beaucoup de nœuds sources (entrées) ainsi que plusieurs nœuds objectifs (sorties) [2]. Il est également important de noter, à ce stade, que nous n'essayons pas encore d'optimiser les chemins, de quelque manière que ce soit. Nous prévoyons de le faire dans le cadre de travaux ultérieurs.

L'approche que nous proposons est un propagateur à deux niveaux qui repose d'abord sur une implémentation simple de la contrainte `connected` [4] sur la version non dirigée du graphe (Alg. 1). Ensuite, un autre niveau de filtrage est ajouté (Alg. 3 et 4) pour garantir la validité des chemins.

La contrainte `connected` garantit que nous n'avons pas de donjons disjoints, mais ne considère pas les graphes dirigés. Nous n'avons donc aucune garantie que le donjon soit jouable (*i.e.* tous les nœuds sont accessibles depuis les entrées, en particulier les nœuds de sortie). Pour surmonter ce problème, nous introduisons un algorithme pour rechercher des chemins à partir de chaque entrée [3]. Pour cette raison, nous avons besoin de connaître les valeurs des tableaux `entries` et `exits` dans le propagateur.

L'algorithme 1 part d'un nœud actif (ligne 2, la fonction selectActiveNode() sélectionne un nœud actif dans le graphe, *i.e.* un nœud x tel que $actives_x = 1$, ou retourne un ensemble vide si aucun nœud actif n'est trouvé). Ensuite, il suit chaque connexion en considérant la version non dirigée du graphe (ligne 12). De la ligne 3 à 7, si nous ne trouvons aucun nœud actif, nous devons vérifier la cohérence du graphe. En effet, si certains des nœuds ne sont pas décidés (leur taille de domaine booléen est de 2), la contrainte est cohérente puisque nous ne pouvons rien filtrer. Sinon, on détecte une incohérence si tous les nœuds sont marqués inactifs (ligne 4-5). Jusqu'à ce que la pile soit vide, nous marquons le nœud courant (ligne 11) et ajoutons ses voisins à la pile (ligne 12). Lorsque la pile est vide, l'algorithme 2 est appelé pour filtrer les nœuds et vérifier la cohérence des contraintes. En effet, pour chaque nœud que nous n'avons pas vu dans la recherche de l'algorithme 1 (donc non connecté au graphe considéré), nous devons mettre leur valeur active à 0. Un conflit est identifié lorsque un nœud déjà décidé actif est traité.

Après l'exécution de l'algorithme 2, le graphe non orienté est assuré d'être connecté. Nous devons maintenant vérifier les chemins dirigés depuis chaque nœud d'entrée et identifier les nœuds non accessibles pour les désactiver. Nous introduisons formellement les algorithmes 3 et 4 pour gérer le graphe dirigé.

2. Un nœud peut bien sûr être une entrée et une sortie en même temps. En pratique, dans les problèmes du monde réel, cela arrive souvent.

3. Cet algorithme peut être adapté pour gérer la taille des chemins, qu'ils soient contraints ou pour obtenir des informations (longueur moyenne, etc.).

Algorithme 4 : checkPaths(G : the graph) : Boolean

```
1  seen ← {∅};
2  possible ← {∅};
3  kept ← {∅};
4  nbValidEntries ← 0;
5  foreach node | entries_node ≠ 0 ∧ actives_node ≠ 0 do
6      if handleNode(node, seen, possible, kept) then
7          nbValidEntry ← nbValidEntry + 1
8      else
9          if entries_node = 1 then
10             return false;            // conflict detected
11         entries_node ← 0
12     possible ← {∅};
13 if nbValidEntry = 0 then
14     return false;          // conflict detected
15 return filterNodes(kept);
```

L'algorithme 4 vérifie qu'il existe un chemin entre chaque entrée et au moins une sortie. Pour cela, trois ensembles sont utilisés : `seen`, `possible` et `kept`. Respectivement, ces ensembles contiennent tous les nœuds vus, les nœuds dans un chemin possiblement valide (*i.e.* qui atteint une sortie) depuis l'entrée considérée, ainsi que les nœuds qui sont conservés si aucun conflit n'est détecté avant la fin de l'algorithme. Le graphe est ensuite parcouru à partir de chaque entrée potentielle grâce à l'algorithme 3 (ligne 6). Si l'entrée considérée est valide, elle est comptabilisées (ligne 7) ; sinon, elle est marquée comme invalide suite à un test de cohérence (ligne 9 à 11). Si aucune entrée valide n'est trouvée (lignes 13 et 14), un conflit est soulevé. Nous appelons ensuite l'algorithme 2 (ligne 15) pour filtrer les nœuds qui n'ont pas été conservés (et éventuellement détecter un conflit).

L'algorithme 3 (appelée à la ligne 6 de l'algorithme 4) permet de tester la validité d'un nœud fourni en paramètre (`node`). En premier lieu, le nœud à tester est ajouté aux ensembles `seen` et `possible` (lignes 2 et 3). Ensuite, si le nœud est déjà dans l'ensemble des nœuds à conserver `kept` ou que ce nœud est une sortie possible, alors il est marqué comme valide et conservé (lignes 4 à 6). Il est à noter que la recherche n'est pas arrêtée lorsqu'un noeud (ou plus particulièrement une entrée) est trouvée valide puisque nous voulons marquer tous les nœuds atteignables à partir du nœud actuellement considéré. La boucle principale (lignes 7 à 13) permet de considérer les fils du nœud considéré (`node`). En effet, d'une manière semblable à la ligne 12 de l'algorithme 1, nous ajoutons les voisins du nœud courant, sans considérer le cas non orienté cette fois-ci. Pour chacun des fils `a` deux cas sont possibles : (1) `a` est déjà dans l'ensemble des nœuds conservés (`kept`) (lignes 8 à 10). Cela implique que `node` est valide car il rejoint un chemin déjà validé précédemment. Le chemin des nœuds possibles est par conséquent conservé (ligne 10). (2) `a` n'a jamais été exploré (lignes 11 à 13). L'algorithme 3 est donc appelé récursivement pour connaître la validité de `a` (ainsi que celle de `node` si `a` est valide [4]). Enfin, avant de retourner la validité du nœud original (`node`), celui-ci est retiré de l'ensemble `possible` (ligne 14).

# variations		W/o GC	Connected+
100	Time	2,7	2,78
	# conflicts	15	173
	% Valid	0	100
1 000	Time	4,33	5,42
	# conflicts	130	1 462
	% Valid	0	1 000
10 000	Time	21,72	32,23
	# conflicts	820	11 781
	% Valid	248	10 000

TABLE 2 – Données des expériences sur l'instance réelle considérée.

Le propagateur `connected+` applique d'abord l'algorithme `connected` (Alg. 1), et si aucun conflit ne survient, alors les chemins sont vérifiés (Alg. 4).

5 Expérimentations

Nous avons implémenté les algorithmes dans le solveur *Nacre* [7]. L'équipe *XCSP3* nous a fourni une version alternative du parseur officiel pour gérer notre nouvelle contrainte globale [5]. Les expériences ont été exécutées sur un ordinateur avec un processeur 6 cœurs cadencé à 3,70 GHz (Intel i7-8700K) avec 32Go de RAM. Nous avons utilisé la méthode de recherche complète (MAC) du solveur *Nacre* sans redémarrage afin de compter les solutions trouvées [6].

Le donjon source (tiré d'un niveau de jeu réel) utilisé pour l'expérience a 52 nœuds, 58 connexions, 7 entrées possibles, 7 sorties possibles et 30 salles finales possibles. Même si ce problème semble petit, il est déjà difficile

4. Il n'est pas nécessaire de répliquer la ligne 10 après la ligne 13 si `a` et `node` sont valides, cela a déjà été fait dans l'appel à la ligne 12
5. Merci à Gilles Audemard et Christophe Lecoutre !
6. `./nacre_mini_release pattern.xml -complete -sols=X`

et il représente bien les problèmes réels (en général entre 40 et 60 nœuds, jusqu'à environ 100 nœuds pour les plus grands). L'instance *XCSP3* associée a 3019 variables et 11169 contraintes (+1 pour la contrainte de graphe). Nous évaluons les deux approches sur la génération de 100, 1000 et 10000 variations, respectivement. La première approche (*W/o GC* dans le tableau 5) n'utilise pas la nouvelle contrainte globale. La seconde approche (*Connected +* dans le tableau 5) utilise la nouvelle contrainte globale et calcule exactement le nombre de variations spécifié (puisqu'elles sont toutes valides). La recherche s'arrête lorsque le nombre spécifié de variations est trouvé (mais pas nécessairement des variantes valides pour l'approche *W/o GC*).

Dans le tableau 5, nous pouvons voir que, comme prévu, avoir le vérificateur de graphes comme propagateur intégré permet de ne produire que des variations valides pour un coût très faible. Nous avons mené d'autres expériences pour mesurer le nombre de variations que la méthode *W/o GC* devrait calculer pour obtenir 100 variations valides. Nous avons constaté que 5390 solutions devraient être calculées (en 7,63 de secondes pour 542 conflits, à comparer avec les 2,78 secondes de notre approche, pour seulement 173 conflits). Nous avons également essayé de calculer 1000 variations valides avec la méthode *W/o GC* mais, après avoir atteint le délai de 2 400 secondes, moins de 300 variations valides ont été générée pour cette instance. Cela doit être comparé à notre approche, qui permet de générer 1000 variations valides en 5,42 secondes.

Nous travaillons à l'enrichissement de la comparaison expérimentale par l'ajout de problèmes générés aléatoirement mais ayant une structure. Cela fait partie de la poursuite du travail présenté ici.

6 Conclusion & Perspectives

Nous avons présenté un nouveau problème d'importance industrielle pour la programmation par contraintes, le *problème des variantes de donjons*, et proposé une première approche pour y faire face. Notre solution est déjà utilisée en pré-production comme un outil interne d'assistance aux *Levels Designers*, supporté par le modèle *XCSP3* et le solveur *Nacre*. Il y a encore des possibilités d'améliorations (par exemple une meilleure structure de données au sein de l'algorithme de filtrage), mais nous pensons que notre approche a déjà démontré son utilité. C'est une solution pragmatique et efficace pour aider les *Levels Designers* dans leur travail quotidien pour l'industrie du jeu.

Nous prévoyons, bien entendu, d'étendre ce travail de plusieurs manières. Nous pouvons utiliser des métriques (mission linearity, map linearity, leniency and path redundancy [8, 11, 12] par exemple) pour noter chaque variation afin de générer un ensemble pertinent de variations de donjon à partir d'un modèle. Ces métriques pourraient être utilisées pour la version d'optimisation (COP) du *problème des variantes de donjons* permettant plus de contrôle sur les donjons générés (un *LD* joue souvent sur la linéarité du *flow*). En utilisant une approche similaire à [2], nous pourrions trouver des chemins où la structure du donjon est faite en utilisant certaines données (ou approximations) : le temps ou la difficulté pour terminer un combat ou un puzzle, par exemple. Nous pourrions alors construire des niveaux en fonction de la difficulté ou du temps.

Nous pourrions également enrichir le modèle avec différentes contraintes pour fournir plus de contrôle et d'automatisation pour le *problème des variantes de donjons* (par exemple, en considérant l'orientation de la salle, en permettant sa rotation). Nous pouvons également penser à des modèles de salles et de connexions (par exemple, une connexion peut être un couloir, une fenêtre, un mur cassable). Nous pouvons également étendre les contraintes de graphes pour gérer les contraintes de distance (par exemple entre les accès, des entrées aux salles de combat). Une dernière amélioration serait de considérer des réseaux de contraintes qualitatives (*QCN*) pour gérer les positions relatives des différentes salles et connexions, permettant au *LD* de spécifier les contraintes topologiques des connexions du donjon vers l'extérieur.

Un objectif à long terme de notre travail est de générer des niveaux à la volée, basés sur l'expérience utilisateur, avec de fortes garanties. Nous pensons que ce travail est le premier pas dans cette direction.

Références

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

[2] Daniel Le Berre, Pierre Marquis et Stéphanie Roussel : Planning personalised museum visits. *In* Daniel Borrajo, Subbarao Kambhampati, Angelo Oddi et Simone Fratini, éditeurs : *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*. AAAI, 2013.

[3] Diego de Uña, Graeme Gange, Peter Schachte et Peter J. Stuckey : A bounded path propagator on directed graphs. *In* Michel Rueher, éditeur : *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 de *Lecture Notes in Computer Science*, pages 189–206. Springer, 2016.

[4] Grégoire Dooms, Yves Deville et Pierre Dupont : Cp(graph) : Introducing a graph computation domain in constraint programming. *In* Peter van Beek, éditeur : *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 de *Lecture Notes in Computer Science*, pages 211–225. Springer, 2005.

[5] Joris Dormans : A handcrafted feel : Unexplored explores cyclic dungeon generation. https ://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/.

[6] Jean-Guillaume Fages : On the use of graphs within constraint-programming. *Constraints An Int. J.*, 20(4):498–499, 2015.

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

[8] R. Lavender : The zelda dungeon generator : Adopting generative grammars to create levels for action-adventure games, 2016.

[9] Patrick Prosser et Chris Unsworth : A connectivity constraint using bridges. *In* Gerhard Brewka, Silvia Coradeschi, Anna Perini et Paolo Traverso, éditeurs : *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 de *Frontiers in Artificial Intelligence and Applications*, pages 707–708. IOS Press, 2006.

[10] Luis Quesada, Peter Van Roy, Yves Deville et Raphaël Collet : Using dominators for solving constrained path problems. *In* Pascal Van Hentenryck, éditeur : *Practical Aspects of Declarative Languages, 8th International Symposium, PADL 2006, Charleston, SC, USA, January 9-10, 2006, Proceedings*, volume 3819 de *Lecture Notes in Computer Science*, pages 73–87. Springer, 2006.

[11] Gillian Smith et Jim Whitehead : Analyzing the expressive range of a level generator. *In Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, PCGames '10, New York, NY, USA, 2010. Association for Computing Machinery.

[12] Thomas Smith, Julian A. Padget et Andrew Vidler : Graph-based generation of action-adventure dungeon levels using answer set programming. *In* Steve Dahlskog, Sebastian Deterding, José M. Font, Mitu Khandaker, Carl Magnus Olsson, Sebastian Risi et Christoph Salge, éditeurs : *Proceedings of the 13th International Conference on the Foundations of Digital Games, FDG 2018, Malmö, Sweden, August 07-10, 2018*, pages 52 :1–52 :10. ACM, 2018.

[13] Valtchan Valtchanov et Joseph Alexander Brown : Evolving dungeon crawler levels with relative placement. *In Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering*, C3S2E '12, page 27–35, New York, NY, USA, 2012. Association for Computing Machinery.

[14] Breno M. F. Viana et Selan R. dos Santos : A survey of procedural dungeon generation. *In 18th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2019, Rio de Janeiro, Brazil, October 28-31, 2019*, pages 29–38. IEEE, 2019.

Actes JFPC 2021

Une approche basée sur l'ASP pour détecter des attracteurs dans les réseaux booléens circulaires

Tarek Khaled **Belaïd Benhamou**

Aix Marseille Université, Université de Toulon, CNRS, LIS, Marseille, France
{tarek.khaled,belaid.benhamou}@univ-amu.fr

Résumé

En biologie, les réseaux booléens sont traditionnellement utilisés pour représenter et simuler les réseaux de régulation de gènes. Les attracteurs font l'objet d'une attention particulière dans l'analyse de la dynamique d'un réseau Booléen. Ils correspondent à des états stables et à des cycles stables, qui jouent un rôle crucial dans les systèmes biologiques.Dans ce travail, nous étudions une nouvelle représentation de la dynamique des réseaux Booléens qui est basée sur une nouvelle sémantique utilisée dans la programmation par ensemble réponse (Answer Set Programming, ASP). Notre méthode est axé sur l'énumération de tous les attracteurs de réseaux Booléens asynchrones pour les graphes d'interaction circulaires. Nous montrons que la sémantique utilisée permet de concevoir une nouvelle approche pour calculer de manière exhaustive les cycles stables et les états stables de tels réseaux. L'énumération de tous les attracteurs et la distinction entre les deux types d'attracteurs est une étape notable pour mieux comprendre certains aspects critiques en biologie. Nous avons appliqué et évalué l'approche proposée sur des réseaux booléens générés aléatoirement. Les résultats obtenus mettent en évidence les avantages de cette approche et correspondent à certains résultats démontrés en biologie.

Abstract

In biology, Boolean networks are conventionally used to represent and simulate gene regulatory networks. The attractors are the subject of special attention in analyzing the dynamics of a Boolean network. They correspond to stable states and stable cycles, which play a crucial role in biological systems. In this work, we study a new representation of the dynamics of Boolean networks that are based on a new semantics used in answer set programming (ASP). Our work is based on the enumeration of all the attractors of asynchronous Boolean networks having interaction graphs which are circuits. We show that the used semantics allows to design a new approach for computing exhaustively both the stable cycles and the stable states of such networks. The enumeration of all the attractors and the distinction between both types of attractors is a significant step to better understand some critical aspects of biology. We applied and evaluated the proposed approach on randomly generated Boolean networks and the obtained results highlight the benefits of this approach, and match with some demonstrated results in biology.

1 Introduction

Un réseau de régulation des gènes est un ensemble de gènes qui interagissent les uns avec les autres. Chaque gène contient l'information qui détermine sa future fonction. Lorsqu'un gène est actif, un processus appelé transcription a lieu, produisant une copie de l'acide ribonucléique (ARN) de l'information génétique. Cette portion d'ARN peut alors régir la production d'une protéine. Un réseau de régulation des gènes est un système biologique spécifique qui représente la manière dont les protéines/gènes interagissent dans une cellule pour la survie, la reproduction ou la mort de la cellule. Plusieurs représentations peuvent être utilisées pour modéliser les réseaux de régulation des gènes [2]. Des représentations quantitatives peuvent être utilisées. Cependant, ces approches nécessitent des paramètres numériques qui doivent d'abord être mesurés ou calculés, et qui sont généralement difficiles à obtenir. L'autre solution consiste à utiliser des représentations qualitatives. Ce choix ne nécessite pas la connaissance des paramètres indispensables aux représentations quantitatives[25, 7, 17, 14]. Les approches qualitatives donnent généralement moins de précision sur la dynamique des systèmes de régulation que les approches quantitatives. Néanmoins, elles permettent de capturer les propriétés les plus importantes, telles que les attracteurs.

Les réseaux booléens ont été proposés comme modèle mathématique pour les réseaux génétiques [9, 10]. Les réseaux booléens offrent un outil qualitative simple et puissant

pour modéliser les réseaux génétiques [22]. Ils transforment la représentation des interactions génétiques en règles logiques qualitatives. Les réseaux booléens ont une structure constituée d'entités qui correspondent aux gènes ou aux protéines. Chaque gène/protéine prend la valeur *on* ou *off*, ce qui signifie que le gène/protéine est ou n'est pas exprimé. Deux gènes sont connectés si l'expression de l'un d'eux module l'expression de l'autre par activation ou inhibition. D'un point de vue logique, un système biologique peut être considéré comme un ensemble d'éléments en interaction.

Malgré la modélisation simplifiée et qualitative de la réalité biologique, il a été démontré que les réseaux booléens expriment et capturent correctement la dynamique des réseaux de régulation des gènes qui sont principalement caractérisés par leurs attracteurs. Les attracteurs sont les ensembles d'états vers lesquels le système converge. Un attracteur correspond généralement aux caractéristiques/phénotypes observés d'un système biologique [10]. En effet, si un réseau contrôle un phénomène de spécialisation, alors la cellule se spécialise en fonction de l'attracteur vers lequel évolue son réseau booléen sous-jacent. Autrement dit, la cellule acquiert un phénotype particulier ou une fonction physiologique spécifique pour l'attracteur vers lequel converge le réseau booléen. Il est alors essentiel d'identifier les attracteurs des réseaux booléens pour étudier leur dynamique.

Notre objectif dans ce travail est de développer une approche exhaustive pour analyser la dynamique des réseaux booléens et capturer tous les états stables possibles et énumérer tous les cycles stables. Nous nous concentrons ici sur les réseaux de gènes qui sont représentés par des graphes d'interactions circulaire. Nous considérons le mode de mise à jour asynchrone et utilisons le cadre ASP pour représenter et résoudre les problèmes mentionnés ci-dessus. L'ASP [19] est un paradigme déclaratif de résolution de problèmes, issu de la programmation logique et du raisonnement non monotone. Plusieurs solveurs ASP [18, 4, 24] sont disponibles. Ils fournissent une variété de structures et de fonctionnalités pour la modélisation de problèmes [3], aidant l'utilisateur à exprimer les problèmes plus naturellement et à les résoudre efficacement. Dans ce travail, nous utilisons la méthode introduite dans [16, 11, 15] pour traiter les réseaux de gènes. Cette méthode s'appuie sur un processus d'énumération booléen défini pour le paradigme ASP conformément à la sémantique introduite dans [1]. Cette sémantique garantit pour tout programme logique consistant, l'existence d'extensions ou de modèles expliquant le programme considéré. Certaines de ces extensions correspondent à des modèles stables et les autres à des extra-modèles. Les extra-modèles correspondent aux extra-extensions qui ne sont pas capturées par la sémantique des modèles stables [5]. Nous observerons que les extra-modèles jouent un rôle essentiel dans l'approche du codage des attracteurs des cycles stables des réseaux booléens. La représentation des graphes d'interaction comme des programmes logiques interprétés dans la sémantique introduite dans [1] donne certains résultats formels que nous utilisons pour identifier les attracteurs des réseaux. Sur la base de ces résultats théoriques [13], nous avons conçu un algorithme pour énumérer tous les attracteurs. La détection des attracteurs se fait sans passer par la simulation des réseaux booléens, contrairement à l'approche proposée dans [12].

Le reste de l'article est organisé comme suit : Nous commençons par rappeler les notions de bases sur les réseaux booléens et sur la sémantique ASP qui est utilisée dans ce papier dans la section 2. Dans la section 3, nous proposons une nouvelle approche pour la recherche et l'énumération des attracteurs. Nous étudions les relations entre le graphe de transition et la représentation logique de son graphe d'interaction dans la section 4. Nous évaluons dans la Section 5 notre approche sur des réseaux booléens générés aléatoirement. Nous concluons le travail et donnons quelques perspectives dans la section 6.

2 Préliminaires

2.1 Réseaux Booléens

Soit $V = \{v_1, ..., v_n\}$ un ensemble fini d'entités booléennes $v_i \in \{0, 1\}$ (1 pour la valeur vrai et 0 pour la valeur faux) représentant les gènes dans les réseaux de régulation. Une configuration $x = (x_1, \ldots, x_n)$ du système est l'affectation d'une valeur de vérité $x_i \in \{0, 1\}$ à chaque élément de V. L'ensemble de toutes les configurations [8], également appelé *l'espace des configurations*, est noté $X = \{0, 1\}^n$. La dynamique d'un tel système est exprimée par une fonction de transition globale f, et un mode de mise à jour qui définit comment les éléments de V sont mis à jour au cours du temps. La fonction de transition globale f est définie comme $f : X \to X$ telle que $x = (x_1, \ldots, x_n) \mapsto f(x) = (f_1(x), \ldots, f_n(x))$, où chaque fonction f_i [1] $: X \to \{0, 1\}$ est *une fonction de transition locale* qui donne l'évolution de l'état x_i du gène v_i au cours du temps. Les réseaux booléens peuvent être considérés comme des abstractions des réseaux de régulation génique où les variables booléennes x_i représentent l'état des gènes v_i. La valeur vraie pour x_i ($x_i = 1$) signifie que le gène correspondant est actif, la valeur fausse ($x_i = 0$) signifie que le gène est inactif.

2.1.1 Graphe de transitions

La dynamique d'un réseau booléen est décrite par un graphe de transition TG qui est défini par une fonction de transition f et un mode de mise à jour, formellement :

Définition 1. *Soit $X = \{0, 1\}^n$ l'espace de configuration d'un réseau booléen, $f : X \to X$ sa fonction de transition*

1. $f_i(x)$ représente le changement local qui est porté sur l'état x_i du gène v_i

globale associée et $f_i : X \to X$, $i \in \{1, \ldots, n\}$ *sont les fonctions de transition locales formant la fonction* f. *Le graphe de transition représentant la dynamique de* f *est le graphe orienté* $TG(f) = (X, T(f))$ *où l'ensemble des sommets est l'ensemble de toutes les configurations de* X *et l'ensemble des arcs est* $T(f) = \{(x, y) \in X^2; x \neq y, x = (x_1, \ldots, x_i, \ldots, x_n), y = (x_1, \ldots, f_i(x), \ldots, x_n)\}$

Le mode asynchrone est un mode de mise à jour dans lequel un seul composant de la configuration x est mis à jour à chaque instant. C'est-à-dire qu'une seule fonction de transition locale f_i est appliquée sur l'état du gène correspondant x_i à chaque instant. Les différents éléments de x pourraient être mis à jour à des intervalles de temps différents. Par conséquent, les transitions ne sont pas déterministes. Il peut y avoir plusieurs configurations successeurs possibles pour une configuration donnée qui est représenté par un nœud dans le graphe de transition.

Une orbite dans $TG(f)$ est une séquence de configurations $(x^0, x^1, x^2, ...)$ telle que soit $(x^t, x^{t+1}) \in T(f)$ soit $x^{t+1} = x^t$ lorsqu'il n'y a pas de successeurs pour x^t. Un cycle de longueur r est une séquence de configurations $(x^1, \ldots, x^r, x^1)$ avec $r \geq 2$ dont les configurations $x^1, \ldots, x^r$ sont différentes. Nous allons à présent définir la notion d'attracteur dans les systèmes dynamiques. Un état / une configuration $x = (x_1, \ldots, x_n)$ du graphe de transition $TG(f)$ est un état / une configuration stable lorsque $\forall x_i \in x, x_i = f_i(x)$, donc $x = f(x)$. Un état/une configuration stable $x = (x_1, \ldots, x_n)$ forme un attracteur trivial de $TG(f)$. Une séquence d'états / configurations $(x^1, x^2, \ldots, x^r, x^1)$ forme un cycle stable de $TG(f)$ lorsque $\forall t < r, x^{t+1}$ est le successeur unique de x^t et x^1 est le successeur unique de x^r. Un cycle stable dans $TG(f)$ forme un attracteur cyclique. Dans ce qui suit, lorsqu'il n'y a pas de confusion, nous représentons l'ensemble des gènes $V = \{v_1, v_2, \ldots, v_n\}$ par leur seule numérotation $V = \{1, 2, \ldots, n\}$.

Exemple 1. *Considérons* $V = \{1, 2, 3\}$, $X = \{0, 1\}^3$ *et les deux fonctions de transition globale suivantes* f *et* g *définies comme suit :* $f(x_1, x_2, x_3) = (x_3, \neg x_1, x_2)$ *et* $g(x_1, x_2, x_3) = (\neg x_3, \neg x_1, x_2)$.

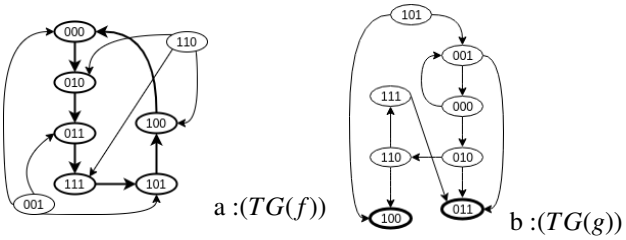


FIGURE 1 – Le graphe de transition d'un circuit booléen positif (b) et d'un circuit booléen négatif (a) de taille 3

Les deux graphes de transition correspondant à la fois à f *et à* g *sont donnés dans la figure 1. Pour chaque arc* (x, y) *des deux graphes de transition, si* $x \neq y$ *alors la configuration* x *diffère de la configuration* y *par une seule composante.* $TG(g)$ *a deux configurations stables* (100) *et* (011)) *indiquées en gras sur la Figure 1 (b). Les deux attracteurs pourraient être écrits comme* $(1, \neg 2, \neg 3)$ *et* $(\neg 1, 2, 3)$ *en prenant en compte les gènes correspondants .* $TG(f)$ *a un cycle stable* ((000), (010), (011), (111), (101), (100)) *de six configurations représentées en gras dans la figure 1 (a). Cet attracteur cyclique pourrait être vu comme* $((\neg 1, \neg 2, \neg 3), (\neg 1, 2, \neg 3), (\neg 1, 2, 3), (1, 2, 3), (1, \neg 2, 3), (1, \neg 2, \neg 3))$ *en considerant les gènes.*

2.1.2 Graphe d'interactions

Les graphes de transition sont un excellent outil pour étudier le comportement dynamique d'une fonction de mise à jour correspondant à un réseau booléen. Cependant, en pratique, les données biologiques proviennent d'expériences qui ne donnent généralement que des corrélations entre les gènes, mais rien sur la dynamique du réseau. Les corrélations entre les gènes dans un réseau de gènes sont traditionnellement représentées par un graphe d'interaction, qui est un graphe dirigé où les arcs sont étiquetés avec un signe - ou +.

Définition 2. *Un graphe d'interaction est un graphe orienté signé* $IG = (V, I)$ *où* $V = \{1, \ldots, n\}$ *est l'ensemble des sommets et* $I \subseteq V \times \{+, -\} \times V$ *est l'ensemble des arcs signés.*

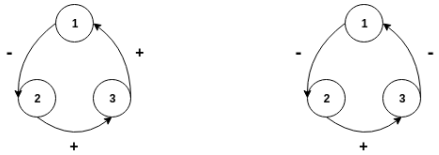
Remarque 1. *Les sommets du graphe d'interaction représentent les différents gènes du réseau de régulation génétique et les arcs expriment les interactions entre eux. Un arc étiqueté par + est dit positif, il dénote une interaction génique positive, tandis qu'un arc étiqueté par - est dit négatif et il exprime une interaction génique négative.*

Définition 3. *Un circuit du graphe d'interactions* $IG = (V, I)$ *de taille* k *est une séquence* $C = (i_1, i_2, \ldots, i_k, i_1)$ *telle que* $(i_j, \{+, -\}, i_{j+1})$ *pour tout* $j \in \{1, \ldots, k-1\}$ *et* $(i_k, \{+, -\}, i_1)$ *sont des arcs du graphe IG. Si tous les sommets de C sont distincts, alors C est dit élémentaire. Si le nombre d'arcs étiquetés par le signe "-" (arcs négatifs) est pair (resp. impair), alors le circuit C est positif (resp. négatif)*

Le graphe d'interaction est une représentation statique des interactions entre les gènes. Chaque nœud v_i du graphe d'interaction est une variable booléenne qui représente l'état du gène i dans le réseau. Plus précisément, si $v_i = 1$ (resp. $v_i = 0$)), alors le gène i est actif (resp. inactif). Un arc positif (resp. négatif) $(v_i, +, v_j)$ (resp. $(v_i, -, v_j)$) compris entre le noeud v_i et le noeud v_j signifie que le gène i est un activateur (resp. ou un inhibiteur) du gène j. Dans la suite, lorsqu'il n'y aura pas de confusion, nous simplifieront la notation en écrivant simplement i pour exprimer le nœud v_i.

Exemple 2. *Considérons le réseau booléen comportant l'ensemble des gènes* $V = \{1, 2, 3\}$, *un espace de configuration*

$X = \{0,1\}^3$ *et deux fonctions de transition globales* f *et* g *définies comme* $f(x_1,x_2,x_3) = (x_3, \neg x_1, x_2)$ *et* $g(x_1,x_2,x_3) = (\neg x_3, \neg x_1, x_2)$. *La figure 2 montre les graphes d'interaction correspondant à la fois à* f *et à* g.



a :$(IG(f))$ b :$(IG(g))$

FIGURE 2 – Les deux graphes d'interaction de circuit correspondant aux fonctions de transition globales f et g.

Nous pouvons voir que la fonction f *induit un circuit négatif de taille 3 (Figure 2 (a)) et g induit un circuit positif de taille 3 (Figure 2 (b)). Ces deux graphes d'interaction circulaires correspondent aux graphes de transition présentés dans la figure 1.*

Les graphes d'interaction sont plus compacts que les graphes de transition, donc plus lisibles. Mais, contrairement au graphe de transition, ils ne donnent que des informations statiques sur les interactions. Dans le contexte des réseaux booléens, de nombreux travaux visent à comprendre les relations formelles entre les graphes d'interaction et de transition. Un sujet de recherche majeur porte sur la construction de graphes de transition en utilisant uniquement les fonctions de transition et les graphes d'interaction correspondants. Les auteurs de [20] montrent qu'un circuit positif de taille n admet deux attracteurs dans le mode de mise à jour asynchrone, à savoir deux configurations stables x et $\neg x$ de taille n où $\neg x$ est la configuration booléenne complémentaire de x obtenue en inversant chaque état génétique dans x. D'autre part, un circuit négatif de taille n n'admet qu'un seul attracteur dans le mode de mise à jour asynchrone correspondant à un cycle stable formé par $2n$ configurations représentant sa longueur.

2.2 Answer Set Programming

2.2.1 La nouvelle sémantique pour les programmes logiques généraux

Un programme logique général π est un ensemble de règles de la forme $r : A_0 \leftarrow A_1, A_2, ..., A_m, not\ A_{m+1}, ..., not\ A_n, (0 \leq m < n)$ où $A_{i\in\{0...n\}}$ est un atome et *not* le symbole exprimant la négation par échec. Le corps positif de la règle r est $body^+(r) = \{A_1, A_2, \ldots, A_m\}$, son corps négatif est $body^-(r) = \{A_{m+1}, A_{m+2}, \ldots, A_n\}$ et A_0 est sa tête. Plusieurs sémantiques ont été introduites dans l'ASP pour donner un sens aux programmes logiques. La sémantique des modèles stables [5] est l'une des plus utilisées en ASP. Une nouvelle sémantique pour les programmes logiques généraux est proposée dans [1]. Cette sémantique capture la sémantique des modèles stables et l'étend. Elle est basée sur un langage propositionnel classique L où deux types de variables sont définis. Le sous-ensemble des variables classiques $V = \{A_i : A_i \in L\}$ et le sous-ensemble des extra-variables $nV = \{not\ A_i : not\ A_i \in L\}$. Pour chaque variable $A_i \in V$, il existe une variable correspondante $not\ A_i \in nV$ désignant une sorte de négation par échec faible. Une relation entre les deux types de variables est exprimée par l'ajout au langage L d'un axiome exprimant l'exclusion mutuelle entre elles. Cet axiome d'exclusion mutuelle est exprimé par un ensemble de clauses binaires $ME = \{(\neg A_i \vee \neg not\ A_i) : A_i \in V\}$. Un programme logique général π=$\{r : A_0 \leftarrow A_1, A_2, ..., A_m, not\ A_{m+1}, ..., not\ A_n\}$, $(0 \leq m < n)$ est exprimé dans le langage propositionnel L par un ensemble de clauses de Horn : $HC(\pi) = \{\bigcup_{r\in\pi}(A_0 \vee \neg A_1 \vee, ..., \neg A_m \vee \neg not\ A_{m+1}, ..., \neg not\ A_n)\} \cup ME = \{(\neg A_i \vee \neg not\ A_i) : A_i \in V\}$. Le strong backdoor (STB) [26] du programme logique π est formée par les littéraux de la forme $not\ A_i$ qui apparaissent dans les corps négatifs de ses règles. Formellement, elle est définie par $STB = \{not\ A_i : \exists r \in \pi, A_i \in body^-(r) \subseteq nV\}$. Étant donné un programme π et son STB, une extension de $HC(\pi)$ par rapport au STB, ou simplement une extension de la paire $(HC(\pi), STB)$ est l'ensemble de toutes les clauses consistantes déduites de $HC(\pi)$ en ajoutant un ensemble maximal de littéraux positifs $not\ A_i \in STB$ à $HC(\pi)$. Formellement :

Définition 4. *Soit* $HC(\pi)$ *le codage CNF d'un programme logique* π, *STB son strong backdoor et un sous-ensemble* $S' \subseteq STB$, *l'ensemble* $\mathrm{E} = \mathrm{HC}(\pi) \cup \mathrm{S}'$ *de clauses est alors une extension de* $(\mathrm{HC}(\pi), \mathrm{STB})$ *si les conditions suivantes sont vérifiées :*

1. E *est consistant,*
2. $\forall \mathrm{not\ A_i} \in \mathrm{STB} - \mathrm{S}', \mathrm{E} \cup \{\mathrm{not\ A_i}\}$ *est inconsistant.*

Il est montré dans [1] que chaque $HC(\pi)$ consistant admet au moins une extension par rapport au STB correspondant, formellement :

Proposition 1. *Soit* π *un programme logique et* STB *son strong backdoor. Si* $HC(\pi)$ *est consistant, alors il existe au moins une extension de la pair* $(HC(\pi), STB)$.

Il est démontré dans [1], que l'ensemble des modèles stables d'un programme logique π est en bijection avec l'ensemble des extensions E de $HC(\pi)$ qui satisfont la condition discriminante $(\forall A_i \in V, E \models \neg A_i \Rightarrow E \models A_i)$. Les deux principales propriétés théoriques démontrées sont données dans les deux théorèmes suivants :

Théorème 1. *Si* X *est un modèle stable d'un programme logique* π, *alors il existe une extension* E *de* $(HC(\pi), STB)$ *telle que* $\mathrm{X} = \{\mathrm{A_i} \in \mathrm{V} : \mathrm{E} \models \mathrm{A_i}\}$. *D'autre part,* E *vérifie la condition dite discriminante :* $(\forall A_i \in V, E \models \neg not\ A_i \Rightarrow E \models A_i)$.

Théorème 2. *Si E est une extension de $(HC(\pi), STB)$, qui vérifie la condition $(\forall L_i \in V, E \models \neg notL_i \Rightarrow E \models L_i)$ alors $X = \{L_i : E \models L_i\}$ est un modèle stable de π.*

Exemple 3. *Considérons le programme logique $\pi = \{q \leftarrow not\ r;\quad r \leftarrow not\ q;\quad p \leftarrow not\ p;\quad p \leftarrow not\ r\}$. La représentation en clause de Horn du programme logique π est formé par l'ensemble $HC(\pi) = CR \cup ME$ où $CR = \{q \vee \neg not\ r\,,\ r \vee \neg not\ q\,,\ p \vee \neg not\ p\,,\ p \vee \neg not\ r\}$, $ME = \{\neg a \vee \neg not\ a, \neg r \vee \neg not\ r, \neg p \vee \neg not\ p\}$ et son strong backdoor est donné par $STB = \{not\ r, not\ q, not\ p\}$. On voit que $(HC(\pi), STB)$ admet deux extensions $E_1 = HC(\pi) \cup \{not\ r\}$ et $E_2 = HC(\pi) \cup \{not\ p\}$. En effet, E_1 et E_2 sont maximalement consistants par rapport à l'ensemble STB. Nous pouvons déduire par résolution unitaire que $E_1 \models \{\neg r, q, p, \neg not\ q, \neg not\ p\}$ and $E_2 \models \{\neg not\ r, r, \neg q, \neg not\ p, \neg p\}$. L'extension E_1 satisfait la condition discriminante, mais E_2 ne la satisfait pas. Ainsi, le programme logique a un modèle stable $M_1 = \{p, q\}$ déduit de E_1 par résolution unitaire et l'extra-extension E_2 induit un extra-modèle $M_2 = \{r\}$ où r est vrai et p et q sont faux. Les modèles stables de π sont en bijections avec les extensions de $(HC(\pi), STB)$ satisfaisant la condition discriminante.*

2.2.2 Comment la sémantique est adoptée pour les programmes étendus

Les programmes généraux permettent de modéliser divers problèmes. Cependant, la négation classique est un concept qui est absolument essentiel lorsque des problèmes réels doivent être modélisés de manière déclarative. La sémantique d'un programme logique étendu peut être définie par sa réduction à un programme général [6]. Cette réduction supprime la négation classique, et la sémantique précédemment résumée pour les programmes généraux peut alors être utilisée pour dériver les ensembles réponses du programme logique étendu. Un programme logique étendu est un ensemble de règles de la forme : $r : L_0 \leftarrow L_1, L_2, ..., L_m, not\ L_{m+1}, ..., not\ L_n, (0 \leq m < n)$ où $L_{i\in\{0...n\}}$ sont des littéraux (Atomes A_i ou sa negation $\neg A_i$). Pour réduire un programme logique étendu π en un programme logique général équivalent π', il faut remplacer tous les littéraux négatifs $\neg L$ apparaissant dans π par un nouvel atome L' dans π', puis ajouter les règles de contrainte d'intégrité $(\leftarrow L, L')$ qui interdisent à L et L' d'être vrais dans le même modèle de π'. Cela évite à L et $\neg L$ d'être vrais dans le même modèle du programme logique étendu π. Ainsi, il nous suffit de calculer les modèles stables du programme général résultant π' à partir desquels nous pouvons déduire les ensembles réponses du programme étendu original π.

3 Représentation de graphes d'interaction à l'aide de programmes logiques

Dans ce qui suit, un graphe d'interaction IG sera traduit en un programme logique P_{IG} et représenté sous sa forme clause de Horn $HC(P_{IG})$. Les propriétés liées aux extensions de $HC(P_{IG})$ sont démontrées. Ces dernières seront ensuite utilisées pour étudier la relation entre $HC(P_{IG})$ et le graphe de transition TG. Nous montrerons comment trouver des configurations et des cycles stables du graphe de transition en utilisant simplement le programme logique P_{IG}. Dans le formalisme des réseaux Booléen, on associe à chaque entité ou gène $i \in \{1, \ldots, n\}$ une variable booléenne v_i. Pour alléger la notation, nous utiliserons i au lieu de v_i lorsque cela sera possible. Notre travail ici s'inspire du travail présenté dans [23]. Dans ce travail, les auteurs ont utilisé la logique des hypothèses [21] pour représenter le graphe d'interaction. Ce cadre non-monotone est puissant pour la représentation des connaissances, mais ne contient pas d'outils algorithmiques efficaces. Pour rectifier cette situation, nous avons choisi le framework ASP et la sémantique introduite dans [1]. Cela nous donne un bon compromis entre l'efficacité et le pouvoir d'expressivité.

Définition 5. *Étant donné le graphe d'interaction IG d'un réseau booléen représentant un réseau de régulation de gène, P_{IG} le programme logique représentant IG et un gène i. Nous définissons alors ce qui suit :*

- *i signifie que le gène i est actif dans la cellule*
- *$\neg i$ signifie que le gène i n'est pas actif.*
- *$not\ \neg i$ (resp. $\neg not\ \neg i$) signifie que la cellule donne (resp. ne donne pas) l'autorisation d'activer i.*
- *$not\ i$ (resp. $\neg not\ i$) signifie que la cellule donne (resp. ne donne pas) le droit de désactiver i.*

Définition 6. *La traduction de IG en un programme logique P_{IG} se fait en traduisant chaque arc de IG en une paire de règles. Plus précisément :*

- *Un arc positif $(i, +, j)$ est traduit en : $j \leftarrow not\ \neg i,\ \neg j \leftarrow not\ i$*
- *Un arc négatif $(i, -, j)$ est traduit en : $j \leftarrow not\ i,\ \neg j \leftarrow not\ \neg i$*

Exemple 4. *Le graphe d'interaction de l'exemple 2 est représenté par le circuit positif qui est exprimé par le programme logique étendus :*

$P(IG(g)) = \{2 \leftarrow not\ 1,\ \neg 2 \leftarrow not\ \neg 1,\ 3 \leftarrow not\ \neg 2,\ \neg 3 \leftarrow not\ 2,\ 1 \leftarrow not\ 3,\ \neg 1 \leftarrow not\ \neg 3\}$.

Le circuit négative est transcrit par le programme logique étendus suivant :

$P(IG(f)) = \{2 \leftarrow not\ 1,\ \neg 2 \leftarrow not\ \neg 1,\ 3 \leftarrow not\ \neg 2,\ \neg 3 \leftarrow not\ 2,\ 1 \leftarrow not\ \neg 3,\ \neg 1 \leftarrow not\ 3\}$.

Le programme logique étendu P_{IG} sera traduit en un programme logique général P'_{IG} qui sera exprimé par un ensemble de clauses de Horn $HC(P'_{IG})$. Une extension de la paire $(HC(P'_{IG}), STB)$ est l'ensemble de toutes les clauses cohérentes dérivées de $HC(P'_{IG})$ lors de l'ajout d'un ensemble maximal de littéraux positifs $not\ A_i \in STB$ dans $HC(P'_{IG})$. Dans ce contexte, l'ensemble STB représente une

collection de permissions pour activer i (Resp. inhiber i). Dans la suite, nous considérerons toujours la représentation clausale $HC(P'_{IG})$ au lieu du programme logique P'_{IG} que nous désignons uniquement par $HC(P_{IG})$ lorsqu'il n'y a pas de confusion. Nous dirons également simplement des extensions de $HC(P_{IG})$ pour signifier des extensions de $(HC(P'_{IG}), STB)$. En ce qui concerne l'utilisation de la nouvelle sémantique [1], le rôle d'une extension semble de collecter un maximum d'autorisations cohérentes. Notez que même si *not* $\neg i$ qui correspond à la cellule donnant la permission de produire i est présente, cette production n'est pas obligatoire. Il peut être produit ou non, selon le contexte (c'est-à-dire l'ensemble de toutes les interactions dans la cellule). De même pour *not* i qui donne l'autorisation de désactiver i. Il est généralement permis d'avoir les deux *not* $\neg i$ et *not* i mais ce n'est pas le cas dans cette représentation. D'un point de vue biologique, nous ne pouvons pas donner la permission d'activer et d'inhiber i en même temps. La proposition 2 ci-dessous confirme cet aspect biologique :

Proposition 2. *Soit $HC(P_{IG})$ un programme logique représentant le graphe d'interaction IG. Alors, Pour chaque $i \in \{1, \ldots, n\}$, $\neg(not\ \neg i \wedge not\ i)$ est vrai.*

Démonstration. Par définition, si IG contient un arc $(i, \{+, -\}, j)$. Avec la traduction de cet arc, nous obtenons deux ensembles de clauses $\{j \vee \neg not\ \neg i,\ \neg j \vee \neg not\ i\}$ ou $\{j \vee \neg not\ i,\ \neg j \vee \neg not\ \neg i\}$. Dans les deux cas, si *not* $\neg i \wedge$ *not* i est vrai. Alors, nous inférons $j \wedge \neg j$. Donc, nous avons une inconsistance. □

Proposition 3. *Soit IG un graphe d'interaction dont l'encodage logique est $HC(P_{IG})$, nous avons :*

1. *$HC(P_{IG})$ est consistant.*
2. *$HC(P_{IG})$ a au moins une seule extension.*

Démonstration.
1. Nous avons noté que $HC(P_{IG})$ est équivalent à un ensemble des clauses de Horn binaires. Chaque clause contient donc au moins un littéral négative et l'affectation de tous les littéraux à faux est un modèle de $HC(P_{IG})$. Alors, $HC(P_{IG})$ est consistant.
2. Comme $HC(P_{IG})$ est consistant, alors $HC(P_{IG})$ a au moins une extension selon la Proposition 1.

□

Les définitions et propositions suivantes sont importantes pour comprendre l'intuition qui se cache derrière la représentation introduite ici. Ils nous permettront de présenter les propriétés et les théorèmes qui établiront des liens entre les graphes d'interactions et de transitions.

Définition 7. *Soit IG un graphe d'interaction ayant l'ensemble d'entités $V = \{v_1, ..., v_n\}$ et E une extension de $HC(P_{IG})$ obtenu en ajoutant à $HC(P_{IG})$ un ensemble maximal consistant de {not i} ou {not $\neg i$}, avec $i \in \{1, \ldots, n\}$.. Nous avons :*

1. *E est complet si pour tous les $i \in V$, not $\neg i \in E$ ou not $i \in E$*
2. *i est libre dans E si $i \notin E$ et $\neg i \notin E$. Autrement, i est fixé.*
3. *Le degré de liberté de E, noté deg(E), est le nombre de sommets libres qui le compose.*
4. *Le miroir de $E = HC(P_{IG}) \cup \{not\ j |\ j \in \{1 \ldots n, \neg 1, \ldots \neg n\}\}$, noté mir(E), est défini comme $mir(E) = HC(P_{IG}) \cup \{not\ \neg j |\ j \in \{1 \ldots n, \neg 1, \ldots \neg n\}\}$.*

Proposition 4. *Soit $HC(P_{IG})$ un programme logique représentant le graphe d'interaction IG et E une extension de $HC(P_{IG})$. Le miroir de E est aussi une extension de $HC(P_{IG})$.*

Démonstration. Par définition, si IG contient un arc $(i, \{+, -\}, j)$ alors sa traduction donne deux ensembles de clauses $\{j \vee \neg not\ \neg i,\ \neg j \vee \neg not\ i\}$ ou $\{j \vee \neg not\ i,\ \neg j \vee \neg not\ \neg i\}$. Une extension est l'ensemble de toutes les clauses cohérentes dérivées de $HC(P_{IG})$ lors de l'ajout d'un ensemble maximal de littéraux positifs *not* i à $HC(P_{IG})$. Si nous inversons chaque littéral *not* i dans l'extension, c'est-à-dire que nous remplaçons *not* i (resp. *not* $\neg i$) par *not* $\neg i$ (resp. *not* i), alors nous avons deux cas :

— Le premier cas correspond à la présence d'un arc positif dans le graphe d'interaction IG. Dans ce cas, nous déduisons j lorsque *not* $\neg i$ est vrai, ou $\neg j$ si *not* i est vrai.

— Le deuxième cas correspond à la présence d'un arc négatif dans le graphe d'interaction IG. Dans ce cas, nous déduisons $\neg j$ lorsque *not* $\neg i$ est vrai et infère j lorsque *not* i est vrai.

Ainsi, il est trivial de voir que l'extension E et son miroir $mir(E)$ sont symétriques. Il en résulte que $mir(E)$ est également une extension. □

Pour certains graphes d'interaction IG particuliers tel que les circuits, nous montrons maintenant que les extensions complètes $HC(P_{IG})$ sont de degré zéro et induisent des ensembles réponses du codage logique $HC(P_{IG})$.

Proposition 5. *Soit IG un graph d'interaction, E est une extension de $HC(P_{IG})$. Si chaque nœud d'un graphe d'interaction IG a au moins un arc entrant, alors toute extension complète de $HC(P_{IG})$ est de degré 0.*

Démonstration. Soit E une extension complète de $HC(P_{IG})$. Pour prouver que E est de degré 0, nous devons prouver que chaque variable j de $HC(P_{IG})$ est liée dans E. En d'autres termes, pour chaque nœud j dans le graphe d'interaction IG, nous avons soit $\neg j \in E$ ou $j \in E$. Par hypothèse, j a un arc entrant positif / négatif $(j, +/-, i)$ dans IG, nous avons donc deux cas :

— Si l'arc est positif, il est exprimé par la paire de clauses $\{j \vee \neg not\ \neg i, \neg j \vee \neg not\ i\ \}$. Puisque E est complet, nous avons soit *not* $i \in E$ ou *not* $\neg i \in E$. Si *not* $\neg i \in E$, alors j est déduit ($j \in E$). Si *not* $i \in E$, alors $\neg j$ est déduit ($\neg j \in E$).

— Le cas d'un arc négatif est traité de la même manière. Nous aurons les clauses $\{j \vee \neg not\ i, \neg j \vee \neg not\ \neg i\ \}$. Si *not* $\neg i \in E$, alors nous déduisons $\neg j$ et si *not* $i \in E$, alors nous déduisons j.

Par conséquent, pour toutes les hypothèses, nous déduisons j ou $\neg j$. Ainsi, chaque élément j est lié dans E et le degré de E est 0. □

Proposition 6. *Soit IG un graphe d'interaction, si chaque sommet de IG a au moins un arc entrant, alors toute extension complète de $HC(P_{IG})$ correspond à un ensemble réponse de $HC(P_{IG})$.*

Démonstration. Soit E une extension complète de $HC(P_{IG})$. E correspond à un ensemble réponse si pour un nœud i donné, la condition discriminante est vérifiée pour i et $\neg i$. Les deux conditions discriminantes sont : (1) $\neg not\ i \in E \Rightarrow i \in E$ et (2) $\neg not\ \neg i \in E \Rightarrow \neg i \in E$. Puisque E est complet, alors il est de degré 0 (Proposition 5). Il en résulte que i ou $\neg i$ sont vrais dans E. Nous avons deux cas :

- Si nous avons $i \in E$. Alors, (1) est trivialement vérifié. Selon l'exclusion mutuelle $ME = \{(\neg i \vee \neg not\, i)\}$, on obtient $\neg not\ i \in E$. Dans ce cas, nous avons $\neg i \notin E$ et supposons maintenant que $\neg not\ \neg i \in E$, cela signifie que $not\ \neg i \notin E$. Comme E est complet, nous avons $not\ i \in E$, ce qui contredit le fait que $\neg not\ i \in E$. Ainsi, la condition (2) est vérifiée.
- Si nous avons $\neg i \in E$, alors la condition (2) est vérifiée trivialement. Selon l'exclusion mutuelle ME, on obtient $\neg not\ \neg i \in E$. Dans ce cas, nous avons $i \notin E$ et supposons maintenant que $\neg not\ i \in E$, donc $not\ i \notin E$. Comme E est complet, alors $not\ \neg i \in E$, ce qui contredit le fait que $\neg not\ \neg i \in E$. La condition (1) est donc vérifiée.

Puisque E vérifie la condition discriminante dans les deux cas, alors E induit un ensemble réponse de $HC(P_{IG})$. □

Maintenant, nous allons prouver que tout ensemble réponse de $HC(P_{IG})$ correspond à une extension de degré 0.

Proposition 7. *Soit IG un graphe d'interaction, si chaque sommet de IG a au moins un arc entrant, alors chaque ensemble réponse de $HC(P_{IG})$ correspond à une extension de degré 0.*

Démonstration. Soit E une extension induisant un ensemble réponse de $HC(P_{IG})$. Par définition, E est maximalement consistant par rapport aux littéraux de la forme $not\ i \in E$ ou $not\ \neg i \in E$ et vérifie la condition discriminante (a)$\neg not\ i \in E \Rightarrow i \in E$ et (b) $\neg not\ \neg i \in E \Rightarrow \neg i \in E$ correspondant à la fois à i et $\neg i$.

L'extension E induit alors un ensemble réponse de $HC(P_{IG})$. Nous devons prouver que pour tout $i \in HC(P_{IG})$, nous avons $i \in E$ ou $\neg i \in E$. Il existe trois cas :

1. Le cas où $not\ i \in E$ et $not\ \neg i \notin E$. Il résulte de la proposition 2 que $\neg not\ \neg i \in E$. Ainsi, de la condition discriminante (b), nous obtenons $\neg i \in E$.
2. Le cas où $not\ \neg i \in E$ et $not\ i \notin E$. De la proposition 2 nous obtenons $\neg not\ i \in E$. Ainsi, $i \in E$ puisque la condition discriminante (a) est vraie.
3. Le cas où $not\ i \notin E$ et $not\ \neg i \notin E$. Dans ce cas, nous avons $not\ i \wedge E \models \square$ et $not\ \neg i \wedge E \models \square$. Ainsi, $E \models \neg not\ i$ et $E \models \neg not\ \neg i$. De (a) et (b), nous avons $E \models i$ et $E \models \neg i$. Ainsi, nous obtenons une incohérence qui contredit le fait que E est une extension.

Il en résulte que seul le premier et le deuxième cas sont possibles. Ainsi, nous avons soit $i \in E$ ou $\neg i \in E$.

□

Dans ce qui suit, nous montrons que pour un graphe d'interaction IG représentant un circuit positif de n nœuds, le codage $HC(P_{IG})$ correspondant admet deux ensembles réponses de taille n.

Proposition 8. *Si le graphe d'interaction IG est un circuit positif de taille n, alors sa forme Horn clausale $HC(P_{IG})$ a deux extensions qui induisent deux ensembles réponses de taille n*

Démonstration. La preuve est basée sur les résultats de la Proposition 6 et le fait que dans un circuit positif, chaque gène agit positivement sur lui même à travers le circuit. En effet, si nous donnons au début l'autorisation d'activer le gène i (en supposant $not\ \neg i$) alors nous allons déduire que i est active, inversement si initialement nous donnons l'autorisation de désactiver i (en supposant $not\ i$) alors nous allons déduire que i est inactive ($\neg i$). On peut alors construire deux extensions complètes de degré 0. La première est faite en supposant au début le littéral $not\ \neg i$ et la seconde est son extension miroir qui est obtenue en supposant au début le littéral $not\ i$. Les deux extensions étant complètes et de degré zéro, on déduit grâce à la proposition 6 que chacune induit un ensemble réponse de taille n. □

Exemple 5. *Considérons le graphe d'interaction dans l'exemple 1 qui représente le circuit positif. Nous traduisons ce graphe en un programme logique étendu puis traduit en un programme logique général :*

$P_{IG}(g) = \{2 \leftarrow not\ 1,\ \neg 2 \leftarrow not\ \neg 1,\ 3 \leftarrow not\ \neg 2,\ \neg 3 \leftarrow not\ 2,\ 1 \leftarrow not\ 3,\ \neg 1 \leftarrow not\ \neg 3\}$

$P'_{IG}(g) = \{2 \leftarrow not\ 1,\ 2' \leftarrow not\ 1',\ 3 \leftarrow not\ 2',\ 3' \leftarrow not\ 2,\ 1 \leftarrow not\ 3,\ 1' \leftarrow not\ 3'\}$.

$HC(P'_{IG}(g))$ a deux extension qui correspondent aux modèles stables $E_1 = HC(P'_{IG}(g)) \cup \{not\ 1,\ not2',\ not3'\}$ et $E_2 = HC(P'_{IG}(g)) \cup \{not\ 1',\ not2,\ not3\}$. E_1 et E_2 sont des modèles stables parce-que pour chaque i, $\neg not\ i \in E \Rightarrow i \in E$ et $\neg not\ i' \in E \Rightarrow i' \in E$. Nous remarquons que E_1 est complet parce-que pour chaque $i \in HC(P'_{IG}(g))$, soit not i' (not $\neg i$) appartient à E_1 ou not i appartient à E_1. Nous avons soit $i \in E_1$ ou $i' = \neg i \in E_1$ pour chaque $i \in HC(P'_{IG}(g))$, ce qui signifie que le degré de liberté de E_1 est 0. L'extension E_2 est le miroir de E_1. Le modèle stable induit par E_1 et E_2 sont $M'_1 = \{1', 2, 3\}$ et $M'_2 = \{1, 2', 3'\}$. Les ensembles réponses du programme étendu $P_{IG}(g)$ correspondants sont $M_1 = \{\neg 1, 2, 3\}$ et $M_2 = \{1, \neg 2, \neg 3\}$. On peut voir que les deux ensembles de réponses précédents correspondent aux deux configurations stables du graphe de transition IG (Figure 1-(b)) du circuit positif de l'Exemple 1.

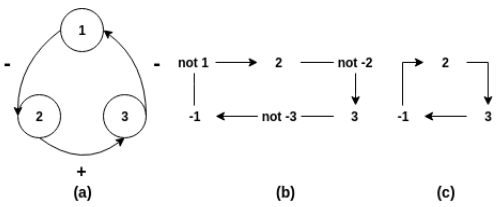


FIGURE 3 – (a) $IG(f)$, (b) Construction de E_1 (c) Le graphe de E_1 (circuit positif)

Une intuition du calcul de E_1 est donné par le processus de construction décrit par la Figure 3. Le graphe d'interaction est représenté par la figure 3-(a).Figure 3-(b) donne les étapes de construction de E_1. En premier E_1 est vide, nous ajoutons à E_1 l'hypothèse not 1. Comme nous avons $2 \leftarrow not\ 1$, nous allons avoir 2. La règle d'exclusion mutuelle ($\neg 2 \vee \neg not\ 2$) dit que c'est impossible d'avoir not 2. La construction de E_1 se poursuit par l'ajout de l'hypothèse not $\neg 2$ à E_1 et nous aurons 3. La règle d'exclusion mutuelle ($\neg 3 \vee \neg not\ 3$) interdit l'application de not 3. Alors, nous ajoutons not $\neg 3$ ce qui donne $\neg 1$. Si nous nous intéressons uniquement aux littéraux i, on obtient alors le graphe restreint de E_1 représenté par la figure 3(c) représentant le modèle stable correspondant M_1. Ce modèle correspond à l'une des deux configurations stables du graphe de transition de l'exemple 1. De la même manière que nous avons construit E_1, nous construisons l'extension E_2 en commençant par not $\neg 1$. D'un point de vue biologique, l'ensemble $\{\neg 1,\ 2,\ 3\}$ de M_1 représente l'état de chaque gène ; 2 et 3 sont active et 1 est inactive. La même chose pour M_2, l'ensemble $\{1,\ \neg 2,\ \neg 3\}$ représente l'état des gènes.

Dans ce qui suit, nous montrerons que chaque graphe d'interaction *IG* représentant un circuit négatif de n nœuds, a $2n$ extra-extensions de degré 1 induisant $2n$ extra-modèles qui encodent un cycle stable de taille $2n$ dans le graphe de transition correspondant.

Proposition 9. *Si le graphe d'interaction IG est un circuit négative de taille n alors $HC(P_{IG})$ a $2n$ extra-extensions de degré 1 qui induisent $2n$ extra-modèles de taille $n-1$.*

Démonstration. La preuve est basée sur le fait que dans un circuit négatif chaque gène agit négativement sur lui même à travers le circuit. En effet, si nous donnons au début de l'autorisation pour activer le gène i (en supposant *not* $\neg i$) alors lorsque nous allons fermer le circuit nous allons déduire que i est inactive ($\neg i$). Inversement, si initialement nous autorisons la désactivation de i (en supposant *not* i) alors nous allons déduire que i est active (i) lorsque nous allons fermer le circuit. Nous obtenons une inconsistance dans les deux cas, parce que nous allons déduire i et $\neg i$ au même moment. Cela signifie que nous ne pouvons pas avoir une extension complète dans les deux cas. Donc, nous obtenons une extension et son miroir qui sont toutes deux incomplètes, il n'y aura ni le littéral *not* j ni le littéral *not* $\neg j$, sachant que j est le prédécesseur de i. Ni i ni $\neg i$ ne sont vrais dans les deux extensions obtenues. En revanche, tous les autres éléments différents de i sont liés dans les deux extensions en question. Il s'ensuit que les deux extensions sont donc de degré 1. Il est également évident que les deux extensions ne remplissent pas la condition discriminante. En effet, dans le premier cas, nous aurons *not* $\neg i$ sans avoir i et dans le cas du miroir, nous aurons *not* $\neg i$ sans avoir $\neg i$. Au final, nous avons donc deux extra-extensions miroir de degré 1 qui induisent deux extra-modèles de taille $n-1$. Chaque fois que nous changeons l'élément de départ i, nous obtiendrons deux extensions supplémentaires miroir de degré 1 qui induiront deux modèles supplémentaires de tailles $n-1$. Au total, il y aura donc $2n$ extra-extensions de degré 1 induisant $2n$ extra-modèles de tailles $n-1$. □

Exemple 6. *Considérons le graphe d'interaction de l'exemple 1 représentant le circuit négative. Nous traduisons ce graphe en un programme logique étendu puis traduit en un programme logique général :* $P_{IG}(f) = \{2 \leftarrow not\ 1,\ \neg 2 \leftarrow not\ \neg 1,\ 3 \leftarrow not\ \neg 2,\ \neg 3 \leftarrow not\ 2,\ 1 \leftarrow not\ \neg 3,\ \neg 1 \leftarrow not\ 3\}$.

$P'_{IG}(f) = \{2 \leftarrow not\ 1,\ 2' \leftarrow not\ 1',\ 3 \leftarrow not\ 2',\ 3' \leftarrow not\ 2,\ 1 \leftarrow not\ 3',\ 1' \leftarrow not\ 3\}$. *Le programme logique $P'_{IG}(f)$ a six extensions qui correspondent aux extra-modèles :*
$E_1 = HC(P'_{IG}(f)) \cup \{not\ 1,\ not\ 2'\}$, $E_1 \models \{not\ 1, 2, not\ 2', 3\}$
$E_2 = HC(P'_{IG}(f)) \cup \{not\ 1,\ not\ 3\}$, $E_2 \models \{not\ 1, 2, not\ 3, 1'\}$
$E_3 = HC(P'_{IG}(f)) \cup \{not\ 1',\ not\ 3'\}$, $E_3 \models \{not\ 1', 2',\ not\ 3, 1\}$
$E_4 = HC(P'_{IG}(f)) \cup \{not\ 1',\ not\ 2\}$, $E_4 \models \{not\ 1', 3',\ not\ 2, 3'\}$
$E_5 = HC(P'_{IG}(f)) \cup \{not\ 2',\ not\ 3'\}$ $E_5 \models \{not\ 2', 3,\ not\ 3', 1\}$
$E_6 = HC(P'_{IG}(f)) \cup \{not\ 2,\ not\ 3\}$, $E_6 \models \{not\ 2, 3', not\ 3, 1'\}$

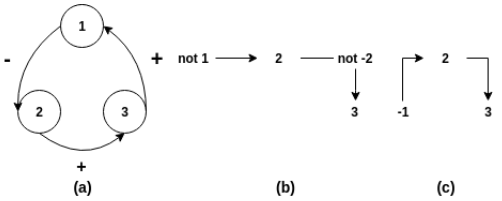


FIGURE 4 – (a) $IG(g)$, (b) E_1 construction (c) Le graphe de E_1 (circuit négatif).

Figure 4(a) montre le circuit négative exprimer par un programme logique. Figure 4(b) montre la construction de l'extension E'_1. Elle est construite en ajoutant à $HC(P'_{IG}(f))$ les littéraux not 1 et not 2'. Nous pouvons remarquer que dans la Figure 4(b), qu'il est impossible de déduire 1'. En effet, pour obtenir 1', nous devons utiliser la règle ($1' \leftarrow not\ 3$). Ce qui est impossible parce-que $\neg not\ 3$ résulte de l'exclusion mutuelle ($3' \vee \neg not\ 3$). D'autre part, nous ne pouvons pas avoir 1. Comme not 1 est vrai, alors de l'exclusion mutuelle ($1' \vee \neg not\ 1$) nous obtenons 1'. Donc, nous ne pouvons pas avoir 1. On peut remarquer que l'extension E'_1 n'est pas complète parce-que elle ne contient ni not 3 ni not 3'. L'Élément 1 est libre dans E'_1 du fait de $1 \notin E_1$ et $1' \notin E'_1$. Il en résulte que, E'_1 est une extension de degré 1. La figure 4(c) donne la construction de E_1 correspondant à l'extra-modèle M_1. Il est important de voir que la notion de degré de liberté joue ici un rôle clé. Nous remarquons que dans cette figure 4–(c) ne forme pas un circuit car il n'y a pas d'arc entre 3 et $\neg 1$. Nous remarquons que dans toutes les extensions, nous avons un seul gène manquant.

4 La relation entre les réseaux Booléens et la representation logique

Dans cette section, nous étudions la relation entre la représentation logique $HC(P_{IG})$ du graphe d'interaction *IG* et son graphe de transition *TG*. Pour ce faire, nous verrons que les sommets du graphe de transition *TG* correspondent à des configurations stables ou à des configurations appartenant à un cycle stable. Ils pourraient représenter en fait des extensions/extra-extensions (ensembles réponses ou extra-modèles) du codage logique $HC(P_{IG})$. Soit un réseau Booléen, *IG* son graphe d'interaction et *TG* le graphe de transition correspondant. Nous avons $HC(P_{IG})$ le forme Horn clausale de P_{IG} exprimant *IG*. Lorsque *IG* est un cir-

cuit positif, nous allons montrer (Théorème 3) que nous avons un isomorphisme entre les configurations stables de TG et les ensembles réponses (modèles stables) de $HC(P_{IG})$ ayant un degré de liberté 0. De plus lorsque IG est un circuit négatif de taille n , nous allons montrer (Théorème 4) que $HC(P_{IG})$ a $2n$ extra-extensions (extra-modèles) de degré 1 correspondant au cycle stable de TG.

Proposition 10. *Soit un réseau booléen représenté par un graphe d'interaction IG où TG est le graphe de transition associé et $HC(P_{IG})$ la représentation en clause de Horn du programme logique P_{IG} exprimant IG. Si s est un sommet (configuration) de TG représentant une extension/extra-extension E de $HC(P_{IG})$ de degré k, alors s a exactement k successeurs.*

Démonstration. Si i est libre dans l'extension/extra-extension E représentant la configuration s, alors soit $\neg i$ ou i est vrai dans une extension/extra-extension correspondant à une configuration s' accessible depuis s.

Par construction de TG, s' est le seul successeur de s qui vérifie cette propriété. Cette dernière est vérifié pour chaque élément libre i dans E. Donc, si le degré de liberté de E est k, alors il y aura exactement k sommets accessible depuis s. □

Théorème 3. *Soit un réseau Booléen où le graphe d'interaction IG est un circuit positif et $HC(P_{IG})$ la représentation en clause de Horn du programme logique P_{IG}. Nous avons alors :*

1. *Si $X = (x_1, x_2, \ldots, x_n)$ est un ensemble réponse de P_{IG} induit par une extension de $HC(P_{IG})$, alors $(x_1, ..., x_n)$ est une configuration stable du graphe de transition TG*
2. *Si $X = (x_1, ..., x_n)$ est une configuration stable du graphe de transition TG, alors X correspond à un ensemble réponse de P_{IG} induit par une extension de $HC(P_{IG})$.*

Démonstration.

1. Soit E une extension induisant l'ensemble réponse $X = (x_1, x_2, \ldots, x_n)$ et $s = (x_1, x_2, \ldots, x_n)$ le sommet représentant E dans TG. Comme E est une extension complète, alors son degré de liberté est 0. Selon la proposition 10, le seul sommet accessible depuis s est lui-même. Donc, s est une configuration stable de TG.
2. Si $s = (x_1, x_2, \ldots, x_n)$ est une configuration stable du graphe de transition TG, alors aucun arc ne sort de s. Le seul sommet accessible depuis s est lui-même. Il s'ensuit que pour chaque élément x_i ($resp.$ $\neg x_i$) de s, soit x_i est vrai ou $\neg x_i$ est vrai. Ensuite, tous les x_i sont liés dans l'extension E correspondant à la configuration s. Autrement dit, le degré de liberté de E est 0. Il résulte de la proposition 7 que $s = (x_1, x_2, \ldots, x_n)$ forme un ensemble réponse de P_{IG}.

□

Exemple 7. *La partie droite de la figure 5 montre les deux extensions obtenues pour le programme logique correspondant au circuit positif de l'exemple 5. Nous pouvons voir que ces extensions induisent deux ensembles réponses qui codent les deux configurations stables du graphe de transition (partie gauche de la figure 5) dessinées en gras.*

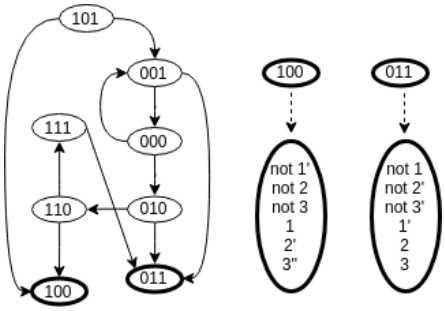


FIGURE 5 – Les configurations stables représentées comme des extensions de $HC(P_{IG})$.

Théorème 4. *Soit un réseau Booléen où le graphe d'interaction IG est un circuit négatif de taille n et P_{IG} le programme logique exprimant IG. L'ensemble $2n$ d'extra-extensions de $HC(P_{IG})$ correspond à un cycle stable du graphe de transition TG associé.*

Démonstration. La proposition 9 garantit l'existence de $2n$ extra-extension (extra-modèles) de degré 1. Nous devons considérer ici le fait que toutes les extra-extension de $2n$ sont de degré 1. Cela implique qu'il y a une seule transition de chaque extra-extension de degré 1 à une autre extra-extension de degré 1, produisant un cycle stable d'extra-extensions de $2n$. Cela correspond à un cycle stable de taille $2n$ dans TG, où chaque extra-extension correspond à une configuration dans le cycle stable de TG.

□

Exemple 8. *La partie droite de la figure 6 montre les extra-extensions obtenues pour le programme logique correspondant au circuit négatif de l'exemple 6. Nous pouvons voir que six extra-extensions de degré 1 induisent six extra-modèles et chacune d'elles identifie une configuration du cycle stable du graphe de transition correspondant (partie gauche de la figure 6, représenté ici en caractères gras).*

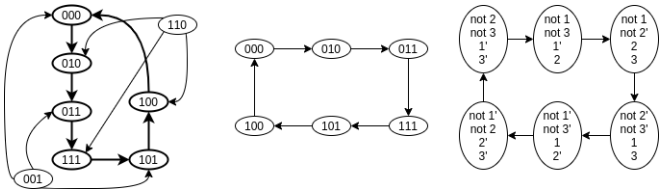


FIGURE 6 – Cycle stable représenté par des extensions de $HC(P_{IG})$.

5 Validation Empirique

Pour démontrer la validité de notre approche sur la découverte des attracteurs de réseaux Booléens, nous l'avons testée sur un vaste ensemble de réseaux générés aléatoirement.

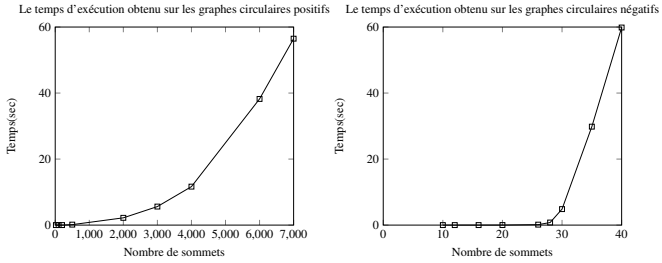


FIGURE 7 – Le temps d'exécution obtenu sur les graphes circulaires générés aléatoirement

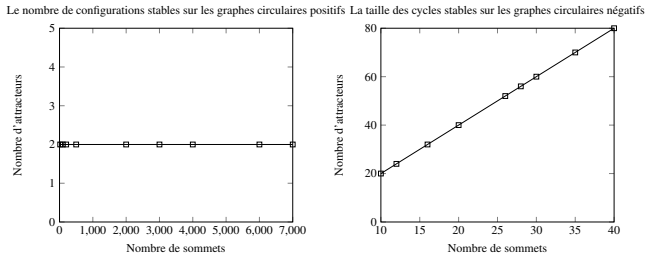


FIGURE 8 – Le nombre d'attracteurs obtenus sur les graphes circulaires générés aléatoirement

Les réseaux circulaires de taille n ont été générés en sélectionnant pour un nœud courant $i \in \{1, 2, \ldots, n\}$ de manière indépendante et uniforme exactement un successeur $j \in \{1, 2, \ldots, i-1, i+1, \ldots, n\}$. L'étiquette s de l'arc (i, s, j) est générée en choisissant au hasard un signe entre positif et négatif ($s \in \{+, -\}$). Le processus est répété pour le nœud j et tous les nœuds successeurs générés jusqu'au dernier nœud, qui doit avoir pour successeur le nœud de départ i rendant le graphe cyclique. Dans ces expérimentations, nous avons appliqué l'algorithme présenté auparavant sur des réseaux Booléens générés de manière aléatoire, jusqu'à 7 000 sommets pour un réseau circulaire positif et jusqu'à 40 pour un réseau circulaire négatif. Le temps d'exécution résultant est présenté dans la figure 7. Comme nous pouvons le constater, le temps d'exécution est très prometteur. Nous calculons les deux attracteurs d'un circuit positif à 7 000 sommets en moins de 60 secondes. Dans le même temps, nous avons calculé pour le circuit négatif des cycles stables pour des réseaux d'une taille de 40 sommets en moins de 60 secondes. Nous pouvons également voir dans la figure 8 que le nombre de configurations stables pour tous les circuits positifs générés est de 2, tandis que pour les circuits négatifs, il n'y a qu'un seul cycle stable de taille $2n$ pour chaque graphe. Ces résultats expérimentaux correspondent bien à la biologie et confirment la validité des propriétés théoriques démontrées dans cet article.

6 Conclusion

Les réseaux booléens représentent une technique de modélisation très répandue pour analyser le comportement dynamique des réseaux de régulation des gènes. En utilisant les réseaux booléens, nous pouvons capturer les attracteurs de réseaux, qui sont souvent utiles pour étudier la fonction biologique d'une cellule. Nous avons discuté dans cet article le cas particulier des circuits qui jouent un rôle essentiel dans les systèmes biologiques. Nous avons prouvé plusieurs propriétés qui établissent une correspondance entre les attracteurs des graphes de transition et les modèles / extra-modèles stables des programmes logiques exprimant les graphes d'interaction circulaire. En particulier, la représentation des cycles stables des circuits négatifs par un ensemble d'extra-modèles montre l'avantage de l'extension offerte par la sémantique utilisée à celle des modèles stables [5]. En plus des résultats théoriques, nous avons proposé une approche qui calcule efficacement tous les ensembles réponses / extra-modèles représentant les configurations stables ou les cycles stables du graphe de transition du réseau booléen considéré. Le fait que la représentation logique d'un circuit positif possède deux modèles miroirs stables et que celle d'un circuit négatif possède un ensemble de $2n$ extra-modèles témoigne bien de la validité de la méthode puisque cela correspond aux résultats connus en biologie [20]. A titre de développement futur, nous souhaitons prendre en compte d'autres modes de mise à jour comme le mode parallèle, puis généraliser l'étude aux blocs séquentiels, qui sont des mises à jour périodiques déterministes. D'autre part, nous nous intéressons à la caractérisation des cycles non stables dans les réseaux booléens.

Références

[1] Belaïd Benhamou et Pierre Siegel : A new semantics for logic programs capturing and extending the stable model semantics. *Tools with Artificial Intelligence (ICTAI)*, pages 25–32, 2012.

[2] Hidde De Jong : Modeling and simulation of genetic regulatory systems : a literature review. *Journal of computational biology*, 9(1):67–103, 2002.

[3] Esra Erdem, Michael Gelfond et Nicola Leone : Applications of answer set programming. *AI Magazine*, 37:53–68, 2016.

[4] Martin Gebser, Benjamin Kaufmann, Andrá Neumann et Torsten Schaub : Conflict-driven answer set solving. *IJCAI*, 7:386–392, 2007.

[5] Michael Gelfond et Vladimir Lifschitz : The stable model semantics for logic programming. *ICLP/SLP*, 50:1070–1080, 1988.

[6] Michael Gelfond et Vladimir Lifschitz : Classical negation in logic programs and disjunctive databases. *New generation computing*, 9:365–385, 1991.

[7] Katsumi Inoue : Logic programming for boolean networks. *In Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[8] François Jacob et Jacques Monod : Genetic regulatory mechanisms in the synthesis of proteins. *Journal of molecular biology*, 3(3):318–356, 1961.

[9] Stuart A Kauffman : Metabolic stability and epigenesis in randomly constructed genetic nets. *Journal of theoretical biology*, 22(3):437–467, 1969.

[10] Stuart A Kauffman : *The origins of order : Self-organization and selection in evolution*. OUP USA, 1993.

[11] Tarek Khaled et Belaid Benhamou : Symmetry breaking in a new stable model search method. *22nd International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR-22), Kalpa Publications in Computing*, 9:58–74, 2018.

[12] Tarek Khaled et Belaid Benhamou : An asp-based approach for attractor enumeration in synchronous and asynchronous boolean networks. *Proceedings 35th International Conference on Logic Programming, ICLP 2019, Las Cruces, NM, USA*, pages 295–301, 2019.

[13] Tarek Khaled et Belaid Benhamou : An asp-based approach for boolean networks representation and attractor detection. *In LPAR*, pages 317–333, 2020.

[14] Tarek Khaled et Belaid Benhamou : Dealing with biology systems in the framework of answer set programming. *Procedia Computer Science*, 176:450–459, 2020.

[15] Tarek Khaled, Belaïd Benhamou et Pierre Siegel : Une nouvelle méthode pour la recherche de modèles stables en programmation logique. *JFPC 2018*, page 63.

[16] Tarek Khaled, Belaïd Benhamou et Pierre Siegel : A new method for computing stable models in logic programming. *Tools with Artificial Intelligence (ICTAI)*, pages 800–807, 2018.

[17] Hannes Klarner, Alexander Bockmayr et Heike Siebert : Computing maximal and minimal trap spaces of boolean networks. *Natural Computing*, 14(4):535–544, 2015.

[18] Fangzhen Lin et Yuting Zhao : Assat : Computing answer sets of a logic program by sat solvers. *Artificial Intelligence*, pages 115–137, 2004.

[19] Victor W. Marek et Miros L. Truszczynski : Stable models and an alternative logic programming paradigm. *The Logic Programming Paradigm*, pages 375–398, 1999.

[20] Elisabeth Remy, Brigitte Mossé, Claudine Chaouiya et Denis Thieffry : A description of dynamical graphs associated to elementary regulatory circuits. *Bioinformatics*, 19(suppl_2):ii172–ii178, 2003.

[21] Camilla Schwind et Pierre Siegel : A modal logic for hypothesis theory. *Fundamenta Informaticae*, 21:89–101, 1994.

[22] Ilya Shmulevich, Edward R Dougherty et Wei Zhang : From boolean to probabilistic boolean networks as models of genetic regulatory networks. *Proceedings of the IEEE*, 90(11):1778–1792, 2002.

[23] Pierre Siegel, Andrei Doncescu, Vincent Risch et Sylvain Sené : Towards a boolean dynamical system representation in a monmonotonic modal logic. 2018.

[24] Patrik Simons, Ilkka Niemelä et Timo Soininen : Extending and implementing the stable model semantic. *Artificial Intelligence*, 138:181–234, 2002.

[25] Nam Tran et Chitta Baral : Hypothesizing about signaling networks. *Journal of Applied Logic*, 7:253–274, 2009.

[26] Ryan Williams, Carla P Gomes et Bart Selman : Backdoors to typical case complexity. *International joint conference on artificial intelligence*, 18:1173–1178, 2003.

Raffiner l'heuristique CHS à l'aide de bandits*†

Mohamed Sami Cherif **Djamal Habet** **Cyril Terrioux**

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{mohamed-sami.cherif, djamal.habet, cyril.terrioux}@univ-amu.fr

Résumé

Récemment, une heuristique efficace, appelée Conflict History Search (CHS), a été introduite pour la résolution du problème de satisfaction de contraintes (CSP). Elle repose sur une technique d'apprentissage par renforcement appelée *Exponential Recency Weighted Average* (ERWA) pour estimer la dureté des contraintes. CHS favorise les variables qui apparaissent souvent dans les échecs récents. Le paramètre de pas utilisé dans CHS est important car il contrôle l'estimation de la dureté des contraintes. Dans cet article, nous envisageons un raffinement de ce paramètre à l'aide d'un bandit manchot. Le bandit sélectionne une valeur appropriée de ce paramètre lors des redémarrages effectués par l'algorithme de recherche. Chaque bras correspond à l'heuristique CHS avec une valeur donnée pour le paramètre de pas et est récompensé selon sa capacité à mener une recherche efficace. Une phase d'entraînement est introduite en amont de la recherche pour aider le bandit à choisir un bras pertinent. L'évaluation expérimentale montre que cette approche conduit à des améliorations significatives.

1 Contexte

Une instance CSP est définie par la donnée d'un triplet (X, D, C), où $X = \{x_1, \ldots, x_n\}$ est un ensemble de n variables, $D = \{d_{x_1}, \ldots, d_{x_n}\}$ est un ensemble de domaines finis de taille au plus d, et $C = \{c_1, \ldots, c_e\}$ est un ensemble de e contraintes. Chaque contrainte c_i est un couple $(S(c_i), R(c_i))$, où $S(c_i) = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$ définit la *portée* de c_i, et $R(c_i) \subseteq d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$ est une *relation de compatibilité*. Une affectation d'un sous-ensemble de X est dite *localement cohérente* si toutes les contraintes couvertes par ce sous-ensemble sont satisfaites. Une *solution* est une affectation localement cohérente de toutes les variables. Déterminer si une instance CSP possède une solution est NP-complet.

Dans ce contexte, l'heuristique de choix de variables permet de désigner la prochaine variable à instancier. Elle joue un rôle important dans la résolution des instances CSP. Son influence sur l'efficacité de la recherche est souvent considérable. Dans la littérature, de nombreuses heuristiques ont été proposées. Les heuristiques dynamiques et adaptatives (par exemple [2, 6, 8]), conduisent généralement aux meilleurs résultats.

Dans ce registre, l'heuristique CHS [4] maintient pour chaque contrainte c_i un score $q(c_i)$ (initialement nul). Lorsque c_i conduit à un échec, $q(c_i)$ est mis à jour avec la formule $q(c_i) = (1-\alpha) \times q(c_i) + \alpha \times r(c_i)$, dérivée d'ERWA. Le paramètre $0 < \alpha < 1$ (dit *paramètre de pas*) définit l'importance donnée à l'ancienne valeur de $q(c_i)$ par rapport à la valeur $r(c_i)$ de la récompense. Sa valeur, initialisée à une valeur α_0 donnée, décroit au cours du temps par pas de 10^{-6} jusqu'à un seuil fixé à $0,06$. La valeur de $r(c_i)$ dépend des échecs rencontrés récemment au niveau de c_i et est définie par $r(c_i) = \frac{1}{\#Conflicts - Conflict(c_i)+1}$. Initialisé à 0, $\#Conflicts$ est le nombre d'échecs rencontrés depuis le début de la recherche tandis que $Conflict(c_i)$ (initialisé à 0) mémorise, pour chaque contrainte c_i, la valeur qu'avait $\#Conflicts$ lors du dernier échec rencontré grâce à c_i. À chaque conflit, $\#Conflicts$ est incrémenté de 1. CHS choisit, comme prochaine variable, celle qui a le plus grand score chv avec $chv(x_j) = \frac{\sum_{c_i \in C \mid x_j \in S(c_i) \wedge |Uvars(c_i)|>1} (q(c_i)+\delta)}{|D_j|}$ où $Uvars(c_i)$ désigne l'ensemble des variables non instanciées de $S(c_i)$. Le paramètre δ permet de débuter la recherche avec des scores initiaux reflétant le nombre de contraintes dans lesquelles apparaît chaque variable. CHS privilégie donc les variables ayant un petit domaine et intervenant dans des contraintes qui sont à l'origine d'échecs récents et récurrents. Enfin, lors de chaque redémarrage, la valeur de α est réinitialisée à α_0 et les valeurs des $q(c_i)$ sont lissées suivant la formule $q(c_i) = q(c_i) \times 0,995^{\#Conflicts - Conflict(c_i)}$.

*Ce papier est un résumé de [3].

†Ce travail est soutenu par l'Agence Nationale de la Recherche dans le cadre du projet DEMOGRAPH (ANR-16-CE40-0028).

2 Raffiner CHS via un bandit manchot

Les résultats expérimentaux de CHS montrent que certaines instances ne sont résolues que pour certaines valeurs de α_0. Afin de gagner en efficacité, nous exploitons un bandit manchot dont chacun des K bras correspond à l'heuristique CHS pour une valeur donnée de α_0. À chaque redémarrage de l'algorithme de résolution, le bandit choisit un de ses bras et donc désigne la variante de CHS à utiliser jusqu'au prochain redémarrage. Ce choix est fait via la politique Upper Confidence Bound (UCB1) [1]. Il repose sur les récompenses attribuées à la fin de chaque relance pour estimer la performance de l'algorithme de résolution par rapport à l'heuristique de recherche employée.

Soit nc_t le nombre de conflits rencontrés durant la relance t. On note p_j le ratio de variables non instanciées lors du j-ème conflit. La fonction de récompense est calculée, à la fin de la relance, selon la formule $R_t(i) = \frac{\sum_{j=1}^{nc_t} p_j}{nc_t}$. Ainsi, cette fonction de récompense consiste en une moyenne des ratios de variables non instanciées rencontrées lors des conflits, évaluant la capacité de l'heuristique à identifier rapidement les instanciations incohérentes. En d'autres termes, moins il y a de variables instanciées, plus la valeur de la récompense sera élevée pour l'heuristique correspondante.

Le nombre de redémarrage semble souvent insuffisant pour évaluer pleinement le potentiel de chaque heuristique candidate, ce qui rend difficile pour le bandit de converger rapidement vers les bras les plus adéquats. Pour cette raison, une phase d'entraînement a été ajoutée. Elle consiste à choisir successivement chaque bras un par un, comme le ferait un algorithme de type tourniquet. Afin de ne pas favoriser un bras au détriment d'un autre, la durée entre deux redémarrages est constante dans cette phase et le nombre de redémarrage est un multiple de K. Cette phase d'entraînement peut être considérée comme une étape d'initialisation des paramètres utilisés par le bandit, notamment des valeurs des récompenses et de leurs moyennes. Une fois l'entraînement terminé, UCB1 prend le relais pour sélectionner un bras parmi les heuristiques candidates. Par ailleurs, durant la phase d'entraînement, les poids des CHS sont mis à jour à chaque exécution lorsqu'un conflit survient ou lors d'un redémarrage par lissage. Enfin, la phase d'entraînement seule peut suffire à résoudre certaines instances.

3 Résultats expérimentaux

Nous comparons, sur 12 901 instances du dépôt XCSP3 [1], notre approche avec entraînement (MAB-CHS+Train) et sans (MAB-CHS) à des heuristiques

1. http ://www.xcsp.org/series

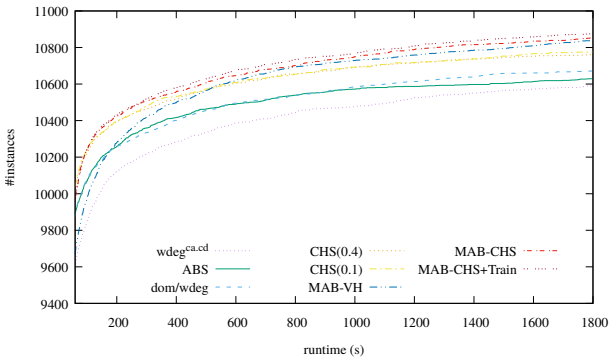


FIGURE 1 – Nombre d'instances résolues en fonction du temps écoulé (à partir de 60 s) pour chaque heuristique.

de l'état de l'art (à savoir *dom/wdeg* [2], ABS [6], wdeg$^{ca.cd}$ [8] et CHS) ainsi qu'à une approche basée sur un bandit multi-heuristiques (MAB-VH [7]). La résolution s'effectue via l'algorithme MAC avec redémarrages et enregistrement de nogoods [5] avec un temps limite de 30 minutes. MAB-CHS(+Train) exploite un bandit à 9 bras. Chaque bras correspond à CHS avec α_0 qui varie de 0,1 à 0,9 par pas de 0,1.

D'abord, d'après la figure 1, l'heuristique MAB-CHS+Train s'avère meilleure que MAB-CHS avec 10 875 instances résolues contre 10 853. Ensuite, comparée aux heuristiques de l'état de l'art, MAB-CHS+Train affiche un gain notable compris entre 95 et 279 instances. Enfin, MAB-CHS+Train se révèle plus efficace que MAB-VH en nombre d'instances résolues (38 instances de plus) et en temps (1,038 h contre 1,068 h).

Références

[1] Peter AUER, Nicolò CESA-BIANCHI et Paul FISCHER : Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[2] F. BOUSSEMART, F. HEMERY, C. LECOUTRE et L. SAÏS : Boosting Systematic Search by Weighting Constraints. *In ECAI*, pages 146–150, 2004.

[3] M. S. CHERIF, D. HABET et C. TERRIOUX : On the Refinement of Conflict History Search Through Multi-Armed Bandit. *In ICTAI*, pages 264–271, 2020.

[4] D. HABET et C. TERRIOUX : Conflict History based Search for Constraint Satisfaction Problem. *In SAC*, pages 1117–1122, 2019.

[5] C. LECOUTRE, L. SAIS, S. TABARY et V. VIDAL : Recording and Minimizing Nogoods from Restarts. *JSAT*, 1(3-4):147–167, 2007.

[6] L. MICHEL et P. Van HENTENRYCK : Activity-based search for black-box constraint programming solvers. *In CPAIOR*, pages 228–243, 2012.

[7] H. WATTEZ, F. KORICHE, C. LECOUTRE, A. PAPARRIZOU et S. TABARY : Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts. *In ECAI*, 2020.

[8] H. WATTEZ, C. LECOUTRE, A. PAPARRIZOU et S. TABARY : Refining Constraint Weighting. *In ICTAI*, pages 71–77, 2019.

On the Refinement of Conflict History Search Through Multi-Armed Bandit

Mohamed Sami Cherif Djamal Habet Cyril Terrioux
Aix-Marseille Univ, Université de Toulon, CNRS, LIS, France
{mohamedsami.cherif, djamal.habet, cyril.terrioux}@lis-lab.fr

***Abstract*—Reinforcement learning has shown its relevance in designing search heuristics for backtracking algorithms dedicated to solving decision problems under constraints. Recently, an efficient heuristic, called Conflict History Search (CHS), based on the history of search failures was introduced for the Constraint Satisfaction Problem (CSP). The Exponential Recency Weighted Average (ERWA) is used to estimate the hardness of constraints and CHS favors the variables that often appear in recent failures. The step parameter is important in CHS since it controls the estimation of the hardness of constraints and its refinement may lead to notable improvements. The current research aims to achieve this objective. Indeed, a Multi-Armed Bandits (MAB) framework can select an appropriate value of this parameter during the restarts performed by the search algorithm. Each arm represents a CHS with a given value for the step parameter and it is rewarded by its ability to improve the search. A training phase is introduced earlier in the search to help MAB choose a relevant arm. The experimental evaluation shows that this approach leads to significant improvements regarding CHS and other state-of-the-art heuristics.**

***Index Terms*—Constraint programming, Conflict History Search, Multi-Armed Bandits**

I. Introduction

The Constraint Satisfaction Problem (CSP) is used successfully in modeling and solving a large variety of academic and real-world problems [1]. In a CSP instance, the variables are defined by their domain values and the constraints specify the relations between variables. The corresponding problem consists in finding an assignment of all the variables that satisfies all the constraints. The solvers dedicated to this NP-complete problem are often based on backtracking algorithms with powerful embedded techniques such as filtering, learning, restarts and search heuristics [1]. A search or branching heuristic determines the manner of visiting the search space by deciding the next variable to assign or to fix. Such a heuristic has a significant impact on the size of the search tree developed by a backtracking algorithm, and consequently on its running time. Although finding the best branching variable while minimizing the search-tree size is NP-hard [2], many search heuristics have been proposed (e.g. [3], [4], [5], [6], [7], [8], [9], [10], [11]) and are intended to dynamically exploit solving related information such as filtering efficiency or constraint hardness.

This work has been published in the proceedings of the 32nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2020, Baltimore, MD, USA, November 9-11, 2020. ISBN 978-1-7281-9228-4 DOI 10.1109/ICTAI50040.2020.00050

Recent research has shown the interest of machine learning in designing efficient search heuristics for CSP [12], [13] as well as for other decision problems [14]. One of the motivations is the difficulty of defining a heuristic which can have high performance on any considered instance. Indeed, a heuristic can perform very well on a family of instances while falling drastically on another [13]. In this context, reinforcement learning under the Multi-Armed Bandit (MAB) framework is employed to tend towards a relevant heuristic among a set of candidates which constitute the arms of the MAB. An arm/heuristic is selected accordingly at each node of the search tree [13] or at each restart of the backtracking algorithm [12].

Another motivation for considering reinforcement learning is to estimate the hardness of constraints while relying on the history of the search steps and, in particular, those leading to dead-ends. This estimation is then used to guide the search on variables appearing in hard constraints. Conflict History Search (CHS) [15] is designed with this in mind. The estimation of the hardness is achieved by the Exponential Recency Weighted Average (ERWA) [16]. ERWA calculates the constraint hardness value by considering its current value and a reward value which becomes higher when the constraint leads to dead-ends over short periods. An important parameter is involved in keeping a balance between these two values, namely the step-size value or step parameter. Although it is identified as a hard task, a relevant initialization of the step parameter may lead to significant improvements [15].

Hence, the main contribution of this paper is refining CHS by dealing with the problem of efficiently setting the step-size parameter in CHS. Indeed, a MAB framework is proposed on the basis of CHS while taking advantage of the restart mechanism in backtracking algorithms. Recall that restarts are widely used to prevent the heavy-tailed phenomena [17]. Each run has a duration expressed in terms of the number of decisions, conflicts or backtracks and a restart policy is chosen to define the effective duration of the runs, such as geometric [18] or Luby [19] sequences. The candidate heuristics or arms of the MAB are CHS instances which differ by the initial value of the step parameter taken from a range of given values.

Important components of the MAB framework are the reward assigned to each arm and the policy that selects the next arm to use after a restart. Even if any policy can be employed in this framework, UCB (Upper Confidence Bound) is used as it has shown its relevance in other work

[20], [12], [13]. Furthermore, any policy is supported by the reward function which should be able to estimate the positive impact of a heuristic on the efficiency of the search. Accordingly, a reward function is proposed to favor the arm that reaches dead-ends quickly. A training phase is also proposed consisting in a series of runs with an identical duration conducted according to a round-robin policy on the arms. The aim of this initialization phase is to speed-up the convergence of the MAB framework. The proposed MAB framework based on CHS, denoted by MAB-CHS, is evaluated on XCSP3 benchmarks[1]http://www.xcsp.org. The empirical results confirm the relevance of the proposed approach as MAB-CHS not only improves the performance of CHS but also shows improvements regarding similar approaches and other powerful heuristics.

The paper is organized as follows. Definitions and notations are given in Section II and related work is described in Section III. The proposed MAB-CHS framework is detailed in Section IV and experimentally evaluated in Section V. We conclude and discuss perspective work in Section VI.

II. Background

An instance of a Constraint Satisfaction Problem (CSP) consists of a triplet (X, D, C) such that $X = \{x_1, \cdots, x_n\}$ is a set of n variables, $D = \{D_1, ..., D_n\}$ is a set of finite domains, and $C = \{c_1, \cdots, c_e\}$ is a set of e constraints. The domain of each variable x_i is D_i. Each constraint c_j is defined by its scope $S(c_j) = \{x_{j_1}, \cdots, x_{j_k}\} \subseteq X$ and its compatibility relation $R(c_j) \subseteq D_{j_1} \times \cdots \times D_{j_k}$. The constraint satisfaction problem consists in finding an assignment of the variables $x_i \in X$, within their respective domains D_i $(1 \leq i \leq n)$, which satisfies each constraint in C. Such an assignment is said to be consistent and is a solution of the given instance. Checking whether a CSP instance has a solution is NP-complete [1].

CSP solving is mainly based on a backtracking algorithm while maintaining a given level of consistency (e.g. the Generalized Arc Consistency property [21]). A usual algorithm is Maintaining Arc Consistency (MAC) [22]. A search tree is constructed where each node corresponds to a decision ($x_i = v$ or $x_i \neq v$) with respect to a selected variable x_i which is not yet fixed and a domain value v. The choice of the variable x_i is taken according to a search heuristic (i.e. variable branching heuristic). MAC can embed sophisticated techniques, such as nogood recording and efficient handling of global constraints [1], leading to high performance. Furthermore, restarts are used to address the heavy-tail phenomena related to irrelevant decisions during the search [17]. The search algorithm performs a series of runs. Each run has a duration expressed in terms of the authorized number of decisions, conflicts, backtracks, etc. The duration of each run is determined following several policies, such as geometric [18] or Luby [19] sequences.

In a Multi-Armed Bandit framework (MAB), a bandit, i.e. an agent, has to choose an arm from a set of candidates by relying on information collected through a sequence of trials. The information available to the bandit are rewards attributed to each arm. One of the earliest MAB models, stochastic MAB, was introduced in [23]. As a decision process, a MAB faces an important dilemma which is the trade-off between exploitation and exploration, i.e. the bandit needs to explore underused arms often enough to have a robust feedback while also exploiting good candidates which have the best rewards. To this end, many policies have been devised such as the ϵ-Greedy strategy [16], which does random exploration, Thompson Sampling [24] and the Upper Confidence Bound (UCB) family [25], [26], [27], which conduct smart exploration adequate for uncertain environments, among others.

[1]http://www.xcsp.org

III. Related Work

The state-of-the-art in terms of search heuristics is very rich. Indeed, many heuristics have been proposed aiming mainly to satisfy the *first-fail principle* [28] which advises "to succeed, try first where you are likely to fail". In this context, efficient heuristics are adaptive and dynamic, by basing the branching decisions on information gathered throughout the solving process, such as the effectiveness of filtering as in ABS [9] and IBS [10] or the hardness of constraints as employed in *dom/wdeg* [5] and its variants [8], [12], [11]. In addition, heuristics such as LC [29] and COS [30] attempt to consider the search history and require the use of auxiliary heuristics.

The Conflict History-Based (CHB) branching heuristic [31], based on the Exponential Recency Weighted Average (ERWA) [16], was introduced for the satisfiability problem. CHB rewards the activity of variables favoring the ones that are involved in recent conflicts. In particular, the reward associated to each variable is updated whenever it is branched on or propagated and its score is calculated after each restart. The Learning Rate Branching (LRB) heuristic [14] extends CHB by exploiting locality and introducing the learning rate of variables. Recently, CHB was also implemented in the context of CSP [32]. The *Conflict-History Search* (CHS) heuristic [15] is also inspired from CHB and similarly considers the history of constraint failures, favoring the variables that are involved in recent failures. Conflicts are dated and the constraints are weighted on the basis of ERWA. Weights are coupled with the variable domains to calculate the Conflict-History scores of the variables. The present work revolves around the CHS heuristic, which we will address more in detail in the next section, and how to refine it using reinforcement learning techniques.

Reinforcement learning techniques have already been studied in constraint programming. In particular, the Multi-Armed Bandit (MAB) framework was used to select adaptively the consistency level of propagation at each node of the search tree [20]. MAB was also used to select a restart strategy in [33]. Moreover, a linear regression method was applied to learn the scoring function of value heuristics [34]. Rewards, calculated for each node of the search tree, were also used to select adaptively a backtracking strategy in [35]. Furthermore, a learning process based on the Least Squares Policy Iteration technique was used to tune adaptively the parameters of stochastic local search algorithms [36]. Upper Confidence Bound (UCB) and Thompson Sampling techniques were employed to select

automatically, at each node of the search tree, a branching heuristic for CSP, among a set of candidate ones [13]. A recent work proposes a MAB framework to select at each restart in a backtracking algorithm a search heuristic among a set of candidate ones, such as *ABS*, *IBS* or CHS [12]. In particular, a reward function was designed to favor, among the set of candidate heuristics, those able to prune large parts of the search tree and the UCB policy was identified as the most powerful in this framework. In our work we use an algorithm in the UCB family, namely UCB1 [26], to refine the CHS heuristic as we will explain in the next section.

IV. MAB Framework for the Refinement of CHS

In this section, we explain how MAB can be used to address the refinement of CHS by selecting a relevant value for the step parameter. In particular, we introduce our reward function which is based on search failures and we explain the UCB policy used in the MAB framework. Finally, we present and describe a training phase which can help initialize the MAB parameters in order for it to converge quickly.

A. An overview of CHS

The Conflict History Search (CHS) heuristic considers the history of constraint failures and favors the variables that often appear in recent ones [15]. The conflicts are dated and the constraints are weighted on the basis of the Exponential Recency Weighted Average (ERWA) [16]. These weights are coupled with the variable domains to calculate the Conflict-History scores of the variables. More precisely, CHS maintains for each constraint c_j a score $q(c_j)$ which is initialized to 0. If c_j leads to a failure during the search because the domain of a variable in $S(c_j)$ is emptied by propagation then $q(c_j)$ is updated by the formula below derived from ERWA.

$$q(c_j) = (1 - \alpha) \times q(c_j) + \alpha \times r(c_j)$$

The parameter $0 < \alpha < 1$ is the step-size, also referred to as the step parameter, and $r(c_j)$ is the reward value for constraint c_j. The parameter α fixes the importance given to the old value of the score q at the expense of the reward r and its value decreases over time. Starting from an initial value α_0, α decreases by 10^{-6} at each constraint failure to a minimum of 0.06. Higher rewards are given to constraints that fail regularly over short periods according to the following formula:

$$r(c_j) = \frac{1}{Conflicts - Conflict(c_j) + 1}$$

$Conflicts$ is the number of conflicts that occurred since the beginning of the search. $Conflict(c_j)$ is updated to the current value of $Conflicts$ when c_j leads to a failure. The Conflict-History score of a variable $x_i \in X$ is defined as follows:

$$chv(x_i) = \frac{\sum_{c_j \in C:\ x_i \in S(c_j) \wedge |Uvars(c_j)|>1} (q(c_j) + \delta)}{|D_i|}$$

$Uvars(c_j)$ denotes the set of unassigned variables in $S(c_j)$ and D_i is the current domain of x_i updated by propagation. The δ parameter is a positive real number close to 0 used to guide branching according to the degree of the variables instead of randomly, at the beginning of the search. CHS chooses the variable to branch on with the highest chv value. It focuses the branching on variables with a small domain size belonging to constraints which appear recently and repetitively in conflicts. When CHS is used in a backtracking algorithm with restarts, $Conflict(c_j)$ and $q(c_j)$ are not reinitialized from a run to another. Nevertheless, the scores of all constraints $c_j \in C$ are smoothed at each restart as follows:

$$q(c_j) = q(c_j) \times 0.995^{Conflicts - Conflict(c_j)}$$

More importantly, for the rest of the paper, CHS resets the value of α to α_0 at each new run.

B. A Multi-Armed Bandit for an Adaptive CHS

The experimental evaluation conducted on CHS reveals that the initial value of the parameter α, i.e. α_0, can be better adjusted to improve the effectiveness of CHS [15]. As an indication, when CHS was tested within a backtracking algorithm with nine different α_0 values, ranging from 0.1 to 0.9 (with a step of 0.1), the Virtual Best Solver (VBS) was able to solve several dozen additional instances [15]. In order to achieve this objective, the present work proposes to use the Multi-Armed Bandit (MAB) framework to choose the relevant values of the α_0 parameter which is of particular importance in CHS. Such a choice will be made at each restart done by the backtracking algorithm. Each arm of the MAB corresponds to a CHS with a different initial α value, i.e. α_0. A training phase is also proposed for a better initialization of the parameters of the arms. Next, we will introduce the proposed MAB framework, detail its use and particularly explain our choice of the reward function.

1) MAB Framework: For a given $i \in \{1 \cdots K\}$, $CHS(\alpha_0^i)$ is defined as CHS where α_0 is set to α_0^i at the beginning of the current restart or run. The arms of the MAB are the candidate heuristics $CHS(\alpha_0^i)$, $i = 1 \cdots K$. Considering a set of K CHS heuristics with different α_0 values and a CSP instance to solve, the proposed framework selects a heuristic $CHS(\alpha_0^i)$ where $i \in \{1 \cdots K\}$ at each restart of the backtracking algorithm according to a MAB policy called Upper Confidence Bound (UCB) [26]. To choose an arm, UCB relies on a reward function calculated at the end of each restart to estimate the performance of the backtracking algorithm regarding the employed search heuristic.

2) Reward Function: An important factor in the efficiency of MAB and, particularly UCB, is the reward function. Indeed, the reward function must reflect the impact of a heuristic on the efficiency of the backtracking algorithm while relying on the information that can be assessed during each run in which it is used. Several criteria can be considered, such as the ability of a heuristic to reduce the size of the search space or to quickly reach failures.

In the proposed MAB framework, any reward function can be used and adapted to a suitable case of use regarding, for example, the instance features or any other criteria. For the current work, several functions have been evaluated through extensive experimentation which highlighted a stronger preference for a reward estimating the ability of a heuristic to reach

failures quickly. This is not surprising as a good part of state-of-the-art heuristics are based on the first-fail principle [28].

Furthermore, the proposed MAB framework is independent from the restart policy, which can be related to the number of conflicts, backtracks, decisions, etc. In this work, the run duration is based on the maximum number of authorized backtracks before achieving a new restart. nc_t denotes the number of conflicts which are permitted during the run t. At each conflict $j = 1 \cdots nc_t$, nu_j variables remain unassigned. Therefore, at conflict j, the ratio p_j of unassigned variables is $p_j = nu_j/n$ where $n = |X|$. If a heuristic $CHS(\alpha_0^i)$ (arm i) is used at the current run t, its reward is calculated at the end of this run as follows:

$$R_t(i) = \frac{\sum_{j=1}^{nc_t} p_j}{nc_t}$$

Hence, this reward function consists in an average of the ratios of unassigned variables encountered during conflicts, assessing the heuristic's ability to quickly identify inconsistent assignments. In other words, the fewer the variables are assigned, the greater the assigned reward value will be for the corresponding heuristic. Finally, one should observe that the values of the rewards are defined in the interval $[0, 1]$.

3) UCB Policy: Consider that t runs are performed since the beginning of the search. The employed MAB policy for selecting the next heuristic is UCB1 [26] which is an algorithm in the Upper Confidence Bound family [25], [26], [27]. UCB1 maintains the following parameters for each candidate arm i:

- $n_t(i)$ is the number of times that an arm i is selected during the t runs,
- $\widehat{R}_t(i)$ is the empirical mean of the rewards of arm i over the t runs.

Accordingly, UCB1 selects the arm that maximizes the following term:

$$\widehat{R}_t(i) + c.\sqrt{\frac{ln(t)}{n_t(i)}} \quad (1)$$

The left side of the formula, i.e. $\widehat{R}_t(i)$, aims to put emphasis on candidate heuristics (arms) that received the highest rewards. Conversely, the right side, i.e. $\sqrt{\frac{ln(t)}{n_t(i)}}$, ensures the exploration of underused arms. The parameter c can help to appropriately balance the interchange between the exploitation and exploration phases in the MAB framework.

A strategy for MAB is evaluated by its expected cumulative regret, i.e., the difference between the cumulative expected value of the reward if the best arm is used at each restart and its cumulative value for the total runs of the MAB, denoted T. If $a_t \in \{1 \cdots K\}$ denotes the arm chosen at run t, the expected cumulative regret is formally defined as follows:

$$R_T = \max_{1 \leq i \leq K} \sum_{t=1}^{T} \mathbf{E}[R_t(i)] - \sum_{t=1}^{T} \mathbf{E}[R_t(a_t)]$$

In particular, UCB1 guaranties an expected cumulative regret no worse than $O(\sqrt{T.K.\ln K})$.

Note that MAB with UCB has been initially defined for rewards following a stationary probability distribution. In our case, the probability distribution is clearly unknown and we cannot ensure that it is stationary. Indeed, the rewards may evolve during the search since we consider MAC with restarts [37] as solving algorithm. As it records some nogoods when a restart occurs, the problem changes over time. There exists variants of UCB adapted for changing environments. However, it is known that the standard version is unlikely to exhibit pathological behavior in such environments [38]. So using UCB1 in this context is relevant.

4) Training Phase: The number of restarts often seems insufficient to fully evaluate the potential of each candidate heuristic thus rendering difficult for the MAB framework to converge quickly to the most adequate arms for the CSP instance. For this reason, a training phase can be conducted. It consists in choosing each arm one at a turn, like a round-robin policy on the different values of α_0. Each arm, among the K candidates, will be attributed the same restart duration limit. In other words, restarts will have a constant duration d_t in this training phase so as to not favor one arm at the expense of another. The round-robin is repeated a certain number of times denoted by d_p.

This training phase can be considered as an initialization step for the parameters used in the MAB framework, especially the reward values and their means. Once the training is completed, UCB1 takes over to select an arm among the candidate search heuristics. It should be noted that by doing so, the expected regret of UCB1 remains in $O(\sqrt{T.K.\ln K})$. Indeed, the training phase is launched first, it has no effect on the expected regret of UCB1. Furthermore, during the training phase, in addition to the initialization of MAB parameters, the weights of CHS are updated during each run when a conflict occurs or at the end of a run through smoothing. This phase may also lead to the discovery of some nogoods if the solver is able to handle them like in [37]. Finally, the training phase alone may suffice to solve some instances.

V. Experimental Evaluation

In this section, we first describe the experimental protocol we use. In Subsection V-B1, we observe the sensitivity of MAB-CHS(+Train) to some parameters. In Subsection V-C, we assess the benefits of the training phase. Afterwards, we compare our MAB approach with state-of-the-art search heuristics in Subsection V-D. Finally, we combine our MAB framework with Last Conflict (LC) heuristic and measure its performance in Subsection V-E.

A. Experimental Protocol

We consider all the CSP instances from the XCSP3 repository[2] and the XCSP3 2018 competition[3], resulting in 16,947 instances. XCSP3, for XML-CSP version 3, is an XML-based format to represent instances of combinatorial constrained problems. Our solver (the same as [15]) is compliant with the rules of the competition except that the global constraints `cumulative`, `circuit` and some variants of the `allDifferent` constraint (namely `except` and

[2]http://www.xcsp.org/series
[3]http://www.cril.univ-artois.fr/XCSP18/

`list`) and the `noOverlap` constraint are not yet supported. Consequently, from the 16,947 obtained instances, we first discard 1,233 unsupported ones. We also remove 2,813 instances which are detected as inconsistent by the initial arc-consistency preprocessing and, thus, having no interest for the present comparison. Hence, our benchmark contains 12,901 instances, including notably structured instances and instances with global constraints.

We consider the following classical heuristics $dom/wdeg$, ABS, wdeg$^{ca.cd}$ [11] and CHS. We do not take into account IBS [10] and CHB [32] which turn to be less relevant [15], [11]. For ABS, we fix the decay parameter γ to 0.999 as in [9]. In CHS, δ is set to 0.0001 while we consider various values of α_0, notably 0.1 and 0.4. For MAB-based heuristics, in addition to our proposed MAB approach, we consider the one defined in [12], which we will refer to as MAB-VH (for MAB with Various Heuristics). MAB-VH exploits five heuristics namely ABS, IBS, dom/ddeg [39], CHS(0.4) and wdeg$^{ca.cd}$ [12]. It aims to take benefit of the heterogeneity of these heuristics in order to identify a relevant one. It relies on a UCB1 policy with $c = 2\sqrt{2}$ and a reward function based on the size of the pruned trees, this size being expressed in terms of the product of the size of the domains of unassigned variables. On the other hand, we discard the MAB approach described in [13] since, in practice, it performs worse than MAB-VH [12]. In order to make the comparison fair, the lexicographic order is used for the choice of the next value to assign within the domains of the variables.

Regarding the solving step, we exploit MAC with restarts [37]. Restarts can be based on various policies. In our experiments, the duration of a run can be measured in the number of backtracks or in the number of decisions. Its evolution can follow a geometric or a Luby sequence. By default, when MAC exploits a classical heuristic alone (resp. MAB-CHS(+Train)), we consider backtrack-based geometric sequences for which the initial cutoff is set to 100 (resp. 50) and the increasing factor to 1.1 (resp. 1.05). For MAB-VH, we consider the setting of [12] with a decision-based Luby sequence whose initial cutoff is set to 100. Moreover, for the sake of simplicity, the duration d_t used in the training phase is equal to the initial cutoff of the restart policy. By default, we set d_p to 10.

We have written our own C++ code to implement all the compared branching heuristics in this section as well as MAC with restarts. By doing so, we avoid any bias related to the way the heuristics or the solver are implemented. In particular, the branching heuristics are all implemented with equal refinement and care, following the recommendations outlined in [40]. The experiments are performed on Dell PowerEdge R440 servers with Intel Xeon Silver 4112 processors (clocked at 2.6 GHz) under Ubuntu 18.04. Each solving process is allocated a slot of 30 minutes and at most 16 GB of memory per instance. The noted time is the cumulative runtime, i.e. the sum of the runtime over all the considered instances.

B. Parameter Sensitivity

1) Exploration vs. Exploitation: We are interested in assessing the impact of the value of parameter c (in the right term of Equation 1) on the ability of MAC with MAB-CHS to solve instances. In the literature, various values of c are considered. This is an important parameter since it controls the balance between the exploitation and exploration phases in the MAB framework. As mentioned before, the reward values in MAB-CHS, and also in MAB-VH, belong to $[0, 1]$. Nonetheless, the values we observe in practice are seldom close to 1. On the contrary, they are often less than 0.5. This trend is even more pronounced when considering $\widehat{R}_t(i)$ due to the effect of the mean. As a consequence, the MAB tends more often towards the exploration part than the exploitation one and, therefore, may be close to a random MAB. Figure 1 depicts the number of solved instances as a function of the elapsed time for different values of c. Clearly, the three values of c we consider lead to close results. However, the value $c = 1$ obtains slightly better results. Indeed it succeeds in solving 6 additional instances (resp. 5) w.r.t. $c = 0.5$ (resp. $c = 2$). Note that this behavior is close to that of MAC with a random MAB, which solves six instances less than MAC with our MAB for $c = 1$. Therefore, in the following experiments, c is set to 1.

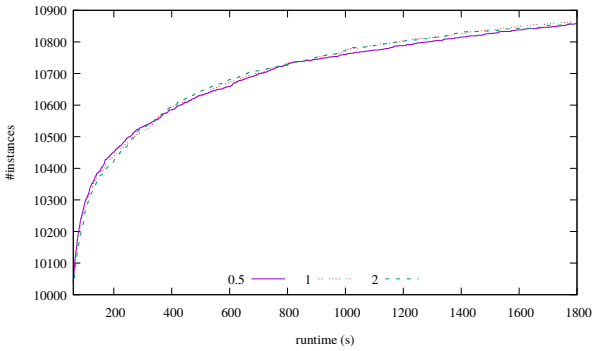


Fig. 1. Number of solved instances as a function of the elapsed time (from 60 s) for MAC with MAB-CHS when c is set respectively to 0.5, 1 or 2.

2) Restart Policy Setting: We consider the increasing factor of the geometric sequence. Indeed this parameter can also influence the behaviour of our MAB. As we can see in Figure 2, values 1.05 and 1.075 turn out to be relevant trade-offs between a weak increase (but more restarts) and a more rapid one (but less restarts). Note that more restarts can mean a better chance for the MAB to make a relevant choice. But, at the same time, it can lead to a less effective search. We can also observe that value 1.05 leads to the best results.

3) Training Phase Setting: We assess the impact of the value of d_t. As the value of d_t is the initial cutoff of the used restart policy, the considered values do not exceed 75 in order to ensure that the duration of runs grows slowly. By doing so, we expect to have enough restarts for the MAB to be able to identify a relevant arm. Figure 3 provides the results of MAC with MAB-CHS+Train. Clearly, $d_t = 50$ turns out to be the best value. Indeed, it allows MAC to solve 10,875 instances against 10,861 and 10,843 respectively for 25 and 75. This gain in number of solved instances is accompanied by a slight reduction of cumulative runtime (1,038 h for 50 against 1,042 h and 1,041 h for 25 and 75). So, $d_t = 50$

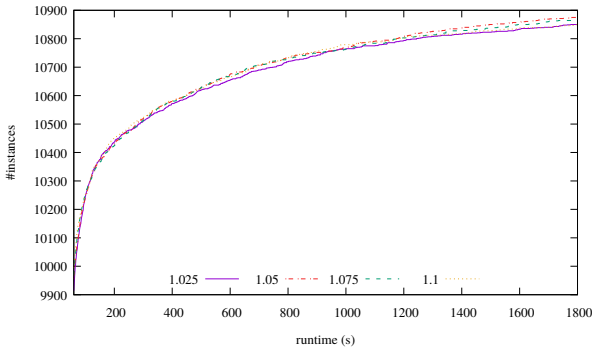


Fig. 2. Number of solved instances as a function of the elapsed time (from 60 s) for MAC with MAB-CHS+Train when the increasing factor is set respectively to 1.025, 1.05, 1.075 or 1.1.

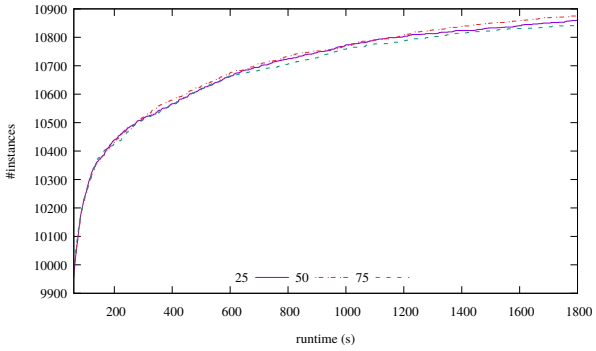


Fig. 3. Number of solved instances as a function of the elapsed time (from 60 s) for MAC with MAB-CHS+Train when d_t is set respectively to 25, 50 or 75.

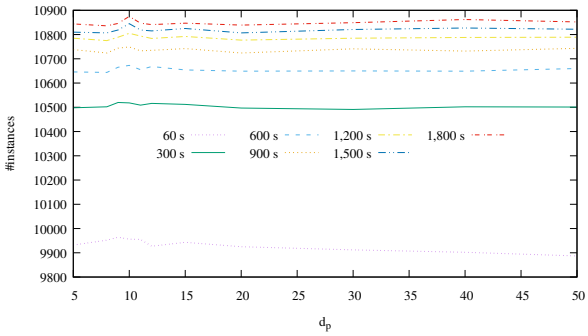


Fig. 4. Number of solved instances within x seconds as a function of the value of d_p for MAC with MAB-CHS+Train.

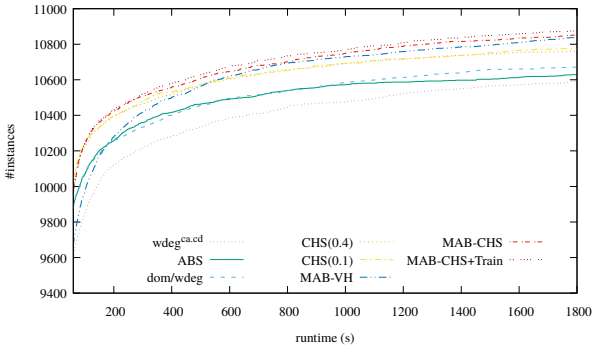


Fig. 5. Number of solved instances as a function of the elapsed time (from 60 s) for MAC with each considered heuristic.

seems to make a good trade-off between the duration of runs and their number.

Finally, We study the MAB-CHS+Train behavior w.r.t. the value of d_p. By varying the value of d_p, the number of instances solved by MAC with MAB-CHS+Train ranges between 10,836 and 10,875. We can clearly observe in Figure 4 the existence of a peak for $d_p = 10$ whatever the limit of runtime we consider above 300 s. Furthermore, the gap between $d_p = 10$ and the other values of d_p is quite significant. For instance, for a timeout of 1,800 s, in most cases, MAC with $d_p = 10$ solves between 20 and 39 additional instances.

To sum up, in the next experiments, we consider the following settings $c = 1$, $d_t = 50$, $d_p = 10$ and an increasing factor of 1.05.

C. Benefits of the Training Phase

First, we compare the results of MAC with MAB-CHS and MAB-CHS+Train (see Figure 5). MAB-CHS+Train leads to the best results. Notably, MAC with MAB-CHS+Train succeeds in solving 10,875 instances against 10,853 for MAC with MAB-CHS. The gain may seem relatively low. However, as we can see in Figure 5, most of the instances are solved easily since, more than 10,000 of them are solved in less than one minute. If we consider the Best Virtual Solver (denoted VBS-CHS) which returns the best answer obtained by MAC with a given $CHS(\alpha_0)$ among the nine considered here, we observe that it solves 10,987 instances. Most of the additional instances it solves are solved by a small number of $CHS(\alpha_0)$ (often only one). So we can suppose that these instances are hard. Note that about 74% of solved instances are solved during the training phase.

Now, we compare MAC with MAB-CHS+Train to VBS-CHS (see Figure 6(a)). Note that by using a MAB approach, our aim is to get as close as possible to VBS-CHS. Of course, MAC with MAB-CHS+Train solves less instances than VBS-CHS and is slower, but this comparison is not fair since the virtual best solver is only a theoretical tool. However, we can observe that the runtime of MAC with MAB-CHS+Train is generally competitive w.r.t. that of VBS-CHS. Interestingly, we observe that MAC with MAB-CHS+Train is able to outperform VBS-CHS on some instances, and even solve instances which are not solved by VBS-CHS within the timeout.

D. MAB-CHS(+Train) vs. Other Search Heuristics

In this subsection, we first compare MAB-CHS(+Train) to some classical branching heuristics of the state-of-the-art. According to Figure 5, MAC with MAB-CHS or MAB-CHS+Train outperforms MAC with any classical heuristic. For MAC with MAB-CHS+Train, the gain in the number of

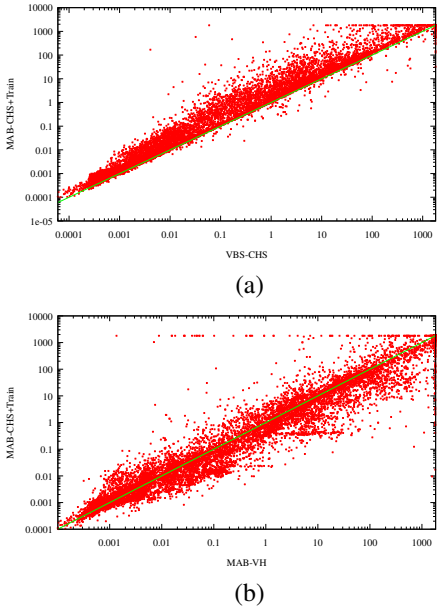


Fig. 6. Runtime comparison of MAB-CHB+Train vs. VBS-CHB (a) and MAB-VH (b).

solved instances ranges between 95 and 279 instances. The best classical heuristic is CHS with $\alpha_0 = 0.1$. Note that 0.1 corresponds to the value of α_0 giving the best results when α_0 varies between 0.1 and 0.9 by step of 0.1. One could consider that the good performance of MAC with MAB-CHS(+Train) is related to the use of the best settings. This is not the case. Indeed, whatever the settings we have considered, we observe that MAC with MAB-CHS(+Train) solves at least 10,836 instances, that is more than 50 additional instances than any classical heuristic.

Next, we compare MAB-CHS(+Train) to another MAB-based approach, namely MAB-VH [12]. MAC with MAB-CHS+Train (resp. MAB-CHS) solves 38 additional instances (resp. 26) than MAB-VH. Roughly, the results of MAB-VH are close to the ones with the worst settings we have considered for MAB-CHS(+Train). Regarding the cumulative runtime, MAB-CHS+Train allows MAC to be faster than MAB-VH (1,038 h vs. 1,068 h). Figure 6(b) confirms this trend where a majority of the instances are solved faster by MAC with MAB-CHS+Train. This result was not a foregone conclusion. Indeed, if both MAB frameworks aim to identify the most relevant heuristic, one may expect that a MAB with different heuristics is more likely to find a relevant one than a MAB using the same heuristic with a different settings in each arm. This can be all the more surprising since the number of instances solved by each CHS variant are close to each other as shown in [15]. One possible explanation lies in the fact that, in practice, different values of α_0 enable us to obtain diversified heuristics. This trend seems to be sustained by the fact that VBS-CHS is able to solve several dozen additional instances than any considered $CHS(\alpha_0^i)$ [15].

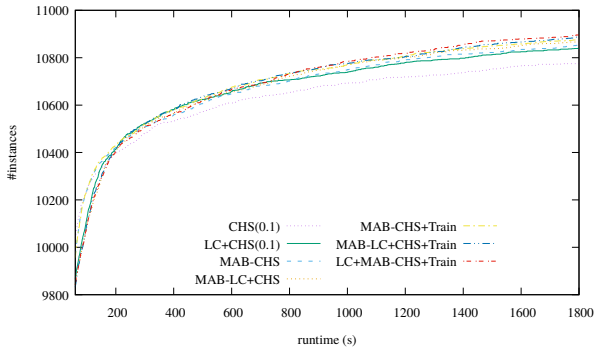


Fig. 7. Number of solved instances as a function of the elapsed time (from 60 s) for MAC with CHS(0.1), MAB-CHS(+Train) and their combination with LC.

E. Combination with LC

The Last Conflict (LC) heuristic [29] aims to focus on the last encountered conflict and chooses a new branching variable only when needed, i.e. when there is no current conflict. In the latter case, it relies on an auxiliary heuristic to make this choice. LC can be used in two different ways with a MAB approach. The first one consists in considering the MAB as the auxiliary heuristic exploited by LC. We denote LC+MAB-CHS(+Train) this version. The second one consists in using LC in each arm of the MAB. In other words, for each arm i, instead of $CHS(\alpha_0^i)$, we consider LC with $CHS(\alpha_0^i)$ as an auxiliary heuristic. We denote MAB-LC+CHS(+Train) this version.

In Figure 7, we compare the behavior of MAC when using these two versions with and without training. We also consider $CHS(0.1)$ and LC+$CHS(0.1)$. Clearly, the use of LC improves the behavior of MAC whatever the auxiliary heuristic we use. We can also observe that the gains, in terms of the number of solved instances, is less important when the auxiliary heuristic alone leads to good results. It seems that the less efficient the heuristic is, the easier it is to solve additional instances. Indeed, incorporating LC has no impact on the hierarchy given in the previous subsection. Moreover, using LC does not make better a given auxiliary heuristic than the next auxiliary heuristic in this hierarchy. For instance, LC+CHS(0.1) does not perform better than CHS-MAB and similarly MAB-LC+CHS performs worse in comparison to MAB-CHS+Train. Finally, the best results are achieved by MAC when using LC+MAB-CHS+Train. It succeeds in solving 22 additional instances w.r.t. MAB-CHS+Train. The gain w.r.t. classical heuristics is even more significant. For example, if we consider the best one, i.e. CHS(0.1), 54 additional instances are solved if we combine LC and CHS, 117 otherwise.

VI. Conclusion and Future Work

Algorithms (or solvers) dedicated to CSP are more and more powerful, but may include parameters which are hard to fix while their values depend on the input instance or any other characteristic. The present work has drawn a framework

based on MAB to refine a single heuristic, namely CHS, regarding an important parameter for estimating the hardness of the constraints following ERWA. The experimental evaluation has validated this approach. Indeed, the performance of CHS was improved and the proposed framework is shown competitive with the state-of-the-art heuristics. MAB-CHS components were also widely investigated. To summarise, our results establish the following hierarchy between the evaluated heuristics in terms of the number of solved instances on the considered benchmark: LC+MAB-CHS+Train $>$ MAB-CHS+Train $>$ MAB-CHS $>$ MAB-VH $>$ classical heuristics ($dom/wdeg$, ABS, wdeg$^{ca.cd}$ and CHS). The originality of this work relies on using MAB with a set of CHS instances differing by the initialization value of the step parameter, while similar work use different heuristics in the MAB framework. The proposed training phase is another contribution which shows its benefits in improving the behavior of the proposed framework.

As perspective of our work, it would be interesting to study the effect of combining our MAB framework with the one introduced in [12], MAB-VH, by adding different heuristics from CHS as arms to MAB-CHS(+Train). Furthermore, refining the reward function by relying on a combination of different criteria [41], [42] may lead to the improvement of our framework as it is difficult to fully reflect the behavior of solvers, which have complex environments, while relying on a single criterion. Finally, it would be relevant to apply the same framework on close problems such as SAT, where the reward function may be related to the quality of clause learning [43], which is an important module in modern SAT solvers.

Acknowledgment

This work has been funded by the Agence Nationale de la Recherche project ANR-16-CE40-0028.

References

[1] F. Rossi, P. van Beek, and T. Walsh, *Handbook of Constraint Programming*, ser. Foundations of Artificial Intelligence. Elsevier, 2006, vol. 2.

[2] P. Liberatore, "On the complexity of choosing the branching literal in DPLL," *Artificial Intelligence*, vol. 116, no. 1-2, 2000.

[3] C. Bessière, A. Chmeiss, and L. Saïs, "Neighborhood-based variable ordering heuristics for the constraint satisfaction problem," in *Proceedings of CP*, 2001, pp. 565–569.

[4] C. Bessière and J.-C. Régin, "MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems," in *Proceedings of CP*, 1996, pp. 61–75.

[5] F. Boussemart, F. Hemery, C. Lecoutre, and L. Saïs, "Boosting Systematic Search by Weighting Constraints," in *Proceedings of ECAI*, 2004, pp. 146–150.

[6] P. A. Geelen, "Dual Viewpoint Heuristics for Binary Constraint Satisfaction Problems," in *Proceedings of ECAI*, 1992, pp. 31–35.

[7] S. W. Golomb and L. D. Baumert, "Backtrack programming," *Journal of the ACM*, vol. 12, pp. 516–524, 1965.

[8] E. Hebrard and M. Siala, "Explanation-Based Weighted Degree," in *Proceedings of CPAIOR*, 2017, pp. 167–175.

[9] L. Michel and P. V. Hentenryck, "Activity-based search for black-box constraint programming solvers," in *Proceedings of CPAIOR*, 2012, pp. 228–243.

[10] P. Refalo, "Impact-based search strategies for constraint programming," in *Proceedings of CP*, 2004, pp. 557–571.

[11] H. Wattez, C. Lecoutre, A. Paparrizou, and S. Tabary, "Refining Constraint Weighting," in *Proceedings of ICTAI*, 2019, pp. 71–77.

[12] H. Wattez, F. Koriche, C. Lecoutre, A. Paparrizou, and S. Tabary, "Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts," in *Proceedings of ECAI*, 2020.

[13] W. Xia and R. H. C. Yap, "Learning Robust Search Strategies Using a Bandit-Based Approach," in *Proceedings of AAAI*, 2018, pp. 6657–6665.

[14] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Learning Rate Based Branching Heuristic for SAT Solvers," in *Proceedings of SAT*, 2016, pp. 123–140.

[15] D. Habet and C. Terrioux, "Conflict History based Search for Constraint Satisfaction Problem," in *Proceeding of SAC, Knowledge Representation and Reasoning Technical Track*, 2019, pp. 1117–1122.

[16] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[17] C. P. Gomes, B. Selman, N. Crato, and H. A. Kautz, "Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems," *Journal of Automated Reasoning*, vol. 24, no. 1/2, pp. 67–100, 2000.

[18] T. Walsh, "Search in a Small World," in *Proceedings of IJCAI*, 1999, pp. 1172–1177.

[19] M. Luby, A. Sinclair, and D. Zuckerman, "Optimal speedup of Las Vegas algorithms," *Information Processing Letters*, vol. 47(4), pp. 173–180, 1993.

[20] A. Balafrej, C. Bessiere, and A. Paparrizou, "Multi-Armed Bandits for Adaptive Constraint Propagation," in *Proceedings of IJCAI*, 2015, pp. 290–296.

[21] A. K. Mackworth, "Consistency in networks of relations," *Artificial Intelligence*, vol. 8, p. 99–118, 1977.

[22] D. Sabin and E. C. Freuder, "Contradicting Conventional Wisdom in Constraint Satisfaction," in *Proceedings of ECAI*, 1994, pp. 125–129.

[23] T. L. Lai and H. Robbins, "Asymptotically efficient adaptive allocation rules," *Advances in applied mathematics*, vol. 6, no. 1, pp. 4–22, 1985.

[24] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, no. 3/4, pp. 285–294, 1933.

[25] R. Agrawal, "Sample mean based index policies by o (log n) regret for the multi-armed bandit problem," *Advances in Applied Probability*, vol. 27, no. 4, pp. 1054–1078, 1995.

[26] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time Analysis of the Multiarmed Bandit Problem," *Mach. Learn.*, vol. 47, no. 2-3, pp. 235–256, 2002.

[27] E. Kaufmann, O. Cappé, and A. Garivier, "On Bayesian Upper Confidence Bounds for Bandit Problems," in *Proceedings of AISTATS*, 2012, pp. 592–600.

[28] R. M. Haralick and G. L. Elliot, "Increasing tree search efficiency for constraint satisfaction problems," *Artificial Intelligence*, vol. 14, pp. 263–313, 1980.

[29] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal, "Last Conflict Based Reasoning," in *Proceedings of ECAI*, 2006, pp. 133–137.

[30] S. Gay, R. Hartert, C. Lecoutre, and P. Schaus, "Conflict Ordering Search for Scheduling Problems," in *Proceedings of CP*, G. Pesant, Ed., 2015, pp. 140–148.

[31] J. H. Liang, V. Ganesh, P. Poupart, and K. Czarnecki, "Exponential Recency Weighted Average Branching Heuristic for SAT Solvers," in *Proceedings of AAAI*, 2016, pp. 3434–3440.

[32] C. Schulte, "Programming branchers," in *Modeling and Programming with Gecode*, C. Schulte, G. Tack, and M. Z. Lagerkvist, Eds., 2018, corresponds to Gecode 6.0.1.

[33] M. Gagliolo and J. Schmidhuber, "Learning Restart Strategies," in *Proceedings of IJCAI*, 2007, pp. 792–797.

[34] G. Chu and P. J. Stuckey, "Learning Value Heuristics for Constraint Programming," in *Integration of AI and OR Techniques in Constraint Programming*. Springer International Publishing, 2015, pp. 108–123.

[35] I. Bachiri, J. Gaudreault, C. Quimper, and B. Chaib-draa, "RLBS: An Adaptive Backtracking Strategy Based on Reinforcement Learning for Combinatorial Optimization," in *Proceedings of ICTAI*, 2015, pp. 936–942.

[36] R. Battiti and P. Campigotto, *An Investigation of Reinforcement Learning for Reactive Search Optimization*. Springer Berlin Heidelberg, 2012, pp. 131–160.

[37] C. Lecoutre, L. Sais, S. Tabary, and V. Vidal, "Recording and Minimizing Nogoods from Restarts," *JSAT*, vol. 1, no. 3-4, pp. 147–167, 2007.

[38] Álvaro Fialho, L. D. Costa, and M. S. Marc Schoenauer, "Analyzing bandit-based adaptive operator selection mechanisms," *Ann. Math. Artif. Intell.*, vol. 60(1-2), pp. 25–64, 2010.

[39] B. M. Smith and S. A. Grant, "Trying Harder to Fail First," in *Proceedings of ECAI*, 1998, pp. 249–253.

[40] J. N. Hooker, "Testing Heuristics: We Have It All Wrong," *Journal of Heuristics*, vol. 1(1), pp. 33–42, 1995.

[41] W. Chu, L. Li, L. Reyzin, and R. Schapire, “Contextual Bandits with Linear Payoff Functions,” in *Proceedings of AISTATS*, 2011, pp. 208–214.

[42] L. Li, W. Chu, J. Langford, and R. E. Schapire, “A Contextual-Bandit Approach to Personalized News Article Recommendation,” in *Proceedings of WWW*, 2010, pp. 661–670.

[43] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Proceedings of SAT*, 2003, pp. 502–518.

Focus sur les heuristiques basées sur la pondération de contraintes

Hugues Wattez[1]* **Frédéric Koriche**[1] **Christophe Lecoutre**[1]
Anastasia Paparrizou[1] **Sébastien Tabary**[1]

[1] CRIL, Univ. Artois & CNRS, 62300 Lens, France
{wattez,koriche,lecoutre,paparrizou,tabary}@cril.fr

Résumé

La recherche avec retour-arrière est une approche complète et traditionnellement utilisée pour la résolution de problèmes de satisfaction et d'optimisation de contraintes. Il est bien connu que l'espace exploré durant la recherche peut drastiquement fluctuer selon l'ordre dans lequel les variables sont instanciées. En considérant que le parfait ordonnancement des variables résulterait en une recherche sans retour-arrière, la recherche d'heuristiques de choix de variables suscite toujours autant l'intérêt des chercheurs. Depuis quinze ans, l'approche basée sur la pondération des contraintes s'est toujours montrée efficace pour guider la recherche. Dans cet article, nous montrons comment les heuristiques génériques de choix de variables `wdeg` et `dom/wdeg` ont été rendues plus robustes ces dernières années par un affinement et le vieillissement de l'information extraite à chaque conflit. Nos résultats expérimentaux avec `ACE`, le nouvel avatar d'`AbsCon`, montrent le réel intérêt de ces heuristiques revisitées.

Abstract

Backtracking search is a complete approach that is traditionally used to solve instances modeled as constraint satisfaction or optimization problems. The space explored during search depends dramatically on the order that variables are instantiated. Considering that a perfect variable ordering might result to a backtrack-free search, finding variable ordering heuristics has always attracted research interest. For fifteen years, constraint weighting has been shown to be a successful approach for guiding backtrack search. In this paper, we show how the popular generic variable ordering heuristics `dom/wdeg` and `wdeg` have been made more robust these two last years by refining and aging the extracted information at each conflict. Our experimental results with `ACE`, the new `AbsCon` avatar, show the practical interest of these revisited constraint weighting heuristics.

*Papier doctorant : Hugues Wattez[1] est auteur principal.

1 Introduction

Les solveurs de contraintes sont constitués de plusieurs composants, dédiés au filtrage de l'espace de recherche, à la conduite de l'exploration, ou encore à l'apprentissage de nogoods, dont la configuration et l'ajustement ont un impact réel sur les performances. Un point particulièrement important dans le choix d'une bonne configuration concerne la sélection de l'heuristique de choix de variables. Une heuristique soigneusement choisie permet de réduire significativement la taille de l'espace de recherche parcouru pour un problème donné. Dans cet article, nous nous attardons sur une famille d'heuristiques génériques de choix de variables ayant démontré leur grande robustesse au cours des ans : les heuristiques basées sur la pondération de contraintes dont `dom/wdeg` et `wdeg` [1] sont les précurseurs. Ces dernières années, de nouvelles avancées ont été faites sur la base de la pondération de contraintes. Tout d'abord, `chs` [4], qui utilise le principe de vieillissement des scores basés sur l'historique des conflits, est venu apporter une nouvelle dimension aux heuristiques d'origine. Ensuite, $\texttt{wdeg}^{\texttt{ca.cd}}$ [13], qui ajuste la pondération en fonction du contexte (arité courante des contraintes, taille des domaines courants), a également semblé démontrer un surcroît de robustesse.

Le but de notre étude est de proposer une uniformisation des différentes heuristiques que nous considérons appartenir à une même famille. La première étape consiste à identifier les principales fonctions de ces heuristiques afin d'établir une interface uniformisant les mécanismes de chacune d'entre elles. L'étape suivante décline chaque heuristique, dans le cadre de cette interface, en accord avec leur descriptif, mais aussi l'implantation effectuée dans le solveur `ACE`. Ainsi, nous espérons que le lecteur aura une meilleure appréhen-

sion des ressemblances et différences des mécanismes de ces heuristiques dont l'apparenté deviendra par la même occasion plus évidente. Pour finir, une évaluation expérimentale est présentée sur la base d'une sélection de trois heuristiques principales en diversifiant légèrement la configuration du solveur, et en utilisant les benchmarks des dernières compétitions XCSP [3, 2, 6], que ce soit pour la satisfaction ou l'optimisation de contraintes.

2 Réseaux de contraintes

Un *réseau de contraintes* (CN pour *Constraint Network*) P est composé d'un ensemble fini de variables $\mathscr{X}$, et d'un ensemble fini de contraintes $\mathscr{C}$. Chaque variable x prend une valeur dans un domaine fini, noté $\texttt{dom}(x)$. Chaque contrainte c représente une relation mathématique $\texttt{rel}(c)$ associé à un ensemble de variables, appelé la portée (*scope*) de c, et noté $\texttt{scp}(c)$. L'*arité* d'une contrainte c est la taille de sa portée.

Une *solution* de P correspond à l'affectation d'une valeur à chaque variable de $\mathscr{X}$ de sorte que toutes les contraintes de $\mathscr{C}$ soient satisfaites. Un réseau de contraintes est *cohérent* lorsqu'il admet au moins une solution. Le *problème de satisfaction de contraintes* (CSP pour *Constraint Satisfaction Problem*) consiste à déterminer si un CN donné est cohérent ou non.

Un réseau de contraintes sous optimisation (CNO pour *Constraint Network under Optimization*) est un réseau de contraintes combiné à une fonction d'objectif $\texttt{obj}$ permettant d'associer à toute solution une valeur dans $\mathbb{R}$. Sans perte de généralité, nous considérons que $\texttt{obj}$ est à minimiser. Une solution S d'un CNO est une solution du CN sous-jacent ; S est *optimale* s'il n'existe pas d'autre solution S' tel que $\texttt{obj}(S') < \texttt{obj}(S)$. Le *problème d'optimisation sous contraintes* (COP pour *Constraint Optimisation Problem*) consiste à trouver une solution optimale à un CNO donné.

Une procédure classique pour résoudre ces problèmes est d'effectuer une recherche avec retour-arrière dans l'espace des solutions partielles, et de maintenir une propriété appelée l'*arc-cohérence généralisée* [8] après chaque décision. Cette procédure, appelée le *maintien de l'arc-cohérence* (MAC pour *Maintaining Arc Consistency*) [9], construit un arbre binaire de recherche $\mathcal{T}$: pour chaque nœud interne ν de $\mathcal{T}$, un couple (x, v) est sélectionné où x est une variable non affectée et v est une valeur dans $\texttt{dom}(x)$. Ainsi, deux cas sont considérés pour l'exploration : l'affectation $x = v$ (décision dite positive) et la réfutation $x \neq v$ (décision dite négative).

L'ordre dans lequel les variables sont choisies durant l'exploration est décidé par une *heuristique de choix de variables*, notée H. Autrement dit, à chaque nœud interne ν de l'arbre de recherche $\mathcal{T}$, la procédure MAC sélectionne une nouvelle variable x en utilisant H, et affecte à x une valeur v en accord avec l'heuristique de choix de valeurs, qui est par exemple tout simplement l'ordre lexicographique sur $\texttt{dom}(x)$. Choisir la bonne heuristique de choix de variables H pour un réseau de contraintes donné est une question clé. Dans la section suivante, nous présentons l'état de l'art des heuristiques de choix de variables basant l'évaluation relative des variables sur le concept de pondération de contraintes.

Pour finir, il est important de noter que les solveurs modernes sont tous équipés d'une politique de redémarrage, afin de permettre une meilleure diversification, et contrer de possibles mauvais choix (décisions) initiaux. C'est pourquoi, dans la section suivante, nous faisons référence à la notion de *run* qui est l'arbre partiel de recherche construit entre deux (re-)démarrages.

3 Heuristiques de choix de variables

Afin d'exprimer simplement, et de manière homogène, les différentes heuristiques qui vont suivre, nous utilisons une interface composée de quatre fonctions.

init() Avant que ne démarre la phase d'exploration par le solveur (c'est-à-dire que le solveur n'entame sa série de runs), l'heuristique de choix de variables initialise ses structures par le biais de cette fonction.

beforeRun() À chaque début de run, cette fonction permet la mise-à-jour des différents paramètres ou scores de l'heuristique.

atConflict(c) Pendant le run courant, le solveur peut rencontrer des conflits (lorsque le domaine d'une variable devient vide lors de l'appel d'un algorithme de filtrage sur l'une des contraintes). À chaque conflit, la fonction atConflict est appelée avec la contrainte c ayant causé le conflit.

score(x) Cette fonction est appelée (sur toutes les variables non déjà instanciées) lors de la sélection de la prochaine variable à instancier par l'heuristique. D'un point de vue global, le solveur cherche à identifier la variable dont le score est maximal : $\max_{x \in \mathscr{X}} \text{SCORE}(x)$. En pratique, seules les variables libres (non-assignées par le solveur) du réseau de contraintes seront considérées.

En cas de meilleurs scores identiques, il existe un mécanisme subsidiaire permettant de discriminer les variables, appelé *tie-breaker*. Ce mécanisme, extérieur à l'interface précédemment déclarée, peut être considéré comme un biais $b(x)$ venant apporter une information supplémentaire sur la variable x. Ce biais peut avoir une

influence certaine sur l'heuristique de choix de variables car il permet notamment de pré-ordonner l'ensemble des variables au début du premier run. Ce biais peut être arbitraire ; basé sur un générateur aléatoire, sur l'ordre lexicographique du nom des variables, ou encore sur un ordre défini par une heuristique secondaire telle que `deg` (plus grand degré) par exemple.

Maintenant que l'interface est définie, la description de chacune des heuristiques auxquelles nous nous intéressons dans cet article peut être donnée. Chacune de ces heuristiques est basée sur une forme de pondération, et est donc notée `wdeg` (weighted degree) en suivant l'appellation choisie initialement [1]. Les différentes variantes de cette famille d'heuristiques sont alors identifiées par un terme placé en exposant, comme par exemple dans $\texttt{wdeg}^{\texttt{ca.cd}}$. Il est également possible d'utiliser `dom/wdeg` à la place de `wdeg` comme base de calcul ; cela sera défini un peu plus loin.

Algorithme 1 : $\texttt{wdeg}^{\texttt{unit-2004}}$

Méthode INIT() :
  pour chaque $c \in \mathscr{C}$ **faire**
    $w_c \leftarrow 0$

Méthode BEFORERUN() :
  Aucune opération

Méthode ATCONFLICT(c) :
  $r \leftarrow 1$
  $w_c \leftarrow w_c + r$

Méthode SCORE(x) :
  retourner $\sum_{c \in \mathscr{C} : x \in scp(c) \wedge |\texttt{fut}(c)|>1} w_c$

$\texttt{wdeg}^{\texttt{unit-2004}}$ Il s'agit de l'heuristique originelle de la famille `wdeg`. Comme structure de données, cette heuristique ne nécessite en fait qu'une variable w_c par contrainte c pour enregistrer le score de celle-ci. À chaque conflit issu d'une contrainte c (lors du processus de filtrage par celle-ci), cette variable, qui sert simplement de compteur, est incrémentée de 1 (*unit* dans l'intitulé faisant référence à une incrémentation unitaire). Ainsi, le score d'une variable x, selon $\texttt{wdeg}^{\texttt{unit-2004}}$, correspond à la somme du poids des contraintes (ayant strictement plus d'une variable non-assignée ; $fut(c)$ désigne l'ensemble des variables *futures* de $scp(c)$, c'est-à-dire des variables non assignées par le solveur, impliquant x.

Algorithme 2 : $\texttt{wdeg}^{\texttt{unit}}$

Méthode INIT() :
  pour chaque $c \in \mathscr{C}$ **faire**
    pour chaque $x \in \texttt{scp}(c)$ **faire**
      $w_c^x \leftarrow 0$

Méthode BEFORERUN() :
  Aucune opération

Méthode ATCONFLICT(c) :
  pour chaque $x \in \texttt{fut}(c)$ **faire**
    $r \leftarrow 1$
    $w_c^x \leftarrow w_c^x + r$

Méthode SCORE(x) :
  retourner $\sum_{c \in \mathscr{C} : x \in scp(c) \wedge |\texttt{fut}(c)|>1} w_c^x$

$\texttt{wdeg}^{\texttt{unit}}$ Une première amélioration, implantée assez rapidement dans `AbsCon`, consiste à affiner la façon dont le poids évolue : non plus globalement pour toutes les variables d'une contrainte, mais seulement pour celles qui sont futures. Il est alors nécessaire d'introduire un tableau pour chaque contrainte c afin d'enregistrer le score (poids) local w_c^x d'une variable x impliquée dans c. La particularité réside ainsi dans la manière d'interpréter un conflit : au lieu d'augmenter globalement le poids de la contrainte ayant causé le conflit, seules les variables futures (non encore assignées) dans la portée de la contrainte voient leur poids incrémenté.

$\texttt{wdeg}^{\texttt{ca.cd}}$ Toujours dans l'optique d'affiner la pondération, $\texttt{wdeg}^{\texttt{ca.cd}}$ [13] propose une nouvelle amélioration. Après chaque conflit, l'incrémentation du score local des variables futures de la contrainte conflictuelle est construit sur l'arité courante (nombre de variables futures) et la taille des domaines courants.

$\texttt{wdeg}^{\texttt{chs}}$ En plus des variables w_c, comme pour $\texttt{wdeg}^{\texttt{unit-2004}}$, une variable t_c pour chaque contrainte c et correspondant au temps du dernier conflit impliquant la contrainte c est introduite pour l'heuristique $\texttt{wdeg}^{\texttt{chs}}$ [4] : le « temps » correspond ici à un compteur de conflits et non à une horloge effective. Par la suite, `time` est introduit et correspond au temps du dernier conflit du solveur : $\texttt{time} = \max_{c \in \mathscr{C}} t_c$. Les valeurs des variables t_c servent par la suite à différencier les contraintes entrant souvent en conflit. La fonction BEFORERUN est utilisée dans l'implantation de $\texttt{wdeg}^{\texttt{chs}}$

Algorithme 3 : $\texttt{wdeg}^{\texttt{ca.cd}}$

Méthode INIT() :
  pour chaque $c \in \mathscr{C}$ **faire**
    pour chaque $x \in \texttt{scp}(c)$ **faire**
      $w_c^x \leftarrow 0$

Méthode BEFORERUN() :
  Aucune opération

Méthode ATCONFLICT(c) :
  pour chaque $x \in \texttt{fut}(c)$ **faire**
    $r \leftarrow \frac{1}{|\texttt{fut}(x)| \times \max(|\texttt{dom}(x)|, 1/2)}$
    $w_c^x \leftarrow w_c^x + r$

Méthode SCORE(x) :
  retourner $\sum_{c \in \mathscr{C} : x \in scp(c) \wedge |\texttt{fut}(c)|>1} w_c^x$

Algorithme 4 : $\texttt{wdeg}^{\texttt{chs}}$

Méthode INIT() :
  $\texttt{time} \leftarrow 0$
  pour chaque $c \in \mathscr{C}$ **faire**
    $w_c \leftarrow 0$
    $t_c \leftarrow 0$

Méthode BEFORERUN() :
  pour chaque $c \in \mathscr{C}$ **faire**
    $w_c \leftarrow w_c \times 0.995^{\texttt{time}-t_c}$
  $\alpha \leftarrow 1/10$

Méthode ATCONFLICT(c) :
  $r \leftarrow \frac{1}{\texttt{time}-t_c+1}$
  $\alpha \leftarrow \max(6/100, \alpha - 10^{-6})$
  $w_c \leftarrow (1-\alpha) \times w_c + \alpha \times r$
  $\texttt{time} \leftarrow \texttt{time} + 1$
  $t_c \leftarrow \texttt{time}$

Méthode SCORE(x) :
  retourner $\sum_{c \in : x \in scp(c) \wedge |\texttt{fut}(c)|>1} w_c$

afin de vieillir la pondération des contraintes en fonction du temps depuis lequel elle n'est pas entrée en conflit : plus ce temps est important, plus l'affaiblissement du score l'est aussi. Une variable α initialisée à $1/10$ et évoluant au cours des conflits est introduite afin de donner plus ou moins d'importance au score donné à une variable lors d'un conflit. La fonction ATCONFLICT reprend ce principe de temps depuis lequel la contrainte a subi son dernier conflit pour calculer l'incrément : $r \leftarrow \frac{1}{\texttt{time}-t_c+1}$. La paramètre α est mis à jour à chaque conflit : $\alpha \leftarrow \max(6/100, \alpha - 10^{-6})$. Plus le nombre de conflits est important, plus α s'affaiblit et plus l'historique de la pondération de la contrainte c sera conservé : $w_c \leftarrow (1-\alpha) \times w_c + \alpha \times r$. Autrement dit, dans les premiers conflits du run, les incréments ont un impact plus grand sur la pondération des contraintes. Enfin, t_c est mis à jour en lui attribuant la valeur du dernier conflit. Les constantes de cette heuristique ont été calculées empiriquement dans l'étude [4] se basant sur des travaux d'une heuristique de branchement pour les solveurs SAT [7].

4 Évalutation expérimentale

Cette section présente les résultats expérimentaux composés de différentes campagnes comparant les trois heuristiques implantées dans ACE telles que présentées dans la littérature : $\texttt{dom/wdeg}^{\texttt{unit}}$, $\texttt{wdeg}^{\texttt{ca.cd}}$ et $\texttt{dom/wdeg}^{\texttt{chs}}$. Par défaut, ACE utilise une suite géométrique de raison 1.1 pour le *cutoff* (point (d'arrêt) du mécanisme de redémarrage (le cutoff du premier run est fixé à 10 mauvaises décisions), lexico comme heuristique de choix de valeurs et *last-conflict* (lc) [5] comme une manière paresseuse de simuler des retours-arrières intelligents en retenant les k derniers conflits.

L'ensemble des campagnes a été exécuté dans les mêmes conditions [1] avec un temps maximal d'exécution, pour chaque instance, fixé à 1,200 secondes. Les différentes heuristiques ont été exécutées sur une large diversité de problèmes de contraintes venant de la distribution XCSP [3, 6]. Nous utilisons deux benchmarks. Un premier, $\mathcal{I}_{\texttt{CSP}}$, correspond à l'ensemble des instances CSP des trois dernières compétitions XCSP : XCSP'17, XCSP'18 et XCSP'19, pour un total de 83 familles de problèmes et de 810 instances. Le second benchmark, $\mathcal{I}_{\texttt{COP}}$, est composé de l'ensemble des instances COP des compétitions XCSP'18 et XCSP'21, pour un total de 51 familles de problèmes et 697 instances.

Sur la base des trois heuristiques sélectionnées, nous souhaitons observer le comportement de chacune d'elles sur trois configurations du solveur ACE :

1. CPU 3.3 GHz Intel XEON E5-2643 et 32 GB RAM

— $\texttt{ACE}^{lc=2}_{b=lex}$: avec le mécanisme *last-conflict* retenant les deux derniers conflits et l'ordre lexicographique comme *tie-breaker* ;
— $\texttt{ACE}^{lc=0}_{b=lex}$: sans le mécanisme *last-conflict* et l'ordre lexicographique comme *tie-breaker* ;
— $\texttt{ACE}^{lc=2}_{b=deg}$: avec le mécanisme *last-conflict* retenant les deux derniers conflits et les poids proposés par `deg` comme *tie-breaker*.

Le solveur `ACE` par défaut correspond à $\texttt{ACE}^{lc=2}_{b=lex}$. Il s'agit par la même occasion de vérifier la robustesse de la configuration par défaut de `ACE` en variant deux de ses paramètres : *lc* et *b*. Pour cela, nous expérimentons les trois heuristiques données sur les trois environnements du solveur à travers deux sections : la première met en concurrence les solveurs sur $\mathcal{I}_{\texttt{CSP}}$ et la deuxième sur $\mathcal{I}_{\texttt{COP}}$.

4.1 Campagne sur les CSPs

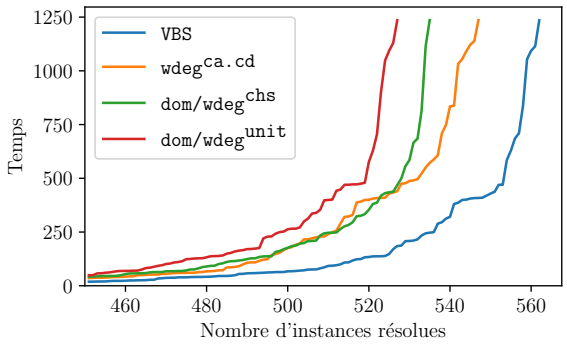


FIGURE 1 – Comparaison des heuristiques avec la configuration $\texttt{ACE}^{lc=2}_{b=lex}$ sur $\mathcal{I}_{\texttt{CSP}}$

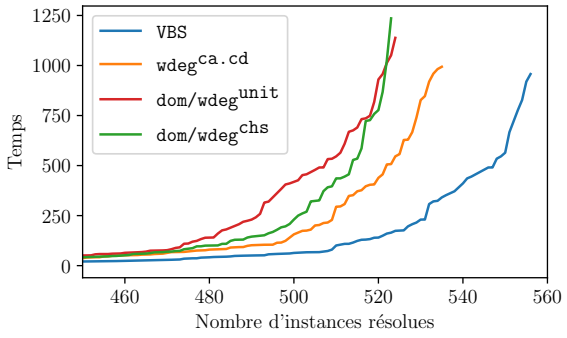


FIGURE 2 – Comparaison des heuristiques avec la configuration $\texttt{ACE}^{lc=0}_{b=lex}$ sur $\mathcal{I}_{\texttt{CSP}}$

Les Figures 1, 2 et 3 montrent les résultats des trois heuristiques et leur `VBS` pour les trois environnements du solveur `ACE`. Un `VBS` (pour *Virtual Best Solver*) correspond à un solveur virtuel simulant les meilleurs

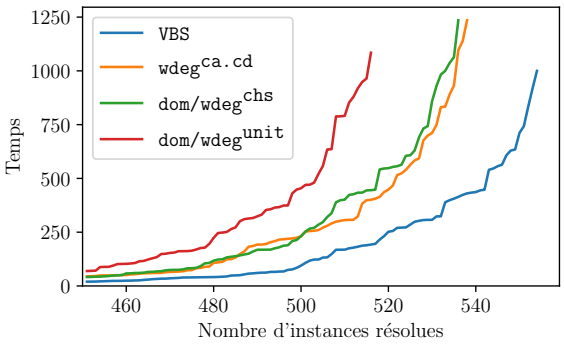


FIGURE 3 – Comparaison des heuristiques avec la configuration $\texttt{ACE}^{lc=2}_{b=deg}$ sur $\mathcal{I}_{\texttt{CSP}}$

résultats des solveurs réels : pour une instance donnée, le `VBS` prend le résultat de la meilleure des heuristiques. Dans notre cas, le `VBS` correspond toujours aux heuristiques courantes de la figure présentée. Un cactus-plot permet de mettre en évidence l'efficacité de chacun des solveurs : chaque solveur est représenté par une courbe représentant le temps cpu de chacune des instances résolues par celui-ci et ceci, dans l'ordre croissant de ce temps de résolution (représenté en y). Ainsi, pour un y donné, plus la courbe d'un solveur tend vers la droite, plus le solveur a résolu un grand nombre d'instances en ce temps donné. Pour simplifier la comparaison entre les cactus-plots, une heuristique est toujours associée à la même couleur. La légende donne le nom des solveurs dans l'ordre décroissant du nombre d'instances résolues.

De manière générale, nous remarquons que l'heuristique $\texttt{dom/wdeg}^{\texttt{unit}}$ est bel et bien dominée par ses successeurs $\texttt{dom/wdeg}^{\texttt{chs}}$ et $\texttt{wdeg}^{\texttt{ca.cd}}$ (bien que la configuration $\texttt{ACE}^{lc=0}_{b=lex}$ fasse légèrement défaut à $\texttt{dom/wdeg}^{\texttt{chs}}$ à partir de $y = 1,000$ secondes). Excepté pour la configuration $\texttt{ACE}^{lc=2}_{b=deg}$ où $\texttt{dom/wdeg}^{\texttt{chs}}$ et $\texttt{wdeg}^{\texttt{ca.cd}}$ semblent sensiblement égales, $\texttt{wdeg}^{\texttt{ca.cd}}$ obtient toujours les meilleurs résultats en terme du nombre d'instances résolues. L'étude des `VBS` apporte une information intéressante : il semblerait que ces trois heuristiques ont un comportement suffisamment différents pour créer un décalage d'une vingtaine à une trentaine d'instances par rapport au meilleur solveur réel. En effet, une analyse complémentaire montre que sur la configuration $\texttt{ACE}^{lc=2}_{b=lex}$, les heuristiques $\texttt{dom/wdeg}^{\texttt{unit}}$, $\texttt{wdeg}^{\texttt{ca.cd}}$ et $\texttt{dom/wdeg}^{\texttt{chs}}$ ont respectivement résolu 4, 17 et 6 instances que les autres heuristiques n'ont pas été en mesure de résoudre.

À l'échelle des configurations de `ACE`, celle apportant les meilleures performances pour les différentes heuristiques semble correspondre à la configuration par défaut, soit $\texttt{ACE}^{lc=2}_{b=lex}$. Cette observation est corroborée

	$\texttt{ACE}^{lc=2}_{b=lex}$	$\texttt{ACE}^{lc=2}_{b=deg}$	$\texttt{ACE}^{lc=0}_{b=lex}$
$\texttt{dom/wdeg}^{\texttt{unit}}$	527 $(75,519s)$	516 $(90,499s)$	524 $(82,409s)$
$\texttt{wdeg}^{\texttt{ca.cd}}$	**547** $(57,394s)$	**538** $(66,927s)$	**535** $(63,877s)$
$\texttt{dom/wdeg}^{\texttt{chs}}$	535 $(63,182s)$	536 $(69,583s)$	523 $(76,540s)$

TABLE 1 – Comparaison des heuristiques avec différentes configurations du solveur ACE sur $\mathcal{I}_{\texttt{CSP}}$

par les résultats donnés par le tableau 1. Notons que les configurations supprimant le mécanisme *last-conflict* ou utilisant le biais basé sur le degré des variables semblent détériorer les résultats (en terme d'instances résolues et/ou de temps).

4.2 Campagne sur les COPs

		$\mathcal{S}_{\texttt{domi}}$	$\mathcal{S}_{\texttt{opti}}$	$\mathcal{S}_{\texttt{agg}}$	$\mathcal{S}_{\texttt{borda}}$
$\texttt{ACE}^{lc=2}_{b=lex}$	$\texttt{dom/wdeg}^{\texttt{unit}}$	338	167	505	5,880.44
	$\texttt{wdeg}^{\texttt{ca.cd}}$	395	181	576	**6,178.23**
	$\texttt{dom/wdeg}^{\texttt{chs}}$	334	176	510	5,785.58
$\texttt{ACE}^{lc=2}_{b=deg}$	$\texttt{dom/wdeg}^{\texttt{unit}}$	366	170	536	5,954.80
	$\texttt{wdeg}^{\texttt{ca.cd}}$	**402**	**184**	**586**	6,143.00
	$\texttt{dom/wdeg}^{\texttt{chs}}$	365	179	544	5,862.42
$\texttt{ACE}^{lc=0}_{b=lex}$	$\texttt{dom/wdeg}^{\texttt{unit}}$	321	158	479	5,743.46
	$\texttt{wdeg}^{\texttt{ca.cd}}$	356	172	528	6,026.50
	$\texttt{dom/wdeg}^{\texttt{chs}}$	300	163	463	5,652.59

TABLE 2 – Comparaison des heuristiques avec différentes configurations du solveur ACE sur $\mathcal{I}_{\texttt{COP}}$

Pour la campagne COP, nous définissons quelques métriques afin d'interpréter les résultats et mener à bien la comparaison des heuristiques avec les différentes configurations de ACE. Pour cela, nous proposons deux métriques de base, que nous agrégeons ensuite en une troisième et présentons une quatrième métrique utilisée pour les compétitions MiniZinc [2] :

- $\mathcal{S}_{\texttt{domi}}$, pour un solveur donné, correspond au nombre d'instances pour lesquelles la meilleure borne trouvée (à l'échelle d'une instance) est au moins aussi bonne que la meilleure borne trouvée par l'ensemble des solveurs de la campagne ;
- $\mathcal{S}_{\texttt{opti}}$, pour un solveur donné, correspond au nombre d'instances dont l'optimalité a été prouvée ;
- $\mathcal{S}_{\texttt{agg}}$ correspond à la somme de $\mathcal{S}_{\texttt{domi}}$ et de $\mathcal{S}_{\texttt{opti}}$;
- $\mathcal{S}_{\texttt{borda}}$ correspond à une agrégation des résultats basée sur la méthode de vote Borda et prenant en compte la qualité de la borne et le temps d'exécution. De façon plus détaillée, chaque instance est considérée comme un votant donnant un rang à chaque solveur en fonction de sa capacité à résoudre efficacement l'instance donnée (comprenant la qualité de la borne et le temps). Ainsi, le score d'un solveur correspond à la somme du nombre de solveurs qu'il bat pour chaque instance.

Le tableau 2 présente l'ensemble des métriques proposées sur l'ensemble des heuristiques et configurations de solveurs. Les résultats mettent en valeur l'heuristique $\texttt{wdeg}^{\texttt{ca.cd}}$. En ce qui concerne les configurations du solveur, les résultats sont tranchés : les métriques basiques $\mathcal{S}_{\texttt{domi}}$, $\mathcal{S}_{\texttt{opti}}$ et $\mathcal{S}_{\texttt{agg}}$ ont tendance à mettre en valeur la configuration utilisant le *tie-breaker* basé sur le degré des variables, là où $\mathcal{S}_{\texttt{borda}}$ attribue un score plus important à la configuration par défaut de ACE. Si nous essayons de classer l'ensemble des heuristiques et configurations en fonction de $\mathcal{S}_{\texttt{agg}}$ et $\mathcal{S}_{\texttt{borda}}$, ces deux meilleurs solveurs se retrouvent premier et deuxième, puis deuxième et premier : il est donc assez compliqué de trancher entre les deux.

5 Conclusion

Sur la base de l'heuristique originelle wdeg, et des différentes optimisations (et affinements) apportées à celle-ci, nous proposons dans cet article une vision uniforme des heuristiques basées sur la pondération de contraintes ; ceci dans l'optique de mieux appréhender la compréhension des différents mécanismes en jeu pour chacun des variantes. Les expérimentations menées sur ACE (notamment la configuration par défaut) démontrent une certaine robustesse de $\texttt{wdeg}^{\texttt{ca.cd}}$ autant en satisfaction de contraintes qu'en optimisation. Toutefois, nous sommes conscients que l'implantation (et le solveur sous-jacent) peuvent jouer un rôle important. Il ne semble pas toujours facile d'obtenir des résultats totalement concordants sur la base d'autres solveurs. Nous faisons notamment référence à Choco [10], et aux discussions que nous avons menées avec C. Prud'Homme.

Perspective En lien direct avec certains travaux cherchant à combiner plusieurs heuristiques tels que la diversification de l'heuristique $\texttt{dom/wdeg}^{\texttt{chs}}$ [4], ou encore la diversification de familles d'heuristiques [12], la recherche d'une méthode permettant de récupérer les meilleures instances des heuristiques de la présente famille pourrait aussi être intéressante. La diversification des méthodes pour pondérer les contraintes semblent non-seulement intéressantes quant aux capacités du VBS présenté dans cet article, mais aussi à travers le VBS d'une étude annexe montrant que la diversification du *tie-breaker* permet aussi de larges bénéfices.

2. https://www.minizinc.org/challenge2020/rules2020.html

Références

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

[2] F. Boussemart, C. Lecoutre, G. Audemard et C. Piette : XCSP3-core : A format for representing constraint satisfaction/optimization problems. *CoRR*, abs/2009.00514, 2020.

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

[4] M. S. Cherif, D. Habet et C. Terrioux : On the Refinement of Conflict History Search Through Multi-Armed Bandit. *In Proceedings of ICTAI'20*, pages 264–271. IEEE, 2020.

[5] C. Lecoutre, L. Sais, S. Tabary et V. Vidal : Reasonning from last conflict(s) in constraint programming. *Artificial Intelligence*, 173(18):1592–1614, 2009.

[6] C. Lecoutre et N. Szczepanski : PYCSP3 : modeling combinatorial constrained problems in python. *CoRR*, abs/2009.00326, 2020.

[7] J. Liang, Vijay Ganesh, P. Poupart et K. Czarnecki : Exponential recency weighted average branching heuristic for sat solvers. *In Proceedings of AAAI'16*, 2016.

[8] U. Montanari : Network of constraints : Fundamental properties and applications to picture processing. *Information Science*, 7:95–132, 1974.

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

[10] The Choco Team : Choco : an open source Java constraint programming library. *In [11]*, pages 8–14, 2008.

[11] M.R.C. van Dongen, C. Lecoutre et O. Roussel, éditeurs. *Proceedings of the third constraint solver competition.* http ://www.cril.univ-artois.fr/CPAI08/Competition-08.pdf, 2008.

[12] H. Wattez, F. Koriche, C. Lecoutre, A. Paparrizou et S. Tabary : Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts. *In Proceedings of ECAI'20*, 2020.

[13] H. Wattez, C. Lecoutre, A. Paparrizou et S. Tabary : Refining constraint weighting. *In Proceedings of ICTAI'19*, pages 71–77, 2019.

Perturbation des heuristiques de branchement dans la résolution de contraintes*

Hugues Wattez† **Frederic Koriche** **Anastasia Paparrizou**

CRIL, Université d'Artois & CNRS,
UFR des Sciences Jean Perrin, Rue Jean Souvraz SP 18, F-62307 Lens, France
{wattez,koriche,paparrizou}@cril.fr

Résumé

L'heuristique de choix de variables est l'un des mécanismes clés d'un solveur de contraintes. Au cours des deux dernières décennies, des heuristiques efficaces ont été proposées, adaptant l'ordre des variables au fur et à mesure que la recherche progresse. Dans le même temps, des méthodes de redémarrage et de randomisation ont été conçues pour rendre les solveurs plus robustes. Alors que les méthodes de redémarrage sont maintenant bien comprises, choisir *comment* et *quand* randomiser une heuristique donnée reste un problème ouvert. Dans cet article, nous présentons plusieurs stratégies de perturbation conceptuellement simples pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrages. La quantité de perturbation est contrôlée et apprise par des bandits sous diverses politiques d'exploration (stationnaire ou non stationnaire). L'évaluation expérimentale montre une amélioration significative des performances pour les heuristiques perturbées par rapport à leurs homologues d'origine.

1 Introduction

Un solveur de contraintes est généralement composé d'un algorithme de recherche avec retour-arrière avec une heuristique de branchement pour guider la recherche, une procédure de filtrage pour réduire l'espace de recherche et un cache mémorisant les branches infructueuses, les "*nogoods*". L'heuristique de branchement est notamment en charge de l'ordonnancement des variables dans l'arbre de recherche. C'est ce qu'on appelle l'heuristique de choix de variables. La sélection de l'heuristique de choix de variables pour résoudre un réseau de contraintes donné est essentiel car le temps de résolution peut varier de façon exponentielle selon l'heuristique. Les heuristiques proposées ces dernières années ont vu cet effet s'atténuer.

Les algorithmes de recherche sont vulnérables à un tel comportement en raison de la nature combinatoire des problèmes. Au début des années 2000, les différences exponentielles des temps ont été beaucoup étudiées avec la découverte des phénomènes de transition de phase [6] et des distributions à queue longue [4]. Ces différences ont été réduites en introduisant des politiques de redémarrage, la randomisation et l'enregistrement de *nogoods* pendant la recherche. Alors que les redémarrages et les *nogoods* sont bien compris dans les solveurs de contraintes, l'usage de la randomisation reste floue. Les solveurs de contraintes et les solveurs SAT (Satisfiability Testing) utilisent des techniques ad hoc qui se sont avérées empiriquement efficaces en pratique.

Dans cet article, nous revisitons la randomisation pour la résolution de contraintes. Même si la randomisation est déjà connue en résolution de contraintes, par exemple dans les algorithmes incomplets, elle n'est pas autant étudiée dans les solveurs complets. La randomisation la plus utilisée dans les solveurs complets est utilisée en tant que *tie-breaker* lors d'ex-aequo pendant l'utilisation d'une heuristique de choix de variables. Un ex-aequo est un ensemble de variables qui reçoivent le même classement par l'heuristique. Il est habituel de sélectionner une de ces variables aléatoirement [5]. Notre approche étudie plus en profondeur l'application de la randomisation. Notre approche provoque une perturbation du processus de sélection des variables en introduisant des choix aléatoires. Ces perturbations ne sont pas arbitraires mais visent à maintenir un bon équilibre entre exploitation et exploration. Nous présentons plusieurs stratégies de perturbation, conceptuellement simples, pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrages.

*Article publié à CP2020
†Papier doctorant : Hugues Wattez est auteur principal.

La quantité de perturbation est contrôlée et apprise par des bandits lors de redémarrages successifs sous diverses politiques d'exploration stationnaire et non stationnaire. Notre évaluation expérimentale a montré des améliorations significatives des performances des heuristiques perturbées par rapport à leur homologue classique, établissant ainsi la nécessité d'incorporer la perturbation dans les solveurs de contraintes modernes.

2 Contribution

Nous avons introduit des stratégies de perturbation conceptuellement simples pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrages. La plupart de nos stratégies sont adaptatives, c'est-à-dire que la quantité de perturbation est apprise lors des redémarrages successifs. Nous exploitons le mécanisme de redémarrage qui existe dans tous les solveurs modernes pour contrôler l'application de choix aléatoires. Nous avons déployé une technique d'apprentissage par renforcement qui détermine à chaque *run* (c'est-à-dire au début du redémarrage), si elle appliquera l'heuristique standard, intégrée dans le solveur, ou une procédure qui fait des choix de branchement aléatoires. Il s'agit d'un problème de décision séquentielle et en tant que tel, il peut être modélisé comme un problème de bandit multi-armé (MAB) [2], plus précisément, comme un problème à double bras.

Dans l'apprentissage par renforcement, la proportion entre l'exploration et l'exploitation est commandée par diverses politiques. Nous en avons essayé plusieurs, telles que *Epsilon-greedy* [7], *EXP3* [3], *Thompson Sampling* [8], *UCB1* [3] et *MOSS* [1]. L'apprentissage provient du retour d'expérience après chaque exécution. Il reflète l'efficacité de l'exécution pour un choix donné, appelée fonction de récompense. Nous proposons également une stratégie statique qui perturbe une heuristique donnée avec une probabilité fixe trouvée empiriquement. Nous avons évalué les stratégies statiques et adaptatives pour plusieurs heuristiques bien connues, montrant des améliorations significatives des performances en faveur du solveur perturbé, indépendamment de l'heuristique utilisée. Nous avons également mené des expériences permettant l'utilisation de *nogoods*, un composant intégrant les solveurs modernes, montrant que les perturbations dominent toujours le solveur standard, car les *nogoods* obtenus lors des exécutions aléatoires ne désorientent pas la recherche.

De nombreuses observations ont été tirées de cette étude :

— Nos techniques de perturbation améliorent constamment les performances du solveur indépendamment de l'heuristique utilisée comme base dans le solveur. La stratégie perturbée surpasse toujours son homologue de base.
— Un solveur perturbé peut compenser un mauvais choix d'heuristique de l'utilisateur, car il améliore automatiquement ses performances en visitant des parties inconnues de l'espace de recherche. Cela est dû aux exécutions aléatoires, au cours desquelles l'heuristique acquiert des connaissances supplémentaires que l'heuristique seule ne peut pas obtenir.
— Les stratégies adaptatives sont toujours plus efficaces que les stratégies statiques car, grâce à leur fonctions de récompense, elles peuvent ajuster leur comportement à l'heuristique et à l'instance à résoudre.
— La présence de *nogoods* n'affecte pas l'efficacité du solveur perturbé. Les *nogoods* produits restent utiles.
— Les résultats expérimentaux ont montré l'avantage des perturbations pour améliorer les performances des solveurs de contraintes, quel que soit leur réglage par défaut.

Références

[1] J-Y. Audibert et S. Bubeck : Minimax policies for adversarial and stochastic bandits. *In COLT*, pages 217–226, Montreal, Canada, 2009.

[2] P. Auer, N. Cesa-Bianchi et P. Fischer : Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, May 2002.

[3] P. Auer, N. Cesa-Bianchi, Y. Freund et R. Schapire : The nonstochastic multiarmed bandit problem. *SIAM Journal on Computing*, 32(1):48–77, 2002.

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

[5] C. P. Gomes, B. Selman et H. Kautz : Boosting combinatorial search through randomization. *In Proceedings of AAAI '98*, pages 431–437, 1998.

[6] T. Hogg, B. A. Huberman et C. P. Williams : Phase transitions and the search problem. *Artificial Intelligence*, 81(1):1 – 15, 1996.

[7] R.S. Sutton et A.G. Barto : Reinforcement learning : An introduction. 9:1054, 02 1998.

[8] W. R. Thompson : On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 12 1933.

Perturbing Branching Heuristics in Constraint Solving

Anastasia Paparrizou and Hugues Wattez

CRIL, University of Artois & CNRS, France
{paparrizou,wattez}@cril.fr

Abstract. Variable ordering heuristics are one of the key settings for an efficient constraint solver. During the last two decades, a considerable effort has been spent for designing dynamic heuristics that iteratively change the order of variables as search progresses. At the same time, restart and randomization methods have been devised for alleviating heavy-tailed phenomena that typically arise in backtrack search. Despite restart methods are now well-understood, choosing *how* and *when* to randomize a given heuristic remains an open issue in the design of modern solvers. In this paper, we present several conceptually simple perturbation strategies for incorporating random choices in constraint solving with restarts. The amount of perturbation is controlled and learned in a bandit-driven framework under various stationary and non-stationary exploration policies, during successive restarts. Our experimental evaluation shows significant performance improvements for the perturbed heuristics compared to their classic counterpart, establishing the need for incorporating perturbation in modern constraint solvers.

1 Introduction

For decades now, researchers in Constraint Programming (CP) have put a tremendous effort in designing constraint solvers and advancing their internal components. Many mechanisms have been combined, leading to a technology that is now widely used to solve combinatorial problems. A constraint solver is typically composed of a backtracking search algorithm, a branching heuristic for guiding search, a filtering procedure for pruning the search space, and no-good recording.

Since the very beginning, the order in which variables are selected (assigned) by the branching heuristic holds a central place. It is referred to as the variable ordering heuristic. Choosing the appropriate variable ordering heuristic for solving a given constraint satisfaction problem is quite important since the solving time may vary by orders of magnitude from one heuristic to the other. Recent heuristics are more stable [15,36].

Backtrack search is also vulnerable to unstable behavior because of its inherent combinatorial nature. In the early '00s, the exponential time differences have been investigated under the phase-transition phenomena [19] and the heavy-tailed distributions of solving time [12]. We can decrease such undesired differences by introducing restart policies, randomization [6] and no-goods recording

during search. While restarts and no-goods are well established in CP solvers [11,18,24,27], randomization remains limited to ad-hoc techniques that have been found to work well in practice.

In this work, we use randomization to perturb the variable selection process. These perturbations are designed to keep a good and controlled balance between exploitation and exploration. We introduce conceptually simple *perturbation strategies* for incorporating random choices in constraint solving with restarts. Most of strategies that we present are adaptive, meaning that the amount of perturbation is learned during successive restarts. We exploit the restart mechanism that exists in all modern solvers to control the application of random choices. We deploy a reinforcement learning technique that determines at each run (i.e., at the beginning of the restart), if it will apply the standard heuristic, embedded in the constraint solver, or a procedure that makes random branching choices. This is a sequential decision problem and as such, it can be modeled as a multi-armed bandit problem (MAB) [3], precisely, as a double-armed. In reinforcement learning, the proportion between exploration and exploitation is specified by various policies. We tried several of them, such as `Epsilon-greedy` [31], `EXP3` [4], `Thompson Sampling` [32], the upper confidence bound `UCB1` [4] and `MOSS` [2]. The learning comes from the feedback taken after each run that reflects the efficiency of the run under a given choice, referred to as a reward function. We also propose a static strategy that perturbs a given heuristic with a fixed probability, found empirically. We evaluate the static and adaptive strategies for several well known heuristics, showing significant performance improvements in favor of the perturbed solver independently of the underlying heuristic used. A perturbed strategy always outperforms its baseline counterpart both in time and number of solved instances. We have also run experiments allowing the use of no-goods, an integral component of solvers nowadays, showing that perturbations still dominate the standard setting, as the no-goods obtained during random runs do not disorientate search.

Many useful observations are derived from this study. The more inefficient a heuristic is, the more effective the perturbation. A perturbed solver can compensate for a potentially bad heuristic choice done by the user, as it permits to automatically improve its performance by visiting unknown parts of the search space. This is due to the random runs, during which the heuristic acquires extra knowledge, other than what obtained when running alone. We show that adaptive strategies always outperform the static ones, as they can adjust their behavior to the instance to be solved and to the heuristic setting. Overall, the results show the benefits of establishing perturbation in CP solvers for improving their overall performance whatever their default setting is.

2 Related Work

Introducing a touch of randomization for better diversifying the search of local and complete procedures has been shown to be quite effective for both SAT (Satisfiability Testing) and CP (Constraint Programming). A stochastic local

search requires the right setting of the "noise" parameter so as to optimize its performance. This parameter determines the likelihood of escaping from local minima by making non-optimal moves. In GSAT [30], it is referred as the random walk parameter and in walkSAT [29], simply as the "noise". Large Neighborhood Search uses randomization to perform jumps in the search space while freezing a fragment of the best solution obtained so far [26].

In complete CP solvers, the first evidence that diversification can boost search dates back to Harvey and Ginsberg research [17]. Harvey and Ginsberg proposed a deterministic backtracking search algorithm that differs from the heuristic path by a small number of decision points, or "discrepancies". Then, Gomes et al. [13,12] showed that a controlled form of randomization eliminates the phenomenon of "heavy-tailed cost distribution" (i.e., a non-negligible probability that a problem instance requires exponentially more time to be solved than any previously solved instances). Randomization was applied as a tie-breaking step: if several choices are ranked equally, choose among them at random. However, if the heuristic function is powerful, it rarely assigns more than one choice the highest score. Hence, the authors introduced a "heuristic equivalence" parameter in order to expand the choice set for random tie-breaking.

More recently, Grimes and Wallace [14] proposed a way to improve the classical `dom/wdeg` heuristic (based on constraint weighting) by using random probing, namely a pre-processing sampling procedure. The main idea is to generate the weights of the variables with numerous but very short runs (i.e., restarts) prior search, in order to make better branching decisions at the beginning of the search. Once the weights are initialized, a complete search is performed during which weights either remain frozen or continue updating.

3 Preliminaries

A *Constraint Network* P consists in a finite set of variables $\texttt{vars}(P)$, and a finite set of constraints $\texttt{ctrs}(P)$. We use n to denote the number of variables. Each variable x takes values from a finite domain, denoted by $\texttt{dom}(x)$. Each constraint c represents a mathematical relation over a set of variables, called the *scope* of c. 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. A constraint network is *satisfiable* if it admits at least one solution, and the corresponding *Constraint Satisfaction Problem (CSP)* is to determine whether a given constraint network is satisfiable, or not. A classical procedure for solving this NP-complete problem is to perform a backtrack search on the space of partial solutions, and to enforce a property called *generalized arc consistency* [23] on each decision node, called *Maintaining Arc Consistency* (MAC) [28]. The MAC procedure selects the next variable to assign according to a *variable ordering heuristic*, denoted H. Then, the selected variable is assigned to a value according to its value ordering heuristic, which is usually the lexicographic order over $\texttt{dom}(x)$.

As mentioned in Section 2, backtrack search algorithms that rely on deterministic variable ordering heuristics have been shown to exhibit heavy-tailed

behavior on both random and real-world CSP instances [12]. This issue can be alleviated using *randomization* and *restart* strategies, which incorporate some random choices in the search process, and iteratively restart the computation from the beginning, with a different variable ordering [6]. Since our randomization method will be discussed in Section 4, we focus here on restart strategies.

Conceptually, a restart strategy is a mapping $\texttt{res} : \mathbb{N} \rightarrow \mathbb{N}$, where $\texttt{res}(t)$ is the maximal number of "steps" which can be performed by the backtracking search algorithm at run, or *trial*, t. A constraint solver, equipped with the MAC procedure and a restart strategy `res`, 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 at run t. After each run, the solver can memorize some relevant information about the sequence $\langle \mathcal{T}^{(1)}, \mathcal{T}^{(2)}, \ldots, \mathcal{T}^{(t-1)} \rangle$, like the number of constraint checks in the previous runs, the no-goods that have appeared frequently in the search trees explored so far [20]. The *cutoff*, $\texttt{res}(t)$, which is the number of allowed steps, may be defined by the number of nodes, the number of wrong decisions [8], the number of seconds, or any other relevant measure. In a *fixed* cutoff restart strategy, the number T of trials is fixed in advance, and $\texttt{res}(t)$ is constant for each trial t, excepted for the Tth trial which allows an unlimited number of steps (in order to maintain a complete algorithm). This strategy is known to be effective in practice [13], but a good cutoff value $\texttt{res}(t)$ has to be found by trial and error. Alternatively, in a *dynamic* cutoff restart strategy, the number T of trials is unknown, but `res` increases geometrically, which guarantees that the whole space of partial solutions is explored after $O(n)$ runs [33]. A commonly used cutoff strategy is driven by the Luby sequence [22].

4 Perturbation strategies

As indicated in Section 3, the process of constraint solving with a restart policy may be viewed as a sequence $\langle 1, 2, \cdots, T \rangle$ of *runs*. For the aforementioned restart functions, the sequence of runs is finite, but the horizon T is not necessarily known in advance. During each run t, the solver calls the MAC algorithm for building a search tree $\mathcal{T}_t$, whose size is determined by the cutoff of the restart policy. If the solver has only access to a single variable ordering heuristic, say H, it will run MAC with H after each restart. Yet, if the solver is also allowed to *randomize* its variable orderings, it is faced with a fundamental choice at each run t: either call MAC with the heuristic H in order to "exploit" previous computations made with this heuristic, or call MAC with a random variable ordering U so as to "explore" new search trees, and potentially better variable orderings. Here, U is any variable ordering drawn at random according to a uniform distribution over the permutation group of $\texttt{vars}(P)$. We need to highlight here, that the intermediate random runs of U perturb the involved classic heuristic H by updating its parameters, which ultimately affects the behavior/performance of H. In other words, the subsequent heuristic runs, will not produce the same orderings as in the traditional solving process, allowing thus the solver to (potentially) tackle instances that neither H nor U would solve stand-alone.

Algorithm 1: Bandit-Driven Perturbations

Input: constraint network P, heuristic H, policy B

1 initArms$_B(H, U)$ // *Initialize the arms and the bandit policy*
2 **for** *each run* $t = 1, \cdots, T$ **do**
3 $a_t \leftarrow$ selectArm$_B()$ // *Select an arm according to the bandit policy*
4 $r_t(a_t) \leftarrow \text{MAC}(P, a_t)$ // *Execute the solver and compute the reward*
5 updateArms$_B\big(r_t(a_t)\big)$ // *Update the bandit policy*

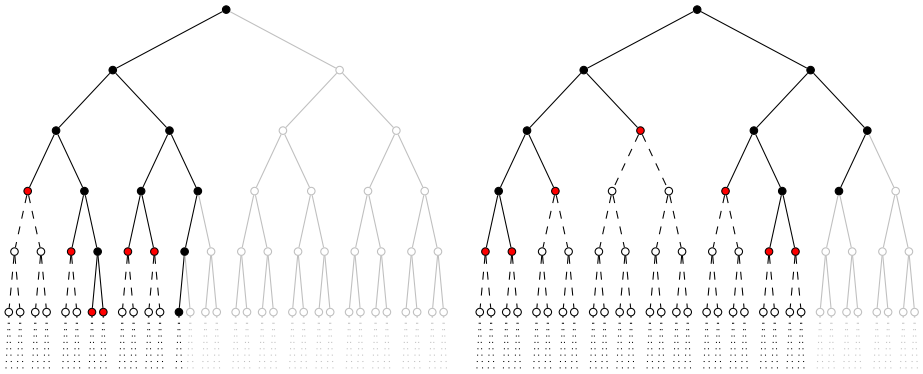
The task of incorporating perturbations into constraint solving with restarts can be viewed as a *double-armed* bandit problem: during each run t, we have to decide whether the MAC algorithm should be called using H (exploitation arm) or U (exploration arm). Once MAC has built a search tree $\mathcal{T}_t$, the performance of the chosen arm can be assessed using a reward function defined according to $\mathcal{T}_t$. The overall goal is to find a policy mapping each run t to a probability distribution over $\{H, U\}$ so as to maximize cumulative rewards.

Multi-armed bandit algorithms have recently been exploited in CP in different contexts, i.e. for guiding search [21], for learning the right level of propagation [5] or the right variable ordering heuristic [34,35,37]. In the framework of Xia and Yap [37], a single search tree is explored (i.e., no restarts), and the bandit algorithm is called at each node of the tree to decide which heuristic to select. The trial is associated with explored subtrees, while in our approach, trials are mapped to runs using a restart mechanism. Our framework makes use of restarts in the same way as the ones of [34,35], as it was shown in [35] that such a framework offers greater improvements compared to the one of [37]. In our case, we utilise a double-armed framework in order to construct our bandit-driven perturbation given by Algorithm 1. The algorithm, takes as input a constraint network P, a variable ordering heuristic H, and a bandit policy B. As indicated above, the bandit policy has access to two arms, H and U, where U is the random variable ordering generated on the fly, during execution. The three main procedures used by the bandit policy are initArms$_B$ for initializing the distribution over $\{H, U\}$ according to policy B, selectArm$_B$ for choosing the arm $a_t \in \{H, U\}$ that will be used to guide the search all along the tth run, and updateArms$_B$ for updating the distribution over $\{H, U\}$ according to the observed reward $r_t(a_t)$ at the end of the tth run.

4.1 Reward function

The feedback $r_t(a_t)$ supplied at the end of each run captures the performance of the MAC algorithm, when it is called using $a_t \in \{H, U\}$. To this end, the reward function r_t maps the search tree $\mathcal{T}_t$ built by MAC(a_t) to a numeric value in $[0, 1]$ that reflects the quality of backtracking search when guided by a_t.

As a reward function, we introduce the measure of the *explored sub-tree* denoted as `esb`. `esb` is given by the number of visited nodes during a run, divided



(a) $\texttt{esb}(\mathcal{T}_{\text{left}}) = \frac{\log(16)}{\log(2^6)} \approx 0.67$ (b) $\texttt{esb}(\mathcal{T}_{\text{right}}) = \frac{\log(16)}{\log(2^5)} \approx 0.80$

Fig. 1: Comparison of two runs with a restart cutoff fixed to 16 nodes.

by the size of the complete sub-tree defined over the variables selected during this run. The later is simply the size of the Cartesian product of the domains of the variables selected during the run. As selected variables, we consider those ones that have been chosen at least once by the arm in question. `esb` represents the search space covered by the solver, under a certain setting of a run, compared to the total possible space on the selected variables. The intuition is that an exploration that discovers failures deeply in the tree (meaning that, many variables are instantiated) will be penalized (due to the big denominator) against an exploration that discovers failures at the top branches.

In formal terms, given a search tree $\mathcal{T}$ generated by the MAC algorithm, let $\texttt{vars}(\mathcal{T})$ be the set of variables that have been selected at least once during exploration of $\mathcal{T}$ and $\texttt{nodes}(\mathcal{T})$ the number of visited nodes. Then,

$$r_t(a_t) = \texttt{esb}(\mathcal{T}_t) = \frac{\log(\texttt{nodes}(\mathcal{T}_t))}{\log\left(\prod_{x \in \texttt{vars}(\mathcal{T}_t)} |\texttt{dom}(x)|\right)}$$

A logarithmic scaling is needed to obtain a better discrimination between the arms as the numerator is usually significantly smaller than the denominator of the fraction. The reward values belong to [0, 1]. The higher the ratio/reward is, the better the performance of a_t is.

Figure 1, shows a motivating example for the reward function. It displays an example of tree explorations done by two different runs. For simplicity, domains are binary and at each level the variable to be instantiated is fixed per run (namely left and right branches are on the same variable). Empty nodes represent non-visited or pruned nodes, solid black nodes are the visited ones, solid red ones denote failures. Below red nodes, there are the pruned sub-trees in dashed style while the non-visited sub-trees are slightly transparent. For both runs we consider the same number of node visits, i.e. 16. At the left run (Figure 1a), the solver goes until level 6 (selecting 6 variables) while the solver at the right

run (Figure 1b) goes until level 5. The left run will take the score of 0.67 and the right one 0.80. Our bandit will prefer the arm that produced the right tree, namely a search that goes faster at the right branches than deeper in the tree (which implies that more search is required). Early failures is a desired effect that many heuristics consider explicitly on implicitly in order to explore smaller search spaces until the solution or unsatisfiability.

4.2 Perturbation rate

As indicated in Algorithm 1, the perturbation framework is conceptually simple: based on a restart mechanism, the solver performs each run by first selecting an arm in $\{H, U\}$, next, observing a reward for that arm, and then, updating its bandit policy according to the observed reward. This simple framework, allows us to use a variety of computationally efficient bandit policies to adapt/control the amount of perturbation applied during search. From this perspective, we have opted for five well-studied "adaptive" policies for the double-armed bandit problem, and one "static" (or stationary) policy which serves as reference for our perturbation methods.

ϵ-Greedy. Arguably, this is the simplest adaptive bandit policy that interleaves exploitation and exploration using a parameter ϵ. The policy maintains the empirical means $\hat{r}$ of observed rewards for H and U. Initially, both $\hat{r}_1(H)$ and $\hat{r}_1(U)$ are set to 0. On each run t, the function $\textsc{selectArm}_{\epsilon G}$ returns with probability $(1-\epsilon)$ the arm a_t that maximizes $\hat{r}_t(a_t)$, and returns with probability ϵ any arm a_t drawn uniformly at random. Finally, based on the observed reward $r_t(a_t)$, the procedure $\textsc{updateArms}_{\epsilon G}$ updates the empirical mean of a_t according to

$$\hat{r}_{t+1}(a_t) = \frac{t}{t+1}\hat{r}_t(a_t) + \frac{1}{t+1}r_t(a_t)$$

EXP3. The EXPonentially weighted forecaster for EXPloration and EXPloitation (EXP3) is the baseline bandit policy operating in "non-stochastic" environments, for which no statistical assumption is made about the reward functions [4]. Several variants of `EXP3` have been proposed in the literature, but we use here the simplest version defined in [10]. Here, the procedure $\textsc{initArms}_{\texttt{EXP3}}$ sets the initial distribution π_1 of arms to the uniform distribution $(1/2, 1/2)$. During each trial t, the procedure $\textsc{selectArm}_{\texttt{EXP3}}$ simply draws an arm a_t according to the distribution π_t. Based on the observed reward $r_t(a_t)$, the procedure $\textsc{updateArms}_{\texttt{EXP3}}$ updates the distribution π_t according to the multiplicative weight-update rule:

$$\pi_{t+1}(a) = \frac{\exp(\eta_t R_t(a))}{\exp(\eta_t R_t(H)) + \exp(\eta_t R_t(U))}$$

η_t corresponds to the learning rate (usually set to $\frac{1}{\sqrt{t}}$),

$$R_t(a) = \sum_{s=1}^{t} \frac{r_s(a)}{\pi_s(a)} \mathbb{1}_{a \sim \pi_s}$$

and $\mathbb{1}_{a \sim \pi_s}$ indicates whether a was the arm picked at trial s, or not.

UCB1. Upper Confidence Bound (UCB) policies are commonly used in "stochastic" environments, where it is assumed that the reward value $r_t(a)$ of each arm a is drawn according to a fixed, but unknown, probability distribution. `UCB1` is the simplest policy in the Upper Confidence Bound family [3]. In the setting of our framework, this algorithm maintains two 2-dimensional vectors, namely, $n_t(a)$ is the number of times the policy has selected arm a on the first t runs, and $\hat{r}_t(a)$ is the empirical mean of $r_t(a)$ during the $n_t(a)$ steps. initArms$_{\texttt{UCB1}}$ sets both vectors to zero and, at each run t, selectArm$_{\texttt{UCB1}}$ selects the arm a_t that maximizes

$$\hat{r}_t(a) + \sqrt{\frac{2\ln(t)}{n_t(a)}}$$

Finally, updateArms$_{\texttt{UCB1}}$ updates the vectors n_t and $\hat{r}_t$ according to a_t and $r_t(a_t)$, respectively.

MOSS. The Minimax Optimal Strategy in the Stochastic case (MOSS) algorithm is an instance of the UCB family. The only difference with `UCB1` lies in the confidence level which not only takes into account the number of plays of individual arms, but also the number of arms (2) and the number of runs (t). Specifically, selectArm$_{\texttt{MOSS}}$ chooses the arm a_t that maximizes

$$\hat{r}_t(a) + \sqrt{\frac{4}{n_t(a)} \ln^+\left(\frac{t}{2n_t(a)}\right)}$$

where $\ln^+(x) = \ln \max\{1, x\}$.

TS. The Thompson Sampling algorithm is another well-known policy used in stochastic environments [1,32]. In essence, the TS algorithm maintains a beta distribution for the rewards of each arm. initArms$_{\texttt{TS}}$ sets $\alpha_1(a)$ and $\beta_1(a)$ to 1 for $a \in \{H, U\}$. On each run t, selectArm$_{\texttt{TS}}$ selects the arm a_t that maximizes $\text{Beta}(\alpha_t(a), \beta_t(a))$, and updateArms$_{\texttt{TS}}$ uses $r_t(a_t)$ to update the beta distribution as follows:

$$\alpha_{t+1}(a) = \alpha_t(a) + \mathbb{1}_{a=a_t} r_t(a_t)$$
$$\beta_{t+1}(a) = \beta_t(a) + \mathbb{1}_{a=a_t}(1 - r_t(a_t))$$

SP. Finally, in addition to the aforementioned adaptive bandit policies which learn a distribution on $\{H, U\}$ according to observed rewards, we shall consider the following Static Policy (SP): on each round t, selectArm$_{\texttt{SP}}$ chooses H with probability $(1 - \epsilon)$, and U with probability ϵ. Although this policy shares some similarities with the ϵ-greedy algorithm, there is one important difference: the distribution over $\{H, U\}$ is fixed in advance, and hence, SP does not take into account the empirical means of observed rewards. In other words, updateArms$_{\texttt{SP}}$ is a dummy procedure that always returns $(1 - \epsilon, \epsilon)$. This stationary policy will serve as reference for the adaptive policies in the experiments.

5 Experimental Evaluation

We have conducted experiments on a large dataset to demonstrate the performance of the proposed perturbations. The set includes all instances (612 in total) from the 2017's XCSP3 competition[1] coming from 60 different problem classes. The experiments have been launched on an 2.66 GHz Intel Xeon and 32 GB RAM nodes. We have used the `AbsCon`[2] solver in which we integrated our perturbation strategies and the strategies of [13] and [14]. We used 2-way branching, generalized arc consistency as the level of consistency, Luby progression based on node visits as restart policy (the constant is fixed to 100 in `AbsCon`) and the timeout set to 1,200 seconds. We have chosen a big variety of variable ordering heuristics, including recent, efficient and state-of-the-art ones: `dom` [16], `dom/ddeg` [7], `activity` [25], `dom/wdeg` [9], `CHS` [15], $\texttt{wdeg}^{\texttt{ca.cd}}$ [36] and finally `rand` which chooses uniformly randomly a variable order. Among these, `dom` and `dom/ddeg` do not record/learn anything between two runs (`dom` and `ddeg` are re-initialized at the root), while all the others learn during each (random) run and maintain this knowledge all along the solving process, which might change/improve their behavior. We have run all these original heuristics separately for a baseline comparison. Note that in our first experiments no-goods recording are switched off in the solver to avoid biasing the results of heuristics and strategies.

Regarding our perturbation strategies, we denote by `SP` the static perturbation and by `e-greedy`, `UCB1`, `MOSS`, `TS` and `EXP3` the various adaptive perturbation (`AP`) strategies. Epsilon of `SP` and `e-greedy` are fixed to 0.1. This value has been fixed offline after a linear search of the best value. Apart from comparing to the default solver settings (i.e., `original` heuristics), we compare to three other perturbation strategies from the bibliography. The one is the `sampling` algorithm of [14] that corresponds to the sampling pre-processing step which is fixed to 40 restarts with a cutoff of n nodes corresponding to the number of variables of each instance. When the probing phase finishes, we continue updating the variable scoring as it produces better results . The second is the `equiv-30` that corresponds to the criterion of equivalence of [13]. This equivalence parameter is set to 30% as authors proposed. Last, we compare to the standard tie-breaking denoted `equiv-0`, where a random choice is done among the top ranked variables scored equally by the underlying heuristic. Note that there are no ties, `equiv-0` has no effect on the heuristic, as opposed to `equiv-30`.

Table 1 displays the results of the aforementioned settings and strategies on the XCSP'17 competition dataset. The comparison is given on the number of solved instances (*#inst*), within 1,200 seconds, the cumulative CPU time (*time*) computed from instances solved by at least one method and the percentage of perturbation (*%perturbation*) which is the mean perturbation of the solved instances, computed by the number of runs with the arm U divided by the total of runs. Each time a setting has not solved an instance that another setting solved, it is penalized by the timeout time. Numbers in bold indicate that a strategy

[1] See http://www.cril.univ-artois.fr/XCSP17
[2] See http://www.cril.fr/~lecoutre/#/softwares

Table 1: Comparison of `original`, `sampling`, `equiv-30` and the proposed perturbed strategies for the XCSP'17 dataset.

		original	sampling	equiv-0	equiv-30	SP	AP e-greedy	AP UCB1	AP MOSS	AP TS	AP EXP3
dom	#inst 287	**321**	**312**	**308**	**315**	**314**	**323**	**322**	**318**	**323**
	time (359) 101,589	**58,496**	**74,064**	**75,992**	**72,460**	**75,474**	**59,527**	**61,171**	**63,856**	**61,842**
	%perturb. 0	-	-	-	10	7.7	32.8	21.0	24.9	44.1
dom/ddeg	#inst 307	**321**	**319**	**324**	**343**	**337**	**345**	**346**	**342**	**343**
	time (365) 85,131	**61,071**	**67,537**	**66,005**	**46,179**	**51,371**	**41,404**	**43,280**	**44,981**	**40,260**
	%perturb. 0	-	-	-	10	8.1	34.3	21.4	26.5	45.3
activity	#inst 342	311	334	329	**356**	**356**	**352**	**353**	**350**	**351**
	time (372) 52,989	82,041	60,304	64,619	**36,688**	**36,125**	**39,552**	**37,463**	**40,306**	**39,371**
	%perturb. 0	-	-	-	10	7.6	33.5	20.7	26.0	44.4
dom/wdeg	#inst 347	342	**349**	**349**	**358**	**346**	**358**	**354**	**356**	**363**
	time (381) 55,599	56,038	**54,276**	**53,263**	**45,615**	58,888	**43,726**	**49,052**	**46,896**	**39,277**
	%perturb. 0	-	-	-	10	11.6	34.3	23.9	28.6	45.2
$\text{wdeg}^{\text{ca.cd}}$	#inst 366	354	**368**	361	**370**	**368**	**371**	**372**	**367**	**369**
	time (389) 42,565	52,344	43,944	49,717	**38,966**	**39,292**	**41,745**	**40,433**	44,941	42,661
	%perturb. 0	-	-	-	10	7.8	32.1	19.9	24.8	42.9
CHS	#inst 370	343	**371**	361	**371**	**372**	367	**373**	367	367
	time (389) 41,462	64,779	42,025	56,639	**37,699**	**38,158**	43,869	**38,190**	46,597	42,444
	%perturb. 0	-	-	-	10	7.3	32.6	19.9	25.7	44.5
rand	#inst 291									
	time (291) 12,921									
	%perturb. 100									

outperformed her corresponding default setting of the solver (i.e., `original`). Underlined numbers show the winning strategy. `dom` and `dom/ddeg`, the unaffected heuristics, appear at the top of the table as they cannot be perturbed by randomized runs. After each run their parameters are reinitialized and not accumulated as for the rest of the heuristics (perturbed ones). Hence, for `dom` and `dom/ddeg`, any additional instance that perturbation strategies are able to solve comes from an intervening run of U.

The existing perturbation techniques, `sampling`, `equiv-30` and `equiv-0`, solve more instances than their respective baseline heuristic for the case of `dom` and `dom/ddeg`. However, this never happens for `sampling` and `equiv-30` on the more sophisticated heuristics (except from `dom/wdeg`), where we see that the perturbation they apply disorientates totally the search, solving constantly less instances than the original heuristics (e.g., `sampling` missed 31 instances for `activity`). `equiv-0` just marginally outperforms pure heuristics by one or two instances while it is far inferior to `AP`s (e.g., it missed 22 instances compared to `e-greedy` on `activity`). `equiv-30` is superior to `sampling` on the perturbed heuristics but still far inferior to all proposed strategies. Among the proposed perturbation strategies, we observe that both `SP` and `AP` strategies constantly outperform the default setting of the solver, while for many heuristics they have close performance (e.g., `activity`). `MOSS` is the best strategy in terms of solved instances and time results, except for `activity` and `dom/wdeg` heuristics, where

SP (e-greedy too) and EXP3 dominate respectively. UCB1 and MOSS are the best strategies for dom and dom/ddeg, with only one instance of difference, showing that a perturbation rate between 20% and 30% is the best choice. On the other side, TS with a similar rate seems not to select that well the right arm for all runs. Similarly for equiv-30, which also applies a randomization of 30% on the top ranked variables, it seems that the way it is applied (i.e., at every decision) is not that efficient. SP and e-greedy, despite winning their original counterparts are less competitive due to their low perturbation rate. EXP3 is also a good candidate policy for dom and dom/ddeg.

rand solved the less instances in total, i.e. 291, in 12,921 seconds, which means that many of them correspond to quite easy instances. As rand is the "bad" arm, a small participation of 10% in SP is just enough to be beneficial for the heuristics that are by themselves efficient (e.g., $\text{wdeg}^{\text{ca.cd}}$, CHS), but as SP is not adaptive, it is rarely better than the AP strategies; it cannot adjust its behavior to heuristics that require more perturbation to improve (e.g., dom). AP strategies, being adaptive, allow usually much more exploration of the U arm that makes them win several instances. An exception is the CHS heuristic, where a perturbation over 20% might be harmful (i.e., in UCB1, TS and EXP3 policies). Each bandit policy follows a general trend that can vary between heuristics (e.g., UCB1 is around 30%, MOSS around 20% and e-greedy with EPX3 represent the two extremities). EXP3 sets the initial distribution π_1 of arms to $(1/2, 1/2)$ such that both arms have equal chances at the beginning. As it is a non-stochastic bandit it needs more exploration than stochastic ones need to converge. Although one would expect that this could deteriorate the solver, it seems that the bandit utilizes the right arm at each run since it is efficient both in time and solved instances. The high perturbation rates come from the mean, that smoothens the high variance between instances. Also, many of the instances are solved fast and EXP3 favorizes a lot U, being the best arm at the early (short) runs, while the long runs at the end are done by H, which explains its good overall performance. TS despite it is better than original heuristics and existing perturbation methods, it is usually inferior to SP by small differences.

We distinguish the MOSS policy, which apart from being the best policy for many heuristics, it never deteriorates the solver for any heuristic (as happens for some strategies on CHS). It applies the exploration when and where needed (more at the early runs) and converges faster to the best heuristic. Regarding the time performance, all perturbation strategies are faster than their baseline heuristic, even for the most efficient heuristics (i.e., CHS and $\text{wdeg}^{\text{ca.cd}}$).

Figure 2, visualizes in a cactus plot the performance of the best AP strategy, namely MOSS, for all heuristics compared to their corresponding original heuristic. On x-axis we see the instances solved as time progresses. y-axis displays the allowed time given to the solver. Dashed lines display the performance of original heuristics and solid lines the perturbed solver. The closer a line is to the right bottom corner the more instances has solved in less time. In general, MOSS policy perturbations appear always at the right side of their respective default heuristic. We observe that for the less efficient heuristics, as dom

and `dom/ddeg`, the performance gap between the `original` and the respective perturbed version is big even for easy instance and increases significantly as time passes, corresponding to more difficult instances. It worths noticing that, `dom/ddeg` after being perturbed, becomes better than `activity` (solved 4 more instances), that originally was much more efficient than `dom/ddeg`. $\texttt{dom}_{\texttt{MOSS}}$ solved in total 35 (resp. 39) more instances than `dom` (resp. `dom/ddeg`). For `activity` and `dom/wdeg`, the curves are closer, though the gap is still significant especially for harder instances. Even for the most efficient heuristics proposed the last two years, namely `CHS` and $\texttt{wdeg}^{\texttt{ca.cd}}$, `MOSS` is almost all the time the best setting for the solver.

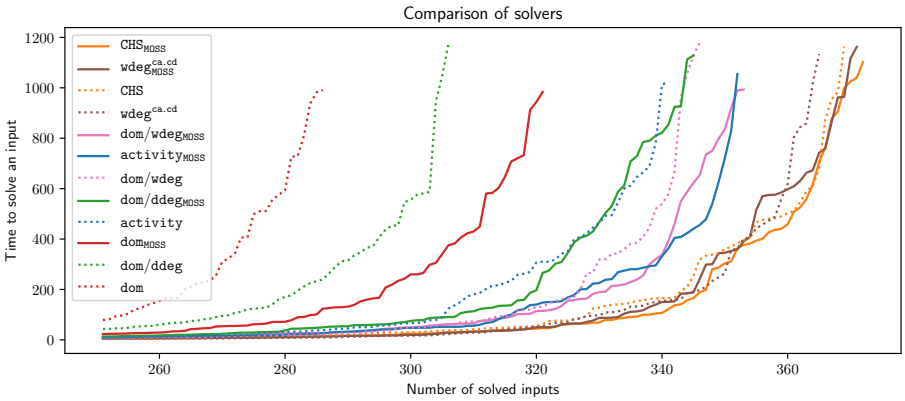


Fig. 2: Comparison of `original` and `MOSS` strategies while increasing the allowed time.

In the following, we do not present results for all adaptive strategies, but only for the most stable and efficient ones. We omit presenting results for `EXP3` because it failed for `CHS`, which is one of the two best heuristics in CP, and for `TS`, because it is never the winning strategy.

Table 2 gives complementary information derived from Table 1 for the best proposed strategies. We have calculated the number of instances solved exclusively by `rand` (i.e., instances that the heuristic alone could not solve), denoted by *#random*, and the number of instances, *#perturb*, that the solver solved due to the perturbation `rand` caused to the corresponding heuristic. Instances in *#perturb* are those that are solved neither by the `original` heuristic nor the `rand` heuristic, making thus the perturbed solver outperform even its corresponding virtual best solver. As expected, `dom` and `dom/ddeg` do not win instances due to perturbation, but only due to a good ordering during a random run (more than 30 instances for each policy). For other heuristics, we see a more balanced distribution of instances in *#random* and *#perturb*. We observe that the most robust heuristics (`CHS` and $\texttt{wdeg}^{\texttt{ca.cd}}$) solve extra instances mainly by perturbation (for

Table 2: Won instances by perturbation or random runs for `SP`, `e-greedy`, `UCB1` and `MOSS` with nodes as cutoff for the XCSP'17 dataset

		dom	dom/ddeg	activity	dom/wdeg	wdeg$^{ca.cd}$	CHS
SP	#perturb.	0	0	8	8	5	4
	#random	28	36	12	10	3	3
e-greedy	#perturb.	0	0	8	5	5	6
	#random	29	31	10	6	3	2
UCB1	#perturb.	0	0	5	7	8	7
	#random	38	41	13	10	3	2
MOSS	#perturb.	0	0	6	7	7	10
	#random	37	41	12	8	3	3

`MOSS`: 10 and 7 respectively) rather than by calling `rand` (3 and 3). Indeed, both are so efficient that the majority of solved instances by `rand` are also solved by them, which explains why their perturbations 'gain' fewer instances than the other heuristics do. In contrast, `activity` and `dom/wdeg`, that are less efficient than `CHS` and wdeg$^{ca.cd}$, gain more instances in total, most of which are gained by `rand`.

Table 3: Comparison of `original`, `SP`, `e-greedy` and `MOSS` strategies on a subset of families from the XCSP'17 dataset.

		original	SP	e-greedy	MOSS
CoveringArray	dom/ddeg	2 (4,803s, 0%)	**3 (4,533s, 10%)**	2 (4,803s, 3%)	**4 (2,952s, 17%)**
	dom/wdeg	4 (2,669s, 0%)	4 (3,450s, 10%)	3 (3,605s, 31%)	**5 (2,156s, 34%)**
	CHS	4 (2,483s, 0%)	**4 (2,424s, 10%)**	**5 (1,705s, 3%)**	**6 (1,491s, 17%)**
SuperSolutions	dom	2 (8,404s, 0%)	**4 (6,972s, 10%)**	**6 (7,471s, 6%)**	**6 (3,912s, 24%)**
	wdeg$^{ca.cd}$	5 (4,968s, 0%)	**7 (3,101s, 10%)**	**6 (4,467s, 9%)**	**7 (3,821s, 23%)**
	CHS	7 (3,507s, 0%)	6 (3,788s, 10%)	6 (3,812s, 6%)	**8 (1,689s, 20%)**
KnightTour	dom/ddeg	5 (7,220s, 0%)	**9 (3,276s, 10%)**	**9 (2,623s, 34%)**	**9 (3,092s, 37%)**
	activity	7 (4,891s, 0%)	**9 (3,530s, 10%)**	**9 (2,823s, 26%)**	**9 (2,513s, 30%)**
	dom/wdeg	5 (7,218s, 0%)	**9 (4,302s, 10%)**	**7 (5,509s, 25%)**	**9 (3,286s, 39%)**
Blackhole	dom	6 (2,609s, 0%)	**6 (2,489s, 10%)**	**6 (2,503s, 4%)**	**6 (2,423s, 16%)**
	dom/ddeg	8 (327s, 0%)	**8 (44s, 10%)**	**8 (55s, 4%)**	**8 (25s, 20%)**
	activity	8 (897s, 0%)	7 (1,618s, 10%)	8 (908s, 6%)	**8 (816s, 17%)**
LatinSquare	dom	8 (5,841s, 0%)	**8 (5,840s, 10%)**	8 (5,928s, 2%)	**8 (5,813s, 13%)**
	dom/wdeg	10 (2,481s, 0%)	**12 (1,143s, 10%)**	10 (3,262s, 44%)	10 (3,351s, 40%)
	CHS	11 (3,031s, 0%)	9 (3,947s, 10%)	9 (4,469s, 3%)	8 (5,866s, 13%)

As the results in Table 1 are quite condensed, in Table 3 we show how strategies operate and adjust for certain problem classes. For each class and each heuristic, we present the number of solved instances, the total time and the perturbation rate. For *CoveringArray*, `MOSS` is the best strategy to apply the ap-

Table 4: Comparison of `original`, `sampling`, `equiv-30`, `SP`, `e-greedy` and `MOSS` with nodes as cutoff and no-goods activated.

		original	sampling	equiv-0	equiv-30	SP	AP e-greedy	AP UCB1	AP MOSS
dom	#inst	309	**321**	306	**316**	**338**	**334**	**342**	**340**
	time (359)	73,367	**57,279**	74,364	**68,446**	**43,327**	**51,168**	**42,219**	**44,027**
	%perturb.	0	-	-	-	10	8.6	33.9	21.4
dom/ddeg	#inst	316	**322**	**320**	**321**	**347**	**346**	**352**	**347**
	time (367)	71,191	**62,375**	**68,703**	**72,550**	**40,042**	**41,530**	**30,770**	**38,995**
	%perturb.	0	-	-	-	10	8.8	34.9	22.5
activity	#inst	352	319	349	346	**356**	**357**	**356**	**355**
	time (373)	40,601	74,680	45,590	46,245	**32,328**	**33,161**	**34,228**	**35,854**
	%perturb.	0	-	-	-	10	7.8	34.1	20.7
dom/wdeg	#inst	352	340	349	344	**359**	**356**	**364**	**364**
	c.time (377)	41,882	53,238	48,073	53,841	**34,253**	**40,100**	**29,559**	**30,995**
	%perturb.	0	-	-	-	10	11.9	34.7	23.9
wdeg$^{ca.cd}$	#inst	373	356	373	361	373	**375**	371	**377**
	time (388)	33,887	50,009	33,053	53,587	**33,412**	**32,359**	33,772	**28,614**
	%perturb.	0	-	-	-	10	7.8	32.8	19.8
CHS	#inst	375	348	372	366	373	373	**375**	**376**
	time (387)	30,990	58,121	36,829	47,159	**30,658**	31,547	31,020	**30,585**
	%perturb.	0	-	-	-	10	7.1	33.0	19.8
rand	#inst	293							
	time (293)	16,992							
	%perturb.	100							

propriate perturbation rate, independently of the heuristic chosen, compared to `SP` and `e-greedy` whose rate is too low. For *SuperSolutions* with `dom` as heuristic method, we see that `e-greedy` and `MOSS` are both winners despite their totally different rates. Though, it is notable that `MOSS` is twice faster than `e-greedy` on the same instances. `SP` with a rate close to `e-greedy` wins the half instances compared to it (and `MOSS`). Such observations are clear evidences, that not only the amount of randomization counts but also the when it appears. Policies as `MOSS` learn to discriminate on which run to apply H or U. Recall that, compared to other policies, `MOSS` considers in SELECTARM$_{\texttt{MOSS}}$ more parameters, as the number or arms and the number of runs. In *KnightTour*, all strategies are efficient, but `AP`s are always better than `SP` in terms of time. For *Blackhole* and *LatinSquare*, perturbation is not fruitful and thus, both `AP`s converge to very low rates even when instances are easy (just few seconds per instance). Notice that, in general `dom/wdeg` is helped a lot by perturbation, as in all classes rates are higher compared to other heuristics (double percentages). Also, the percentage of perturbation varies a lot depending on the heuristic and the problem class, which is the reason of the success of `AP`s.

As modern solvers exploit no-goods to improve their overall performance, we repeated our experiments by activating no-goods in order to examine the robustness of the proposed strategies and the interaction between no-goods and perturbation. Table 4 displays the results `sampling`, `equiv-0`, `equiv-30`, `SP` and

the best AP strategies, namely e-greedy, UCB1 and MOSS. As seen in Table 1, sampling and equiv-30 make some improvements on the less efficient heuristics as dom and dom/ddeg, but are still inferior to SP and MOSS, while they are inefficient on all other heuristics. Surprisingly, equiv-0, despite being still more efficient than sampling and equiv-30, seems to interact badly with the presence of no-goods, as it can no longer improve the solver for any heuristic (just marginally dom/ddeg). SP, despite being static, it remains efficient apart from the case of CHS. Regarding the AP strategies, e-greedy (resp. UCB1) wins almost always the underlying heuristic except from the case of CHS (resp. $\texttt{wdeg}^{\texttt{ca.cd}}$). MOSS is again the most stable strategy, being able to improve all heuristics it perturbed. Note that the presence of no-goods has improved the performance of both the default and the perturbed solver. Therefore, there are slightly smaller differences between them compared to Table 1. The proposed perturbation strategies are robust to this fundamental parameter for solvers compared to existing strategies and adapt their behavior, especially MOSS.

6 Conclusion

We presented several strategies that significantly improve the performance and robustness of the solver by perturbing the default branching heuristic. It is the first time an approach tries to learn how and when to apply randomization in an on-line and parameter-free fashion. Controlled random search runs help variable ordering heuristics to acquire extra knowledge from parts of the search space that they were not dedicated to explore. We summarize the benefits of our approach which are manifold:

- Our perturbation techniques constantly improve the performance of the solver independently of the heuristic used as baseline in the solver. A perturbed strategy always outperforms its baseline counterpart both in time and solved instances.
- The presence of no-goods does not impact the efficacy of the perturbed solver. The produced no-goods are still fruitful.
- Perturbed heuristics can compensate for a wrong heuristic choice done by the user. Thanks to perturbation, the performance of the solver with a bad initial heuristic can reach or even outperform the performance of the solver with a better baseline heuristic. This is a step towards autonomous and adaptive solving, where the solver learns and adjusts its behavior to the instance being solved.
- Our approach is generic and easy to embed in any solver that exploits restarts.

Acknowledgments. The authors would like to thank Frederic Koriche for his valuable advices on the Machine Learning aspects of the paper as well as the anonymous reviewers for their constructive remarks. This work has been partially supported by the project *Emergence* 2020 BAUTOM of INS2I and the project CPER Data from the region "Hauts-de-France".

References

1. Agrawal, S., Goyal, N.: Near-optimal regret bounds for Thompson Sampling. J. ACM **64**(5), 30:1–30:24 (2017)
2. Audibert, J.Y., Bubeck, S.: Minimax policies for adversarial and stochastic bandits. In: COLT. pp. 217–226. Montreal, Canada (2009)
3. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning **47**(2), 235–256 (May 2002)
4. Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.: The nonstochastic multiarmed bandit problem. SIAM Journal on Computing **32**(1), 48–77 (2002)
5. Balafrej, A., Bessiere, C., Paparrizou, A.: Multi-armed bandits for adaptive constraint propagation. In: Proceedings of IJCAI'15. pp. 290–296 (2015)
6. van Beek, P.: Backtracking search algorithms. In: Handbook of Constraint Programming, chap. 4, pp. 85–134. Elsevier (2006)
7. Bessiere, C., Régin, J.: MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In: Proceedings of CP'96. pp. 61–75 (1996)
8. Bessiere, C., Zanuttini, B., Fernandez, C.: Measuring search trees. In: Proceedings of ECAI'04 workshop on Modelling and Solving Problems with Constraints. pp. 31–40 (2004)
9. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proceedings of ECAI'04. pp. 146–150 (2004)
10. Bubeck, S., Cesa-Bianchi, N.: Regret Analysis of Stochastic and Nonstochastic Multi-armed Bandit Problems. Foundations and Trends in Machine Learning, Now Publishers (2012)
11. Gecode Team: Gecode: Generic constraint development environment (2006), available from `http://www.gecode.org`
12. Gomes, C., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. Journal of Automated Reasoning **24**(1), 67–100 (2000)
13. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings of AAAI '98. pp. 431–437 (1998)
14. Grimes, D., Wallace, R.: Sampling strategies and variable selection in weighted degree heuristics. In: Proceedings of CP'07. pp. 831–838 (2007)
15. Habet, D., Terrioux, C.: Conflict History Based Branching Heuristic for CSP Solving. In: Proceedings of the 8th International Workshop on Combinations of Intelligent Methods and Applications (CIMA). Volos, Greece (Nov 2018)
16. Haralick, R., Elliott, G.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14**, 263–313 (1980)
17. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1. p. 607–613. IJCAI'95 (1995)
18. Hebrard, E.: Mistral, a constraint satisfaction library. Proceedings of the Third International CSP Solver Competition **3**(3), 31–39
19. Hogg, T., Huberman, B.A., Williams, C.P.: Phase transitions and the search problem. Artificial Intelligence **81**(1), 1 – 15 (1996)
20. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Recording and minimizing nogoods from restarts. Journal on Satisfiability, Boolean Modeling and Computation (JSAT) **1**, 147–167 (2007)
21. Loth, M., Sebag, M., Hamadi, Y., Schoenauer, M.: Bandit-based search for constraint programming. In: Proceedings of CP'13. pp. 464–480 (2013)

22. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. Information Processing Letters **47**(4), 173–180 (1993)
23. Mackworth, A.: On reading sketch maps. In: Proceedings of IJCAI'77. pp. 598–606 (1977)
24. Merchez, S., Lecoutre, C., Boussemart, F.: Abscon: a prototype to solve CSPs with abstraction. In: Proceedings of CP'01. pp. 730–744 (2001)
25. Michel, L., Hentenryck, P.V.: Activity-based search for black-box constraint programming solvers. In: Proceedings of CPAIOR'12. pp. 228–243 (2012)
26. Pisinger, D., Ropke, S.: Large neighborhood search. In: Handbook of metaheuristics. pp. 399–419. Springer (2010)
27. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Solver Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2016), http://www.choco-solver.org
28. Sabin, D., Freuder, E.: Contradicting conventional wisdom in constraint satisfaction. In: Proceedings of CP'94. pp. 10–20 (1994)
29. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: Proceedings of AAAI '94. pp. 337–343 (1994)
30. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: Proceedings of the Tenth National Conference on Artificial Intelligence. pp. 440–446. AAAI'92, AAAI Press (1992), http://dl.acm.org/citation.cfm?id=1867135.1867203
31. Sutton, R., G. Barto, A.: Reinforcement learning: An introduction **9**, 1054 (02 1998)
32. Thompson, W.R.: On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. Biometrika **25**(3-4), 285–294 (12 1933)
33. Walsh, T.: Search in a small world. In: Proceedings of IJCAI'99. pp. 1172–1177 (1999)
34. Wattez, H., Koriche, F., Lecoutre, C., Paparrizou, A., Tabary, S.: Heuristiques de recherche : un bandit pour les gouverner toutes. In: 15es Journées Francophones de Programmation par Contraintes – JFPC 2019 (2019), https://hal.archives-ouvertes.fr/hal-02414288
35. Wattez, H., Koriche, F., Lecoutre, C., Paparrizou, A., Tabary, S.: Learning variable ordering heuristics with multi-armed bandits and restarts. In: Proceedings of ECAI'20 (to appear)
36. Wattez, H., Lecoutre, C., Paparrizou, A., Tabary, S.: Refining constraint weighting. In: Proceedings of ICTAI'19. pp. 71–77 (2019)
37. Xia, W., Yap, R.H.C.: Learning robust search strategies using a bandit-based approach. In: Proceedings of AAAI'18. pp. 6657–6665 (2018)

Actes JFPC 2021

Tirage de solutions par ajout de contraintes tables aléatoires

Mathieu Vavrille[†]*[,] **Charlotte Truchet**[†] **Charles Prud'homme**[‡]

[†]Laboratoire des Sciences du Numérique de Nantes, 44322 Nantes
[‡]TASC, IMT-Atlantique, LS2N-CNRS, F-44307 Nantes
[†]prenom.nom@univ-nantes.fr [‡]prenom.nom@imt-atlantique.fr

Résumé

Les solveurs de contraintes actuels mettent à disposition des utilisateurs des algorithmes efficaces pour traiter les problèmes de satisfaction et d'optimisation combinatoires. Ceux-ci sont inadaptées à de nouveaux usages, comme celui du tirage aléatoire de solutions. Nous proposons ici un algorithme pour tirer des solutions aléatoirement, se basant sur l'ajout de contraintes tables générées aléatoirement, sans modifier le modèle du problème. Nous avons implémenté cette méthode de résolution en utilisant un solveur de contraintes existant. Nos expériences montrent que cet algorithme est une amélioration par rapport à une stratégie de branchement aléatoire en terme de qualité de l'aléatoire du tirage.

Abstract

Constraint solvers feature efficient algorithms to handle satisfaction and optimisation combinatorial problems. These features are not suited to new usages, such as the sampling of solutions. We propose here an algorithm to randomly sample solutions, based on the addition of randomly generated table constraints, without modifying the model of the problem. We implemented this method of resolution using an existing constraint solver. Our experiments show that this algorithm is an improvement over a random branching strategy in terms of quality of the randomness of the sampling.

1 Introduction

Les solveurs de contraintes s'améliorent constamment en terme d'expressivité, de facilité d'utilisation et de performances, ce qui les ouvre à de nouvelles utilisations. Classiquement les solveurs permettent de trouver une, ou la meilleure solution pour un problème. Parmi les applications des solveurs de contraintes il y a par exemple les problèmes de planification ou de configuration. Or, pour ces applications, il est fréquent que l'utilisateur souhaite avoir plusieurs solutions significativement différentes pour faciliter l'aide à la décision. Il peut également exiger une forme d'équité entre les solutions, si les décisions prises à partir de ces solutions ont des impacts humains. On se trouve donc face à deux nouvelles problématiques : renvoyer plusieurs solutions variées, ou choisir équitablement une solution (tirage aléatoire uniforme). Les mécanismes internes des solveurs de contraintes ne peuvent pas garantir ces propriétés car la recherche en profondeur renvoie séquentiellement des solutions proches.

Dans cet article, nous nous intéressons au problème de tirer aléatoirement des solutions, uniformément parmi l'ensemble des solutions. En effet, des solutions renvoyées aléatoirement permettent d'avoir une certaine diversité, car il est peu probable de toujours avoir des solutions proches, et garantit une certaine équité car la recherche n'est pas biaisée par la modélisation du problème. Il est toujours possible de choisir aléatoirement variables et valeurs avec la stratégie de recherche RandomVarDom mais les solutions sont loin d'être renvoyées uniformément parmi toutes les solutions.

1.1 Contributions

Cet article présente une nouvelle approche permettant de tirer aléatoirement une solution, utilisant un solveur de contraintes comme une boîte noire, et sans ré-écrire le modèle du problème.

Nous proposons un algorithme simple reposant sur l'ajout au problème de contraintes tables générées aléatoirement. Nous avons implémenté cet algorithme en utilisant le solveur Choco-solver[12] et nous le comparons à RandomVarDom sur des plusieurs types de

*Papier doctorant : Mathieu Vavrille[†] est auteur principal.

problèmes. Les expériences montrent que l'approche tire *plus uniformément* que la stratégie RandomVarDom, et atteint l'uniformité sur certain problèmes. Bien que, pour garantir ces propriétés supplémentaires, on s'attende à une dégradation du temps de calcul, nous constatons que ce dernier reste dans un ordre de grandeur comparable.

1.2 Travaux liés

L'approche utilisée dans cet article est fortement inspirée des travaux de Kuldeep Singh Meel [10] en SAT, où les contraintes ajoutées sont des contraintes XOR générées aléatoirement. L'algorithme d'ajout de contraintes, associé à ce choix spécifique de contraintes XOR, permet de contrôler la proximité entre l'échantillonnage effectué, et un tirage uniforme. En comparaison, nous avons fait le choix de contraintes tables, efficaces et déjà existantes dans les solveurs CP, et d'un algorithme de tirage plus économique. En contrepartie, notre méthode ne donne pas de garanties sur l'uniformité du tirage, et nous l'étudions donc expérimentalement.

Le tirage aléatoire de solutions en programmation par contraintes a été étudié d'abord dans [4], puis [6] en utilisant des réseaux bayésiens. Ces approches permettent d'avoir un tirage uniforme (ou de choisir la distribution des solutions), mais ont l'inconvénient majeur d'être exponentielles en la largeur induite du graphe de contraintes, ce qui rend impossible l'utilisation sur de grosses instances, et force l'utilisation d'approximations. D'autres approches, dans [7, 15], garantissent la diversité des solutions, qui est une tâche différente de l'échantillonnage car elle consiste à trouver des solutions éloignées les unes des autres (pour une métrique adaptée). Ces approches utilisent une réécriture du problème, ou une heuristique de recherche visant à explorer les espaces de solutions éloignés des solutions déjà trouvées.

1.3 Organisation de l'article

L'article est organisé de la manière suivante : la section 2 présente les pré-requis théoriques nécessaires à la compréhension de l'article, notamment le test du χ^2 et la stratégie RandomVarDom ; la section 3 présente la nouvelle approche de tirage par tables ; pour finir, la section 4 présente les résultats expérimentaux pour évaluer la qualité de l'aléatoire ainsi que le temps de calcul.

2 Pré-requis théoriques

2.1 Programmation par contraintes

Dans cet article nous sommes intéressés par les problèmes de satisfaction de contraintes. Un problème de satisfaction de contraintes (CSP) $\mathcal{P}$ est un triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ où

- $\mathcal{X} = \{X_1, \ldots, X_n\}$ est un ensemble de variables,
- $\mathcal{D}$ est une fonction associant un domaine à chaque variable,
- $\mathcal{C}$ est un ensemble de contraintes, chaque contrainte $C \in \mathcal{C}$ est constituée :
  - d'un tuple de variables appelé *scope* de la contrainte $scp(C) = (X_{i_1}, \ldots, X_{i_r})$, où r est l'arité de la contrainte
  - d'une relation, i.e. un ensemble d'instanciations

$$rel(C) \subseteq \prod_{k=1}^{r} \mathcal{D}(X_{i_k})$$

Une contrainte est dite satisfaite si chaque variable $X_{i_k} \in scp(C)$ est instanciée à une valeur de son domaine $x_{i_k} \in \mathcal{D}(X_{i_k})$ et que $(x_{i_1}, \ldots, x_{i_r}) \in rel(C)$. Les contraintes peuvent être données en extension (appelées contraintes de table [5]) en donnant explicitement $rel(C)$, ou en intention par une expression dans un langage de plus haut niveau. Par exemple, l'expression $x + y \leq 1$ pour des domaines $\{0, 1\}$ représente $rel(C) = \{(0, 0), (0, 1), (1, 0)\}$.

La résolution d'un CSP est la recherche d'une, une partie ou toutes les solutions, c'est à dire l'affectation d'une valeur à chaque variable, telle que toutes les contraintes sont satisfaites. Les problèmes d'optimisation (COP) sont des CSP auxquels ont été ajoutés une fonction objectif obj à minimiser (ou maximiser).

Notation : Soit un problème $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, et une contrainte C, par abus de notation nous notons $\mathcal{P} \wedge C$ le CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \cup \{C\} \rangle$. Nous notons $Sols(\mathcal{P})$ l'ensemble des solutions du problème $\mathcal{P}$.

Par la suite, nous ne considérerons que des problèmes de satisfaction, mais les problèmes d'optimisations pourront être traités grâce à la transformation suivante. Soit un COP $(\mathcal{P}, obj)$ à minimiser (resp. maximiser), et soit opt la valeur minimum (resp. maximum) de l'objectif. Soit $\epsilon \geq 0$, on transforme alors ce problème en un CSP $\mathcal{P} \wedge (obj \leq opt + \epsilon)$ (resp. $\mathcal{P} \wedge (obj \geq opt - \epsilon)$). Plus l'écart ϵ est élevé, plus les solutions cherchées seront éloignées de la valeur optimale.

2.2 Test du chi-deux

Évaluer le caractère aléatoire d'un système est une tâche compliquée, du fait que les processus aléatoires

peuvent prendre des valeurs étonnantes sans pour autant être biaisés (par exemple, une pièce tombant 10 fois de suite sur pile). Le test du chi-deux (ou χ^2) permet de confronter un résultat d'expérience aléatoire a une distribution de probabilité attendue. Il provient d'un résultat de convergence d'une loi vers la loi du χ^2 énoncé dans [11] et rappelé ici. Soit Y une variable aléatoire prenant la valeur k avec probabilité p_k pour $1 \leq k \leq d$. Soient $Y_1, \ldots, Y_n$ des variables aléatoires indépendantes de même loi que Y. Soit $N_n^{(k)}$ le nombre de variables $Y_i, 1 \leq i \leq n$ égales à k.

Théorème 1 ([11]). *Quand n tend vers l'infini, la fonction de répartition de la variable aléatoire*

$$Z_n = \sum_{k=1}^{d} \frac{\left(N_n^{(k)} - n \cdot p_k\right)^2}{n \cdot p_k}$$

tend vers la fonction de répartition de la loi du χ^2 à $(d-1)$ degrés de liberté (notée χ^2_{d-1}).

Le test du χ^2 revient à tirer des valeurs en faisant l'hypothèse qu'elles suivent la loi de Y, de calculer la valeur expérimentale z_n^{exp}, puis calculer la probabilité (appelée valeur-p)

$$\mathbb{P}(Z_n \geq z_n^{exp}) \approx \mathbb{P}(\chi^2_{d-1} \geq z_n^{exp})$$

Si cette probabilité est faible, cela signifie qu'il est improbable d'avoir un résultat plus extrême que celui obtenu, ce qui signifie que l'hypothèse selon laquelle les valeurs tirées suivent la même loi que Y peut être rejetée avec confiance.

2.3 Stratégie de recherche aléatoire

L'algorithme de recherche de solution d'un problème de contraintes alterne :

- une recherche en profondeur où l'espace de recherche est réduit en ajoutant une contrainte (appelée décision), par exemple, une affectation d'une variable à une valeur de son domaine ;
- une phase de propagation qui vérifie la satisfiabilité des contraintes du CSP.

Une stratégie de recherche naturelle pour avoir une recherche aléatoire de solutions est la stratégie RandomVarDom, qui consiste à choisir aléatoirement (uniformément) une variable X parmi toutes les variables non instanciées, puis une valeur $x \in \mathcal{D}(X)$, et appliquer la décision $X = x$. Cette stratégie a l'avantage d'être très facilement implémentée dans n'importe quel solveur de contraintes. En revanche elle empêche d'utiliser une autre stratégie d'exploration plus efficace, et elle n'offre pas de garanties sur la répartition des solutions tirées,

Fonction *TableAleatoire*$(\mathcal{P}, v, p)$
  Données : Un CSP $\mathcal{P} = \langle \{X_1, \ldots, X_n\}, \mathcal{D}, \mathcal{C} \rangle$, $v > 0$, $0 < p < 1$
  Résultat : Une contrainte table aléatoire
  $T \leftarrow \{\}$;
  $i_1, \ldots, i_v \leftarrow$ TireIndices$(\mathcal{P}, v)$;
  pour chaque $(x_{i_1}, \ldots, x_{i_v}) \in \prod_{k=1}^{v} \mathcal{D}(X_{i_k})$ **faire**
    si Random() $< p$ **alors**
      $T.add((x_{i_1}, \ldots, x_{i_v}))$;
  retourner Table$([X_{i_1}, \ldots, X_{i_v}], T)$;

Algorithme 1 : Algorithme de tirage de contrainte table aléatoire

notamment la répartition des solutions peut ne pas être uniforme. Par exemple sur le problème $\mathcal{P} = \langle \{X, X\}, \{X \to \{0,1\}, Y \to \{0,1\}\}, \{X + Y > 0\} \rangle$, soit s la solution renvoyée par un solveur configuré pour faire une recherche aléatoire, alors,

$$\mathbb{P}(s = \{X \mapsto 0, Y \mapsto 1\}) = \frac{3}{8}$$
$$\mathbb{P}(s = \{X \mapsto 1, Y \mapsto 0\}) = \frac{3}{8}$$
$$\mathbb{P}(s = \{X \mapsto 1, Y \mapsto 1\}) = \frac{1}{4}$$

Les implications sur le temps de calcul pour trouver des solutions sur différents problèmes sont évoquées dans la section 4.

3 Nouvelle approche d'échantillonage

Nous présentons ici une nouvelle approche pour tirer des solutions. Cette approche se base sur l'ajout de contraintes tables générées aléatoirement pour réduire le nombre de solutions du problème, solutions qui seront alors énumérées.

3.1 Contraintes tables aléatoires

L'algorithme de tirage aléatoire de tables est présenté dans l'algorithme 1. On suppose accessibles des fonctions Random() qui renvoie un nombre flottant aléatoire entre 0 et 1, TireIndices$(\mathcal{P}, v)$ qui tire v indices $i_1, \ldots, i_v$ tels que $|\mathcal{D}(X_{i_k})| \neq 1, 1 \leq k \leq v$ et Table(X, T) qui crée une contrainte table C telle que $scp(C) = X$ et $rel(C) = T$. Les deux paramètres de l'algorithme sont : v le nombre de variables dans la table, et p la probabilité d'ajouter un tuple dans la table. L'algorithme choisit aléatoirement v variables parmi les variables dont le domaine n'est pas un singleton, puis parcourt toutes les instanciations de ces v

variables, en ajoutant chacune à la table avec probabilité p. Le théorème qui suit montre qu'ajouter une table réduit le nombre de solutions du problème d'un facteur p en moyenne.

Théorème 2. *Soit $\mathcal{P}$ un CSP, et T une contrainte table générée aléatoirement avec probabilité p. Alors*

$$\mathbb{E}\left(|Sols(\mathcal{P} \wedge T)|\right) = p|Sols(\mathcal{P})|$$

Démonstration. Pour $\sigma \in Sols(\mathcal{P})$, soit γ_σ une variable aléatoire valant 1 si et seulement si $\sigma \in Sols(\mathcal{P} \wedge T)$. Pour connaître $\mathbb{P}(\gamma_\sigma = 1)$ il suffit de savoir avec quelle probabilité σ satisfait T. Soient $X_{i_1}, \ldots, X_{i_v}$ les variables choisies dans la table T. Chaque instanciation de ces variables a été ajoutée dans la table avec probabilité p, notamment l'instanciation $(\sigma(X_{i_1}), \ldots, \sigma(X_{i_v}))$. Cela signifie que σ satisfait la contrainte table T avec probabilité p. On a donc $p = \mathbb{P}(\gamma_\sigma = 1) = \mathbb{E}(\gamma_\sigma)$. Cela nous donne

$$\begin{aligned}
\mathbb{E}\left(|Sols(\mathcal{P} \wedge T)|\right) &= \mathbb{E}\left(\sum_{\sigma \in Sols(\mathcal{P})} \gamma_\sigma\right) \\
&= \sum_{\sigma \in Sols(\mathcal{P})} \mathbb{E}(\gamma_\sigma) \\
&= \sum_{\sigma \in Sols(\mathcal{P})} p \\
&= p|Sols(\mathcal{P})|
\end{aligned}$$

□

3.2 Algorithme d'échantillonage

Avant de présenter l'algorithme d'échantillonage, il faut présenter les fonctions auxilliaires qui sont utilisées. La première fonction est ElementAleatoire(S) qui renvoie un élément aléatoire pris uniformément dans S. La seconde fonction est TrouveSolutions($\mathcal{P}, s$), qui énumère des solutions jusqu'à en avoir trouvé s, puis les renvoie. Ce qu'il faut remarquer sur cette fonction est que si elle renvoie s solutions, alors $|Sols(\mathcal{P})| \geq s$, et si elle renvoie strictement moins de s solutions, alors toutes les solutions ont été trouvées. La recherche en profondeur faite dans les solveurs de contraintes facilite l'implémentation de cette fonction.

L'algorithme d'échantillonage de solutions fonctionne de la manière suivante : des contraintes tables sont ajoutées au problème pour réduire le nombre de solutions, et quand il y a moins de solutions qu'une valeur pivot préalablement choisie, une solution est renvoyée aléatoirement et uniformément parmi les solutions restantes. L'algorithme est présenté en détail dans l'algorithme 2. Une valeur pivot κ est choisie pour l'énumération successive des solutions des problèmes intermédiaires,

```
1  Fonction TirageParTables(P, κ, v, p)
       Données : Un CSP P, κ ≥ 2, v > 0,
                 0 < p < 1
       Résultat : Une solution du problème P
2      S ← TrouveSolutions(P, κ);
3      si |S| = 0 alors
4          retourner "Pas de solution";
5      tant que |S| = 0 ∨ |S| = κ faire
6          t ← TableAleatoire(P, v, p);
7          S ← TrouveSolutions(P ∧ t, κ);
8          si |S| ≠ 0 alors
9              P ← P ∧ t;
10     retourner ElementAleatoire(S);
```

Algorithme 2 : Algorithme de tirage par ajout de tables

ainsi que le nombre de variables par tables v, et la probabilité d'ajouter un tuple dans la table p.

L'algorithme commence par énumérer κ solutions du problème initial. Il s'arrête immédiatement si le problème n'a pas de solutions, ou qu'il y a moins de κ solutions. Si le problème a plus de κ solutions, à chaque étape une nouvelle contrainte table est générée aléatoirement et si le problème avec cette contrainte a toujours des solutions, la contrainte est ajoutée au problème définitivement. L'algorithme s'arrête quand le problème est satisfiable et qu'il y a moins de κ solutions, et une solution est choisie aléatoirement et uniformément parmi les solutions du problème.

3.3 Ajout dichotomique des tables

Il est possible d'améliorer l'algorithme en modifiant le nombre de tables ajoutées à chaque étape. Au début de la recherche une table a une faible probabilité de rendre le problème inconsistant, donc il est plus judicieux pour réduire le nombre de résolutions d'ajouter plusieurs tables d'un coup au problème. Cet algorithme est inspiré de la recherche dichotomique dans un espace non borné : commencer par trouver i tel que la solution est entre 2^i et $2^{i+1} - 1$, puis faire une recherche dichotomique habituelle entre 2^i et $2^{i+1} - 1$.

L'algorithme d'ajout dichotomique de tables est présenté dans l'algorithme 3 et vise à remplacer les lignes 6 à 9 de l'algorithme 2. Soit n le nombre de tables ajoutées à l'étape précédente, on choisit $nbTables = 1$ si $n = 0$, ou $nbTables = 2n$ sinon, puis $nbTables$ sont générées et stockées dans la liste $\mathcal{T}$. L'algorithme énumère ensuite κ solutions au problème auquel on a ajouté toutes les contraintes de $\mathcal{T}$, et si il n'y a pas de solution il supprime la moitié des contraintes présentes dans $\mathcal{T}$. La procédure s'arrête quand le problème est satisfiable, ou que $|\mathcal{T}| = 0$.

1 **Fonction** AjoutDichotomique($\mathcal{P}, nbTables, \kappa, v, p$)
  Données : Un CSP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, $nbTables > 0$, $\kappa \geq 2$, $v > 0$, $0 < p < 1$
  Résultat : $\mathcal{P}$ augmenté des nouvelles contraintes table, et le nombre de tables ajoutées
2 $\mathcal{T} \leftarrow$ tableau de taille $nbTables$;
3 **pour** $i = 0$ *to* $nbTables - 1$ **faire**
4 $\mathcal{T}[i] \leftarrow$ TableAleatoire($\mathcal{P}, v, p$);
5 $S \leftarrow$ TrouveSolutions($\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa$);
6 **tant que** $|S| = 0 \wedge |\mathcal{T}| > 0$ **faire**
7 $\mathcal{T} \leftarrow \mathcal{T}[0 : |\mathcal{T}|/2[$;
8 $S \leftarrow$ TrouveSolutions($\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, \kappa$);
9 **retourner** $\mathcal{P} \wedge \bigwedge_{t \in \mathcal{T}} t, |\mathcal{T}|$;

Algorithme 3 : Algorithme dichotomique d'ajout de tables

3.4 Discussion

Qualité de la division des tables : Dans la preuve du théorème 2, les variables aléatoires $(\gamma_\sigma)_{\sigma \in Sols(\mathcal{P})}$ ne sont pas indépendantes. Par exemple, soient σ_1 et σ_2 deux solutions du problème qui ne diffèrent que sur une seule variable X, alors

$$\mathbb{P}(\gamma_{\sigma_2} = 1 | \gamma_{\sigma_1} = 1) = p \cdot \mathbb{P}(X \in scp(T)) + 1 \cdot \mathbb{P}(X \notin scp(T)) \quad (1)$$

Cela signifie que si la table ne porte pas sur toutes les variables, alors elle ne sépare pas bien les regroupements de solutions qui prennent la même valeur sur plusieurs variables. Cette notion d'indépendance est centrale dans les approches de Kuldeep Meel [10] pour prouver l'uniformité du tirage. Ici le tirage n'est pas uniforme, mais est quand même une amélioration par rapport à la stratégie RandomVarDom.

La formule 1 sur la non indépendance nous montre aussi que plus la table contient de variables, plus les variables aléatoires γ_σ sont indépendantes. Des tables contenant toutes les variables du problème rendraient indépendantes les variables aléatoires γ_σ, et donneraient ainsi une garantie théorique, mais elles seraient impossible à générer en pratique.

Influence des paramètres Trois paramètres doivent être choisis pour exécuter l'algorithme. Nous pouvons déjà estimer quel est l'impact de chaque paramètre sur le temps de calcul, et sur la qualité de l'aléatoire.

— Le pivot détermine combien de solutions au maximum seront énumérées à chaque étape. Avoir un pivot élevé nécessite plus de résolution car à chaque étape il faut trouver plus de solutions.
— Comme vu précédemment, augmenter le nombre de variables doit améliorer l'aléatoire, mais va aussi augmenter exponentiellement le nombre de tuples dans les tables ce qui aura un impact négatif sur le temps de calcul.
— Réduire la probabilité d'ajouter un tuple à une table doit améliorer le temps de calcul, car les tables sont alors plus petites donc la propagation de ces tables sera plus rapide, et le nombre de tables ajoutées sera aussi plus faible car le problème sera plus vite réduit.

Ces hypothèses sont vérifiées dans la section 4 expérimentalement.

4 Expériences

Cette section présente les expériences faites pour tester notre nouvelle approche. Le premier but est d'évaluer le comportement aléatoire de l'approche, et ensuite d'évaluer le temps de calcul, en se comparant à la stratégie RandomVarDom. Le code est disponible en ligne [1], avec les scripts pour générer les figures présentées dans cet article.

4.1 Implémentation

L'implémentation a été faite en Java 11 en utilisant le solveur de contraintes `choco-solver` version 4.10.6 [12]. Il est possible de créer le modèle directement en utilisant `choco-solver`, ou alors en donnant en paramètre un fichier au format FlatZinc. La stratégie de recherche utilisée est celle par défaut du solveur (`dom/Wdeg` [3] et `lastConflict` [9]). Si le fichier FlatZinc déclare une stratégie, elle est utilisée.

Une amélioration a été apportée en ajoutant une étape de propagation du solveur avant la génération d'une table (avant la ligne 6 de l'algorithme 2). Cela permet lors de la génération des tables de ne pas avoir à énumérer des tuples qui auraient été immédiatement éliminés par la propagation.

Par la suite, l'algorithme utilisé est le tirage par tables amélioré avec l'ajout dichotomique des tables.

Gestion de l'aléatoire Le générateur de nombres aléatoires utilisé est celui présent de base en Java : `java.util.Random`. Ce générateur utilise une formule de congruence linaire pour modifier une graine sur 48 bits donnée en entrée. La documentation de Java pointe vers [8], section 3.2.1 pour plus d'information. Ce générateur d'aléatoire a des défauts (notamment

1. https://github.com/MathieuVavrille/tableSampling

une periode de 2^{48}), mais pour l'utilisation qui est faite ici il est suffisant (comme montré dans [2]).

L'implémentation utilise une unique instance du générateur aléatoire qui est passé en argument à toutes les fonctions ayant besoin de générer des nombres aléatoires. Cela permet d'éviter des effets de non indépendance qui peuvent advenir à cause d'une mauvaise création de graines aléatoires.

4.2 Les problèmes utilisés

L'approche mise en place est indépendante des contraintes des modèles résolus, cela nous a permis de l'appliquer sur quatre problèmes différents, dont trois issus de problèmes réels. Nous présentons ici les modèles et leurs caractéristiques.

N-reines Le premier problème est celui des N-reines, qui consiste à placer N reines sur un plateau d'échec de taille $N \times N$ de sorte à ce qu'aucune n'en attaque une autre (les reines attaquent les pièces dans les huit directions et aussi loin que possible). Nous avons choisi la modélisation classique avec N variables qui ont un domaine $[1, N]$, une contrainte `all_different`, et des contraintes binaires d'inégalité (pour les attaques en diagonale). Le nombre de solutions du problème est connu pour les premières valeurs de N [16].

Configuration des Renault Méganes Il s'agit d'un problème de configuration des Renault Mégane introduit dans [1], et déjà utilisé dans [7] pour la recherche de solutions diverses. Il y a 101 variables, avec des domaines contenant jusqu'à 43 valeurs, et les 113 contraintes sont modélisées par des contraintes tables, dont la plupart non-binaires. Ce problème est peu contraint, et a ainsi plus de $1.4 \cdot 10^{12}$ solutions.

On call rostering Ce problème modélise le système de "gardes", notamment utilisé par les personnels de santé. Le problème est disponible dans les benchmarks de MiniZinc [2], et contient différents types de contraintes, comme des contraintes linéaires, des contraintes globales `count`, des valeurs absolues, des implications, et des contraintes tables. Plusieurs jeux de données sont disponibles, mais seul le plus petit (`4s-10d.dzn`) a été utilisé ici. Il s'agit d'un problème d'optimisation (de minimisation), donc il a fallu le transformer en problème de satisfaction en bornant l'objectif. Ce problème a pour valeur optimale 1 :

— Il y a 136 solutions en fixand $obj \leq 1$
— Il y a 2099 solution en fixant $obj \leq 2$
— Il y a plus de 10000 solutions en fixant $obj \leq 3$

2. https://github.com/MiniZinc/minizinc-benchmarks/tree/master/on-call-rostering

La génération aléatoire de solutions permet d'utiliser le solveur comme un outil d'aide à la décision pour la personne créant les plannings (lui donner plusieurs plannings à comparer), et apporte une forme d'équité entre les personnels. En effet, des méthodes de recherche orientées peuvent favoriser certaines personnes, au détriment d'autres.

Feature Models Les *Feature Models* sont des problèmes de gestion de logiciels, et visent à aider à la prise de décision quant à l'ordre d'implémentation des fonctionnalités d'un logiciel. Le problème est spécifié au format MiniZinc dans l'article [13], en utilisant les données de [14]. Il s'agit encore ici d'un problème d'optimisation (de maximisation), la valeur optimale étant 20222 la résolution va ajouter la contrainte $obj \geq 17738$ ce qui en fait un problème de satisfaction à 95 solutions.

4.3 Qualité de l'aléatoire

Les premières expériences conduites visent à évaluer la qualité de l'aléatoire, c'est à dire, savoir si les solutions sont tirées aléatoirement et uniformément. Les résultats qui suivent montrent que les solutions ne sont pas tirées uniformément, mais que l'approche par tables est *plus uniforme* que la stratégie RandomVarDom.

4.3.1 Évaluation de l'uniformité

Pour avoir une mesure quantitative de l'uniformité du tirage des solutions, nous avons utilisé le test du χ^2. Connaissant le nombre de solutions $nbSols$ d'un problème (et en numérotant ces solutions), $nbSample$ tirages sont fait et le nombre d'occurences $nbOcc_i$ de chaque solution $i \in \{1, \ldots, nbSols\}$ est compté. Nous calculons la valeur de la variable

$$z_{exp} = \sum_{k=1}^{nbSols} \frac{(nbOcc_k - nbSamples/nbSols)^2}{nSamples/nbSols}$$

et ensuite la valeur-p du test [3] (c'est à dire, la probabilité que la loi du χ^2 prenne une valeur plus extrême que z_{exp}). Cette valeur-p nous donne une évaluation quantitative de la qualité de l'aléatoire. Plus particulièrement, un très grand nombre de tirages sont faits, et l'évolution de la valeur-p en fonction du nombre de tirage est affichée. Dans notre cas, comme le tirage n'est pas uniforme, la valeur-p va tendre vers 0 quand le nombre de tirages augmente, mais nous remarquerons que le tirage par tables a une valeur-p qui tend moins vite vers 0 que le tirage en utilisant RandomVarDom.

3. Nous utilisons librairie "Apache Commons Mathematics Library" (https://commons.apache.org/proper/commons-math/) pour les calculs de probabilités

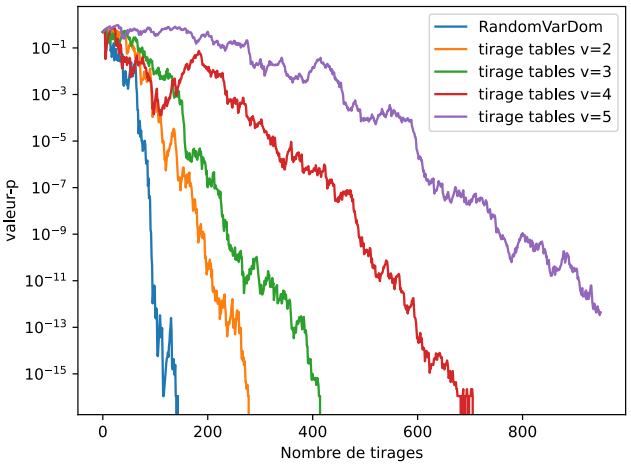


FIGURE 1 – Évolution de la valeur-p sur le problème Feature Models, avec différents nombre de variables dans les tables où $\kappa = 2$ et $p = 1/2$

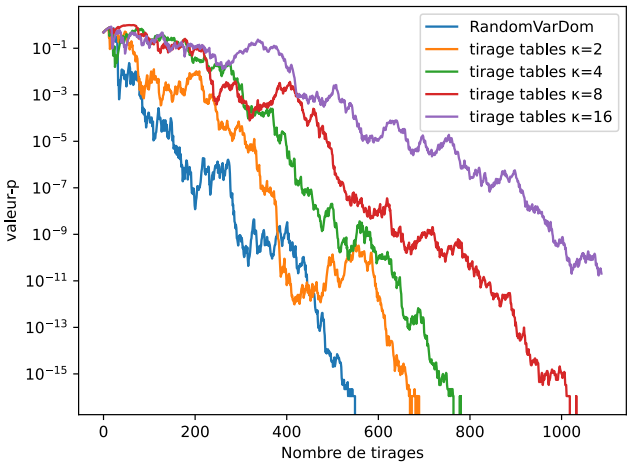


FIGURE 2 – Évolution de la valeur-p sur le problème On Call Rostering avec $obj \leq 1$, avec différents pivots et où $v = 4$ et $p = 1/8$

Pour effectuer ce test, il faut connaître le nombre de solutions *nbSols*, et pouvoir tirer plusieurs fois *nbSols* solutions, donc l'évaluation de l'aléatoire ne peut se faire que sur des petits problèmes. Les problèmes qui ont été utilisés : 9-reines (352 solutions), Feature Models avec la contrainte $obj \geq 17738$ (95 solutions), et On Call Rostering avec les contraintes $obj \leq 1$ et $obj \leq 2$ (136 et 2099 solutions).

Nous voulons aussi évaluer l'impact de l'évolution d'un paramètre (nombre de variables, pivot, ou probabilité) de l'algorithme de tirage par tables sur la qualité de l'aléatoire. Pour cela, dans les figures qui suivent sont affichés l'évolution de la valeur-p pour la stratégie RANDOMVARDOM, ainsi que pour différents paramètres du tirage par tables, en faisant varier un paramètre à la fois. La légende donne les paramètres associés à chaque exécution (v pour le nombre de variables, κ pour le pivot et p pour la probabilité).

Remarque. *Les figures montrent la valeur-p sur une échelle logarithmique, car elle tend très vite vers 0. De plus, comme les calculs sont fait en utilisant une représentation en machine des nombres flottants, une valeur de valeur-p inférieure à* 10^{-16} *sera considérée comme nulle.*

4.3.2 Impact du nombre de variables

Nous commençons par faire varier le nombre de variable dans les tables, la figure 1 montre l'évolution de la valeur-p, où des tirages ont été fait avec différentes valeurs pour v, et où on a fixé $\kappa = 2$ et $p = 1/2$. On constate premièrement une amélioration par rapport à la stratégie RANDOMVARDOM, car la valeur-p décroît beaucoup plus vite. Ensuite, notre intuition quand à l'utilisation de tables avec plus de variables est vérifiée, car quand les tables portent sur plus de variables, la valeur-p décroît plus lentement.

4.3.3 Impact du pivot

Nous voulons ensuite évaluer l'impact du pivot sur la qualité de l'aléatoire, la figure 2 montre l'évolution de la valeur-p, avec différentes valeurs de κ, en fixant $v = 4$ et $p = 1/8$, sur le problème On Call Rostering, avec la contrainte $obj \leq 1$. Quand le pivot augmente, la valeur-p décroît moins vite. Plus le pivot est grand, moins de tables seront ajoutées au problème, le cas extrême étant le cas où le pivot est plus grand que le nombre de solutions et dans ce cas l'algorithme revient à énumérer toutes les solutions, et en piocher une aléatoirement (ce n'est pas le cas ici, car il y a 136 solutions, et le pivot vaut au maximum 16).

4.3.4 Impact de la probabilité

Nous avons fini par évaluer l'impact de la probabilité d'ajouter un tuple dans les tables générées. La figure 3 montre l'évolution de la valeur-p sur le problème On Call Rostering, où la contrainte $obj \leq 2$ a été ajoutée, en fixant les paramètres $v = 5$ et $\kappa = 16$ et en faisant varier p. Encore une fois, le tirage par tables montre une amélioration par rapport à la stratégie RANDOMVARDOM, mais il n'y a pas d'influence claire de la probabilité d'ajouter un tuple dans la table sur la qualité de l'aléatoire.

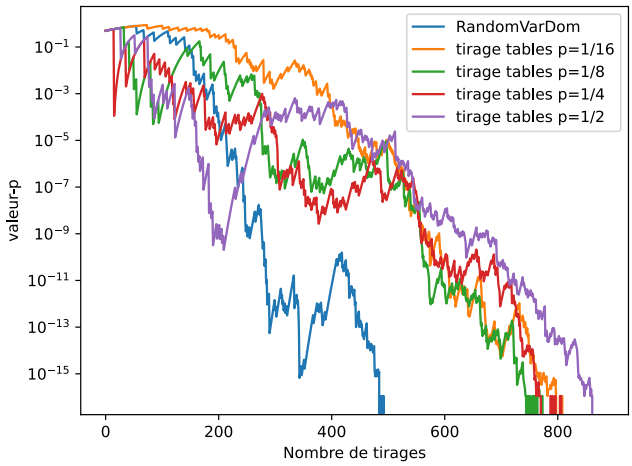


FIGURE 3 – Évolution de la valeur-p sur le problem On Call Rostering avec $obj \leq 2$, avec différentes probabilités et où $\kappa = 16$ et $v = 5$

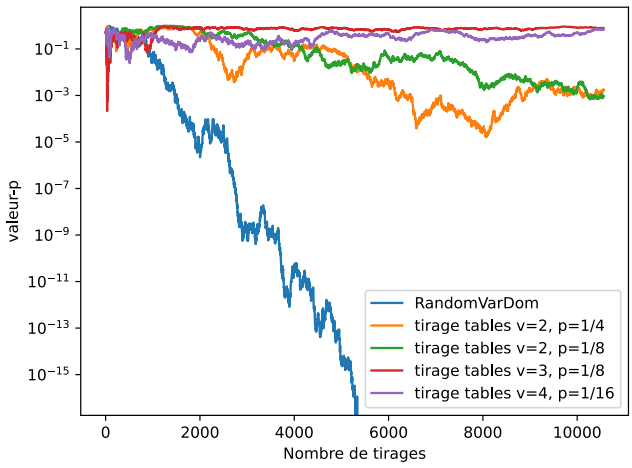


FIGURE 4 – Évolution de la valeur-p sur le problème des 9-reines, avec différents paramètres pour v et p pour le tirage par tables, avec $\kappa = 8$

4.3.5 Qualité de l'aléatoire obtenu

On constate que sur le problème des N-reines, la qualité de l'échantillonnage est particulièrement bonne. La figure 4 montre l'évolution de la valeur-p sur ce problème, avec différents paramètres pour le tirage par tables. On constate que la valeur-p ne tend pas vers 0 avec le tirage par tables, alors qu'elle tend toujours vers 0 avec la stratégie RANDOMVARDOM. Nous supposons que les solutions sont assez bien réparties dans l'espace de recherche, ce qui expliquerait ces bonnes performances.

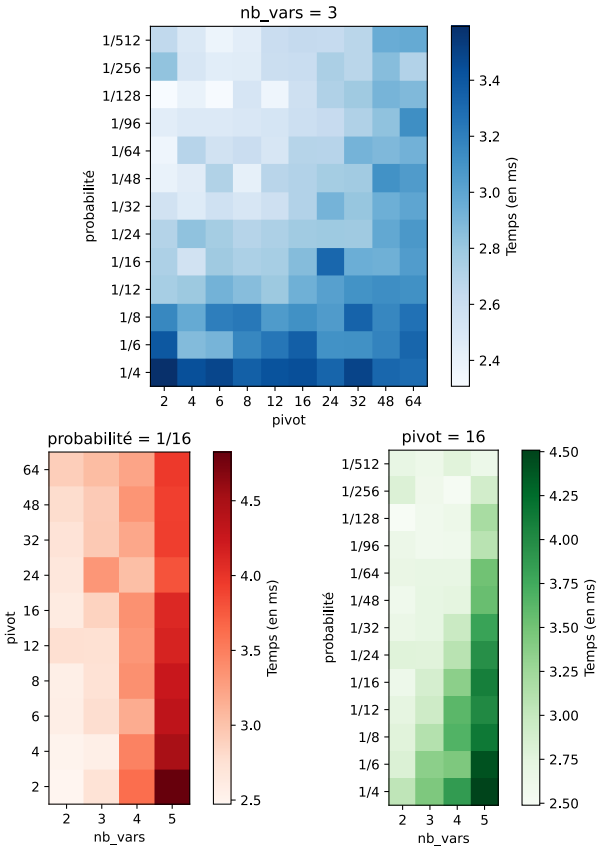


FIGURE 5 – Cartes de chaleur du temps pour tirer une solution, en fixant différent paramètres, sur le problème d'On Call Rostering, avec la contrainte $obj \leq 3$

4.4 Temps d'exécution

L'évaluation du temps d'exécution se fait en deux parties. Le choix des paramètres est important dans le temps d'exécution, donc il faut trouver quels paramètres ont un impact positif sur le temps de calcul. Par la suite, nous utiliserons des problèmes avec plus de 10000 solutions, et 50 tirages seront fait pour faire une moyenne du temps pour tirer une solution avec les différentes méthodes.

4.4.1 Impact des paramètres

Pour montrer l'impact des paramètres sur le temps pour tirer une solution, nous fixons un des paramètres, et faisons varier les deux autres pour ainsi avoir des cartes de chaleur du temps en fonction de différents paramètres. La figure 5 montre les 3 cartes de chaleur obtenues en fixant le nombre de variables, le pivot ou la probabilité. Les hypothèses faites en section 3.4 sont vérifiées :

a. diminuer le nombre de variables dans les tables améliore le temps de calcul

b. augmenter le pivot augmente le temps de calcul

c. diminuer la probabilité d'ajouter un tuple aux tables améliore le temps de calcul

Comme nous l'avons vu précédemment, augmenter le pivot améliore la qualité de l'aléatoire. Nous avons donc ici un compromis à faire entre le temps de calcul (réduire le pivot ou le nombre de variables) et la qualité de l'aléatoire (augmenter le pivot ou le nombre de variables).

Nous avons aussi vu que la probabilité n'avait pas d'impact significatif sur la qualité de l'aléatoire, donc il est possible de choisir des probabilités très faibles pour que les tables soient plus petites, et ainsi avoir une propagation plus rapide, ainsi que moins de tables ajoutées lors de l'algorithme.

4.4.2 Comparaison par rapport à RandomVarDom

Pour finir, il faut comparer les temps à la stratégie RandomVarDom. La table 1 montre pour huit ensembles de paramètres différents le temps pour tirer une solution avec le tirage par tables, contre le temps en utilisant la stratégie RandomVarDom, la dernière colonne donnant le ratio $T_{tables}/T_{RandomVarDom}$. Nous remarquons que notre approche est plus lente que la stratégie RandomVarDom sur les problèmes de configuration de la Renault Mégane, ainsi que sur les 12-reines. En revanche sur le problème On Call Rostering (où l'objectif a été borné par 3, le problème a ainsi plus de 10000 solutions), les performances sont meilleures.

La raison pour laquelle le tirage par tables fonctionne particulièrement bien sur On Call Rostering est que ce problème a très peu de solutions, et surtout très peu de valeurs rendent le problème satisfiable pour chaque variable. Les problèmes de configuration de la Renault Mégane et des 12-reines ont très peu de valeurs qui sont interdites, ce qui fait que peu importe le choix fait par la stratégie d'exploration, les retours en arrière sont très rares. En revanche sur le problème On Call Rostering, beaucoup de valeurs initialement dans les domaines ne débouchent pas sur une solution. Cela signifie que la stratégie RandomVarDom va faire des choix de couples (variable,valeur) qui ont une forte probabilité de ne pas donner de solution, ainsi obligeant un retour en arrière dans l'arbre de décision. Une stratégie optimisée pour la résolution de problème, même simple va fortement améliorer le temps de calcul, et notre approche de tirage par tables permet de laisser le solveur choisir la stratégie de recherche en fonction de la structure du problème.

Problème	RandomVarDom	Tirage par tables				Ratio
		v	κ	p	Temps	
Megane	32 ms	2	16	1/16	55 ms	1.7
				1/32	59 ms	1.8
			32	1/16	89 ms	2.8
				1/32	86 ms	2.7
		3	16	1/16	74 ms	2.3
				1/32	67 ms	2.1
			32	1/16	83 ms	2.6
				1/32	65 ms	2.0
On Call Rostering	38 ms	2	16	1/16	14 ms	0.36
				1/32	14 ms	0.38
			32	1/16	15 ms	0.4
				1/32	18 ms	0.46
		3	16	1/16	18 ms	0.47
				1/32	15 ms	0.4
			32	1/16	19 ms	0.5
				1/32	17 ms	0.44
12-reines	2 ms	2	16	1/16	4 ms	2.4
				1/32	4 ms	2.3
			32	1/16	7 ms	4.1
				1/32	7 ms	3.9
		3	16	1/16	10 ms	5.6
				1/32	8 ms	4.3
			32	1/16	12 ms	6.9
				1/32	11 ms	6.0

Table 1 – Comparaison du temps pour tirer une solution entre RandomVarDom et le tirage par tables. Les temps sont les moyenne des temps pour 50 tirages

5 Conclusion

Nous avons présenté ici une nouvelle approche pour tirer des solutions aléatoirement en ajoutant au problème des contraintes tables tirées aléatoirement jusqu'à ce que le problème soit assez petit pour pouvoir être énuméré. Bien qu'il n'y ait pas de garantie d'uniformité, l'approche montre expérimentalement une amélioration de la qualité de l'aléatoire par rapport à la stratégie de recherche RandomVarDom. De plus, le fait que la stratégie de recherche ne soit pas fixée permet de garder un temps de calcul du même ordre de grandeur qu'avec la stratégie RandomVarDom.

L'algorithme de tirage étant indépendant des contraintes utilisées pour diviser l'espace, il serait intéressant d'étudier si d'autres contraintes peuvent être générées aléatoirement et permettre un tirage plus uniforme des solutions, sans devenir trop coûteuses. Une analyse plus approfondie de la structure des problèmes ou des solutions permettrait aussi de choisir des paramètres plus appropriés, tant pour avoir un plus faible temps de calcul que pour avoir un tirage plus uniforme. Il serait aussi intéressant d'étudier plus particulièrement les problèmes d'optimisation, et les notions de tirage dans ces problèmes.

Références

[1] Jérôme Amilhastre, Hélene Fargier et Pierre Marquis : Consistency restoration and explanations in dynamic csps—application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[2] Eric Bach : Realistic analysis of some randomized algorithms. *In Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 453–461, 1987.

[3] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre et Lakhdar Sais : Boosting systematic search by weighting constraints. *In ECAI*, volume 16, page 146, 2004.

[4] Rina Dechter, Kalev Kask, Eyal Bin, Roy Emek *et al.* : Generating random solutions for constraint satisfaction problems. *In AAAI/IAAI*, pages 15–21, 2002.

[5] Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin et Pierre Schaus : Compact-table : efficiently filtering table constraints with reversible sparse bit-sets. *In International Conference on Principles and Practice of Constraint Programming*, pages 207–223. Springer, 2016.

[6] Vibhav Gogate et Rina Dechter : A new algorithm for sampling csp solutions uniformly at random. *In International Conference on Principles and Practice of Constraint Programming*, pages 711–715. Springer, 2006.

[7] Emmanuel Hebrard, Brahim Hnich, Barry O'Sullivan et Toby Walsh : Finding diverse and similar solutions in constraint programming. *In AAAI*, volume 5, pages 372–377, 2005.

[8] Donald E Knuth : *Art of computer programming, volume 2 : Seminumerical algorithms*. Addison-Wesley Professional, 2014.

[9] Christophe Lecoutre, Lakhdar Sais, Sébastien Tabary et Vincent Vidal : Last conflict based reasoning. *In* Gerhard Brewka, Silvia Coradeschi, Anna Perini et Paolo Traverso, éditeurs : *ECAI 2006, 17th European Conference on Artificial Intelligence, August 29 - September 1, 2006, Riva del Garda, Italy, Including Prestigious Applications of Intelligent Systems (PAIS 2006), Proceedings*, volume 141 de *Frontiers in Artificial Intelligence and Applications*, pages 133–137. IOS Press, 2006.

[10] Kuldeep S Meel : Constrained counting and sampling : bridging the gap between theory and practice. *arXiv preprint arXiv :1806.02239*, 2018.

[11] Karl Pearson : X. on the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 50(302):157–175, 1900.

[12] Charles Prud'homme, Jean-Guillaume Fages et Xavier Lorca : *Choco Solver Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.

[13] Björn Regnell et Krzysztof Kuchcinski : Exploring software product management decision problems with constraint solving-opportunities for prioritization and release planning. *In 2011 Fifth International Workshop on Software Product Management (IWSPM)*, pages 47–56. IEEE, 2011.

[14] Günther Ruhe et Moshood Omolade Saliu : The art and science of software release planning. *IEEE software*, 22(6):47–53, 2005.

[15] Yevgeny Schreiber : Value-ordering heuristics : Search performance vs. solution diversity. *In International Conference on Principles and Practice of Constraint Programming*, pages 429–444. Springer, 2010.

[16] Neil James Alexander Sloane : https://oeis.org/A000170.

Actes JFPC 2021

Metrics : Mission Expérimentations

Thibault Falque[1] **Romain Wallon**[2] **Hugues Wattez**[3]

[1] Exakis Nelite

[2] LIX, Laboratoire d'Informatique de l'X, École Polytechnique, Chaire X-Uber

[3] CRIL, Univ Artois & CNRS

thibault.falque@exakis-nelite.com wallon@lix.polytechnique.fr wattez@cril.fr

Résumé

Le développement de solveurs de contraintes s'accompagne nécessairement d'une phase d'expérimentation, permettant d'évaluer les performances des fonctionnalités implantées. Pour chacun des solveurs considérés, il faut alors collecter et analyser des statistisques relatives à leur exécution et qui, le plus souvent, restent les mêmes. Néanmoins, il existe probablement autant de scripts permettant d'extraire ces statistiques et de tracer les figures associées que de chercheurs du domaine. Un outil unifiant et facilitant les analyses de résultats expérimentaux paraît donc nécessaire, notamment pour favoriser le partage et la reproductibilité de ces résultats. Partant de ce constat, nous avons développé la bibliothèque *Metrics*, dont l'ambition est de fournir une chaîne complète d'outils conçue dans cette optique. Dans cet article, nous présentons cette bibliothèque et illustrons son utilisation en rejouant l'analyse des résultats de la compétition XCSP'19.

Abstract

Developing constraint solvers comes with the need to perform experiments, in order to evaluate the performance of the implemented features. For each considered solver, one needs to collect and analyze statistics about its execution which, most of the time, remain the same. However, there exist probably as many scripts allowing to extract such statistics and to draw the associated figures as researchers in the domain. A unifying tool making easier the analysis of experimental results seems thus required, especially to favor the sharing and the reproducibility of these results. Based on this observation, we have developed the *Metrics* library, whose ambition is to provide a complete toolchain designed in that purpose. In this paper, we present this library and illustrate its use through the analysis of the results of the XCSP'19 competition.

1 Introduction

Dans le cadre de la recherche en informatique (et plus généralement, dans n'importe quel domaine nécessitant la conception de logiciels), il est nécessaire de réaliser des expérimentations pour s'assurer que les programmes développés fonctionnent comme prévu. En particulier, il est important de vérifier que les ressources utilisées par ces programmes restent raisonnables. Dans ce but, différentes solutions logicielles telles que *runsolver* [7] ont été proposées, à la fois pour mesurer et limiter l'utilisation des ressources temporelles et spatiales par le programme en cours d'évaluation. Cependant, respecter les limites fixées est en général insuffisant pour évaluer le comportement du programme. Il est souvent nécessaire de collecter des statistiques supplémentaires, qui peuvent être fournies par le programme lui-même (par exemple, via ses *logs*) ou par l'environnement d'exécution (par exemple, *runsolver*). Les données collectées doivent ensuite être aggrégées pour évaluer la qualité des résultats du programme au travers d'une analyse statistique.

Dans ce domaine, de nombreux outils mathématiques peuvent être utilisés, et faire un choix parmi ceux-ci peut introduire des biais dans les résultats ou leur analyse. Il est donc important d'appliquer les principes de *science ouverte* et de *reproductibilité* pour permettre de rejouer l'analyse des résultats. Ainsi, ces principes ont fait l'objet d'une recommendation de l'OCDE [6], et des chercheurs ont déjà introduit diverses approches favorisant la reproductibilité dans le contexte d'expérimentations logicielles [3, 5]. En particulier, il est recommandé de rendre disponible le code source des logiciels étudiés (ou, à minima, leurs exécutables sous forme binaire), et de diffuser les données utilisées pour les évaluer (par exemple, dans des forges logicielles). Il

est également important de rendre l'analyse des résultats reproductibles, en utilisant des outils tels que le *RMarkdown* ou les *notebooks Jupyter*.

Dans la communauté CP (tout comme dans les communautés SAT, PB, MaxSAT ou QBF par exemple), il y a souvent peu de différences sur la manière d'exécuter les solveurs. En effet, ces derniers doivent proposer des interfaces en lignes de commande qui respectent les règles imposées par l'environnement dans lequel ils sont exécutés (par exemple, durant les compétitions). De plus, la plupart des données collectées lors de l'exécution des solveurs restent le plus souvent les mêmes (par exemple, le temps d'exécution, l'utilisation de la mémoire, etc.). Dans ce contexte, la création d'un outil permettant d'exécuter le programme, de collecter les données qu'il produit et de les analyser présenterait plusieurs avantages : tester de nouvelles fonctionnalités serait plus simple, à la fois en termes d'exécution et d'analyse, et la reproductibilité des résultats serait systématiquement assurée.

Partant de ces observations, cet article présente *Metrics* (*mETRICS* signifie *rEproducible sofTware peRformance analysIs in perfeCt Simplicity*), une bibliothèque Python visant à unifier et faciliter l'analyse des expérimentations réalisées sur des solveurs. L'ambition de *Metrics* est de fournir une chaîne complète d'outils de l'exécution du solveur à l'analyse de ses performances. Actuellement, cette bibliothèque possède deux composants principaux : *Scalpel* (*sCAlPEL* signifie *extraCting dAta of exPeriments from softwarE Logs*) et *Wallet* (*wALLET* signifie *Automated tooL for expLoiting Experimental resulTs*). D'une part, *Scalpel* est conçu pour simplifier la récupération des données expérimentales. Ce module est capable de gérer une grande variété de fichiers, incluant les formats CSV, XML, JSON ou encore la sortie produite par le solveur, qui peut être décrite par l'utilisateur dans un fichier de configuration. Cette approche fait de *Scalpel* un outil à la fois flexible et simple à configurer. D'autre part, *Wallet* propose une interface simple d'utilisation pour tracer les diagrammes les plus communément utilisés pour l'analyse de solveurs (tels que les *scatter plots* et les *cactus plots*) et pour calculer diverses statistiques relatives à leur exécution (en particulier, leurs scores en utilisant différentes mesures classiques). La conception de *Wallet* facilite l'intégration de l'analyse dans des *notebooks Jupyter* qui peuvent être facilement partagés en ligne (par exemple, *GitHub* ou *GitLab* sont capables d'afficher de tels fichiers), favorisant également la reproductibilité de l'analyse.

Dans cet article, nous présentons l'utilisation de *Metrics* de la manière suivante. En guise de préliminaires, nous introduisons les différentes notions de vocabulaires utilisées par la suite. Nous enchaînons avec la présentation des différentes étapes d'une analyse expérimentale réalisée avec *Metrics*, de l'extraction des données au tracé des diagrammes permettant leur exploitation, en rejouant l'analyse des résultats de la compétition XCSP'19 [1]. Nous concluons enfin avec quelques perspectives d'amélioration pour *Metrics*, qui permettront d'en faire une bibliothèque unifiée pour l'expérimentation de solveurs.

2 Vocabulaire de *Metrics*

Cette section introduit quelques notions de vocabulaire utilisées pour identifier les données manipulées par *Metrics* et pour décrire les analyses réalisées par cette bibliothèque.

L'objet central dans *Metrics* est la *campagne*, qui contient toutes les données expérimentales qui ont été collectées, et définit la configuration de l'environnment d'exécution des solveurs (limites temporelle et spatiale, configuration des machines, etc.). Au cours d'une campagne des *expérimentaciels* (ou *experimentwares* en anglais) sont évalués. Ce néologisme est utilisé pour caractériser n'importe quel logiciel pouvant être expérimenté, en guise de notion plus générale que le terme de *solveur*. Notons que différentes configurations d'un même programme sont considérées comme différents expérimentaciels, de même que des programmes complètement différents. Une campagne se caractérise par l'ensemble des fichiers d'entrée (*input-set*) utilisés pour cette campagne. Dans ce contexte, *tous* les expérimentaciels sont exécutés sur le *même* ensemble de fichiers. Enfin, une *expérimentation* correspond à l'exécution d'un expérimentaciel donné sur un fichier d'entrée donné, de sorte que l'ensemble des expérimentations correspond au produit cartésien de l'ensemble des fichiers d'entrée et de celui des expérimentaciels. Chaque expérimentation est caractérisée par les données qui sont pertinentes au vu des analyses à réaliser, telles que le temps d'exécution ou la mémoire utilisée par l'expérimentaciel (et bien d'autres, dépendant de la configuration fournie par l'utilisateur).

Exemple 1. *Considérons une* campagne *dans laquelle nous souhaitons comparer les solveurs* AbsCon *et* Nacre *[4]. Ces deux solveurs sont nos* expérimentaciels. *Supposons que nous souhaitions comparer ces deux solveurs sur deux* entrées, *à savoir* `AllInterval.xml` *et* `CarSequencing.xml`. *L'ensemble des fichiers d'entrée se compose de ces deux fichiers. Les expérimentations de cette campagne sont alors :*

- *l'exécution d'*AbsCon *sur* `AllInterval.xml` ;
- *l'exécution d'*AbsCon *sur* `CarSequencing.xml` ;
- *l'exécution de* Nacre *sur* `AllInterval.xml` ;
- *l'exécution de* Nacre *sur* `CarSequencing.xml`.

3 Présentation détaillée de *Metrics*

Dans cette section, nous proposons une description de notre bibliothèque *Metrics*. En particulier, nous motivons l'intérêt d'une telle bibliothèque, et illustrons son utilisation en rejouant l'analyse des résultats de la compétition XCSP'19 [1].

3.1 Motivation

La bibliothèque *Metrics* a été conçue dans le but de fournir tous les outils nécessaires pour l'analyse de résultats expérimentaux obtenus dans le cadre de l'évaluation de solveurs. Dans la communauté, plusieurs outils sont fréquemment utilisés dans ce but. C'est par exemple le cas de *GNUPlot*, qui est une bonne bibliothèque pour tracer des diagrammes (mais un peu classique), qui ne permet cependant pas de gérer toutes les fonctionnalités envisagées pour *Metrics*. L'utilisation du langage *R* a un temps été envisagée, notamment en raison de son support natif de la reproductibilité au travers du format *RMarkdown*. Son utilisation reste toutefois limitée au sein de la communauté, ce qui peut être un frein pour l'adoption de *Metrics*.

Notre choix s'est donc porté sur le langage Python, qui offre un large écosystème pour l'analyse de données. *Metrics* est en particulier fondé sur deux bibliothèques bien connues dans ce domaine, à savoir *pandas*, qui propose une implantation de *data-frames* nous permettant de gérer et manipuler les données extraites des expérimentations réalisées, et *matplotlib*, que nous utilisons pour tracer des figures statiques (des figures dynamiques sont également proposées grâce à la bibliothèque *plotly*). Concernant les figures, nous notons que des outils ont déjà été implantés en Python par la communauté, comme par exemple *mkplot*[1], mais ceux-ci ne fournissent pas une chaîne complète d'outils comme *Metrics* souhaite le faire.

3.2 *Metrics* en action

Pour présenter les capacités de *Metrics*, nous proposons dans cette section de rejouer l'analyse des résultats de la compétition XCSP'19 [1]. Dans la *main track* de cette compétition, 26 expérimentaciels (ou *solveurs*) ont été soumis et évalués sur 600 entrées (en l'occurence, 300 problèmes de décision et 300 problèmes d'optimisation décrits au format XCSP3 [2]). Chaque expérimentation (c'est-à-dire, chaque exécution d'un solveur donné sur une instance donnée) était limitée à 2400 secondes, et ne pouvait utiliser plus de 16 Go de mémoire pour les solveurs séquentiels, et 32 Go pour les solveurs parallèles.

1. https://github.com/alexeyignatiev/mkplot.git

Le lecteur intéressé pourra retrouver et reproduire l'analyse proposée ci-après en consultant le dépôt *GitHub* de *Metrics*[2].

3.2.1 Extraction des données avec *Scalpel*

La première étape pour analyser les résultats est d'extraire les données nécessaires à l'analyse. C'est le rôle du module *sCAlPEL* (*« extraCt dAta of exPeriments from softwarE Logs »*) fourni par *Metrics*, qui construit la représentation d'une campagne à partir d'un ou plusieurs fichiers produits au cours de celle-ci.

Plus précisément, *Scalpel* est capable de lire nativement différents formats de fichiers classiquement utilisés, tels que le format CSV (et ses variantes), ainsi que les formats JSON et XML (en utilisant une notation pointée pour identifier chacun des attributs apparaissant dans le fichier). *Scalpel* est également capable d'extraire des données directement depuis la sortie des solveurs, sans que ceux-ci n'aient à respecter un format particulier. Dans ce dernier cas, *Scalpel* permet notamment d'explorer une hiérarchie complète de fichiers de log, en extrayant les données de chacun des fichiers pertinents rencontrés au cours de cette exploration. Ceci est rendu possible par l'utilisation d'un fichier de configuration, dans lequel is est possible de décrire et de nommer les données pertinentes apparaissant dans cess fichiers, en utilisant des expressions régulières, ou des motifs simplifiés, permettant d'identifier des valeurs communes (comme des valeurs booléennes, entières, réelles, ou des mots).

Dans le cas qui nous intéresse ici, nous procédons à l'analyse des résultats de la compétition XCSP'19 à partir du fichier texte fourni sur le site web de la compétition. Il s'agit d'un fichier ayant un format particulier, que nous appellons `evaluation` dans la suite. Comme ce format est utilisé dans un certain nombre de compétitions, *Scalpel* propose un lecteur natif pour les fichiers de ce type. La configuration de *Scalpel* présentée ici est donc relativement simple, et par souci de concision, nous ne pouvons pas décrire toute la puissance de lecture offerte par ce module. Le lecteur intéressé pourra retrouver plus d'informations sur la documentation de *Metrics*[3].

Construisons maintenant la configuration de *Scalpel* pour le cas particulier de la compétition XCSP'19. Cette configuration s'effectue dans un fichier YAML. Les premières informations à saisir sont celles permettant d'identifier de manière unique la campagne.

```
name: Compétition XCSP 2019
date: 1 Octobre 2019
```

2. https://github.com/crillab/metrics
3. https://metrics.readthedocs.io

Puis, il faut donner la configuration de l'environnement où les solveurs ont été exécutés, ce qui permet de pouvoir facilement reproduire les expérimentations par la suite.

```
setup:
  os: Linux CentOS 7 (x86_64)
  cpu: Intel XEON X5550
  ram: 32GB
  timeout: 2400
  memout: 16384
```

Il faut ensuite préciser où se trouvent les fichiers que *Scalpel* doit lire, au travers du champ `source` du fichier de configuration.

```
source:
  path: path/to/XCSP19.txt
  format: evaluation
  is-success:
    - ${Checked answer} in ["SAT", "UNSAT"]
```

Ici, nous pouvons reconnaître le chemin vers le fichier de la campagne, en l'occurence `path/to/XCSP19.txt`. Notons qu'une liste de fichiers peut également être précisée, auquel cas *Scalpel* extraiera les données de chacun des fichiers de la liste. Ici, le format `evaluation` est précisé pour indiquer à *Scalpel* comment lire le fichier de la campagne, comme l'extension `.txt` est trop vague pour déterminer son format.

Attardons-nous maintenant quelques instants sur la valeur de `is-success` donnée dans cet exemple. Afin de déterminer si une expérimentation donnée s'est terminée correctement, il est possible de saisir des conditions dans un langage d'expressions relativement simple (la liste des conditions étant interprétée conjonctivement). Ici, une expérimentation est considérée comme un succès si la réponse produite par le solveur est soit `SAT`, soit `UNSAT`. Cette information se trouve dans la colonne `Checked answer` du fichier de la compétition.

Considérons maintenant ce qui est probablement la partie la plus importante du fichier de configuration, qui décrit *comment* retrouver les données à partir des fichiers fournis à *Scalpel*. Dans notre exemple, comme il s'agit d'un format reconnu par *Scalpel*, il y a relativement peu d'informations à donner dans le champ `data` présenté ici. La seule information pertinente à considérer est le `mapping`, qui permet d'identifier quelle(s) colonne(s) du fichier correspond(ent) aux données attendues par *Scalpel*.

```
data:
  mapping:
    experiment_ware:
      - Solver name
      - Solver version
      cpu_time: CPU time
      input: Instance name
```

Notons que les champs indiqués dans cet exemple doivent obligatoirement être précisés, comme les noms des colonnes ne respectent pas les conventions de nommage de *Metrics*. De plus, les informations relatives aux expérimentaciels se trouvent dans deux colonnes distinctes, d'où la liste mentionnée ici.

Comme mentionné plus haut, l'analyse de la compétition XCSP'19 ne nécessite pas de lire des fichiers de log produits au cours de la campagne, les données nécessaires ayant déjà été extraites et mises à disposition dans le fichier structuré `XCSP19.txt`. Toutefois, par souci de complétude, nous présentons brièvement comment des fichiers de log pourraient être lus par *Scalpel*. Dans ce contexte les données à extraire doivent être décrite de la manière suivante.

```
source:
  path: path/to/my/campaign
  format: dir
data:
  raw-data:
    - log-data: status
      file: execution.out
      pattern: 's {word}'
    - log-data: cpu_time
      file: execution.out
      regex: 'c real time : (\d+\.\d+)'
      group: 1
```

La configuration présentée ci-dessus permet d'extraire des données à partir d'une hiérarchie de fichiers. Dans cette hiérarchie, les fichiers produits par les solveurs lors d'une expérimentation sont placés dans un répertoire dédié à cette expérimentation. Dans ce répertoire, le fichier qui va nous intéresser est `execution.out`, qui contient deux données qui vont être extraites par *Scalpel* : le statut de la résolution (décrit via un motif simplifié) et le temps CPU de l'exécution du solveur (décrit via une expression régulière). Notons qu'il est également possible d'extraire des informations à partir du nom des fichiers ou des répertoires explorés, en utilisant le champ `file_name_meta` et en appliquant une configuration similaire à celle illustrée ici.

D'autres informations relatives à la campagne peuvent par ailleurs être précisées dans ce fichier, comme par exemple la liste des expérimentaciels et celle des entrées considérées dans la campagne. Par défaut, il n'est pas nécessaire de les préciser : *Scalpel* les découvrira automagiquement en lisant les fichiers de la campagne. Dans le cas de la compétition XCSP'19, il n'est donc pas nécessaire de préciser ces informations.

Nous remarquons cependant que cela peut s'avérer utile dans le cas où des informations pertinentes relatives aux expérimentaciels ne sont pas disponibles dans les fichiers fournis à *Scalpel* (par exemple, l'empreinte SHA des logiciels, leur ligne de commande, etc.).

Ceci termine la présentation du module *Scalpel* de *Metrics*. Conscients de la violence des informations présentées dans cette section, nous préférons vous montrer cette illustration, consacrée au logo de *Metrics*, avant de passer à la suite.

FIGURE 1 – Le logo de *Metrics*. Contrairement à une idée largement répandue, le panda et le python s'entendent très bien, notamment pour développer une bibliothèque plus humaine.

3.2.2 Exploiter les données avec *Wallet*

Une fois les données extraites, analysons-les ![4] C'est le rôle du module *wALLET* (*« Automated tooL for expLoiting Experimental resulTs »*) fourni par *Metrics*. Il permet, entre autres, de calculer des statistiques et de tracer des diagrammes communément utilisés dans le cadre de l'expérimentation de solveurs. *Wallet* est principalement conçu pour être utilisé au sein d'un *notebook Jupyter*, mais il est parfaitement envisageable de l'utiliser dans n'importe quel environnement Python convenablement configuré.

Wallet est capable de produire différents types de figures. Un premier type correspond aux graphiques statiques communément utilisés par la communauté, notamment les *scatter plots* et *cactus plots*, ainsi que leurs homologues, les *CDF plots*, que nous détaillons plus bas, et les *box plots* (aussi appelées *boîtes à moustaches*). Ces graphiques sont générés en interne par la bibliothèque *matplotlib* et peuvent être exportées sous différents formats tels que des images PNG ou vectorisées (SVG ou EPS). Ces graphiques statiques se veulent hautement personnalisables afin de s'adapter à leur environnement final (présentation, article, etc.). Entre autres, il est possible de modifier la police d'écriture, d'interpréter des formules LaTeX, de lier les expérimentaciels à différentes couleurs et différents styles, de personnaliser la légende, etc.

Un second type de figures correspond aux versions dynamisées des figures mentionnéees ci-dessus, générées grâce à la bibliothèque *plotly*. Cette bibliothèque rend possible des interactions avec l'utilisateur, par exemple, en permettant de zoomer en avant ou en arrière sur les graphiques, de sélectionner un sous-ensemble de la légende ou d'afficher des données supplémentaires par le biais d'événements tels que le survol de la souris. L'utilisateur peut aussi personnaliser ces graphiques à travers l'interface proposée par les *notebooks Jupyter* et les différents paramètres mis à disposition lors de la création des graphiques.

Enfin, *Wallet* propose aussi différentes figures sous forme tabulaire, affichant diverses statistiques discriminant les expérimentaciels sous plusieurs points de vue. Ces données tabulaires sont exportables en LaTeX.

Manipulation des données Afin de réaliser l'analyse, la première étape est de charger la campagne grâce au fichier de configuration de *Scalpel* préparée dans la précédente section :

```
from metrics.wallet import Analysis
analysis = Analysis(
  'path/to/config.yml', [...]
)
```

Une *analyse* est un objet *Metrics* permettant de gérer et de manipuler simplement les expérimentations. Par souci de clarté et de concision, nous ne présentons ici qu'une forme simplifiée [5]. Il est néanmoins possible, grâce à des informations supplémentaires fournies par l'utilisateur, de vérifier la cohérence des résultats obtenus, à la fois pour chaque expérimentation individuelle et à l'échelle d'une instance. Par exemple, il serait incohérent d'observer qu'un solveur trouve une solution pour une instance, et qu'un autre solveur identifie cette même instance comme insatisfaisable. Naturellement, l'analyse remplace les expérimentations manquantes ou incohérentes par leurs ajouts invalidés et interprétés

4. « C'est un alexandrin »

5. L'analyse complète est ici : https://github.com/crillab/metrics/tree/master/example/xcsp-19/

comme un *timeout*. L'utilisateur est prévenu de toute opération appliquée sur l'analyse et peut vérifier les instances et expérimentaciels concernés.

Une fois cette première étape consolidant les données de l'analyse effectuée, l'utilisateur peut enfin les manipuler. Dans le cadre de la compétition XCSP'19, et souhaitant nous attarder sur l'analyse de la résolution *séquentielle* de CSPs (*Problème de Satisfaction de Contraintes*), une première opération doit être réalisé :

```
analysis_no_para = analysis.filter_analysis(
    function=lambda x:
      'parallel' not in x['experiment_ware']
)
```

Cette opération permet de filtrer l'ensemble des solveurs comprenant le mot-clé *parallel* dans leur nom et de ne garder que les solveurs séquentiels. Une autre manipulation, extrayant la famille des instances, permettra d'analyser par la suite plus finement la campagne. Ici, nous produisons une nouvelle variable (une nouvelle métrique) permettant d'associer directement chacune des instances à sa famille :

```
import re
family_re = re.compile(r'^XCSP\d\d/(.*?)/')

new_analysis = analysis.add_variable(
    new_var='family',
    function=lambda x:
      family_re.match(x['input']).group(1)
)
```

Les instances de la compétition étant décrites sous forme de lien où chacune d'elles se trouve dans le dossier de sa famille, il est donc possible d'appliquer une expression régulière pour récupérer la donnée intéressante. Ainsi, la méthode `add_variable` permet de produire un nouvelle variable `family` stockant désormais la famille de l'instance. Nous proposons une dernière manipulation de l'analyse actuelle en créant un solveur virtuel :

```
from metrics.wallet
  import find_best_cpu_time_input

analysis_plus_vbs =
  analysis.add_virtual_experiment_ware(
      function=find_best_cpu_time_input,
      name='VBS'
  )
```

Parmi les solveurs virtuels, nous pouvons citer le *VBS* (pour *Virtual Best Solver*), obtenu ici en renseignant la fonction `find_best_cpu_time_input`. De manière générale, un solveur virtuel est un solveur composé d'expérimentations de solveurs réels (pour chaque instance, l'expérimentation du solveur souhaité est choisie). La particularité de la fonction importée est de choisir, pour chaque instance, l'expérimentation d'un solveur réel ayant produit le meilleur résultat (le temps le plus faible dans ce cas). L'utilisateur est libre de personnaliser cette fonction afin de produire un solveur virtuel selon d'autres critères.

Comme il n'est pas possible de présenter l'entièreté des manipulations d'analyse mise à disposition, nous résumons celles-ci par la capacité à :

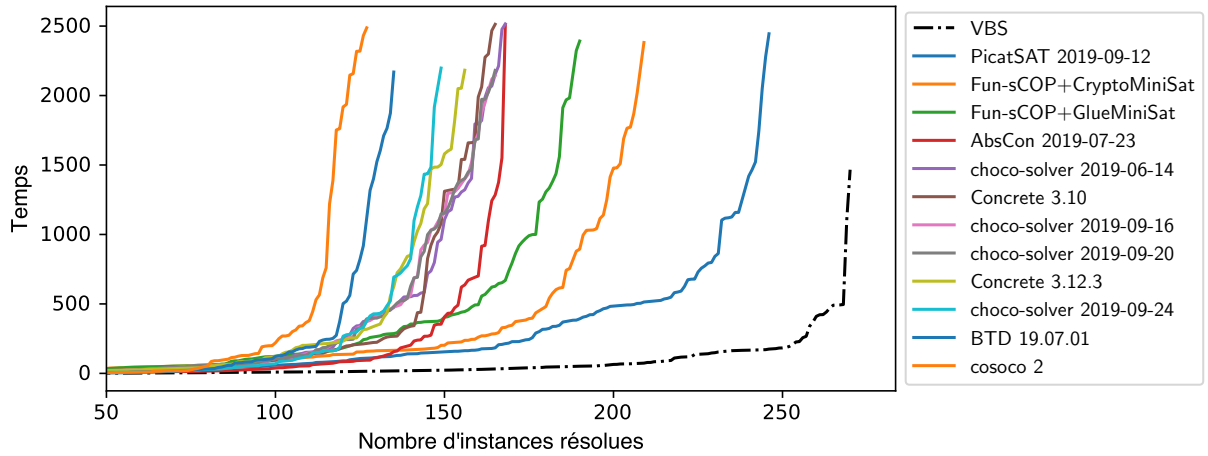
- ajouter et supprimer des variables ;
- ajouter une analyse ou une data-frame déjà existante tout en appliquant les précédents tests de cohérences ;
- filtrer l'analyse (par exemple, en supprimant des solveurs ou des instances) ;
- grouper l'analyse (par exemple, par famille) et sous-analyses ;
- produire autant d'analyse que de paires de solveurs pour les analyser plus finement.

Génération des Figures Une fois notre analyse entièrement configurée, nous pouvons construire l'ensemble des figures proposées par *Metrics*. Tout d'abord, faisons un tour d'horizon des solveurs soumis à la compétition avec un cactus plot. Pour ce faire, il suffit d'appeler la méthode `cactus_plot()` depuis l'analyse de la campagne :

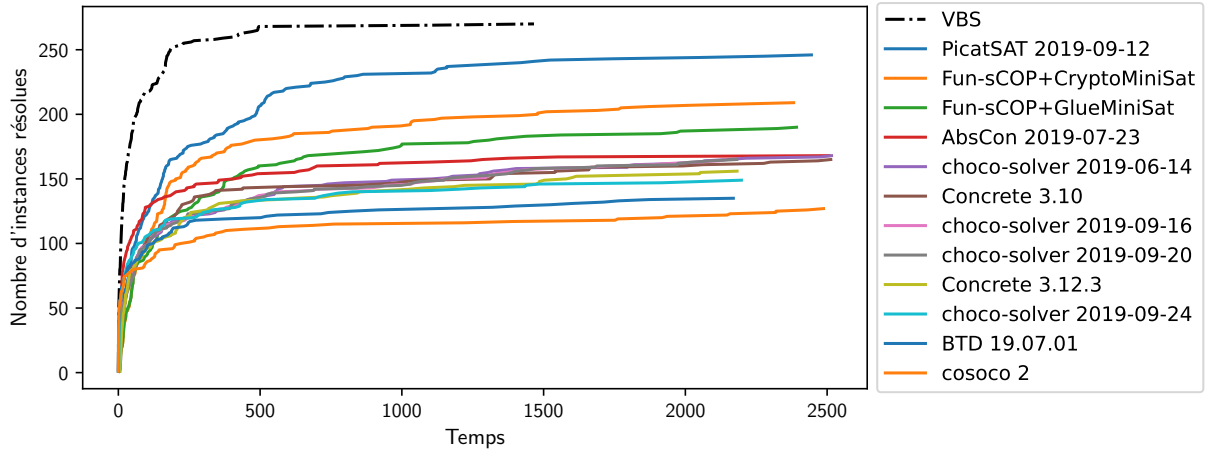
```
analysis.cactus_plot(
  cactus_col='cpu_time',
  x_min=75,
  [...]
)
```

Ici, nous avons explicitement défini `cactus_col` à `cpu_time` (la valeur par défaut) pour montrer la capacité du cactus plot (et des autres graphiques) à adapter son comportement à tout type de valeur numérique. Au-delà de ce comportement, le cactus plot et les autres graphiques produits par *Wallet* sont configurables par le biais de nombreux paramètres. En outre, il est possible de :

- changer le titre du graphique et de ses axes ;
- modifier les limites des axes x et y et changer leur échelle (en logarithmique par exemple) ;
- adapter la taille de la figure ;
- afficher des marqueurs (personnalisables selon le graphique) ;
- associer le nom des solveurs à une couleur ou un motif de tracé de courbe ;
- modifier la police d'écriture (famille, taille, couleur) ;
- interpréter du LaTeX.



(a) Cactus plot des solveurs soumis à la compétition XCSP'19.



(b) CDF plot des solveurs soumis à la compétition XCSP'19.

FIGURE 2 – Cactus plot vs. CDF plot [6]

La Figure 2a nous présente une vue globale des solveurs séquentiels soumis et expérimentés pour la compétition XCSP'19. Nous pouvons observer distinctement l'ensemble des solveurs et noter l'avance qu'a le gagnant de la compétition : le solveur *PicatSAT*. Avec plus d'avance, nous pouvons observer le VBS ayant récupéré les meilleurs contributions de chaque solveur.

Plus le nombre de solveurs est important, plus le graphique est complexe à interpréter. Dans le cas où cela se produit, il existe deux manières de répondre à ces complications. D'une part, *Wallet* est doté d'un autre module graphique permettant de passer d'un modèle statique à un modèle dynamique par le simple changement du paramètre `dynamic` à `True`. Une fois ce passage fait, le modèle dynamique dévoile des outils supplémentaires :

— ajouter ou supprimer des solveurs par simple clic sur leur nom dans la légende ;
— zoomer en avant ou en arrière sur une partie intéressante du graphique ;
— survoler pour obtenir des informations supplémentaires (le temps CPU exact, le nom d'une instance, etc.).

Notons que l'ensemble des graphiques statiques ont leur homologue dynamique. Une autre possibilité pour améliorer la lisibilité des graphiques est de restreindre l'analyse des solveurs au sous-ensemble de ces derniers qui nous intéresse dans le cadre de l'analyse.

De manière équivalente, le CDF plot (*Cumulative Distribution Function*) montre les mêmes résultats avec les axes x et y inversés et le nombre d'instances résolues (Figure 2b). Le CDF est intéressant car il est plus communément utilisé dans d'autres communautés et sa lecture est assez naturelle : l'ordre des courbes suit exactement l'ordre de la légende. Comme dit précédemment, il est également important de considérer le temps

d'exécution des différents solveurs afin de diversifier les informations à considérer. Nous pouvons observer sur ces deux graphiques que *PicatSAT* est nettement devant avec presque 250 instances de résolues.

L'extraction des métriques n'étant pas précise à travers les graphiques, il est aussi intéressant d'observer plus en détails celles-ci à travers le format tabulaire :

```
analysis.stat_table([...])
```

Par souci de concision, nous omettons certains paramètres, qui permettent de personnaliser les nombres en les entourant de dollars et en séparant les ordres de grandeur de 10^3 par des virgules. Dans leur finalité, ces tableaux sont aussi exportables au format LaTeX.

Le Tableau 1 montre, de haut en bas, les solveurs ordonnés par le nombre d'instances résolues. Évidemment, nous pouvons observer que le VBS possède le meilleur classement. Il est suivi par les solveurs *PicatSAT* et *Fun-sCOP+CryptoMiniSat*. Ce Tableau est composé d'une diversité de colonnes présentant le nombre de résolution (`count`), la somme du temps CPU (`sum`), les méthodes `PARx` pénalisant les instances non-résolues pour un solveur en multipliant son timeout par x (de ce fait, `sum` et `PAR1` sont semblables), le nombre d'instances communément résolues (`common count`) et le temps CPU pour les résoudre (`common sum`), `uncommon count` étant le nombre d'instances qu'au moins un solveur n'a pas su résoudre (celles-ci sont considérées plus compliquées) et enfin le nombre `total` d'instances.

Une autre information intéressante à extraire du VBS est la contribution de chaque solveur qui le compose. Le Tableau 2 fournit la contribution de l'ensemble des solveurs en comparaison au VBS. Les quatre premières colonnes correspondent au VBS contraint par une *delta* minimale en secondes. Par exemple, $0s$ correspond au VBS original, tandis que $100s$ correspond aux instances pour lesquelles le meilleur solveur possède une avance de 100 secondes sur le second meilleur solveur. Ainsi, ces quatre colonnes exhibent la distribution des instances communes entre le VBS et les solveurs de la compétition en fonction de ce delta de temps. La dernière colonne `contribution` correspond au nombre d'instances que seul le solveur courant a réussi à résoudre. Ainsi, nous pouvons observer que *PicatSAT* est le plus grand contributeur concernant le monopole de résolutions : il a le plus grand nombre d'instances que seul lui a résolu. Nous pouvons aussi remarquer que *PicatSAT* et *Fun-sCOP+CryptoMiniSat* ont respectivement résolu 25 et 18 instances 100 secondes plus rapidement que le second meilleur solveur (pour chaque instance).

Une autre façon d'observer en détail le comportement de *PicatSAT* et *Fun-sCOP+CryptoMiniSat* est

6. « Le cactus ne s'associe pas avec le CDF. »

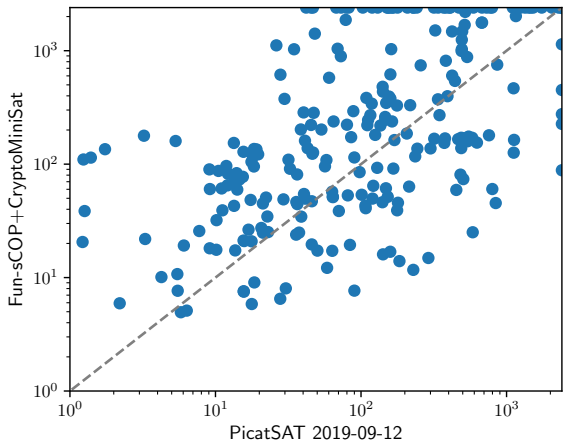


FIGURE 3 – Scatter plot des deux meilleurs solveurs de la compétition XCSP'19.

d'utiliser un scatter plot :

```
analysis.scatter_plot(
  scatter_col='cpu_time',
  [...]
)
```

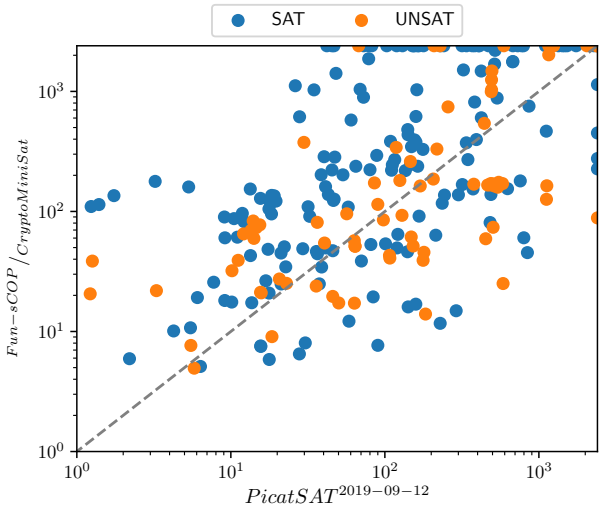


FIGURE 4 – Scatter plot des deux meilleurs solveurs de la compétition XCSP'19.

Le scatter plot, tel qu'illustré dans la Figure 3, montre que beaucoup d'instances résolues rapidement par *PicatSAT* sont en *timeout* pour *Fun-sCOP+CryptoMiniSat*. Si nous omettons les instances en *timeout*, il y a peu de différences entre les deux solveurs. L'utilisation du paramètre `color_col="Checked answer"`, dans la Figure 4, met en évidence les instances *SAT*/*UNSAT* et permet

	count	sum	PAR1	PAR2	PAR10	common count	common sum	uncommon count	total
VBS	270	90,388	90,388	162,388	738,388	65	405	205	300
PicatSAT 2019-09-12	246	192,377	192,377	321,977	1,358,777	65	11,093	181	300
Fun-sCOP hybrid+CryptoMiniSat (2019-06-15)	209	274,323	274,323	492,723	2,239,923	65	16,472	144	300
Fun-sCOP order+GlueMiniSat (2019-06-15)	190	320,070	320,070	584,070	2,696,070	65	14,632	125	300
AbsCon 2019-07-23	168	341,387	341,387	658,187	3,192,587	65	2,805	103	300
choco-solver 2019-06-14	168	369,846	369,846	686,646	3,221,046	65	7,875	103	300
Concrete 3.10	165	369,615	369,615	693,615	3,285,615	65	5,182	100	300
choco-solver 2019-09-16	165	372,266	372,266	696,266	3,288,266	65	7,790	100	300
choco-solver 2019-09-20	165	372,316	372,316	696,316	3,288,316	65	7,754	100	300
Concrete 3.12.3	156	386,276	386,276	731,876	3,496,676	65	7,198	91	300
choco-solver 2019-09-24	149	390,634	390,634	753,034	3,652,234	65	2,570	84	300
BTD 19.07.01	135	421,087	421,087	817,087	3,985,087	65	6,718	70	300
cosoco 2	127	448,425	448,425	863,625	4,185,225	65	6,810	62	300

TABLE 1 – Tableau des instances résolues et des statistiques de temps d'exécution des principaux solveurs de la compétition XCSP'19.

	vbew simple	vbew 1s	vbew 10s	vbew 100s	contribution
PicatSAT 2019-09-12	70	65	49	25	3
BTD 19.07.01	49	29	4	1	0
Fun-sCOP hybrid+CryptoMiniSat (2019-06-15)	41	37	30	18	1
cosoco 2	41	25	14	1	0
AbsCon 2019-07-23	17	16	11	2	0
choco-solver 2019-09-24	16	13	7	2	0
Fun-sCOP order+GlueMiniSat (2019-06-15)	15	10	8	2	0
Concrete 3.10	6	5	2	1	0
choco-solver 2019-06-14	6	5	4	1	0
Concrete 3.12.3	4	2	1	0	0
choco-solver 2019-09-16	4	2	0	0	0
choco-solver 2019-09-20	1	1	0	0	0

TABLE 2 – Tableau de contribution des principaux solveurs de la compétition XCSP'19.

de montrer, par exemple, que les *timeout* de *Fun-sCOP+CryptoMiniSat* sont le plus souvent sur des instances *SAT*. Cette même figure montre quelques manipulations possibles dans la transformation des noms de solveurs avec interprétation LaTeX. Enfin, *Metrics* donne la possibilité de survoler les points avec la souris et d'afficher les métadonnées des différentes instances (pour la version dynamique du scatter plot avec l'option `dynamic=True`). Cet outil peut être très pratique pour obtenir plus d'informations sur les instances en *timeout* par exemple.

Enfin, nous présentons un nouveau type de graphique qui complète les informations fournies par le cactus plot : le box plot. Ce graphique affiche des informations sur la moyenne et les quartiles. Comme précédemment, nous avons juste besoin d'appeler la méthode `box_plot()` de l'analyse que nous souhaitons observer :

```
analysis.box_plot(
  box_col='cpu_time',
  [...]
)
```

Les box plots résultants sont présentés dans la Figure 5. Chaque box plot est composé, de gauche à droite, de la valeur minimale, du premier et du troisième quartile (la boîte) et du maximum. Les valeurs aberrantes sont représentées par des cercles.

Nous pouvons observer que le premier solveur à résoudre des instances est *PicatSAT*. Dans la box, entre le premier et troisième quartile nous pouvons observer la médiane. Nous observons aussi que *PicatSAT* est le seul solveur à ne pas avoir son troisième quartile (représentant donc 75% des instances) confondu avec la moustache droite représentant la valeur maximale calculée.

Enfin, à travers les statistiques et les chiffres fournis par *Metrics*, nous observons que posséder une vision globale de la campagne pour la compétition XCSP'19 est important. Grâce aux opérations de filtrage, de génération de vues et de statistiques, nous avons rapidement identifié les meilleurs solveurs et avons pu les comparer pour obtenir des conclusions plus précises. Il reste difficile de trouver une unique méthode pour départager les solveurs à travers leurs performances : c'est pourquoi une diversité d'outils ont été conçus pour clarifier la situation.

4 Conclusion

Dans cet article, nous avons présenté *Metrics*, une bibliothèque fournissant une chaîne d'outils simple d'utilisation pour collecter et analyser des résultats expérimentaux en calculant des statistiques et en traçant différents types de diagrammes. Les utilisateurs sont

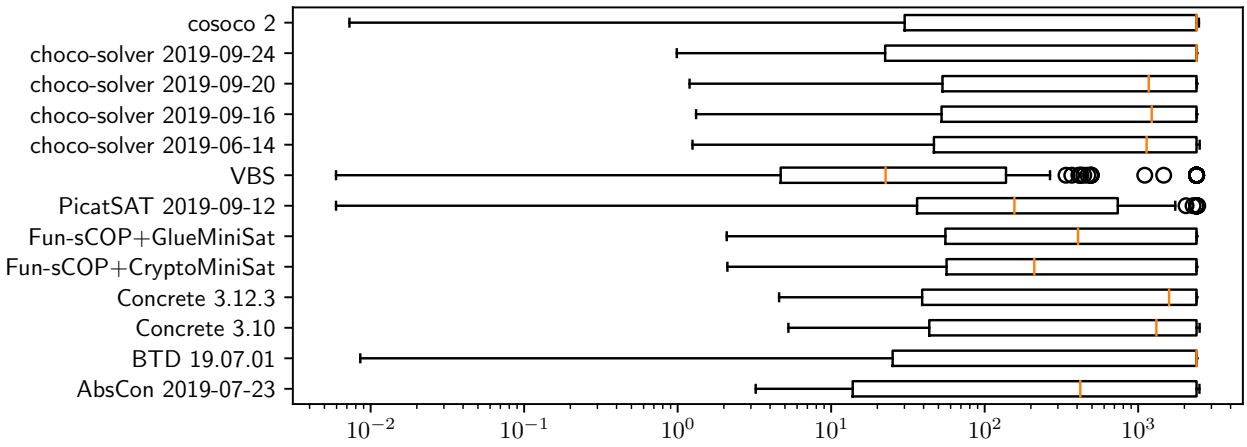


FIGURE 5 – Box plots des temps d'exécution des meilleurs solveurs de la compétition XCSP'19.

laissés libres d'organiser et de personnaliser l'analyse produite selon leurs besoins grâce à l'utilisation de *notebooks Jupyter*. Ceci permet de partager les résultats des expérimentations réalisées tout en rendant possible la reproductibilité de l'analyse.

A ce jour, *Metrics* est un projet *jeune*, qui propose un œil neuf sur la manière d'analyser ses expérimentations. Nous envisageons d'ajouter de nouvelles fonctionnalités à cette bibliothèque, de sorte à permettre des analyses plus approfondies des résultats, par exemple en permettant l'utilisation d'autres types de figures (par exemple, dans le cas particulier des problèmes d'optimisation). Bien que les utilisateurs peuvent utiliser le *data-frame* de la campagne et réaliser leurs analyses avec les fonctions déjà fournies par *pandas* et *matplotlib*, nous pensons que *Metrics* doit être en mesure de fournir ces fonctionnalités par défaut, afin de simplifier les actions à réaliser par l'utilisateur.

De plus, pour devenir une chaîne *complète* d'outils, *Metrics* a l'ambition d'étendre ses capacités en proposant une interface en lignes de commande ainsi qu'une interface web au travers desquelles il serait possible d'automatiser le processus d'exécution des solveurs et celui de collecte et d'analyse des données. En particulier, en configurant leurs installations de *Metrics* suivant leurs besoins, les utilisateurs pourront soumettre leur(s) logiciel(s) directement dans l'application, de sorte que toutes les étapes entre l'exécution du solveur (éventuellement, sur des machines distantes, sur un cluster ou dans le *cloud*) à la production d'un rapport complet, en minimisant les efforts à fournir par l'utilisateur. Notre but ultime est de faire de *Metrics* un outil incontournable : pas de *Metrics*, pas d'expé', pas d'expé', pas de papier, pas de papier… pas de papier !

Références

[1] 2019 XCSP3 Competition. http://www.cril.univ-artois.fr/XCSP19/, 2019 (accessed April 13, 2021).

[2] Gilles AUDEMARD, Frédéric BOUSSEMART, Christophe LECOUTRE, Cédric PIETTE et Olivier ROUSSEL : Xcsp^3 and its ecosystem. *Constraints An Int. J.*, 25(1-2):47–69, 2020.

[3] Juliana FREIRE, Norbert FUHR et Andreas RAUBER : Reproducibility of Data-Oriented Experiments in e-Science (Dagstuhl Seminar 16041). *Dagstuhl Reports*, 6(1):108–159, 2016.

[4] Gael GLORIAN, Jean-Marie LAGNIEZ et Christophe LECOUTRE : NACRE - A nogood and clause reasoning engine. *In* Elvira ALBERT et Laura KOVÁCS, éditeurs : *LPAR 2020 : 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020*, volume 73 de *EPiC Series in Computing*, pages 249–259. EasyChair, 2020.

[5] Yang-Min KIM, Jean-Baptiste POLINE et Guillaume DUMAS : Experimenting with reproducibility : a case study of robustness in bioinformatics. *GigaScience*, 7(7):giy077, juillet 2018.

[6] Dirk PILAT et Yukiko FUKASAKU : Oecd principles and guidelines for access to research data from public funding. *Data Science Journal*, 6:4–11, 06 2007.

[7] Olivier ROUSSEL : Controlling a Solver Execution : the runsolver Tool. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:139–144, 2011.

Actes JFPC 2021

Un bandit manchot pour combiner CHB et VSIDS*

Mohamed Sami Cherif† **Djamal Habet** **Cyril Terrioux**

Aix Marseille Univ, Université de Toulon, CNRS, LIS, Marseille, France
{mohamed-sami.cherif, djamal.habet, cyril.terrioux}@univ-amu.fr

Résumé

Les solveurs *Conflict Driven Clause Learning* (CDCL) sont efficaces pour résoudre des instances structurées avec un grand nombre de variables et de clauses. Un composant important de ces solveurs est l'heuristique de branchement qui sélectionne la prochaine variable de décision. Dans cet article, on propose d'utiliser un bandit manchot pour combiner deux heuristiques de l'état de l'art pour SAT, à savoir *Variable State Independent Decaying Sum* (VSIDS) et *Conflict History-Based* (CHB). Le bandit évalue et choisit de manière adaptative une heuristique adéquate à chaque redémarrage. Une évaluation expérimentale est menée et montre que la combinaison de VSIDS et CHB avec un bandit est compétitive et surpasse les deux heuristiques.

Abstract

Conflict Driven Clause Learning (CDCL) solvers are known to be efficient on structured instances and manage to solve ones with a large number of variables and clauses. An important component in such solvers is the branching heuristic which picks the next variable to branch on. In this paper, we propose a Multi-Armed Bandit (MAB) framework to combine two state-of-the-art heuristics for SAT, namely the Variable State Independent Decaying Sum (VSIDS) and the Conflict History-Based (CHB) branching heuristic. The MAB takes advantage of the restart mechanism to evaluate and adaptively choose an adequate heuristic. We conduct an experimental evaluation which shows that the combination of VSIDS and CHB using MAB is competitive and even outperforms both heuristics.

1 Introduction

Résoudre le problème de Satisfiabilité (SAT) consiste à déterminer, étant donné une formule booléenne en Forme Normale Conjonctive (FNC), s'il existe une affectation des variables qui la satisfait. SAT est au cœur de nombreuses applications dans différents domaines et est utilisé pour modéliser une grande variété de problèmes académiques et industriels [25, 11, 18]. C'est aussi le premier problème de décision prouvé NP-complet [10]. Néanmoins, les solveurs modernes, appelés Conflict Driven Clause Learning (CDCL) [26], parviennent à résoudre des instances impliquant un grand nombre de variables et de clauses. Un composant important de ces solveurs est l'heuristique de branchement qui sélectionne la prochaine variable sur laquelle brancher. Variable State Independent Decaying Sum (VSIDS) [27] est l'heuristique dominante depuis son introduction il y a deux décennies. Récemment, Liang et al. ont conçu une nouvelle heuristique pour SAT, appelée Conflict History-Based (CHB) [22], et ont montré qu'elle est compétitive avec VSIDS. Ces dernières années, la majorité des solveurs CDCL présentés dans les compétitions SAT incorporent une variante d'une de ces deux heuristiques.

Des travaux récents ont montré l'intérêt de l'apprentissage automatique dans la conception d'heuristiques de recherche efficaces pour SAT [22, 23, 19] ainsi que pour d'autres problèmes de décision [33, 32, 28, 8]. L'un des principaux défis est de définir une heuristique avec des performances élevées sur n'importe quelle instance considérée. En effet, une heuristique peut très bien fonctionner sur une famille d'instances tout en échouant drastiquement sur une autre. Pour cela, on utilise l'apprentissage par renforcement sous la forme de bandit manchot afin de choisir une heuristique adéquate parmi CHB et VSIDS pour chaque instance. Le bandit évalue et choisit de manière adaptative une heuristique adéquate à chaque redémarrage. L'évaluation est réalisée grâce à une fonction de récompense, qui doit estimer l'efficacité d'une heuristique en se basant sur les informations acquises lors des exécutions entre les redémarrages. On a choisi une fonction de récompense

*Ce travail est soutenu par l'Agence Nationale de la Recherche dans le cadre du projet DEMOGRAPH (ANR-16-CE40-0028).

†Papier doctorant : Mohamed Sami Cherif est auteur principal.

qui estime la capacité d'une heuristique à atteindre des conflits rapidement et efficacement. Le bandit utilise la stratégie Upper Confidence Bound (UCB) [4] pour sélectionner le bras adéquat à chaque redémarrage. L'évaluation expérimentale menée montre que le bandit réalise un gain substantiel, principalement en termes d'instances satisfiables, par rapport à VSIDS et CHB.

Cet article est organisé de la manière suivante. Un aperçu des algorithmes CDCL est donné dans la section 2. Les heuristiques VSIDS et CHB ainsi que le problème du bandit manchot sont décrits dans la section 3. L'idée proposée est détaillée dans la section 4 et évaluée expérimentalement dans la section 5. Enfin, on conclut et on discute les perspectives de ce travail dans la section 6.

2 Préliminaires

Soit X un ensemble des variables propositionnelles. Un littéral est une variable $x \in X$ ou sa négation $\overline{x}$. Une clause est une disjonction de littéraux. Une formule en Forme Normale Conjonctive (FNC) est une conjonction de clauses. Une affectation $I : X \rightarrow \{vrai, faux\}$ associe à chaque variable une valeur booléenne et peut être représentée comme un ensemble de littéraux. un littéral est satisfait par une affectation I si $l \in I$, sinon il est falsifié par I. Une clause est satisfaite par une affectation I si au moins un de ses littéraux est satisfait par I, sinon elle est falsifiée par I. Une formule FNC est satisfiable s'il existe une affectation qui satisfait toutes ses clauses, sinon elle est insatisfiable. La résolution du problème de Satisfiabilité (SAT) consiste à déterminer si une formule FNC donnée est satisfiable.

Bien que SAT soit NP-complet [10], les solveurs CDCL (Conflict Driven Clause Learning) sont efficaces et parviennent à résoudre des instances impliquant un grand nombre de variables et de clauses. Ces solveurs reposent sur des heuristiques de branchement puissantes ainsi que sur plusieurs techniques de résolution, à savoir la propagation unitaire, l'apprentissage de clauses et les redémarrages, entre autres. À chaque étape, la propagation unitaire est appliquée pour simplifier la formule en propageant les littéraux dans les clauses unitaires. Ensuite, une heuristique de branchement sélectionne une variable en fonction des informations acquises tout au long de la recherche. Plus important encore, lorsqu'un conflit est détecté, c'est-à-dire qu'une clause est falsifiée par l'affectation en cours, les étapes de l'algorithme sont retracées et les clauses impliquées dans le conflit sont résolues jusqu'au premier point d'implication unitaire (FUIP) dans le graphe d'implication [26]. La clause produite par ce processus est apprise, c'est-à-dire ajoutée à la formule. Cela permet d'éviter de revisiter un sous-espace de l'arborescence de recherche déjà exploré. Initialement introduits pour traiter les phénomènes à queue lourde (heavy-tailed) dans SAT [13], les redémarrages sont également un composant important dans les solveurs CDCL. Au début de chaque redémarrage, les paramètres du solveur et ses structures de données sont réinitialisés afin de lancer l'exploration ailleurs dans l'espace de recherche. Il existe deux stratégies principales de redémarrage, à savoir les redémarrages géométriques [31] et Luby [24]. La plupart des solveurs CDCL modernes utilisent les redémarrages Luby car ils sont plus performants [14].

3 Travaux connexes

3.1 Heuristiques de branchement pour SAT

L'heuristique de branchement est l'un des composants les plus importants des solveurs CDCL modernes et a un impact direct sur leur efficacité. Une heuristique de branchement peut être considérée comme une fonction qui classe les variables à l'aide d'un score mise à jour tout au long de la recherche. Dans cette section, on décrit deux des principales heuristiques de branchement de l'état de l'art.

3.1.1 VSIDS

Variable State Independent Decaying Sum (VSIDS) [27] est l'heuristique la plus utilisée depuis son introduction il y a environ deux décennies. Cette heuristique maintient un score pour chaque variable, appelé activité et initialement fixé à 0. Lorsqu'un conflit survient, l'activité de certaines variables est augmentée de 1 (bump). De plus, les activités des variables sont réduites (decay) périodiquement, généralement après chaque conflit. Plus précisément, les activités des variables sont multipliées par un facteur appartenant à $]0, 1[$. Il existe plusieurs variantes de VSIDS. Par exemple, MiniSat [12] augmente les activités des variables apparaissant dans la clause apprise tandis que Chaff [27] le fait pour toutes les variables impliquées dans le conflit, c'est-à-dire les variables résolues y compris celles de la clause apprise.

3.1.2 CHB

L'heuristique de branchement CHB (Conflict History-Based) a été récemment introduite dans [22]. Cette heuristique, basée sur Exponential Recency Weighted Average (ERWA) [29], favorise les variables impliquées dans les conflits récents comme dans VSIDS. CHB maintient un score (ou une activité) $Q(x)$ pour chaque variable x, initialement fixé à 0. Le score $Q(x)$ est mis

à jour lorsqu'une variable x est branchée, propagée ou affirmée à l'aide de la formule d'ERWA comme suit :

$$Q(x) = (1 - \alpha) \times Q(x) + \alpha \times r(x)$$

Le paramètre $0 < \alpha < 1$ est la taille du pas, initialement fixée à $0,4$ et diminuée de 10^{-6} après chaque conflit jusqu'à un minimum de $0,06$. $r(x)$ est la valeur de récompense de la variable x qui peut diminuer ou augmenter la probabilité de brancher sur x. Des récompenses plus élevées sont accordées aux variables impliquées dans des conflits récents selon la formule suivante :

$$r(x) = \frac{multiplier}{Conflicts - lastConflict(x) + 1}$$

$Conflicts$ indique le nombre de conflits survenus depuis le début de la recherche. $lastConflict(x)$ est mis à jour à la valeur actuelle de $Conflicts$ chaque fois que x est présent dans les clauses analysées durant un conflit. $multiplier$ est fixé à 1,0 lorsque le branchement, la propagation ou l'affirmation de la variable qui a déclenché la mise à jour du score conduit à un conflit, sinon il est réglé à 0,9. L'idée est de donner des récompenses supplémentaires pour les variables qui produisent un conflit.

3.2 Bandit Manchot

Le bandit manchot est un problème d'apprentissage par renforcement constitué d'un agent et d'un ensemble de bras candidats parmi lesquels l'agent doit choisir tout en maximisant le gain attendu. L'agent s'appuie sur des informations sous forme de récompenses données à chaque bras et collectées à travers une séquence d'essais. Un bandit est toujours confronté à un dilemme important qui est le compromis entre l'exploitation et l'exploration. En effet, l'agent doit explorer les bras sous-utilisés assez souvent pour avoir un retour d'information solide tout en exploitant les bons candidats qui ont les meilleures récompenses. Le premier modèle du bandit manchot, appelé bandit manchot stochastique, a été introduit dans [20]. Ensuite, différentes politiques ont été conçues pour le bandit manchot telles que la stratégie ϵ-Greedy [29], qui effectue une exploration aléatoire, Thompson Sampling [30] et la famille Upper Confidence Bound (UCB) [1, 4], qui permettent une exploration plus intelligente.

Ces dernières années, il y a eu un afflux d'intérêt pour l'application des techniques d'apprentissage par renforcement et, en particulier, celles basées sur le bandit manchot dans le contexte de SAT. Par exemple, CHB [22] et LRB [23] (une extension de CHB) sont des heuristiques basées sur ERWA [29] qui est utilisée dans le cadre des problèmes bandit manchot non stationnaires pour estimer les récompenses moyennes de chaque bras. De plus, une nouvelle approche, appelée Bandit Ensemble for parallel SAT Solving (BESS), a été conçue dans [21] pour contrôler la topologie de coopération dans des solveurs SAT parallèles. Des approches similaires sont également utilisées dans le contexte du problèmes de satisfaction de contraintes (CSP) pour choisir une heuristique de branchement parmi un ensemble de candidats à chaque nœud de l'arbre de recherche [33] ou à chaque redémarrage [32]. Enfin, des stratégies de perturbation simples pilotées par des bandits ont également été introduites et évaluées dans [28] pour incorporer des choix aléatoires dans la résolution de contraintes avec redémarrages.

4 Bandit Manchot pour SAT

Soit $A = \{a_1, \ldots, a_K\}$ l'ensemble des bras du bandit contenant différentes heuristiques candidates. Le bandit sélectionne une heuristique a_i où $i \in \{1 \ldots K\}$ à chaque redémarrage de l'algorithme conformément à la politique Upper Confidence Bound (UCB) [4]. Pour choisir un bras, UCB s'appuie sur une fonction de récompense calculée à chaque exécution pour estimer les performances des heuristiques de branchement. La fonction de récompense joue un rôle important dans le cadre proposé et a un impact direct sur son efficacité. Pour cela, on a choisi une fonction de récompense qui estime la capacité d'une heuristique à atteindre des conflits rapidement et efficacement. Si t désigne l'exécution en cours, la récompense du bras $a \in A$ est calculée comme suit :

$$r_t(a) = \frac{log_2(decisions_t)}{decidedVars_t}$$

$decisions_t$ et $decidedVars_t$ désignent respectivement le nombre de décisions et le nombre de variables décidées, c'est-à-dire les variables sur lesquelles l'algorithme a branché au moins une fois au cours de l'exécution t. Par conséquent, plus les conflits sont rencontrés tôt dans l'arbre de recherche et moins il y a de variables instanciées, plus la valeur de récompense attribuée sera élevée pour l'heuristique correspondante. $r_t(a)$ est clairement dans $[0, 1]$ puisque $decisions_t \leq 2^{decidedVars_t}$. Cette fonction de récompense est adaptée de la mesure de sous-arbre explorée (explored sub-tree) introduite dans [28].

L'algorithme UCB1 [4] est utilisé pour sélectionner l'heuristique de branchement suivante dans l'ensemble des heuristiques candidates A. Les paramètres suivants sont conservés pour chaque bras candidat $a \in A$:

- $n_t(a)$: le nombre de fois où le bras a est sélectionné pendant les t premières exécutions
- $\widehat{r}_t(a)$: la moyenne empirique des récompenses du bras a sur les t premières exécutions

UCB1 sélectionne donc le bras $a \in A$ qui maximise $UCB(a)$ défini ci-dessous. Le terme de gauche de $UCB(a)$ vise à mettre l'accent sur les bras qui ont reçu les récompenses les plus élevées. À l'inverse, le terme de droite assure l'exploration des bras sous-utilisés. Le paramètre c peut aider à équilibrer de manière appropriée les phases d'exploitation et d'exploration.

$$UCB(a) = \widehat{r}_t(a) + c.\sqrt{\frac{ln(t)}{n_t(a)}}$$

Enfin, une stratégie pour le bandit manchot est évaluée par son regret cumulé espéré, c'est-à-dire la différence entre la valeur cumulée espérée de la récompense si le meilleur bras est utilisé à chaque redémarrage et sa valeur cumulée réelle pour toutes les exécutions. Le regret cumulé espéré R_T, où T est le nombre total d'exécutions et $a_t \in A$ désigne le bras choisi à l'exécution t, est formellement défini ci-dessous. En particulier, UCB1 garantit un regret cumulé en $O(\sqrt{K.T.\ln T})$.

$$R_T = \max_{a \in A} \sum_{t=1}^{T} \mathbf{E}[r_t(a)] - \sum_{t=1}^{T} \mathbf{E}[r_t(a_t)]$$

5 Évaluation Expérimentale

5.1 Protocole Expérimental

Dans cette section, on décrit l'évaluation expérimentale de la performance du bandit manchot pour combiner VSIDS et CHB. On considère les instances du Main Track des trois dernières compétitions SAT, totalisant 1 200 instances. Pour les tests, on utilise le solveur Kissat [7] qui a remporté la première place du Main Track de la Competition SAT 2020. Il faut noter que ce solveur est une réimplémentation condensée et améliorée du solveur référence et compétitif CaDiCaL [6, 7] en langage C. Des données fournies par A. Bierre et M. Heule [1] montrent que Kissat est très compétitif et surpasse les vainqueurs des compétitions antérieures sur les benchmarks de 2020 et 2019.

Pour le bandit manchot, on fixe $K = 2$ et on considère les heuristiques de l'état de l'art introduites dans la section 3, à savoir VSIDS et CHB. Ce choix sera discuté à la fin de la section 5.4. On maintient la variante VSIDS déjà implémentée dans Kissat qui est similaire à Chaff où toutes les variables analysées sont bumpées après chaque conflit [27]. On augmente également le solveur avec l'heuristique CHB comme spécifié dans [22] sauf la mise à jour des scores qu'on réalise seulement pour les variables au dernier niveau de décision après la propagation unitaire. De plus, on a implémenté le bandit manchot, qu'on dénotera MAB (pour Multi-Armed Bandit), comme spécifié dans la section 4 avec c fixé hors ligne à 2 après une recherche linéaire de la meilleure valeur. Les récompenses dans UCB sont initialisées en lançant chaque heuristique une fois lors des premiers redémarrages. Il est important de noter que les seuls composants modifiés du solveur sont le composant de décision et le composant de redémarrage, c'est-à-dire que tous les autres composants ainsi que les paramètres par défaut du solveur restent inchangés. Les expériences sont réalisées sur des serveurs Dell PowerEdge M620 avec des processeurs Intel Xeon Silver E5-2609 sous Ubuntu 18.04. Enfin, nous utilisons une limite de temps de 5 000 s pour chaque instance.

5.2 MAB vs (VSIDS,CHB)

Dans le tableau 1, on présente les résultats en termes du nombre d'instances résolues pour CHB, VSIDS et MAB. On inclut également les résultats du Virtual Best Solver (VBS) entre VSIDS et CHB. Avant de discuter des résultats, on rappelle que « l'amélioration des solveurs SAT est souvent un monde cruel. Pour donner une idée, améliorer un solveur en résolvant au moins dix instances supplémentaires (sur un ensemble fixe de benchmarks d'une compétition) est généralement considéré comme une nouvelle fonctionnalité critique. En général, le gagnant d'une compétition est choisi sur la base de quelques instances supplémentaires résolus » [2]. Les résultats indiquent clairement que MAB surpasse les deux heuristiques. En effet, MAB parvient à résoudre 30 instances supplémentaires au total (+3,9%) par rapport à la meilleure heuristique. De plus, bien que les résultats globaux obtenus par VSIDS et CHB soient comparables, ils peuvent avoir des comportements différents sur les benchmarks individuels. Poutant, MAB parvient à capturer le comportement de la meilleure heuristique et même à le surpasser pour chaque benchmark individuel. Les résultats obtenus par MAB sont également très proches du VBS. En particulier, MAB réalise plus de 98% (resp. 99%) des performances du VBS sur l'ensemble du benchmark en termes de nombre d'instances résolues (resp. nombre d'instances satisfiables résolues) alors que la meilleure heuristique ne dépasse pas 95% (resp. 93%).

Cependant, il est important de noter que le gain est principalement au niveau des instances satisfiables alors que, pour les instances insatisfiables, MAB surpasse CHB mais pas VSIDS. Néanmoins, MAB reste compétitif avec VSIDS car il ne résout que 3 instances de moins, avec une instance de moins seulement chaque année. Cela équivaut à une perte de performance de 0,9% par rapport aux instances insatisfiables alors que

1. http ://fmv.jku.at/kissat/

2. traduite à partir de [3] de G. Audemard et L. Simon.

		VSIDS	CHB	RD_N	RD_R	MAB	VBS
Compétition 2018 (400 instances)	SAT	160	159	160	164	**167**	169
	UNSAT	**111**	109	109	110	110	113
	TOTAL	271	268	268	274	**277**	282
Race 2019 (400 instances)	SAT	158	149	155	158	**161**	162
	UNSAT	**97**	95	95	96	96	99
	TOTAL	255	244	250	254	**257**	261
Compétition 2020 (400 instances)	SAT	131	146	146	151	**154**	157
	UNSAT	**121**	119	117	120	120	123
	TOTAL	252	265	263	271	**274**	280
TOTAL (1 200 instances)	SAT	449	454	461	473	**482**	488
	UNSAT	**329**	323	321	326	326	335
	BOTH	778	777	782	799	**808**	823

TABLE 1 – Comparaison entre VSIDS, CHB, RD_N, RD_R, MAB et VBS en termes de nombre d'instances résolues dans Kissat. Pour chaque ligne, les meilleurs résultats obtenus sans considérer le VBS sont écrits en gras.

MAB réalise un gain de 6,7% en termes d'instances satisfiables. Ceci peut être dû à de nombreux facteurs. En effet, les résultats en termes d'instances insatisfiables semblent très homogènes pour chaque année et sont très proches des résultats obtenus par le VBS car les deux heuristiques (resp. la meilleure heuristique) atteignent plus de 96% (resp. 98%) de ses performances. Un autre facteur pourrait être la politique de redémarrage de Kissat qui alterne entre les phases stables et non stables comme c'est le cas dans CaDiCaL [6], renommé en mode stable et mode focalisé dans [7]. Les heuristiques mentionnées ne sont utilisées que dans les phases stables qui ciblent principalement les instances satisfiables. Pendant les phases focalisés, le solveur ne calcule pas les scores et déplace en revanche les variables analysées au début de la file de priorités des décisions. Cela peut également aider à expliquer l'homogénéité des résultats obtenus par les différentes heuristiques (y compris MAB) pour les instances insatisfiables. Un troisième facteur possible, quoique dans une moindre mesure, peut être la fonction de récompense qui oriente en partie la recherche de manière à générer plus de conflits. En effet, générer plus de conflits et donc apprendre plus de clauses sans contrôler leur qualité peut impacter négativement la performance du solveur sur les instances insatisfiables, pour lesquelles il doit explorer tout l'espace de recherche. Or, bien que la fonction de récompense ait été conçue pour générer efficacement des conflits, elle ne peut pas garantir une bonne qualité des clauses apprises.

Dans la suite, on veut évaluer MAB en termes de temps de résolution. Puisque UCB1 mène une exploration continue afin de garantir la sélection du bras le plus adéquat à chaque redémarrage, on pourrait s'attendre à des performances moindre du MAB comparé aux heuristiques considérées en terme de temps de résolution. Or, cela ne semble pas être le cas. En effet, conduire

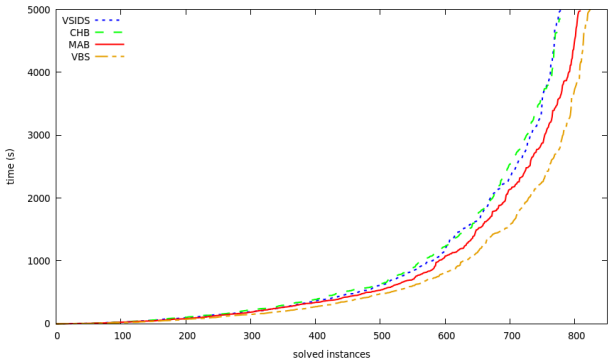


FIGURE 1 – Temps CPU (en secondes) en fonction des instances résolues pour VSIDS, CHB, MAB et VBS.

l'exploitation avec le meilleur bras et alterner les heuristiques semble compenser cet inconvénient. Dans la figure 1, on représente le temps de résolution en fonction des instances résolues du benchmark. On observe que MAB surpasse à la fois VSIDS et CHB. En fait, MAB réalise plus de 4% (resp. 5%) de gain en termes de temps de résolution sur l'ensemble du benchmark par rapport à la meilleure heuristique si l'on donne une pénalité de 5 000 s (resp. 10 000 s, ce qui équivaut à deux fois le temps limite) aux instances non résolues. Ce gain est important d'autant plus qu'on réalise les tests avec le solveur compétitif Kissat, vainqueur de la compétition SAT 2020. Un autre résultat intéressant est la performance de MAB sur des instances « dures », c'est-à-dire dont le temps de résolution dépasse 4 000 s, qui se rapproche beaucoup du VBS comme le montre la figure 1. Les temps de résolution de VSIDS, CHB et MAB sont respectivement 110%, 110,9% et 105,6% plus élevés que le VBS sur l'ensemble du benchmark. Ces résultats montrent que MAB réalise un gain substantiel non seulement en termes de nombre d'instances

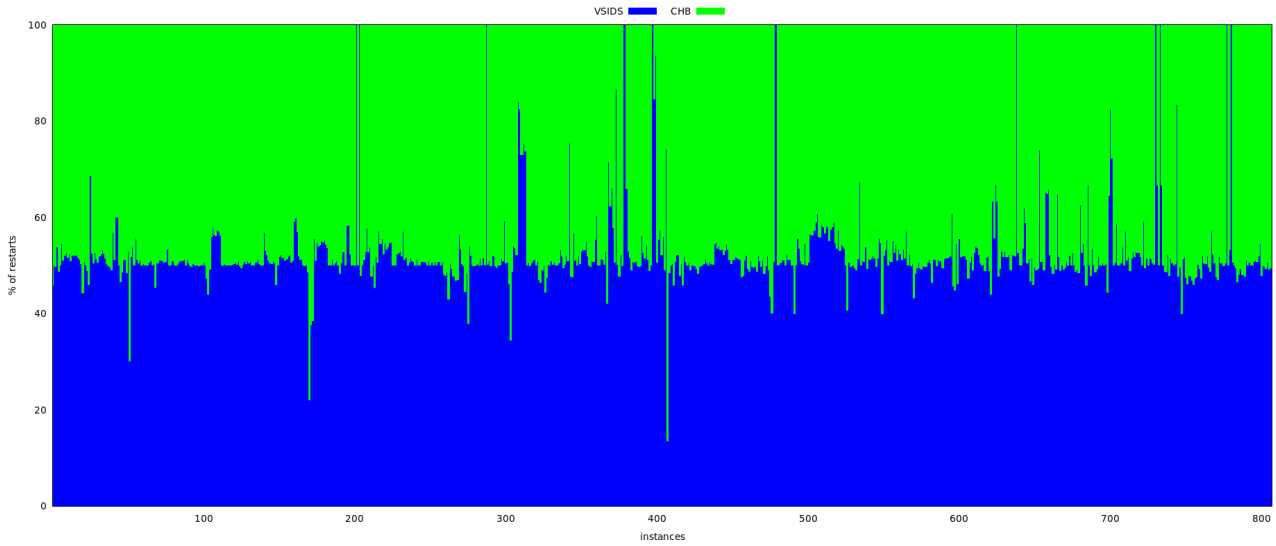


FIGURE 2 – Pourcentages d'utilisation de chaque bras dans MAB par rapport à l'ensemble des instances.

résolues mais aussi en termes de temps de résolution et ses performances sont très proches du VBS.

5.3 Comportement de MAB

Ci-après, on veut analyser plus en détail le comportement du MAB. Tout d'abord, on se focalise sur l'utilisation des bras du bandit. Dans la figure 2, on représente les pourcentages d'utilisation de chaque bras, à savoir VSIDS et CHB. Plus précisément, on représente le pourcentage de redémarrages où chaque bras est choisi par UCB1. On observe que MAB alterne entre les deux heuristiques mais les pourcentages d'utilisation se situent principalement dans l'intervalle $[40\%, 60\%]$ et sont souvent proches de 50%. Il est important de rappeler que les pourcentages d'utilisation des bras sont directement influencés par la politique de redémarrage et par le paramètre c. Cependant, ces résultats ne sont pas surprenants compte tenu du nombre généralement très faible de redémarrages stables dans Kissat au cours desquels les heuristiques VSIDS et CHB sont utilisées. Pour donner une idée, le nombre moyen de redémarrages stables effectués par Kissat sur l'ensemble du benchmark est de 771 alors que la valeur médiane est beaucoup plus faible et s'élève à 313. Par conséquent, les pourcentages obtenus semblent adéquats surtout que le solveur doit parvenir à un bon compromis entre l'exploration et l'exploitation.

De plus, il est important de noter que tirer parti du mécanisme de redémarrage pour combiner VSIDS et CHB via l'utilisation du bandit n'était pas un choix arbitraire. En effet, on a mené une expérience afin de choisir le niveau approprié pour combiner les deux heuristiques grâce au bandit manchot. On a implémenté deux versions de Kissat, RD_N et RD_R, dans lesquelles une heuristique parmi VSIDS et CHB est choisie au hasard respectivement à chaque décision ou à chaque redémarrage. Les résultats moyens (sur 10 exécutions avec des graines différentes) de RD_N et RD_R sont représentés dans le tableau 1 et indiquent que RD_R surpasse RD_N sur l'ensemble du benchmark avec un gain de plus de 2% en termes d'instances résolues et de 40% en termes de temps de résolution (si on attribue une pénalité de 10 000 s pour les instances non résolues). En effet, la combinaison des deux heuristiques au niveau des décisions peut provoquer des interférences et peut ne pas permettre à chaque heuristique de mener un apprentissage robuste puisqu'elles sont constamment interchangées. Étonnamment, les deux versions sont compétitives avec CHB et VSIDS. En particulier, RD_R les surpasse et résout, en moyenne, 21 instances de plus (+2,7%) que la meilleure heuristique. Ceci est dû à la randomisation et à la diversification qui aident à éviter les phénomènes à queue lourde dans SAT et qui peuvent rendre les solveurs plus efficaces [15, 13]. Néanmoins, MAB reste plus efficace que RD_R puisqu'il résout 9 instances supplémentaires (+1,1%) et réalise un gain de 1% en termes de temps de résolution sachant que les résultats individuels obtenus pour RD_R varient et que la différence devient beaucoup plus importante si on considère les tests avec les moindres résultats.

5.4 MAB et familles d'instances

On termine cette section par une analyse plus approfondie des résultats obtenus par MAB sur les familles

Famille		VSIDS		CHB		MAB		VBS	
nom	#inst	#inst	temps	#inst	temps	#inst	temps	#inst	temps
Antibandwidth	14	2	1 010	7	9 804	9	15 651	7	9 628
Keystream Generator Cryptanalysis	18	18	14 494	14	15 104	18	13 118	18	19 240
Baseball-lineup	13	18	3 317	12	2 949	12	2 947	12	2 906
Core-based	14	13	8 438	13	10 860	14	14 165	13	7 239
Chromatic Number (CNP)	20	20	1 708	20	1 972	20	1 180	20	1 194
Edge-Matching Puzzle †	14	3	4 476	3	6 201	4	8 674	4	8 823
Logical Cryptanalysis	20	20	5 606	20	10 476	20	4 241	20	4 946
Hgen	13	12	3 168	12	2 423	12	783	12	2 365
Course Scheduling	20	14	14 439	14	15 363	15	9 323	14	9 654
Relativized Pigeonhole Principle (RPHP)	20	11	14 890	10	11 344	11	14 065	11	14 890
Station Repacking	12	6	15 286	12	10 656	12	8 766	12	10 656
Stedman Triples †	27	10	8 766	11	11 508	12	7 399	11	6 947
Timetable †	26	1	1 565	10	5 082	11	6 247	10	5 082
Vlsat	14	3	103	7	4 457	7	500	7	3 934

TABLE 2 – Comparaison entre VSIDS, CHB, MAB et VBS en termes de nombre d'instances résolues et de temps de résolution cumulé en secondes (pour les instances résolues) dans Kissat pour certaines familles d'instances du benchmark. Les résultats des familles marquées par † sont joints à partir de deux benchmarks annuels différents.

d'instances. Dans le tableau 2, on décrit quelques résultats intéressants obtenus sur certaines familles d'instances au sein du benchmark considéré [17, 16, 5] pour lesquelles MAB surpasse à la fois CHB et VSIDS en termes d'instances résolues ou de temps de résolution, ou les deux. Plus important encore, pour certaines familles, MAB parvient à obtenir de meilleurs résultats, principalement en termes de temps de résolution, que ceux obtenus par le VBS. Par exemple, MAB améliore considérablement le temps de résolution des familles `ASG` (-54%), `Hgen` (-66,9%), `Station Repacking` (-17%) et `Vlsat` (-87,3%) par rapport au VBS. De plus, MAB est également capable de résoudre des instances qui n'ont pas été résolues par le VBS comme dans les familles `Antibandwidth`, `Core-based`, `Polynomial Multiplication`, `Ofer` et `Stedman Triples`.

Enfin, on discute brièvement du choix de combiner VSIDS et CHB même si MAB permet d'inclure plus d'heuristiques. En effet, on pourrait argumenter que l'ajout d'autres heuristiques pourrait permettre de traiter avec succès plus de familles et d'instances grâce à la diversification. Cependant, il faut noter que les solveurs SAT modernes, et en particulier Kissat, sont très compétitifs et s'appuient sur des heuristiques puissantes pour obtenir des résultats impressionnants. Une mauvaise heuristique ou un mauvais réglage des paramètres peut considérablement détériorer les performances d'un solveur. De plus, pratiquement toutes les heuristiques utilisées dans les solveurs SAT modernes sont des variantes de VSIDS, qui a été l'heuristique dominante depuis son introduction en 2001 [27]. Seulement récemment, l'heuristique CHB a été introduite et a été démontrée compétitive avec VSIDS [22]. Les résultats reportées dans le tableau 1 montrent également que CHB peut atteindre de nouvelles instances (le VBS réalise un gain de plus de 5,7% en termes d'instances résolues) tout en restant globalement compétitive et comparable à VSIDS sur les benchmarks des dernières compétitions. Un autre facteur qui peut rendre inefficace l'augmentation du nombre de bras est le nombre très faible de redémarrages et, en particulier, de redémarrages stables dans Kissat. En effet, augmenter le nombre d'heuristiques rendrait le solveur incapable d'identifier le bras adéquat à chaque redémarrage en raison du manque d'essais. On pourrait également penser que l'augmentation du nombre de redémarrages en modifiant les paramètres du solveur peut facilement résoudre ce problème. Ce n'est pas le cas, car la modification des paramètres des solveurs compétitifs, qui sont généralement fixés finement après un réglage approfondi, entraîne généralement une détérioration drastique de leurs performances. C'est pourquoi, après de nombreuses expérimentations sur Kissat, la diversification qui peut être obtenue en combinant plusieurs heuristiques via MAB, y compris des heuristiques non compétitives, va détériorer les performances du solveur.

6 Conclusion

Dans cet article, on a proposé d'utiliser une approche par bandit manchot pour combiner deux heuristiques de l'état de l'art, à savoir VSIDS et CHB. Le bandit tire parti du mécanisme de redémarrage pour sélectionner une heuristique pertinente tout en maintenant un bon équilibre entre l'exploration et l'exploitation. En outre, les heuristiques sont évaluées sur leur capacité à résoudre les conflits rapidement et efficacement et sont sélectionnées par le biais de la politique Upper Confidence Bound (UCB). L'évaluation expérimentale menée montre que MAB surpasse les heuristiques considérées à la fois en termes d'instances résolues, principalement

les instances satisfiables, et en termes de temps de résolution. Le gain est d'autant plus important qu'on a réussi à améliorer Kissat qui est non seulement un solveur de référence (réimplémentation de CaDiCaL en C) mais aussi un solveur des plus compétitifs qui a remporté la compétition SAT 2020. De plus, MAB obtient des résultats très proches du VBS sur VSIDS et CHB et parvient même à les surpasser pour certaines familles d'instances.

Comme perspective de ce travail, il serait intéressant d'affiner la fonction de récompense en s'appuyant sur une combinaison de différents critères [9] afin d'améliorer la performance du bandit, notamment sur les instances insatisfiables. Il serait également intéressant de se focaliser sur une heuristique et d'essayer de la raffiner en utilisant le bandit manchot, une approche qui s'est avérée pertinente dans le contexte du problème de satisfaction des contraintes [8]. Enfin, il serait intéressant de concevoir une approche similaire pour améliorer d'autres composants des solveurs SAT modernes tels que la suppression de clauses apprises [2].

Références

[1] Rajeev Agrawal : Sample mean based index policies by o (log n) regret for the multi-armed bandit problem. *Advances in Applied Probability*, 27(4):1054–1078, 1995.

[2] Gilles Audemard et Laurent Simon : Predicting learnt clauses quality in modern SAT solvers. *In Proceedings of the International Joint Conference on Artificial Intelligence*, 2009.

[3] Gilles Audemard et Laurent Simon : Refining restarts strategies for sat and unsat. *In International Conference on Principles and Practice of Constraint Programming*, pages 118–126. Springer, 2012.

[4] Peter Auer, Nicolò Cesa-Bianchi et Paul Fischer : Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[5] Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo et Martin Suda : Proceedings of SAT Competition 2020 : Solver and Benchmark Descriptions. 2020.

[6] Armin Biere : CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2017. *In* Tomáš Balyo, Marijn Heule et Matti Järvisalo, éditeurs : *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 de *Department of Computer Science Series of Publications B*, pages 14–15. University of Helsinki, 2017.

[7] Armin Biere, Katalin Fazekas, Mathias Fleury et Maximillian Heisinger : CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. *In* Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo et Martin Suda, éditeurs : *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 de *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[8] Mohamed Sami Cherif, Djamal Habet et Cyril Terrioux : On the refinement of conflict history search through multi-armed bandit. *In* Miltos Alamaniotis et Shimei Pan, éditeurs : *Proceedings of 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 264–271. IEEE, 2020.

[9] Wei Chu, Lihong Li, Lev Reyzin et Robert Schapire : Contextual Bandits with Linear Payoff Functions. *In Proceedings of International Conference on Artificial Intelligence and Statistics*, pages 208–214, 2011.

[10] Stephen A. Cook : The Complexity of Theorem-proving Procedures. *In Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC'71, pages 151–158, New York, NY, USA, 1971. ACM.

[11] Todd Deshane, Wenjin Hu, Patty Jablonski, Hai Lin, Christopher Lynch et Ralph Eric McGregor : Encoding first order proofs in SAT. *In Proceedings of the International Conference on Automated Deduction*, pages 476–491, 2007.

[12] Niklas Eén et Niklas Sörensson : An extensible SAT-solver. *In Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[13] Carla P Gomes, Bart Selman, Nuno Crato et Henry Kautz : Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of automated reasoning*, 24(1-2):67–100, 2000.

[14] Shai Haim et Marijn Heule : Towards Ultra Rapid Restarts. *CoRR*, abs/1402.4413, 2014.

[15] William D. Harvey et Matthew L. Ginsberg : Limited discrepancy search. *In IJCAI'95*, page 607–613, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[16] Marijn Heule, Matti Järvisalo et Martin Suda : Proceedings of SAT Race 2019 : Solver and Benchmark Descriptions. 2019.

[17] Marijn Heule, Matti Juhani Järvisalo, Martin Suda *et al.* : Proceedings of SAT Competition 2018 : Solver and Benchmark Descriptions. 2018.

[18] T. Hong, Y. Li, S. Park, D. Mui, D. Lin, Z. A. Kaleq, N. Hakim, H. Naeimi, D. S. Gardner et S. Mitra : QED : Quick Error Detection tests for effective post-silicon validation. *In Proceedings of the International Test Conference*, pages 154–163, 2010.

[19] Vitaly Kurin, Saad Godil, Shimon Whiteson et Bryan Catanzaro : Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning. *CoRR*, 2019.

[20] Tze Leung Lai et Herbert Robbins : Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics*, 6(1):4–22, 1985.

[21] Nadjib Lazaar, Youssef Hamadi, Said Jabbour et Michèle Sebag : Cooperation control in Parallel SAT Solving : a Multi-armed Bandit Approach. Research Report RR-8070, INRIA, septembre 2012.

[22] Jia Hui Liang, Vijay Ganesh, Pascal Poupart et Krzysztof Czarnecki : Exponential Recency Weighted Average Branching Heuristic for SAT Solvers. *In Proceedings of the AAAI Conference on Artificial Intelligence*, pages 3434–3440, 2016.

[23] Jia Hui Liang, Vijay Ganesh, Pascal Poupart et Krzysztof Czarnecki : Learning rate based branching heuristic for SAT solvers. *In Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 123–140, 2016.

[24] Michael Luby, Alistair Sinclair et David Zuckerman : Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[25] Inês Lynce et João P. Marques-Silva : SAT in Bioinformatics : Making the Case with Haplotype Inference. *In Proceedings of International Conference on Theory and Applications of Satisfiability Testing*, pages 136–141, 2006.

[26] João P Marques-Silva et Karem A Sakallah : Grasp : A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[27] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang et Sharad Malik : Chaff : Engineering an efficient SAT solver. *In Proceedings of the Design Automation Conference*, pages 530–535, 2001.

[28] Anastasia Paparrizou et Hugues Wattez : Perturbing branching heuristics in constraint solving. *In* Helmut Simonis, éditeur : *Principles and Practice of Constraint Programming*, pages 496–513, Cham, 2020. Springer International Publishing.

[29] Richard S. Sutton et Andrew G. Barto : *Reinforcement Learning : An Introduction*. MIT Press, Cambridge, MA, USA, 1st édition, 1998.

[30] William R Thompson : On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4): 285–294, 1933.

[31] Toby Walsh : Search in a Small World. *Proceedings of International Joint Conference on Artificial Intelligence*, pages 1172–1177, 2002.

[32] Hugues Wattez, Frederic Koriche, Christophe Lecoutre, Anastasia Paparrizou et Sébastien Tabary : Learning Variable Ordering Heuristics with Multi-Armed Bandits and Restarts. *In Proceedings of the European Conference on Artificial Intelligence*, 2020.

[33] Wei Xia et Roland H. C. Yap : Learning Robust Search Strategies Using a Bandit-Based Approach. *In Proceedings of the AAAI Conference on Artificial Intelligence*, pages 6657–6665, 2018.

Des réfutations SAT aux réfutations Max-SAT*

Matthieu Py† **Mohamed Sami Cherif** **Djamal Habet**

Aix-Marseille Université, Université de Toulon, CNRS, LIS, Marseille, France
{matthieu.py, mohamed-sami.cherif, djamal.habet}@univ-amu.fr

Résumé

Adapter une preuve SAT par résolution en une preuve valide pour Max-SAT sans augmenter considérablement sa taille est une question longtemps restée ouverte. Cet article, qui résume les travaux publiés dans [3], contribue à y répondre en présentant des adaptations linéaires de réfutations SAT en réfutations Max-SAT, dans les cas *tree-like regular*, *tree-like* et *semi-tree-like*. On étend également ces résultats en proposant une adaptation complète pour toute réfutation SAT qui est exponentielle dans le pire des cas.

1 Introduction

Étant donnée une formule sous Forme Normale Conjonctive, le problème Max-SAT consiste à déterminer le nombre maximum de clauses qu'il est possible de satisfaire par une affectation des variables alors que le problème SAT consiste simplement à déterminer si la formule est satisfiable. Dans le contexte du problème SAT, une formule peut être démontrée insatisfiable à l'aide d'une séquence de résolutions [4], appelée réfutation par résolution, qui déduit de la formule initiale de nouvelles clauses jusqu'à en déduire la clause vide qui, par définition, est impossible à satisfaire. De même, pour Max-SAT, on utilise un système de preuve bien connu basé sur la règle de la max-résolution [2], qui étend la règle de résolution utilisée dans SAT. Les séquences de max-résolutions sont plus contraintes que les séquences de résolutions car la max-résolution remplace les prémisses par les conclusions, ce qui consomme les prémisses et empêche de les utiliser à nouveau. Adapter une réfutation SAT (par résolution) en une réfutation valide pour Max-SAT est par conséquent simple si la formule est *read-once*, c'est à dire si chaque clause est utilisée une seule fois comme prémisse d'une résolution. Dans ce cas, il suffit de remplacer chaque étape de résolution par une max-résolution pour obtenir une réfutation valide pour Max-SAT dont la taille (exprimée en nombres d'étapes de transformation) est linéaire par rapport à la taille de la réfutation SAT [1]. En revanche, adapter une réfutation SAT en une réfutation Max-SAT dans le cas général et sans augmenter considérablement sa taille reste une question ouverte.

Dans cet article, on propose d'apporter une contribution pour répondre à cette question. Pour cela, on augmente la règle de la max-résolution avec la règle du *split*, permettant de remplacer une clause par deux clauses en y ajoutant un littéral supplémentaire. Ainsi, on est désormais capable d'adapter n'importe quelle réfutation SAT *tree-like regular* en une réfutation Max-SAT de taille linéaire. On étend ensuite ce résultat aux cas *tree-like* et *semi-tree-like*. Enfin, on propose une adaptation des réfutations SAT dans le cas général mais dont la taille de la réfutation Max-SAT obtenue peut être exponentielle dans le pire des cas.

2 Des réfutations tree-like regular aux réfutations Max-SAT

Considérons tout d'abord le cas des réfutations par résolution *tree-like regular*. Une réfutation est dite *tree-like* si aucune clause intermédiaire (déduite par une résolution) n'est utilisée plusieurs fois en tant que prémisse d'une étape de résolution. Une réfutation est dite *regular* si chacune de ses branches (séquences de résolutions commençant depuis des clauses de la formule et se terminant par la déduction de la clause vide) contient au plus une résolution par variable.

Exemple 1. $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$. *La réfutation par résolution de* ϕ *représentée dans la figure 1 n'est pas read-once mais elle est tree-like regular.*

*Cet article est un résumé de [3].
†Papier doctorant : Matthieu Py est auteur principal.

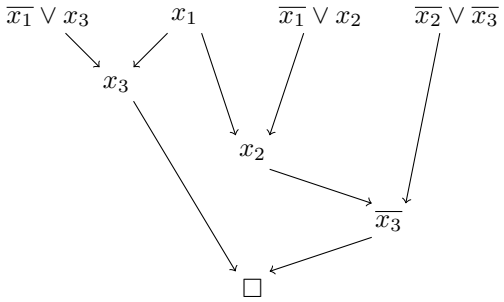


FIGURE 1 – Réfutation par résolution

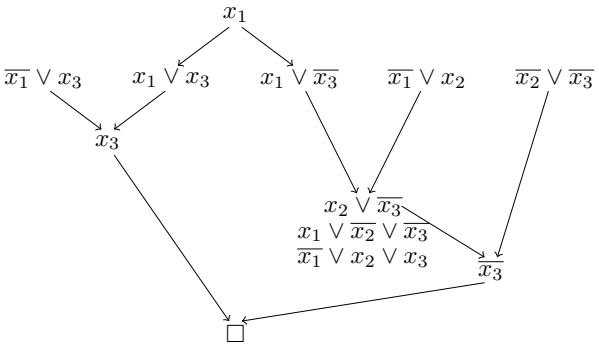


FIGURE 2 – Application de la règle *split* pour générer une réfutation Max-SAT

Théorème 1. *Étant donnée une formule insatisfiable ϕ et une réfutation par résolution tree-like regular P de ϕ, il existe une réfutation Max-SAT de ϕ contenant $O(|P|)$ étapes d'inférence.*

Comme énoncé dans le théorème 1, il est possible d'adapter une réfutation *tree-like regular* en une réfutation Max-SAT de taille linéaire par rapport à la taille de la réfutation SAT. Pour cela, si une clause est utilisée plusieurs fois, on cherche le point de jonction de toutes les branches partant de cette clause. Ce point de jonction est une étape de résolution sur une variable telle que l'application de la règle du *split* sur la clause initiale et cette variable génère deux clauses qui permettent de la remplacer sans impacter la validité de la preuve. En répétant cette opération autant de fois que nécessaire, on retombe sur une réfutation que l'on peut qualifier de *read-once* et on peut remplacer chaque résolution par une max-résolution pour obtenir une réfutation Max-SAT de la formule de départ.

Exemple 2. *Dans la réfutation de l'exemple 1, la clause (x_1) est utilisée deux fois et le point de jonction des deux branches qui partent de (x_1) est une résolution sur la variable x_3. On applique donc une étape de split sur la clause (x_1) et la variable x_3 pour obtenir deux nouvelles clauses permettant de remplacer (x_1). On remplace enfin chaque résolution par une max-résolution et on obtient la réfutation Max-SAT de la figure 2.*

3 Extension aux cas tree-like, semi-tree-like et général

Le résultat linéaire sur les réfutations *tree-like regular* s'étend aux réfutations *tree-like* en utilisant un résultat connu permettant de transformer n'importe quelle réfutation *tree-like* en une réfutation *tree-like regular* plus petite [5]. Pour cela, à chaque fois que l'on détecte deux résolutions sur la même variable dans une branche, on supprime la première résolution ainsi que toutes les autres résolutions devenues inapplicables. Ce résultat s'étend également au cas *semi-tree-like*, où chaque branche de la réfutation contient au plus une clause utilisée plusieurs fois. Dans ce cas-là, on découpe la réfutation en deux morceaux indépendants, le premier *read-once* et le second *tree-like* et on les traite de manière indépendante avant de les refusionner.

Théorème 2. *Étant donnée une formule insatisfiable ϕ et une réfutation par résolution semi-tree-like P de ϕ, il existe une réfutation Max-SAT de ϕ contenant $O(|P|)$ étapes d'inférence.*

Enfin, dans le cas général, il est possible de rendre la réfutation semi-tree-like moyennant une augmentation au pire exponentielle de sa taille. Pour cela, on duplique chaque portion de la preuve conduisant à une clause intermédiaire utilisée plusieurs fois afin d'en créer autant de copies que nécessaire, ce qui permet de rendre la preuve semi-tree-like et de revenir au cas précédent.

Théorème 3. *Étant donnée une formule insatisfiable ϕ et une réfutation par résolution P de ϕ, il existe une réfutation Max-SAT de ϕ contenant $O(2^{\mu(P)} \times |P|)$ étapes d'inférence, où $\mu(P)$ est le nombre de multi-utilisation des clauses intermédiaires de P.*

Références

[1] Federico HERAS et Joao MARQUES-SILVA : Read-Once resolution for Unsatisfiability-Based Max-SAT Algorithms. *In IJCAI*, 2011.

[2] María Luisa BONET, Jordi LEVY et Felip MANYÀ : Resolution for Max-SAT. *Artificial Intelligence*, 2007.

[3] M. PY, M. S. CHERIF et D. HABET : Towards Bridging the Gap Between SAT and Max-SAT Refutations. *In 2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*, 2020.

[4] J. A. ROBINSON : A machine-oriented logic based on the resolution principle. *In Journal of the Association for Computing Machinery*, 1965.

[5] Alasdair URQUHART : The Complexity of Propositional Proof. *Bulletin of Symbolic Logic*, 1995.

Towards Bridging the Gap Between SAT and Max-SAT Refutations

Matthieu Py, Mohamed Sami Cherif and Djamal Habet

Aix-Marseille Université, Université de Toulon, CNRS, LIS, Marseille, France
{matthieu.py, mohamed-sami.cherif, djamal.habet}@univ-amu.fr

***Abstract*—Adapting a resolution proof for SAT to a Max-SAT resolution proof without increasing considerably the size of the proof is an open question. This paper contributes to this topic by exhibiting linear adaptations, in terms of the input SAT proof size, in restricted cases which are regular tree resolution refutations, tree resolution refutations and a new introduced class of refutations that we refer to as semi-tree resolution refutations. We also extend these results by proposing a complete adaptation for any unrestricted SAT refutation to a Max-SAT refutation, which is exponential in the worst case.**

***Index Terms*—Resolution, Max-SAT Resolution, Refutation, Regular Resolution, Tree Resolution**

I. Introduction

Given a Boolean formula in Conjunctive Normal Form (CNF), the Max-SAT problem consists in determining the maximum number of clauses that it is possible to satisfy by an assignment of the variables, while the SAT problem asks for the existence of an assignment which satisfies all the clauses. A well-known proof system for Max-SAT is Max-SAT resolution [20] which extends the resolution rule [24] used in the context of SAT. Max-SAT resolution plays a prominent role in Max-SAT as it is the most studied inference rule, both in theory and practice [1], [2], [7], [18], [20], [22].

In the context of SAT, an unsatisfiable formula can be refuted with a sequence of resolution steps which leads to the empty clause. Sequences of Max-SAT resolution steps are more constrained than sequences of resolution steps. Indeed, resolution adds the conclusion to the premises whereas the premise clauses are replaced by the conclusions when applying Max-SAT resolution. Switching from a read-once resolution proof, where each clause is used once, to a Max-SAT resolution proof is possible and well-known [11]. However, the adaptation of any resolution proof to a Max-SAT resolution one, especially in the context of a refutation, is an established problem. Bonet et al. state that "it seems difficult to adapt a classical resolution proof to get a Max-SAT resolution proof, and it is an open question if this is possible without increasing substantially the size of the proof" [20].

This paper attempts to contribute to this open question on refutation proofs by proposing a way to deal with non-read-once clauses, i.e. clauses used several times as a premise of a resolution step. Indeed, to adapt any resolution refutation, it is necessary to duplicate the non-read-once clauses in some form while also preserving Max-SAT equivalence. To this end, we augment Max-SAT resolution with a simple split rule which allows to generate two clauses subsumed by the original clause. Intuitively, applying the split rule on a non-read-once clause will duplicate it since only literals that will not affect the rest of the proof will be added.

Accordingly, we deal first with regular tree resolution refutations [4], [26], showing that a linear adaptation of a SAT refutation to a Max-SAT one is possible in this case. Then, we extend this result to tree resolution refutations using a known result in [25] which stipulates that a minimal tree resolution refutation is regular. Furthermore, we introduce a new class of refutations that we refer to as semi-tree-like, which is a generalization of tree resolution refutations, and we extend our linear result to this class of refutations. Finally, we propose a complete adaptation of any (or unrestricted) resolution refutation to a Max-SAT refutation, although with an exponential factor, using the fact that any resolution refutation can be made tree-like with an exponential cost.

This paper is organized as follows. Section II gives some necessary definitions and notations. Sections III to VI describe the above contributions in detail. Finally, we conclude and discuss future work in Section VII.

II. Preliminaries

A. Definitions and Notations

Let X be the set of propositional variables. A literal l is a variable $x \in X$ or its negation $\overline{x}$. A clause c is a disjunction (or a set) of literals $(l_1 \vee l_2 \vee ... \vee l_k)$. A formula in Conjunctive Normal Form (CNF) ϕ is a conjunction (or a multiset) of clauses $\phi = c_1 \wedge c_2 \wedge ... \wedge c_m$. An assignment $I : X \rightarrow \{true, false\}$ maps each variable to a boolean value and can be represented as a set of literals. A literal l is satisfied (resp. falsified) by an assignment I if $l \in I$ (resp. $\overline{l} \in I$). A clause c is satisfied by an assignment I if at least one of its literals is satisfied by I, otherwise it is falsified by I. The empty clause $\square$ contains zero literals and is always falsified. A clause c opposes a clause c' if c contains a literal whose negation is in c', i.e. $\exists l \in c,\ \overline{l} \in c'$. A clause c subsumes a clause c' if each literal of c is a literal of c', i.e. $\forall l \in c,\ l \in c'$. We denote $var(c)$ the variables appearing in the clause c. A CNF formula ϕ is satisfied by an assignment I, that we call model of ϕ, if each clause $c \in \phi$ is satisfied by I, otherwise it is falsified by I. Solving the Satisfiability (SAT) problem consists in determining whether there exists an

assignment I that satisfies a given CNF formula ϕ. In the case where such an assignment exists, we say that ϕ is satisfiable, otherwise we say that ϕ is unsatisfiable or inconsistent. The cost of an assignment I, denoted $cost_I(\phi)$, is the number of clauses falsified by I. The Maximum Satisfiability (Max-SAT) problem is an optimization extension of SAT which, for a given CNF formula ϕ, consists in determining the maximum number of clauses that can be satisfied by an assignment of ϕ. Equivalently, it consists in determining the minimum number of clauses that each assignment must falsify, i.e. $\min_I cost_I(\phi)$.

B. Resolution Refutations in SAT

To certify that a CNF formula is satisfiable, it is sufficient to simply exhibit a model of the formula. On the other hand, to prove that a CNF formula is unsatisfiable, we need to refute the existence of a model. To this end, we can exhibit a SAT refutation which consists of a sequence of equivalence-preserving transformations (in the sense of SAT as defined below) starting from the formula and ultimately deducing an empty clause.

Definition 1 (SAT Equivalence)**.** *Let ϕ and ϕ' be two CNF formulas. We say that ϕ is equivalent (in the sense of SAT) to ϕ' if for any assignment $I : var(\phi) \cup var(\phi') \rightarrow \{true, false\}$, I is a model of ϕ if and only if I is a model of ϕ'.*

A well-known SAT refutation system is based on an inference rule for SAT called resolution [24]. Refutations in this system are referred to as resolution refutations. The resolution rule, defined below, deduces a clause called resolvent from two opposed clauses which can be added to the formula while preserving SAT equivalence. Resolution plays an important role in the context of Conflict Driven Clause Learning (CDCL) [21]. Furthermore, it was shown that CDCL can polynomially simulate general resolution [23]. As showcased in Example 1, a resolution proof can be represented as a Directed Acyclic Graph (DAG) whose nodes are clauses in the proof either having two or zero incoming arcs (resp. if they are resolvents or clauses of the initial formula).

Definition 2 (Resolution [24])**.** *Given two clauses $c_1 = (x \vee A)$ and $c_2 = (\overline{x} \vee B)$, the resolution rule is defined as follows:*

$$\frac{c_1 = (x \vee A) \quad c_2 = (\overline{x} \vee B)}{c_3 = (A \vee B)}$$

Example 1. *We consider the CNF formula $\phi = (\overline{x_1} \vee x_3) \wedge (x_1) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_2} \vee \overline{x_3})$. A resolution refutation of ϕ is represented as a DAG in Fig. 1.*

Many restricted classes of resolution refutations have been studied in the literature namely linear resolution [19], unit resolution [12], input resolution [12], regular resolution [26], read-once resolution [14] and tree (or tree-like) resolution [4] refutations among others. In particular, a resolution refutation is tree-like if every intermediate clause, i.e. resolvent, is used at most once in the proof. It is known that DPLL algorithms [8] on unsatisfiable instances correspond to tree resolution refutations [9]. Similarly, a resolution refutation is read-once if each clause is used at most once in the proof. Clearly, read-once resolution refutations are also tree-like since they form a restricted class of tree resolution refutations. It was shown in [14] that there exists unsatisfiable CNF formulas which cannot be refuted using read-once resolution. Finally, a resolution is regular if every variable is resolved on at most once in each branch of the DAG, i.e. path from a clause of the initial formula to the empty clause. It was shown that CDCL without restarts can polynomially simulate regular resolution [6]. We say that an irregularity is a sequence of clauses (each clause must be deduced using the previous one as premise) such that the first clause and the last one contain a literal l but at least one of the intermediate clauses does not contain this literal l. In other words, an irregularity is a certificate that a resolution refutation is not regular.

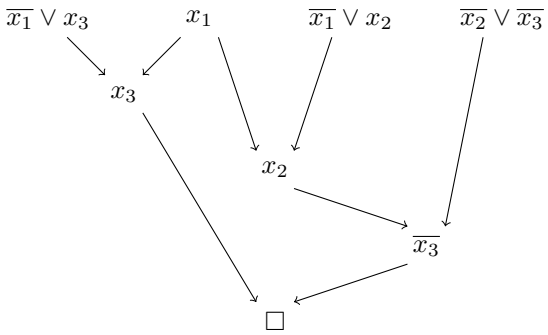


Fig. 1. Resolution refutation

Example 2. *We consider the refutation of ϕ in Example 1. The refutation is clearly tree-like but it is not read-once since clause (x_1) is used two times as a premise of a resolution step. The refutation is also regular as every variable is resolved on at most once in every branch of the DAG in Fig. 1.*

C. Max-SAT Refutations

Several complete proof systems for Max-SAT were introduced in the litterature, namely the Max-SAT resolution Calculus in [20] and the Clause Tableau Calculus in [17]. In particular, Max-SAT resolution, one of the first known complete systems for Max-SAT, was inspired from Resolution. The aim of complete Max-SAT systems is not to refute the formula per se but to compute the Max-SAT optimum of a given CNF formula, i.e. the maximum number of falsified clauses. The formula is thus refuted as many times as its optimum through equivalence-preserving transformations in the sense of Max-SAT as defined below.

Definition 3 (Max-SAT Equivalence)**.** *Let ϕ and ϕ' be two CNF formulas. We say that ϕ is equivalent (in the sense of Max-SAT) to ϕ' if for any assignment $I : var(\phi) \cup var(\phi') \rightarrow \{true, false\}$, we have $cost_I(\phi) = cost_I(\phi')$.*

The Max-SAT resolution proof system relies on an inference rule that extends resolution for Max-SAT. Other than the resol-

vent clause, this rule, called Max-SAT resolution and defined below, introduces new clauses referred to as compensation clauses essential to preserve Max-SAT equivalence. As a sound and complete rule for Max-SAT [20], Max-SAT resolution plays an important role in the context of Max-SAT theory and solving. In particular, it is extensively used in the context of Branch and Bound algorithms for Max-SAT to transform inconsistent subsets [2], [15], [18] as well as in the context of SAT-based algorithms to transform cores returned by SAT oracles [11], [22]. For a given CNF formula, it is possible to generate a Max-SAT resolution proof of its optimum by applying the saturation algorithm [20] to deduce empty clauses. As showcased in Example 3, a Max-SAT resolution proof can also be represented as a DAG whose nodes are multisets of clauses either having two or zero incoming arcs (resp. if they are clauses produced by a Max-SAT resolution step or clauses of the initial formula.

Definition 4 (Max-SAT resolution [5], [16], [20])**.** *Given two clauses $c_1 = x \vee A$ and $c_2 = \overline{x} \vee B$ with $A = a_1 \vee ... \vee a_s$ and $B = b_1 \vee ... \vee b_t$. The Max-SAT resolution rule is defined as follows:*

$$\frac{c_1 = x \vee A \qquad c_2 = \overline{x} \vee B}{\begin{array}{c} c_r = A \vee B \\ cc_1 = x \vee A \vee \overline{b_1} \\ cc_2 = x \vee A \vee b_1 \vee \overline{b_2} \\ \vdots \\ cc_t = x \vee A \vee b_1 \vee ... \vee b_{t-1} \vee \overline{b_t} \\ cc_{t+1} = \overline{x} \vee B \vee \overline{a_1} \\ cc_{t+2} = \overline{x} \vee B \vee a_1 \vee \overline{a_2} \\ \vdots \\ cc_{t+s} = \overline{x} \vee B \vee a_1 \vee ... \vee a_{s-1} \vee \overline{a_s} \end{array}}$$

where c_r is the resolvent clause and $cc_1, ..., cc_{t+s}$ are compensation clauses.

Remark 1. *Unlike resolution, the Max-SAT resolution rule replaces the premises by the conclusions.*

Example 3. *We consider the CNF formula from Example 1. A hand-made Max-SAT resolution refutation of ϕ was proposed in [20] and is represented in Fig. 2.*

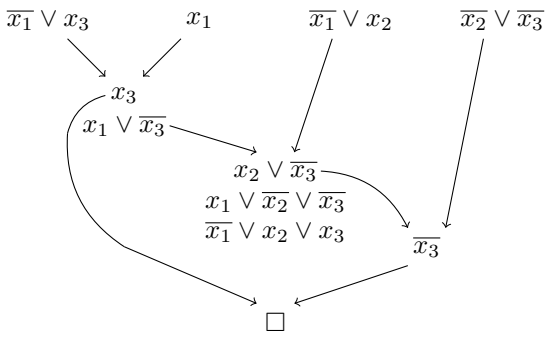


Fig. 2. A Max-SAT resolution proof

In this paper, we also augment Max-SAT resolution with the split rule defined below. Intuitively, this rule allows to duplicate a clause by adding one literal.

Definition 5 (Split rule)**.** *Given a clause $c_1 = (A)$ where A is a disjunction of literals and x a variable, the Max-SAT split rule is defined as follows:*

$$\frac{c_1 = (A)}{c_2 = (x \vee A) \qquad c_3 = (\overline{x} \vee A)}$$

Remark 2. *Like the Max-SAT resolution rule, the split rule replaces the premise by the conclusions.*

Finally, we choose to maintain the designation 'refutation' in the context of Max-SAT. A Max-SAT refutation (or max-refutation) will thus consist of a sequence of Max-SAT preserving transformations, namely Max-SAT resolutions and splits in this paper, allowing to deduce an empty clause from a given unsatisfiable formula. The size of a (SAT or Max-SAT) refutation is the number of its inference steps.

III. From Regular Tree Resolution Refutations to Max-Refutations

In this section, we show how it is possible to adapt a regular tree resolution refutation to obtain a max-refutation with linear size. If a clause c is used k times ($k > 1$) as a premise of a resolution step, we use the split rule to duplicate clause c into k distinct clauses subsumed by c. We will then use these new clauses to replace c as a premise of a resolution step. Given a branch starting from a clause c, we say that this branch accepts the substitution of c by $c \vee l$ if updating the branch after the substitution of c by $c \vee l$ does not affect the validity of the resolution refutation. The following lemma guarantees that, for a given non-read-once clause, there exists a variable x such that some branches starting from c accept the substitution of c by $c \vee x$ while the rest accept the substitution of c by $c \vee \overline{x}$.

Lemma 1. *Given a non-read-once regular tree resolution refutation P and a non-read-once clause c in P, there exists a variable $x \notin var(c)$ such that it is possible to partition the branches starting from c into two non-empty subsets of branches, the branches in the first subset accepting the substitution of c by $c \vee x$ and the branches in the second accepting the substitution of c by $c \vee \overline{x}$.*

Proof. Let P a non-read-once regular tree resolution refutation and c a non-read-once clause in P. There exists a node v of the DAG of P representing a resolution step on variable x such that v is the first junction point of all the paths starting from c. The existence is ensured since this junction point is eventually the empty clause. Furthermore, every path starting from the clause c leads to one (and only one) of the premises of the resolution step in the node v. Indeed, a path leading to both premises entails the existence of an intermediate non-read-once clause which is not possible since the refutation is tree-like. We partition the branches starting from c into two subsets containing respectively the paths leading to the first and second premise of the resolution step in the node v. Each partition is

non empty since if there exists an empty subset v can't be the first junction point of the branches. Let x be the variable eliminated at this resolution step and suppose w.l.o.g that the first premise contains literal x while the second contains literal $\overline{x}$. As P is regular, x is not a variable of c and the subset of branches starting from c leading to the first premise accepts the substitution of c by $c \vee x$ while the subset of branches leading to the second premise accepts the substitution of c by $c \vee \overline{x}$.

■

The result established in Lemma 1 ensures the possibility to fix any non-read-once clause used $k > 1$ times by using the split rule. Indeed, we can apply this rule to replace a non-read-once clause used $k > 1$ times by two clauses used respectively $1 \leq k_1 < k$ and $1 \leq k_2 < k$ such that $k = k_1 + k_2$. By iterating this method, we can fix every non-read-once clause. Then, we only need to replace the resolution rule by the Max-SAT resolution rule to obtain an adaptation from any regular tree resolution refutation to a max-refutation in linear size.

Theorem 1. *Given an unsatisfiable formula ϕ and a regular tree resolution refutation P of ϕ, there exists a max-refutation of ϕ containing $O(|P|)$ inference steps.*

Proof. Let P be a regular tree resolution refutation of ϕ. We set $T_1 = \emptyset$ and $T_2 = MR(P)$, where $MR(P)$ is obtained from P after replacing each resolution by Max-SAT resolution. If P is read-once, T_2 is a max-refutation of ϕ containing $|P|$ inference steps (which is obviously in $O(|P|)$). Now, let c be a non-read-once clause of P. Using Lemma 1, there exists a variable $x \notin var(c)$ and a partition of the branches starting from c into two non-empty subsets, the first accepting $c \vee x$ and the second accepting $c \vee \overline{x}$. We apply the Max-SAT split rule on c to obtain $c \vee x$ and $c \vee \overline{x}$ and we replace c as premise by $c \vee x$ on the first subset of branches and c by $c \vee \overline{x}$ on the second. Doing this, we augment T_1 by adding one split and we change T_2 by replacing the premise clause c as described above. As T_2 is a tree-like regular resolution refutation of $(\phi \backslash c) \wedge (c \vee x) \wedge (c \vee \overline{x})$, it is possible to iteratively apply this operation on T_2 until we obtain a read-once regular tree resolution refutation. Therefore, after the last iteration, we have a couple (T_1, T_2) such that T_1 is a sequence of applications of the split rule transforming ϕ into a Max-SAT equivalent ϕ' and T_2 is a read-once regular max-refutation of ϕ'. Therefore, these transformations form a max-refutation of ϕ.

To prove that the size of the max-refutation is in $O(|P|)$, we first consider how to fix a leaf clause of P (i.e. how to replace it by read-once clauses). If c is a leaf clause of P used k times, we prove by induction on k that it is possible to fix this clause using at most $k - 1$ splits:

- If $k = 1$, we clearly need 0 splits to fix the read-once clause c.
- Suppose that the assertion is true for any $k' < k$ and let c be a clause used k times. Using Lemma 1, it is possible to use 1 split to replace c by two clauses c_1 and c_2 respectively used k_1 and k_2 times with $k_1, k_2 > 0$ and $k_1 + k_2 = k$. Using our assertion for k_1 and k_2, it is possible to fix c_1 with at most $k_1 - 1$ splits and c_2 with at most $k_2 - 1$ splits. Therefore, it is possible to fix c with at most $1 + (k_1 - 1) + (k_2 - 1) = k - 1$ splits.

Let $c_1, ..., c_p$ be the leaf clauses of P used respectively $k^1, k^2, ..., k^p$ times. Notice that $k^1 + k^2 + ... + k^p = |P| + 1$ since P has exactly $2|P|$ premises, i.e. uses of clauses, and $|P| - 1$ intermediate clauses (the empty clause is not used and we neglect the trivial cases where a non-empty intermediate clause is not used and where the proof produces several empty clauses). Using the previous induction, we need at most $k^1 - 1 + k^2 - 1 + ... + k^p - 1 \leq |P|$ splits to fix every non-read-once leaf clause of P. Consequently, $|T_1| \leq |P|$. On the other hand, the number of Max-SAT resolutions in T_2 is by construction equal to the number of resolution steps in P and, therefore, $|T_2| = |P|$. We conclude that the complete max-refutation contains at most $2|P|$ inference steps, which is in $O(|P|)$.

■

Example 4. *We consider the regular tree resolution refutation from Example 1 represented by the DAG in Fig. 1. We observe that the original clause (x_1) is used two times as a premise of a resolution step. The junction point of the left and right branches eliminates variable x_3 such that the branch on the left leads to the premise containing literal x_3 and the branch on right leads to the premise containing literal $\overline{x_3}$. We apply the split rule on clause (x_1) to get $(x_1 \vee x_3)$ and $(x_1 \vee \overline{x_3})$ and we replace (x_1) by $(x_1 \vee x_3)$ and $(x_1 \vee \overline{x_3})$ respectively on the left and right branches. Finally, we replace all resolutions by Max-SAT resolutions to obtain the complete max-refutation.*

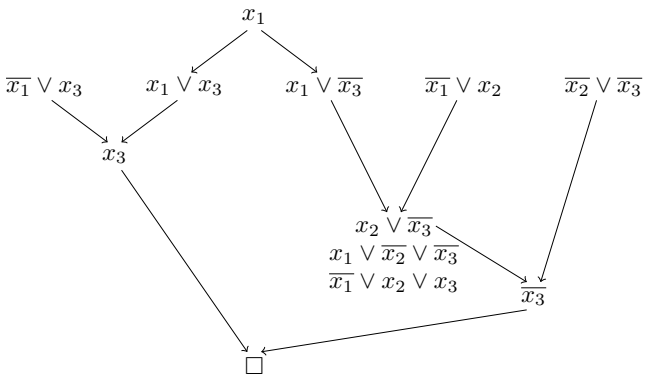


Fig. 3. Applying the split rule to deal with a non-read once clause

IV. From Tree Resolution Refutations to Max-Refutations

In the previous section, we proposed a linear adaptation from regular tree resolution refutations to max-refutations. We propose in this section to extend the case where this adaptation guarantees linear size of the obtained max-refutation to tree resolution refutations. To this end, we simply exhibit a known transformation from any tree resolution refutation to a regular

tree resolution refutation without increasing its size. This result was proved in [25] in the form of the following lemma (cf. Lemma 5.1 in [25]). The proof relies on a transformation which consists in iteratively discarding the first resolution in the case of an irregularity and updating the rest of the resolution proof accordingly, potentially discarding other resolution steps which are no longer necessary.

Lemma 2. *[25] A tree resolution refutation of minimal size is regular.*

Example 5. *We consider the tree resolution refutation represented in Fig. 5. This refutation is not regular since x_1 is eliminated two times in the same branch. As shown in Fig. 5, this refutation can be minimized and thus made regular by discarding the first resolution on variable x_1 in the irregularity and updating the rest of the proof, Notice that after the transformation, clauses $(\overline{x_1} \vee \overline{x_3})$ and (x_3) are no longer used in the refutation.*

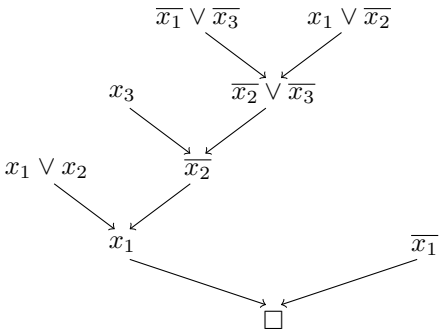


Fig. 4. Tree resolution refutation containing an irregularity on variable x_1

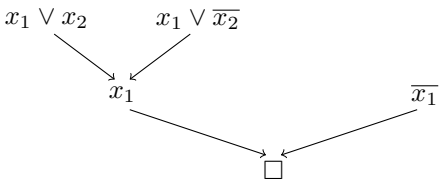


Fig. 5. Regular tree resolution refutation after minimization

Since it is possible to make a tree resolution refutation regular without increasing the size of the proof, we can apply the adaptation in Theorem 1 to produce a max-refutation with linear size as shown in the following corollary.

Corollary 1. *Given an unsatisfiable formula ϕ and a tree resolution refutation P of ϕ, there exists a max-refutation of ϕ containing $O(|P|)$ inference steps.*

Proof. Using Lemma 2, there exists a regular tree resolution refutation P_2 such that $|P_2| = O(|P|)$. By applying Theorem 1, we obtain a max-refutation containing $O(|P_2|) = O(|P|)$ inference steps. ■

V. From Semi-Tree Resolution Refutations to Max-Refutations

In Section IV, we proposed a linear adaptation from any tree resolution refutation to a max-refutation. We propose in this section to extend this linear result to semi-tree resolution refutations defined below. As shown in Proposition 1, this class of refutations extends tree resolution refutations, i.e. every tree resolution refutation is semi-tree-like.

Definition 6 (semi-tree resolution refutation)**.** *A resolution refutation is semi-tree-like if, for any branch of the refutation, at most one clause is non-read-once.*

Example 6. *We consider the resolution refutation P in Fig. 6. P is clearly semi-tree-like since in each branch at most one clause is non-read-once. Notice also that P is not tree-like since (x_1) is an intermediate non-read-once clause.*

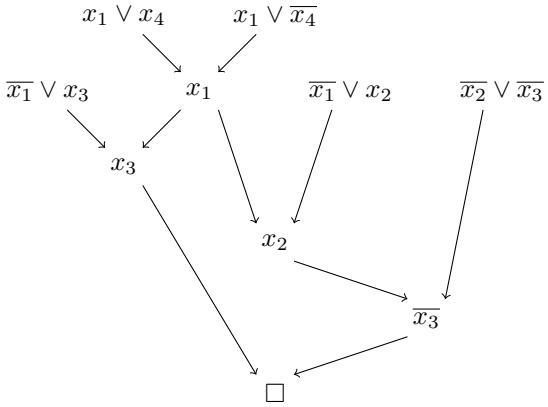


Fig. 6. Semi-tree-like resolution refutation

Proposition 1. *Let P be a resolution refutation. If P is tree-like then P is semi-tree-like.*

Proof. Suppose that P is tree-like. By definition, each intermediate clause is read-once. In each branch, the only clause that can be non-read-once in P is by definition a leaf. Therefore, at most one clause is non-read-once in each branch and we conclude that P is semi-tree-like. ■

To extend our result to semi-tree-like resolution refutations, we propose a method which relies on the fact that semi-tree resolution refutations can be partitioned into two parts where the first part is a read-once sequence of resolutions and the second part is a tree-like resolution refutation. As the first part is a read-once sequence of resolutions, it is possible to adapt it for Max-SAT using a similar method to the one in [11], i.e. replacing each resolution by a Max-SAT resolution. As the second part is a tree resolution refutation, it is possible to adapt it for Max-SAT using the result in Corollary 1. After transforming the two parts, we glue them back to construct the complete max-refutation.

Theorem 2. *Given an unsatisfiable formula ϕ and a semi-tree resolution refutation P of ϕ, there exists a max-refutation of ϕ containing $O(|P|)$ inference steps.*

Proof. As P is semi-tree-like, each branch of P contains at most one non-read-once clause. We partition P into two parts P_1 and P_2 as follows:

- For each branch containing one non-read-once clause, the transformations until this clause are put in P_1 and the transformations after this clause are put in P_2.
- For each branch not containing a non-read-once clause, the transformations are put in P_2.

By construction, P_1 is a read-once sequence of resolutions so it is possible to adapt it to obtain a Max-SAT transformation P'_1 containing exactly $|P_1|$ inference steps by replacing resolutions with Max-SAT resolutions as in [11]. Furthermore, P_2 is a tree resolution refutation because the non-read-once clauses of P are leaf clauses in P_2. Consequently, it is possible to adapt P_2 into a max-refutation P'_2 containing $O(|P_2|)$ inference steps using the result in Corollary 1. Finally, we can combine P'_1 and P'_2 to obtain a Max-SAT refutation containing at most $O(|P_1| + |P_2|)$ and we conclude that the complete adaptation contains $O(|P|)$ inference steps. ■

Example 7. *We consider the semi-tree resolution refutation in Example 6, represented in Fig. 6. To adapt this semi-tree resolution refutation to a max-refutation, we put aside the top resolution on variable x_1 taking clauses $(x_1 \vee x_4)$ and $(x_1 \vee \overline{x_4})$ and we obtain the tree-like resolution refutation in Example 1, represented in Fig. 1. We adapt this tree-like resolution refutation as in Example 4 and we replace the resolution step on x_1 by a Max-SAT resolution step (in this case no compensation clauses are generated). We glue back the two parts to obtain the complete max-refutation represented in Fig. 7.*

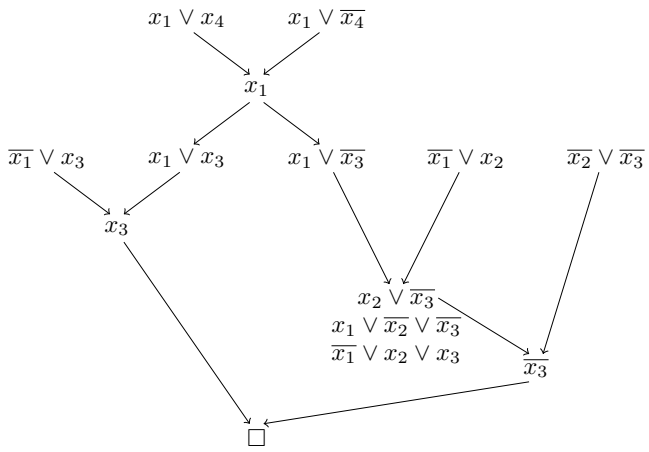


Fig. 7. Adapting a semi-tree resolution refutation to a max-refutation

VI. From Unrestricted Resolution Refutations to Max-Refutations

In Sections III, IV and V, we proposed linear adaptations from specific classes of resolution refutations to max-refutations. In this section, we propose an adaptation from any resolution refutation to a max-refutation. To make this adaptation, we will simply extend the adaptation described in section IV by adding a first transformation to make the initial resolution refutation tree-like as described in Lemma 3. Notice that we could make the initial resolution refutation semi-tree-like (instead of tree-like) but this choice does not affect the theoretical size of the obtained max-refutation.

To achieve this first intermediate transformation, we will iteratively search in the proof for the first non-read-once intermediate clause c. If this clause is used $k > 1$ times as a premise of another resolution step, we consider the part of the proof leading to c and we dupplicate it k times in order to get a tree-like sequence of resolutions generating k resolvents $c_1, c_2, ..., c_k$ (with $c_1 = c$), each resolvent c_i containing exactly the same literals as c and is generated by a similar sequence of resolution steps. Consequently, c is no longer used several times as a premise of a resolution step, the input clauses are. Repeating this operation forces the resolution refutation to become tree-like. Fixing a non-read-once intermediate clause can, in the worst case, double the size of the current resolution refutation. As such, the size of the obtained tree-like resolution refutation is exponentially bounded by the size of the initial unrestricted resolution refutation. To polish this upper bound, we introduce a new parameter defined below, which is the number of multi-uses of intermediate clauses. Notice how, in the definition, we subtract 1 use for each clause. Intuitively, we consider the first use of any non-read-once intermediate clause as authorized.

Definition 7. *Let P be a resolution refutation. The number of multi-uses of intermediate non-read-once clauses, denoted $\mu(P)$, is defined as follows:*

$$\mu(P) = \sum_{c \text{ intermediate non-read-once in } P} (d^+(c) - 1)$$

where $d^+(c)$ denotes the number of uses of the clause c, i.e. the number of outgoing arcs from c in the DAG representation of P.

Lemma 3. *Given an unsatisfiable formula ϕ and a resolution refutation P of ϕ, there exists a tree resolution refutation of ϕ containing $O(2^{\mu(P)} \times |P|)$ resolution steps.*

Proof. Let P be a resolution refutation of ϕ. We iteratively make the intermediate non-read-once clauses read-once. Each time, we pick the first intermediate non-read-once clause c and duplicate the sub-proof deriving c exactly $d^+(c) - 1$ times. Each iteration decrements the number of intermediate non-read-once clauses by 1 until the resolution refutation becomes tree-like. Clearly, for each duplication, the size of the proof is doubled in the worst case and we perform exactly $\mu(P)$ duplications. We conclude that the size of the obtained tree

resolution refutation is bounded by $O(2^{\mu(P)} \times |P|)$.

■

Now that we can transform any resolution refutation to a tree-like resolution refutation, we just have to adapt the obtained tree-like resolution refutation with the method described in section IV as shown in the following Theorem.

Theorem 3. *Given an unsatisfiable formula ϕ and an unrestricted resolution refutation P of ϕ, there exists a max-refutation of ϕ with $O(2^{\mu(P)} \times |P|)$ inference steps.*

Proof. Let P be an unrestricted resolution refutation of ϕ. We adapt P to obtain a tree resolution refutation P_t of size $O(2^{\mu(P)} \times |P|)$ using Lemma 3. Then, using Theorem 1, we obtain a max-refutation of size $O(2^{\mu(P)} \times |P|)$.

■

Example 8. *We consider the resolution refutation represented in Fig. 8. This refutation is not semi-tree-like since the clauses (x_1) and (x_4) are two non read-once clauses in the same branch. First, we duplicate the resolutions leading to (x_1) and we obtain the tree-like resolution refutation represented in Fig. 9. Then, we apply the transformations described in Section IV to get the max-refutation represented in Fig. 10.*

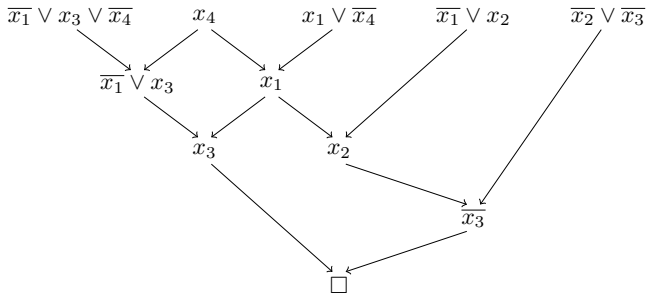


Fig. 8. Unrestricted resolution refutation

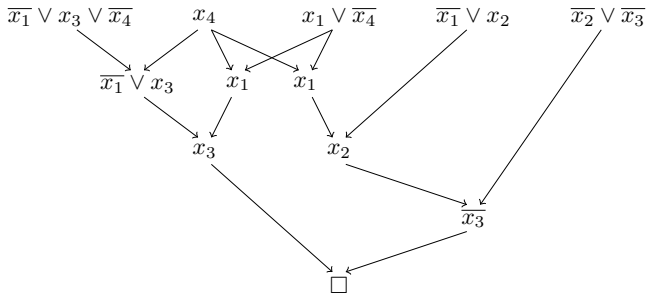


Fig. 9. Adapting a resolution refutation to a tree-like resolution refutation

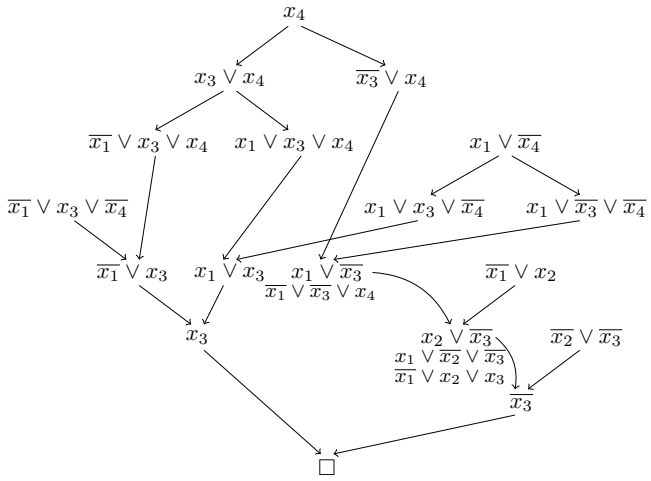


Fig. 10. Adapting an unrestricted resolution refutation to a max-refutation

We finish this section by exhibiting resolution refutations whose adaptations as in Theorem 3 is exponential. To this end, we introduce in the following definition a new pattern which we will use to build such refutations.

Definition 8 (Diamond pattern)**.** *Let A be a disjunction of literals and let $x \notin var(A)$ and $y \notin var(A)$ two distinct variables. We define the diamond pattern (x, y, A) as the sequence of resolutions represented in Fig.11.*

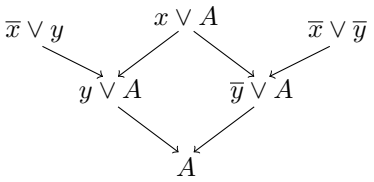


Fig. 11. Diamond pattern (x, y, A)

Fig. 12. Simplified representation of a diamond pattern

We can represent this pattern by a diamond as in Fig. 12. Notice that in particular, the diamond pattern $(x, y, \square)$ is a resolution refutation. Now, imagine that the topmost clause of $(x, y, \square)$ is derived through another diamond pattern. We iterate the same reasoning to define a k-stacked diamonds pattern as follows:

Definition 9 (k-stacked diamond pattern)**.** *Let $k \geq 1$ be a natural number and let x_i and y_i where $1 \leq i \leq k$ be distinct variables. A k-stacked diamond pattern is formed by k diamond patterns (x_i, y_i, A_i) where $1 \leq i \leq k$ such that $A_1 = \square$ and $A_i = (x_1 \vee \cdots \vee x_{i-1})$ for $1 < i \leq k$. Each diamond (x_i, y_i, A_i) is stacked on top of $(x_{i-1}, y_{i-1}, A_{i-1})$ such that the last conclusion of the former is the topmost central premise of the latter.*

A k-stacked diamond pattern is represented as a stack of diamonds as shown in Fig.13 for $k = 3$. Clearly, k-stacked diamond are resolution refutations as they deduce the empty clause $\square$. In particular, when $k > 2$, a k-stacked diamond is not semi-tree-like. The size of a k-stacked diamond P is $|P| = 3k$. Furthermore, we have $\mu(P) = k - 1$. Therefore, after the application of the adaptation described in Theorem 3, we obtain a max-refutation whose size is at least 2^{k-1} showing that the proposed adaptation can be exponential in the worst case.

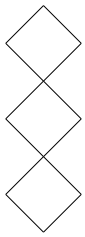


Fig. 13. Simplified representation of a 3-stacked diamond pattern

VII. Conclusions and Future Work

The contributions of this work are related to adapting resolution refutations to Max-SAT refutations. In particular, we have proposed linear adaptations regarding the size of the resolution refutations in the following cases: regular tree resolution, tree resolution and semi-tree resolution. These results are achieved by augmenting Max-SAT resolution with the split rule which enabled us to duplicate clauses by adding literals when necessary. We have also generalised our adaptation to unrestricted resolution refutations, even though the proposed transformation can produce a max-refutation whose size is exponential in the worst case. Notice that our results remain valid for weighted Max-SAT formulas as we simply need to augment the previous rules with another split rule for weights. Indeed, the overhead of this rule is linear in terms of the size of the refutation since we need to apply it once on the clauses in the weighted MAX-SAT formula to produce clauses with same weight, i.e. the minimum of all the weights as done in the context of SAT-based (weighted) Max-SAT algorithms [3].

These results may help to exhibit proofs based on Max-SAT resolution for Max-SAT algorithms which remains an unexplored topic whereas, in SAT, practically all modern solvers are able to compute a resolution proof of unsatisfiability (in different formats [10], [13]). Indeed, it would be interesting to include the proposed adaptations in SAT-based algorithms for Max-SAT. Such extended algorithms would thus iteratively call a SAT oracle to get a resolution refutation, adapt this resolution refutation to get a Max-SAT refutation based on our results and transform the formula accordingly. This treatment is repeated until reaching a satisfiable formula and a set of empty clauses whose size is the optimum value of the formula. Such implementation would require an efficient SAT oracle which returns a resolution refutation. Finally, the existence of an adaptation that does not increase substantially the size of an unrestricted resolution proof remains an open question. We will continue to investigate this topic either by exhibiting a polynomial adaptation or refuting its existence.

References

[1] A. Abramé and D. Habet. On the resiliency of unit propagation to max-resolution. In *IJCAI*, 2015.

[2] A. Abramé and D. Habet. ahmaxsat: Description and evaluation of a branch and bound max-sat solver. In *Journal on Satisfiability, Boolean Modeling and Computation. Vol. 9, pp. 89–128*, 2015.

[3] C. Ansótegui, M. L. Bonet, and J. Levy. Solving (weighted) partial maxsat through satisfiability testing. In O. Kullmann, editor, *SAT 2009*, volume 5584 of *Lecture Notes in Computer Science*, pages 427–440. Springer, 2009.

[4] E. Ben-sasson, R. Impagliazzo, and A. Wigderson. Near optimal separation of tree-like and general resolution. *Combinatorica*, 24:585–603, 09 2004.

[5] M. Bonet, J. Levy, and F. Manyà. A complete calculus for max-sat. In *SAT 2006*, volume 4121, pages 240–251, 08 2006.

[6] S. Buss, J. Hoffmann, and J. Johannsen. Resolution Trees with Lemmas: Resolution Refinements that Characterize DLL Algorithms with Clause Learning. *Logical Methods in Computer Science*, 4, 11 2008.

[7] M. S. Cherif and D. Habet. Towards the characterization of max-resolution transformations of ucss by up-resilience. In T. Schiex and S. de Givry, editors, *CP*, pages 91–107, Cham, 2019. Springer International Publishing.

[8] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[9] A. V. Gelder. Extracting (easily) checkable proofs from a satisfiability solver that employs both preorder and postorder resolution. In *ISAIM*, 2002.

[10] A. V. Gelder. Verifying rup proofs of propositional unsatisfiability. In *ISAIM*, 2008.

[11] F. Heras and J. Marques-Silva. Read-once resolution for unsatisfiability-based max-sat algorithms. In *IJCAI*, 2011.

[12] A. Hertel and A. Urquhart. Algorithms and complexity results for input and unit resolution. *Journal of Satisfiability, Boolean Modeling and Computation*, 6, 05 2009.

[13] M. J. Heule, W. A. Hunt, and N. Wetzler. Trimming while checking clausal proofs. In *2013 Formal Methods in Computer-Aided Design*, pages 181–188. IEEE, 2013.

[14] K. Iwama and E. Miyano. Intractability of read-once resolution. In *Proceedings of Structure in Complexity Theory. Tenth Annual IEEE Conference*, 1995.

[15] A. Küegel. Improved exact solver for the weighted max-sat problem. In *Daniel Le Berre (editor). POS-10. Pragmatics of SAT, vol 8, pages 15–27*, 2012.

[16] J. Larrosa and F. Heras. Resolution in max-sat and its relation to local consistency in weighted csps. In *IJCAI*, pages 193–198, 01 2005.

[17] C.-M. Li, F. Manyà, and J. R. Soler. A clause tableau calculus for maxsat. In *IJCAI*, IJCAI'16, page 766–772. AAAI Press, 2016.

[18] C.-M. Li, F. Manyà, and J. Planes. New inference rules for max-sat. *J. Artif. Intell. Res. (JAIR)*, 30:321–359, 09 2007.

[19] D. Loveland. A linear format for resolution. *Symposium on Automatic Demonstration*, pages 147–162, 01 1970.

[20] M. Luisa Bonet, J. Levy, and F. Manyà. Resolution for max-sat. *Artificial Intelligence Volume 171, Issues 8–9, June 2007, Pages 606-618*, 2007.

[21] J. P. Marques Silva and K. A. Sakallah. Grasp-a new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.

[22] N. Narodytska and F. Bacchus. Maximum satisfiability using core-guided maxsat resolution. In *AAAI*, 2014.

[23] K. Pipatsrisawat and A. Darwiche. On the power of clause-learning sat solvers as resolution engines. *Artificial Intelligence*, 175(2):512 – 525, 2011.

[24] J. A. Robinson. A machine-oriented logic based on the resolution principle. In *Journal of the Association for Computing Machinery, vol. 12 (1965), pp. 23–41.*, 1965.

[25] A. Urquhart. The complexity of propositional proofs. *Bull. Symbolic Logic*, 1(4):425–467, 12 1995.

[26] A. Urquhart. A near-optimal separation of regular and general resolution. *SIAM J. Comput.*, 40:107–121, 01 2011.

Actes JFPC 2021

Approche basée sur la Relaxation pour la Fouille de Motifs Fermés et Diversifiés

A. Hien[1]* **S. Loudni**[3] **N. Aribi**[2] **Y. Lebbah**[2] **M. Laghzaoui**[2] **A. Ouali**[1] **A. Zimmermann**[1]

[1] Normandie Univ., UNICAEN, CNRS – UMR GREYC, France
[2] University of Oran1, Lab. LITIO, 31000 Oran, Algeria
[3] TASC (LS2N-CNRS), IMT Atlantique, FR – 44307 Nantes, France

La fouille interactive de motifs a fait évolué la fouille de données vers un modèle centré utilisateur [3, 6]. Il s'agit de prendre en compte les préférences de l'utilisateur afin de guider la recherche vers des motifs intéressants pour lui. Cela est rendu possible par l'introduction de mécanismes de *feedback* qui permettent à l'utilisateur de spécifier ses préférences sur les motifs qu'on lui présente [3]. Un aspect important de ce cadre est la nécessité de présenter à l'utilisateur des *résultats diversifiés* [1, 2].

Dans cet article, nous utilisons la programmation par contraintes pour extraire efficacement les motifs fréquents fermés diversifiés. La diversité est contrôlée par un seuil sur l'indice de `Jaccard`. Nous montrons que cette mesure n'a pas de propriété de monotonie, ce qui rend le processus d'extraction infaisable. Pour y remédier, nous proposons une nouvelle contrainte globale, CLOSEDDIVERSITY, qui exploite une relaxation anti-monotone de l'indice de Jaccard pour élaguer les motifs non diversifiés. Une seconde relaxation, basée sur une borne supérieure, est exploité via une nouvelle heuristique de branchement. [1].

Notion de diversité entre motifs. Soit $\mathcal{I}$ un ensemble de n *items*, un *motif* P est un sous-ensemble non vide de $\mathcal{I}$. Une base transactionnelle $\mathcal{D}$ est un multi-ensemble de transactions sur $\mathcal{I}$, où chaque *transaction* t est un sous-ensemble de $\mathcal{I}$, i.e., $t \subseteq \mathcal{I}$. Un motif P *apparaît* dans une transaction t, ssi $P \subseteq t$. La *couverture* de P dans $\mathcal{D}$ est l'ensemble des transactions qui le supportent : $\boldsymbol{t}(P) = \{t \in \mathcal{D} \mid p \subseteq t\}$. Le *support* de P dans $\mathcal{D}$ est le cardinal de sa couverture : $sup(P) = |\boldsymbol{t}(P)|$. Un motif P est dit *fréquent* si son support dépasse un seuil de fréquence minimal θ, $sup(P) \geq \theta$. The *clôture* d'un motif P, notée $Clos(P)$, est l'ensemble des items communs à toutes les transactions dans $\boldsymbol{t}(P)$: $Clos(P) = \{i \in \mathcal{I} \mid \forall t \in \boldsymbol{t}(P), i \in t\}$. Un motif P est dit *fermé* ssi $Clos(P) = P$.

L'indice de Jaccard est une mesure de similarité classique sur les ensembles. Nous l'utilisons pour quantifier le chevauchement des couvertures entre deux motifs.

Definition 1 (Indice de Jaccard) *Soient deux motifs P et Q, l'indice de Jaccard est le rapport entre le cardinal de l'intersection et le cardinal de l'union des couvertures des deux motifs :* $Jac(P, Q) = \frac{|\boldsymbol{t}(P) \cap \boldsymbol{t}(Q)|}{|\boldsymbol{t}(P) \cup \boldsymbol{t}(Q)|}$.

Un indice de Jaccard plus petit est synonyme d'une faible similarité en termes de couverture entre motifs et peut donc être utilisé comme mesure de diversité entre des paires de motifs.

Definition 2 (Contrainte de Jaccard/Diversité) *Soit P et Q deux motifs, Jac la mesure de Jaccard et J_{max} un seuil de diversité, la paire P et Q est dite diversifiée ssi $Jac(P, Q) \leq J_{max}$.*

Notre objectif est d'exploiter la contrainte de Jaccard durant la recherche pour élaguer les motifs non-diversifiés. Toutefois, comme l'indique la proposition 1, la contrainte de Jaccard n'est ni monotone ni anti-monotone, ce qui implique un élagage limité lors de la recherche. Pour faire face à ce problème, nous proposons deux relaxations anti-monotones : (i) Une relaxation par la borne inférieure, permettant d'élaguer les motifs non-diversifiés lors de la recherche, (ii) une relaxation par la borne supérieure pour trouver les items menant vers des motifs diversifiés.

Proposition 1 *Soit P, Q et P' trois motifs avec $P \subset P'$. $Jac(P, Q)$ peut être plus petit, égal ou supérieur à $Jac(P', Q)$.*

Relaxations de la contrainte de Jaccard. L'idée clé de notre approche est d'exploiter une relaxation de la contrainte de Jaccard ayant des propriétés de monotonicité afin de les exploiter pour la réduction de l'espace de recherche.

Definition 3 (Formulation du problème) *Soit $\mathcal{H}$ l'historique courant contenant les paires de motifs clos fréquents diversifiés, J_{max} un seuil de diversité, LB_J et UB_J les bornes inférieure et supérieure de l'indice de Jaccard, le problème relaxé consiste à extraire les motifs P tels que $\forall H \in \mathcal{H}$, $LB_J(P, H) \leq J_{max}$. Quand $UB_J(P, H) \leq J_{max}$, pour tout $H \in \mathcal{H}$, alors la contrainte Jaccard est satisfaite.*

Nous présentons à présent une borne inférieure et supérieure de l'indice de Jaccard et montrons comment les exploiter. Pour cela, nous introduisons la notion de *couverture propre* d'un motif.

*Papier doctorant : A. Hien[1] est auteur principal.

1. La version originale de ce papier est publiée dans [4].

Definition 4 (Couverture propre) *Soit P et Q deux motifs. La couverture propre de P par rapport à Q est définie par $\boldsymbol{t}_Q^{pr}(P) = \boldsymbol{t}(P) \backslash \{\boldsymbol{t}(P) \cap \boldsymbol{t}(Q)\}$.*

Proposition 2 (Borne inférieure) *Soit un motif $H \in \mathcal{H}$. Soit P un motif rencontré durant la recherche tel que $sup(P) \geq \theta$, et $\boldsymbol{t}_H^{pr}(P)$ est la couverture propre de P par rapport à H. $LB_J(P,H) = \frac{\theta - |\boldsymbol{t}_H^{pr}(P)|}{|\boldsymbol{t}(P)| + |\boldsymbol{t}(H)| + |\boldsymbol{t}_H^{pr}(P)| - \theta}$ est un minorant de $Jac(P,H)$.*

Proposition 3 (Monotonicité du LB_J) *Soit $H \in \mathcal{H}$ un motif. Pour toute paire de motifs P et Q avec $P \subseteq Q$, nous avons $LB_J(P,H) \leq LB_J(Q,H)$.*

La propriété 3 définit une condition nécessaire d'élagage basée sur la monotonie de la borne inférieure : Si $LB_J(P,H) > J_{max}$, alors aucun motif $Q \supseteq P$ ne pourra satisfaire la contrainte de Jaccard (car LB_J est une borne inférieure), rendant la contrainte elle-même *anti-monotone*. Ainsi, le motif Q peut être élagué.

Comme notre relaxation est une sur-estimation des solutions de la contrainte Jaccard, on pourrait alors avoir des motifs P tels que $LB_J(P,H) < J_{max}$ mais $Jac(P,H) > J_{max}$. Pour traiter ce cas, nous définissons une borne supérieure sur l'indice Jaccard pour évaluer la satisfaction de la contrainte, c'est-à-dire les cas avec $UB_J(P,H) \leq J_{max}, \forall\, H \in \mathcal{H}$.

Proposition 4 (Borne supérieure) *Soit un élément H de l'historique $\mathcal{H}$, et un motif P tel que $sup(P) \geq \theta$. $UB_J(P,H) = \frac{|\boldsymbol{t}(H) \cap \boldsymbol{t}(P)|}{|\boldsymbol{t}_P^{pr}(H)| + \max\{\theta, |\boldsymbol{t}(H)| \cap |\boldsymbol{t}(P)|\}}$ est un majorant de $Jac(P,H)$.*

Proposition 5 (Anti-monotonicité de UB_J) *Soit $H \in \mathcal{H}$ un motif. Pour toute paire de motifs P et Q avec $P \subseteq Q$, nous avons $UB_J(P,H) \geq UB_J(Q,H)$.*

La contrainte globale CLOSEDDIVERSITY. Pour représenter un motif inconnu, nous utilisons un vecteur x de variables booléennes $(x_1, \ldots, x_{|\mathcal{I}|})$, où x_i désigne la présence de l'item $i \in \mathcal{I}$ dans le motif. Nous utiliserons les notations suivantes : $x^+ = \{i \in \mathcal{I} \,|\, dom(x_i) = \{1\}\}$ les items présents, $x^- = \{i \in \mathcal{I} \,|\, dom(x_i) = \{0\}\}$ les items absents, et $x^* = \{i \in \mathcal{I} \,|\, i \notin x^+ \cup x^-\}$ l'ensemble des items non-assignés.

Definition 5 (CLOSEDDIVERSITY) *Soit x une instanciation complète des variables d'items, $\mathcal{H}$ un historique de motifs clos fréquents diversifiés (initialement vide), θ un seuil de fréquence, J_{max} un seuil de diversité et $\mathcal{D}$ une base transactionnelle. La contrainte* CLOSEDDIVERSITY$_{\mathcal{D},\theta}(x, \mathcal{H}, J_{max})$ *est satisfaite ssi : (1) $Clos(Px^+) = x^+$; (2) $sup(x^+) \geq \theta$; (3) $\forall\, H \in \mathcal{H}, LB_J(x^+, H) \leq J_{max}$.*

Notre contrainte globale permet de mettre à jour $\mathcal{H}$ de manière incrémentale durant la recherche. Le propagateur de CLOSEDDIVERSITY exploite les règles de filtrage de la contrainte CLOSEDPATTERNS [5]. Il utilise également notre relaxation LB pour supprimer les items qui ne peuvent mener à une solution contenant x^+. Notons par x^-_{Freq} les ietms filtrés par la règle des items non fréquents et par x^-_{Div} les items filtrés par notre règle sur LB.

Proposition 6 (Filtrage de CLOSEDDIVERSITY) *Soit $\mathcal{H}$ un historique de motifs clos fréquents diversifiés, x^+ une instanciation partielle sur x, et un item libre $i \in x^*$, $x^+ \cup \{i\}$ ne peut mener à un motif diversifié dans l'un des deux cas suivants : (1) si $\exists\, H \in \mathcal{H}$ s.t. $LB_J(x^+ \cup \{i\}, H) > J_{max}$, alors $1 \notin dom(x_i)$; (2) if $\exists\, k \in x^-_{Div}$ s.t. $\boldsymbol{t}(x^+ \cup \{i\}) \subseteq \boldsymbol{t}(x^+ \cup \{k\})$, alors $LB_J(x^+ \cup \{i\}, H) > LB_J(x^+ \cup \{k\}, H) > J_{max}$ et $1 \notin dom(x_i)$.*

Exploiter la borne UB_J. On peut avoir $LB_J(x^+, H) \leq J_{max}$ mais $Jac(x^+, H) > J_{max}$. Nous proposons d'exploiter notre relaxation UB pour garantir la satisfaction de la contrainte Jaccard. Pendant la recherche, nous calculons incrémentalement $UB(x^+ \cup \{i\}, H)$ de toute extension de l'affectation partielle x^+ avec un item libre i. Si, $\forall H \in \mathcal{H}, UB_J(x^+ \cup \{j\}, H) \leq J_{max})$, alors la contrainte est satisfaite (i.e. propriété de *temoin*). De plus, grâce à l'anti-monotonie de UB_J (voir Proposition 5), tous les sur-ensembles de $x^+ \cup \{i\}$ satisferont également la contrainte de Jaccard. Pour accélérer la certification des solutions, nous proposons une nouvelle heuristique d'ordre sur les variables qui *sélectionne le premier item libre satisfaisant la propriété témoin*, sinon nous sélectionnons la variable ayant le plus petit support estimé (voir [4] pour plus de détails).

Résultats expérimentaux. Nous avons évalué les performances de notre contrainte globale CLOSEDDIV, ainsi que la qualité des résultats (en terme de diversité) sur les jeux de données UCI. Les résultats confirment l'efficacité de la règle de filtrage LB pour l'élagage des motifs non diversifiés. Ces résultats montrent aussi que la borne supérieure se rapproche beaucoup de la valeur de l'indice Jaccard, ce qui signifie que notre borne UB fournit une relaxation serrée. Ceci est révélateur de la qualité des motifs trouvés en termes de diversité et démontre l'intérêt et la force de notre règle de branchement pour obtenir des motifs diversifiés par rapport à la contrainte globale CLOSEDPATTERNS.

Dans ce papier, nous avons proposé une approche introduite sous la forme d'une contrainte globale appelée CLOSEDDIV. Cette contrainte exploite deux relaxations LB/UB(anti-)monotones, qui permettent d'avoir un filtrage et une règle de branchement efficaces, dynamisant l'ensemble du processus de recherche.

Références

[1] G. BOSC, J-F. BOULICAUT, Ch. RAÏSSI et M. KAYTOUE : Anytime discovery of a diverse set of patterns with monte carlo tree search. *Data mining and knowledge discovery*, 32(3):604–650, 2018.

[2] B. BRINGMANN et A. ZIMMERMANN : The chosen few : On identifying valuable patterns. *In Proceedings of ICDM 2007*, pages 63–72, 2007.

[3] V. DZYUBA et M. van LEEUWEN : Interactive discovery of interesting subgroup sets. *In International Symposium on Intelligent Data Analysis*, pages 150–161. Springer, 2013.

[4] A. HIEN, S. LOUDNI, N. ARIBI, Y. LEBBAH, M. LAGHZAOUI, A. OUALI et A. ZIMMERMANN : A relaxation-based approach for mining diverse closed patterns. *In Proceedings of ECML PKDD 2020*, volume 12457, pages 36–54. Springer, 2020.

[5] N. LAZAAR, Y. LEBBAH, S. LOUDNI, M. MAAMAR, V. LEMIÈRE, C. BESSIERE et P. BOIZUMAULT : A global constraint for closed frequent pattern mining. *In Proceedings of the 22nd CP*, pages 333–349, 2016.

[6] M. van LEEUWEN : *Interactive Data Exploration Using Pattern Mining*, pages 169–182. Springer, 2014.

A Relaxation-based Approach for Mining Diverse Closed Patterns

Arnold Hien[2], Samir Loudni[2,3]✉, Noureddine Aribi[1], Yahia Lebbah[1], Mohammed Laghzaoui[1], Abdelkader Ouali[2], and Albrecht Zimmermann[2]

[1] University of Oran1, Lab. LITIO, 31000 Oran, Algeria
[2] Normandie Univ., UNICAEN, CNRS – UMR GREYC, France
[3] TASC (LS2N-CNRS), IMT Atlantique, FR – 44307 Nantes, France

Abstract. In recent years, pattern mining has moved from a slow-moving repeated three-step process to a much more agile iterative/user-centric mining model. A vital ingredient of this framework is the ability to *quickly* present a set of *diverse* patterns to the user. In this paper, we use constraint programming (well-suited to user-centric mining due to its rich constraint language) to efficiently mine a diverse set of closed patterns. Diversity is controlled through a threshold on the Jaccard similarity of pattern occurrences. We show that the Jaccard measure has no monotonicity property, which prevents usual pruning techniques and makes classical pattern mining unworkable. This is why we propose anti-monotonic lower and upper bound relaxations, which allow effective pruning, with an efficient branching rule, boosting the whole search process. We show experimentally that our approach significantly reduces the number of patterns and is very efficient in terms of running times, particularly on dense data sets.

1 Introduction

The original data analysis model based on pattern mining consists of three steps in a kind of *multi-waterfall cycle*: 1) a user chooses the values of one or several mining parameters, 2) an underlying engine extracts patterns (often taking not inconsiderable time to do so), and 3) the user sifts through a (potentially very large) set of result patterns and interprets them, using their insights to return to the first step and repeat the cycle.

Recently, this approach has been challenged by an increasing focus on *user-centered*, *interactive*, and *anytime* pattern mining [14]. This new paradigm stresses that users should be presented quickly with patterns likely to be interesting to them, and typically affect later iterations of the mining process by giving feedback. A powerful framework for taking a variety of user feedback into account is pattern mining via constraint programming (CP). Much of the current focus in this domain is on user-centered/interactive mining, particularly the ability to elicit and exploit user feedback [9, 14, 18]. An important aspect of requesting such feedback is that the user be quickly presented with *diverse* results. If patterns are too similar to each other, deciding which one to prefer can become challenging, and if they appear in several successive iterations, it eventually becomes a slog. Similarly, a method that produces diverse results but takes a long time to do so, risks that the user checks out of the process. Older work on diversity either post-process patterns derived from the process described above [5, 12, 21], use heuristics [20] or view

it purely from the point of view of speeding up the extraction process [8]. Recent work, on the other hand, pushes diversity constraints into the mining process itself [3, 4]. At the algorithmic level, additional user-specified constraints often require new implementations to filter out the patterns violating or satisfying the user's constraints, which can be computationally infeasible for large databases.

In the last decade, data mining has been combined with constraint programming to model various data mining problems [2,6,13,19]. The main advantage of CP for pattern mining is its declarativity and flexibility, which include the ability to incorporate new user-specified constraints without the need to modify the underlying system. Moreover, CP allows to define flexible search strategies.[4] In this paper, we propose to add to the literature on *explicitly* taking the diversity of patterns (in terms of the data instances they describe) into account and to use an exhaustive process to find candidates for inclusion into a result set. To achieve this, we use the widely accepted Jaccard index to compare patterns and formulate a diversity constraint, which has no monotonicity property, implying limited pruning during search. To cope with this problem, we propose two anti-monotonic relaxations: (i) A lower bound relaxation, which allows to prune non-diverse items during search. This is integrated in our constraint programming based approach through a new global constraint taking into account diversity with its filtering algorithms (aka, propagators); (ii) An upper bound relaxation to find items ensuring diversity. This is exploited through a new branching rule, boosting the search process towards diverse patterns. We demonstrate the performance of our proposed method experimentally, comparing to the state-of-the-art in CP-based closed pattern mining.

2 Preliminaries

2.1 Itemset Mining

Let $\mathcal{I} = \{1, ..., n\}$ be a set of n *items*, an *itemset* (or pattern) P is a non-empty subset of $\mathcal{I}$. The language of itemsets corresponds to $\mathcal{L}_\mathcal{I} = 2^\mathcal{I} \backslash \emptyset$. A transactional dataset $\mathcal{D}$ is a bag (or multiset) of transactions over $\mathcal{I}$, where each *transaction* t is a subset of $\mathcal{I}$, i.e., $t \subseteq \mathcal{I}$; $\mathcal{T} = \{1, ..., m\}$ a set of m *transaction* indices. An itemset P *occurs* in a transaction t, iff $P \subseteq t$. The *cover* of P in $\mathcal{D}$ is the set of transactions in which it occurs: $\mathcal{V}_\mathcal{D}(P) = \{t \in \mathcal{D} \mid p \subseteq t\}$. The *support* of P in $\mathcal{D}$ is the size of its cover: $sup_\mathcal{D}(P) = |\mathcal{V}_\mathcal{D}(P)|$. An itemset P is said to be *frequent* when its support exceeds a user-specified minimal threshold θ, $sup_\mathcal{D}(P) \geq \theta$. Given $S \subseteq \mathcal{D}$, $items(S)$ is the set of common items belonging to all transactions in S: $items(S) = \{i \in \mathcal{I} \mid \forall t \in S, i \in t\}$. The *closure* of an itemset P, denoted by $Clos(P)$, is the set of common items that belong to all transactions in $\mathcal{V}_\mathcal{D}(P)$: $Clos(P) = \{i \in \mathcal{I} \mid \forall t \in \mathcal{V}_\mathcal{D}(P), i \in t\}$. An itemset P is said to be *closed* iff $Clos(P) = P$. Constraint-based pattern mining aims at extracting all patterns P of $\mathcal{L}_\mathcal{I}$ satisfying a selection predicate c (called *constraint*) which is usually called *theory* [5]: $Th(c)$. A common example is the frequency measure leading to the minimal support constraint, which can be combined with the closure constraint to mine closed frequent itemsets.

[4] Opposed to more rigid search in classical pattern mining algorithms, which often rely on exploiting the properties of a particular constraint.

Example 1. Figure 1 shows the itemset lattice derived from a toy dataset with five items and 100 transactions. As the figure shows, there exist 26 frequent closed itemsets with $\theta = 7$.

Most constraint-based mining algorithms take advantage of monotonicity which offers pruning conditions to safely discard non-promising patterns from the search space. Several frameworks exploit this principle to mine with a monotone or an anti-monotone constraint. Other classes of constraints have also been considered [15, 16]. However, for constraints that are not anti-monotone, pushing them into the discovery algorithm might lead to less effective pruning phases. Thus, we propose in this paper to exploit the *witness* concept introduced in [11] to handle such constraints. A witness is a single itemset on which we can test whether a constraint holds and derive information about properties of other itemsets.

Definition 1 (Witness). *Let P, Q itemsets, and $C : \mathcal{I} \mapsto \{true, false\}$, then W, $P \subseteq W \subseteq P \cup Q$, is called a positive (negative) witness iff $\forall P', P \subseteq P' \subseteq P \cup Q : C(W) = true \Rightarrow C(P') = true$ ($C(W) = false \Rightarrow C(P') = false$).*

2.2 Diversity of Itemsets

The Jaccard index is a classical similarity measure on sets. We use it to quantify the overlap of the covers of itemsets.

Definition 2 (Jaccard index). *Given two itemsets P and Q, the Jaccard index is the relative size of the overlap of their covers : $Jac(P, Q) = \frac{|\mathcal{V}_{\mathcal{D}}(P) \cap \mathcal{V}_{\mathcal{D}}(Q)|}{|\mathcal{V}_{\mathcal{D}}(P) \cup \mathcal{V}_{\mathcal{D}}(Q)|}$.*

A lower Jaccard indicates low similarity between itemset covers, and can thus be used as a measure of diversity between pairs of itemsets.

Definition 3 (Diversity/Jaccard constraint). *Let P and Q be two itemsets. Given the Jac measure and a diversity threshold J_{max}, we say that P and Q are pairwise diverse iff $Jac(P, Q) \leq J_{max}$. We will denote this constraint by c_{Jac}.*

Our aim is to push the Jaccard constraint during pattern discovery to prune non-diverse itemsets. To achieve this, we maintain a *history* $\mathcal{H}$ of extracted pairwise diverse itemsets during search and constrain the next mined itemsets to respect a maximum Jaccard constraint with all itemsets already included in $\mathcal{H}$. This problem can be formalized as follows.

Definition 4 (k diverse frequent itemsets). *Given a current history $\mathcal{H} = \{H_1, \ldots, H_k\}$ of k pairwise diverse frequent closed itemsets, the Jac measure and a diversity threshold J_{max}, the task is to mine new itemsets P such that $\forall\, H \in \mathcal{H}$, $Jac(P, H) \leq J_{max}$.*

Example 2. The lattice in Figure 1 depicts the set of diverse FCIs (marked with blue and green solid line circles) with $J_{max} = 0.19$ and $\mathcal{H} = \{BE\}$. ACE is a diverse FCI (i.e., $Jac(ACE, BE) = 0.147 < 0.19$).

Proposition 1. *Let P, Q and P' be three itemsets s.t. $P \subset P'$. $Jac(P, Q)$ may be smaller, equal or greater than $Jac(P', Q)$.*

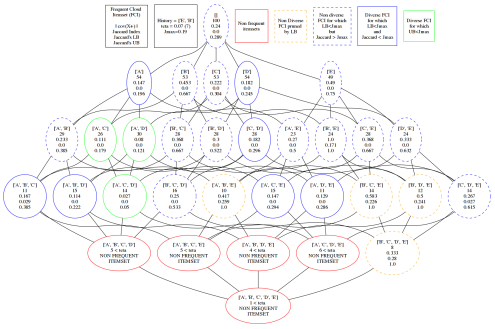


Fig. 1: The powerset lattice of frequent closed itemsets ($\theta = 7$) for the dataset $\mathcal{D}$ of Example 1.

Based on the above proposition, the anti-monotonicity of the maximum Jaccard constraint does not hold, which disables pruning. Thus, instead of solving the problem of Definition 4 directly, we introduce bounds in Section 3 that allow us to prune the search space using a relaxation of the Jaccard constraint. The appeal of this approach is that we are able to infer monotone and anti-monotone properties from this relaxation.

2.3 Constraint Programming (CP)

Constraint programming [10] is a powerful paradigm which offers a generic and modular approach to model and solve combinatorial problems. A CP model consists of a set of variables $X = \{x_1, \ldots, x_n\}$, a set of domains D mapping each variable $x_i \in X$ to a finite set of possible values $dom(x_i)$, and a set of constraints $\mathcal{C}$ on X. A constraint $c \in \mathcal{C}$ is a relation that specifies the allowed combinations of values for its variables $X(c)$. An assignment on a set $Y \subseteq X$ of variables is a mapping from variables in Y to values in their domains. A solution is an assignment on X satisfying all constraints. Constraint solvers typically use backtracking search to explore the search space of partial assignments. Algorithm 1 provides a general overview of a CP solver. At each node of the search tree, procedure *Constraint-Search* selects an unassigned variable (line 8) according to user-defined heuristics and assigns it a value (line 9). It backtracks when a constraint cannot be satisfied, i.e. when at least one domain is empty (line 5). A solution is obtained (line 12) when each domain $dom(x_i)$ is reduced to a singleton and all constraints are satisfied. The main concept used to speed up the search is constraint propagation by *Filtering algorithms*. At each assignment, constraint filtering algorithms prune the search space by enforcing local consistency properties like *domain consistency*. A constraint c on $X(c)$ is domain consistent, if and only if, for every $x_i \in X(c)$ and every $v \in dom(x_i)$, there is an assignment satisfying c such that $(x_i = v)$. Global constraints are families of constraints defined by a relation on any number of variables [10].

2.4 A CP Model for Frequent Closed Itemset Mining

The first constraint programming model for frequent closed itemset mining (FCIM) was introduced in [6]. It is based on reified constraints to connect item variables to transac-

Algorithm 1: Constraint-Search(D)

```
1  In: X : a set of decision variables; C : a set of constraints;
2  InOut: D : a set of variable domains;
3  begin
4      D ← Filtering(D, C)
5      if there exists x_i ∈ X s.t. dom(x_i) is empty then
6          return failure
7      if there exists x_i ∈ X s.t. |dom(x_i)| > 1 then
8          Select x_i ∈ X s.t. |dom(x_i)| > 1
9          forall v ∈ dom(x_i) do
10             Constraint-Search(Dom ∪ {x_i → {v}})
11     else
12         output solution D
```

tion variables. The first global constraint CLOSEDPATTERNS for mining frequent closed itemsets was proposed in [13]. The global constraint COVERSIZE for computing the exact size of the cover of an itemset was introduced in [19]. It offers more flexibility in modeling problems. We present the global constraint CLOSEDPATTERNS.

Global Constraint CLOSEDPATTERNS. Most declarative methods use a vector x of Boolean variables $(x_1, \ldots, x_{|\mathcal{I}|})$ for representing itemsets, where x_i represents the presence of the item $i \in \mathcal{I}$ in the itemset. We will use the following notations: $x^+ = \{i \in \mathcal{I} \,|\, dom(x_i) = \{1\}\}$, $x^- = \{i \in \mathcal{I} \,|\, dom(x_i) = \{0\}\}$ and $x^* = \{i \in \mathcal{I} \,|\, i \notin x^+ \cup x^-\}$.

Definition 5 (CLOSEDPATTERNS). *Let x be a vector of Boolean variables, θ a support threshold and $\mathcal{D}$ a dataset. The global constraint* CLOSEDPATTERNS$_{\mathcal{D},\theta}(x)$ *holds if and only if x^+ is a closed frequent itemset w.r.t. the threshold θ.*

Definition 6 (Closure extension [22]). *A non-empty itemset P is a closure extension of Q iff $\mathcal{V}_{\mathcal{D}}(P \cup Q) = \mathcal{V}_{\mathcal{D}}(Q)$.*

Filtering of CLOSEDPATTERNS. [13] also introduced a complete filtering algorithm for CLOSEDPATTERNS based on three rules. The first rule filters 0 from $dom(x_i)$ if $\{i\}$ is a closure extension of x^+ (see Definition 6). The second rule filters 1 from $dom(x_i)$ if the itemset $x^+ \cup \{i\}$ is infrequent w.r.t. θ. Finally, the third rule filters 1 from $dom(x_i)$ if $\mathcal{V}_{\mathcal{D}}(x^+ \cup \{i\})$ is a subset of $\mathcal{V}_{\mathcal{D}}(x^+ \cup \{j\})$ where j is an absent item, i.e. $j \in x^-$.

To show the strength and the flexibility of the CP approach in taking into account user's constraints, we formulate a CP model to extract more specific patterns using the following four global constraints : $\mathcal{C} = \{$CLOSEDP$_{\mathcal{D},\theta}(X)$, ATLEAST(X, lb), KNAPSACK(X, z, w), REGULAR$(X, DFA)\}$ The ATLEAST constraint enforces that at least lb variables in X are assigned to 1; the KNAPSACK constraint restricts a weighted linear sum to be no more than a given capacity z, i.e. $\sum_i w_i X_i \leq z$; the REGULAR constraint imposes that X is accepted by deterministic finite automaton (DFA), which recognizes a regular expression.

Example 3. Let us consider the example of Figure 1 using $lb = 2$, $w = \langle 8, 7, 5, 14, 16 \rangle$, $z = 20$, $\theta = 7$ and the regular expression $0^* \, 1^+ \, 0^*$ ensuring items' contiguity. Solving this CP model provides the solution: $Th(\mathcal{C}) = \{\{AB\}, \{BC\}, \{CD\}, \{ABC\}\}$.

3 A CP Model for Mining Diverse Frequent Closed Itemsets

We present our approach for computing diverse FCIs. The key idea is to compute an approximation of the set of diverse FCIs by defining two bounds on the Jaccard index that allow us to reduce the search space. All the proofs are given in the Supp. material [1].

3.1 Problem Reformulation

Proposition 1 states that the Jaccard constraint is neither monotonic nor anti-monotonic. So, we propose to approximate the theory of the original constraint c_{Jac} by a larger collection corresponding to the solution space of its *relaxation* c^r_{Jac}: $Th(c_{Jac}) \subseteq Th(c^r_{Jac})$. The key idea is to formulate a relaxed constraint having suitable monotonicity properties in order to exploit them for search space reduction. More precisely, we want to exploit *upper and lower bounding* operators to derive a monotone relaxation and an anti-monotone one of c_{Jac}.

Definition 7 (Problem reformulation). *Given a current history $\mathcal{H} = \{H_1, \ldots, H_k\}$ of extracted k pairwise diverse frequent closed itemsets, a diversity threshold J_{max}, a lower bound LB_J and an upper bound UB_J on the Jaccard index, the relaxed problem consists of mining candidate itemsets P such that $\forall H \in \mathcal{H}$, $LB_J(P, H) \leq J_{max}$. When $UB_J(P, H) \leq J_{max}$, for all $H \in \mathcal{H}$, the Jaccard constraint is fully satisfied.*

3.2 Jaccard Lower Bound

Let us now formalize how to compute the lower bound and how to exploit it. To arrive at a lower bound for the Jaccard value between two itemsets, we need to consider the situation where the *overlap* between them has become as small as possible, while the coverage that is proper to each itemset remains as large as possible.

Definition 8 (Proper cover). *Let P and Q be two itemsets. The proper cover of P w.r.t. Q is defined as $\mathcal{V}^{pr}_Q(P) = \mathcal{V}_\mathcal{D}(P) \backslash \{\mathcal{V}_\mathcal{D}(P) \cap \mathcal{V}_\mathcal{D}(Q)\}$.*

The lowest possible Jaccard would reduce the numerator to 0, which is however not possible under the minimum support threshold θ. The denominator, on the other hand, consists of $|\mathcal{V}_\mathcal{D}(H)|$ (which cannot change) and the part of P's coverage that does not overlap with H, i.e. $\mathcal{V}^{pr}_H(P)$.

Proposition 2 (Lower bound). *Consider a member pattern H of the history $\mathcal{H}$. Let P an itemset encountered during search such that $sup_\mathcal{D}(P) \geq \theta$, and $\mathcal{V}^{pr}_H(P)$ be the proper cover of P w.r.t. H. $LB_J(P, H) = \dfrac{\theta - |\mathcal{V}^{pr}_H(P)|}{|\mathcal{V}_\mathcal{D}(P)| + |\mathcal{V}_\mathcal{D}(H)| + |\mathcal{V}^{pr}_H(P)| - \theta}$ is a lower bound of $Jac(P, H)$.*

The lower bound on the Jaccard index enables us to discard some non-diverse itemsets, i.e., those with an LB_J value greater than J_{max} are *negative witnesses*.

Example 4. The set of all non diverse FCIs with a lower bound value greater than J_{max} are marked in Figure 1 with orange line circles.

Proposition 3 (Monotonicity of LB_J). *Let $H \in \mathcal{H}$ be an itemset. For any two itemsets $P \subseteq Q$, the relationship $LB_J(P,H) \leq LB_J(Q,H)$ holds.*

Property 3 establishes an important result to define a pruning condition based on the monotonicity of the lower bound (cf. Section 3.4). If $LB_J(P,H) > J_{max}$, then no itemset $Q \supseteq P$ will satisfy the Jaccard constraint (because LB_J is a lower bound), rendering the constraint itself *anti-monotone*. So, we can safely prune Q.

3.3 Jaccard Upper Bound

As our relaxation approximates the theory of the Jaccard constraint, i.e. $Th(c_{Jac}) \subseteq Th(c^r_{Jac})$, one could have itemsets P such that $LB_J(P,H) < J_{max}$ but $Jac(P,H) > J_{max}$ (see itemsets marked with blue dashed line circles in Figure 1). To tackle this case, we define an upper bound on the Jaccard index to evaluate the satisfaction of the Jaccard constraint, i.e., those with $UB_J(P,H) \leq J_{max}$, $\forall H \in \mathcal{H}$, are *positive witnesses*.

To derive the upper bound, we need to follow the opposite argument as for the lower bound: the highest possible Jaccard will be achieved if $\mathcal{V}_\mathcal{D}(H) \cap \mathcal{V}_\mathcal{D}(P)$ stays unchanged but the set $\mathcal{V}^{pr}_H(P)$ is reduced as much possible (under the minimum support constraint). If the intersection is greater than or equal θ, in the worst case scenario (leading to the highest Jaccard), a future P' covers only transactions in the intersection. If not, the denominator needs to contain a few elements of $\mathcal{V}^{pr}_H(P)$, $\theta - |\mathcal{V}_\mathcal{D}(H) \cap \mathcal{V}_\mathcal{D}(P)|$, to be exact.

Proposition 4 (Upper bound). *Given a member pattern H of the history $\mathcal{H}$, and an itemset P such that $sup_\mathcal{D}(P) \geq \theta$. $UB_J(P,H) = \dfrac{|\mathcal{V}_\mathcal{D}(H) \cap \mathcal{V}_\mathcal{D}(P)|}{|\mathcal{V}^{pr}_P(H)| + \max\{\theta, |\mathcal{V}_\mathcal{D}(H)| \cap |\mathcal{V}_\mathcal{D}(P)|\}}$ is an upper bound of $Jac(P,H)$.*

Example 5. The set of all diverse FCIs with UB_J values less than J_{max} are marked with green line circles in Figure 1.

Our upper bound can be exploited to evaluate the Jaccard constraint during mining. More precisely, in the enumeration procedure, if the upper bound of the current candidate itemset P is less than J_{max}, then c_{Jac} is fully satisfied. Moreover, if the upper bound is *monotonically decreasing* (or anti-monotonic), then all itemsets Q derived from P are also diverse (see Proposition 5).

Proposition 5 (Anti-monotonicity of UB_J). *Let H be a member pattern of the history $\mathcal{H}$. For any two itemsets $P \subseteq Q$, the relationship $UB_J(P,H) \geq UB_J(Q,H)$ holds.*

3.4 The Global Constraint CLOSEDDIVERSITY

This section presents our new global constraint CLOSEDDIVERSITY that exploits the LB relaxation to mine pairwise diverse frequent closed itemsets.

Algorithm 2: Filtering for CLOSEDDIVERSITY

```
1  In: θ, J_max : frequency and diversity thresholds; H : history of solutions encountered during search;
2  InOut: x = {x_1 ... x_n} : Boolean item variables;
3  begin
4      if ( |V_D(x^+)| < θ ∨ !PGrowth_LB(x^+, H, J_max)) then return false;
5      foreach i ∈ x^* do
6          if (|V_D(x^+ ∪ {i})| < θ) then
7              dom(x_i) ← dom(x_i) − {1}; x^-_Freq ← x^-_Freq ∪ {i}; x^* ← x^* \ {i}; continue;
8          if (|V_D(x^+ ∪ {i})| = |V_D(x^+)|) then
9              dom(x_i) ← dom(x_i) − {0}; x^+ ← x^+ ∪ {i};    x^* ← x^* \ {i};
10         if (! PGrowth_LB(x^+ ∪ {i}, H, J_max)) then
11             dom(x_i) ← dom(x_i) − {1}; x^-_Div ← x^-_Div ∪ {i}; x^* ← x^* \ {i}; continue;
12         foreach k ∈ (x^-_Freq ∪ x^-_Div) do
13             if (V_D(x^+ ∪ {i}) ⊆ V_D(x^+ ∪ {k})) then
14                 dom(x_i) ← dom(x_i) − {1}
15                 if k ∈ x^-_Freq then x^-_Freq ← x^-_Freq ∪ {i};
16                 else x^-_Div ← x^-_Div ∪ {i};
17                 x^* ← x^* \ {i}; break;
18     return true;
19 Function PGrowth_LB(x, H, J_max): Boolean
20     foreach H ∈ H do
21         if (LB_J(x, H) > J_max) then return false
22     return true
```

Definition 9 (CLOSEDDIVERSITY). *Let x be a vector of Boolean item variables, $\mathcal{H}$ a history of pairwise diverse frequent closed itemsets (initially empty), θ a support threshold, J_{max} a diversity threshold and $\mathcal{D}$ a dataset. The* CLOSEDDIVERSITY$_{\mathcal{D},\theta}(x, \mathcal{H}, J_{max})$ *global constraint holds if and only if: (1) x^+ is closed; (2) x^+ is frequent, $sup_{\mathcal{D}}(x^+) \geq \theta$; (3) x^+ is diverse, $\forall\, H \in \mathcal{H}, LB_J(x^+, H) \leq J_{max}$.*

Initially, the history $\mathcal{H}$ is empty. Our global constraint allows to incrementally update $\mathcal{H}$ with diverse FCIs encountered during search. Condition (3) expresses a necessary condition ensuring that x^+ is diverse. Indeed, one could have $LB_J(x^+, H) \leq J_{max}$ but $Jac(x^+, H) > J_{max}$. Thus, we propose in Section 4 to exploit our UB relaxation to guarantee the satisfaction of the Jaccard constraint.

The propagator for CLOSEDDIVERSITY exploits the filtering rules of CLOSEDPATTERNS (see Section 2.4). It also uses our LB relaxation to remove items i that cannot belong to a solution containing x^+. We denote by x^-_{Freq} the set of items filtered by the rule of infrequent items and by x^-_{Div} the set of items filtered by our LB rule.

Proposition 6 (CLOSEDDIVERSITY Filtering rule). *Given a history $\mathcal{H}$ of pairwise diverse frequent closed itemsets, a partial assignment on x, and a free item $i \in x^*$, $x^+ \cup \{i\}$ cannot lead to a diverse itemset if one of the two cases holds:*
1) if $\exists\, H \in \mathcal{H}$ s.t. $LB_J(x^+ \cup \{i\}, H) > J_{max}$, then we remove 1 from $dom(x_i)$.
2) if $\exists\, k \in x^-_{Div}$ s.t. $\mathcal{V}_{\mathcal{D}}(x^+ \cup \{i\}) \subseteq \mathcal{V}_{\mathcal{D}}(x^+ \cup \{k\})$, then $LB_J(x^+ \cup \{i\}, H) > LB_J(x^+ \cup \{k\}, H) > J_{max}$, thus we remove 1 from $dom(x_i)$.

Algorithm. The propagator for CLOSEDDIVERSITY is presented in Algorithm 2. It takes as input the variables x, the support threshold θ, the diversity threshold J_{max} and the current history $\mathcal{H}$ of pairwise diverse frequent closed itemsets. It starts by computing the cover of the itemset x^+ and checks if x^+ is either infrequent or not diverse (see function $\mathcal{P}Growth_{LB}$), if so the constraint is violated and a fail is returned (line 4). Algorithm 2 extends the filtering rules of CLOSEDPATTERNS (see Section 2.4) by examining the diversity condition of the itemset $x^+ \cup \{i\}$ (see Proposition 3). For each element $H \in \mathcal{H}$, the function $\mathcal{P}Growth_{LB}(x^+ \cup \{i\}, \mathcal{H}, J_{max})$ computes the value of $LB_J(x^+ \cup \{i\}, H)$ and tests if there exists an H s.t. $LB_J(x^+ \cup \{i\}, H) > J_{max}$ (lines 20-21). If so, we return $false$ (line 21) because $x^+ \cup \{i\}$ cannot lead to a diverse itemset w.r.t. $\mathcal{H}$, remove 1 from $dom(x_i)$ (line 11), update x^-_{Div} and x^* and we continue with the next free item. Otherwise, we return true. Second, we remove 1 from each free item variable $i \in x^*$ such that its cover is a superset of the cover of an absent item $k \in (x^-_{Freq} \cup x^-_{Div})$ (lines 12-17). The LB filtering rule associated to the case $k \in x^-_{Div}$ is a new rule taking its originality from the reasoning made on absent items.

Proposition 7 (Consistency and time complexity). *Algorithm 2 enforces Generalized Arc Consistency (GAC) (a.k.a. domain consistency [10]) in* $\mathcal{O}(n^2 \times m)$.

4 Using witnesses and the estimated frequency within the search

In this section, we show how to exploit the witness property and the estimated frequency so as to design a more informed search algorithm.

Positive Witness. During search, we compute incrementally the $UB(x^+ \cup \{i\}, H)$ of any extension of the partial assignment x^+ with a free item i. If, for each $H \in \mathcal{H}$, this upper bound is less or equal to J_{max}, then c_{Jac} is fully satisfied and $x^+ \cup \{i\}$ is a *positive witness*. Moreover, thanks to the anti-monotonicity of UB_J (see Proposition 5), all supersets of $x^+ \cup \{i\}$ will satisfy the Jaccard constraint.

Estimated frequency. The frequency of an itemset can be computed as the cardinality of the intersection of its items' cover: $sup_{\mathcal{D}}(x^+) = |\cap_{i \in x^+} \mathcal{V}_{\mathcal{D}}(i)|$, the intersection between 2 covers being performed by a bitwise-AND. To limit the number of intersections, we use an estimation of the frequency of each item $i \in \mathcal{I}$ w.r.t the set of present items x^+, denoted $eSup_{\mathcal{D}}(i, x^+)$. This estimation constitutes a *lower bound* of $|\mathcal{V}_{\mathcal{D}}(x^+ \cup \{i\})|$. Interestingly, if $eSup_{\mathcal{D}}(i, x^+) \geq \theta$ then $|\mathcal{V}_{\mathcal{D}}(x^+ \cup \{i\})| \geq \theta$, meaning that the intersection between covers is performed only if $eSup_{\mathcal{D}}(i, x^+) < \theta$, thereby leading to performance enhancement. In addition, we argue that the estimated support is an interesting heuristic to reinforce the witness branching rule. Indeed, branching on the variable having the minimum estimated support (using the lower bound of the real support) will probably activate our filtering rules (see Algorithm 2), thus reducing the search space. It will be denoted as MINCOV variable ordering heuristic.

We propose Algorithm 3 as a branching procedure (returns the next variable to branch on). When the search begins, for each item $i \in x^*$, its estimated frequency is initialized to $eSup_{\mathcal{D}}(i, \emptyset) = |\mathcal{V}_{\mathcal{D}}(i)|$. Once an item j has been added to the partial solution, the estimated frequencies of unbound items must be updated (see lines 4-9). Thus, we first find the variable x^{es} having the minimal estimated support (line 4).

Algorithm 3: Branching for CLOSEDDIVERSITY

```
1  In: J_max : diversity thresholds; H : history of solutions ;
2  Out: First witness index or x^es as the item with the smallest estimated support
3  begin
4      x^es ← argmin_{i∈x*}(eSup_D(i, x^+));
5      diff ← (|V_D(x^+)| − |V_D(x^+ ∪ {x^es})|);
6      foreach i ∈ x* \ {x^es} do
7          eSup_D(i, x^+ ∪ {x^es}) ← eSup_D(i, x^+) − diff;
8          if (eSup_D(i, x^+ ∪ {x^es}) < θ) then
9              eSup_D(i, x^+ ∪ {x^es}) ← |V_D(x^+ ∪ {x^es}) ∩ V_D(i)|;
10     foreach i ∈ x* do
11         if (PGrowth_UB(x^+ ∪ {i}, H, J_max)) then
12             return ⟨i, true⟩;
13     return ⟨x^es, false⟩
14 Function PGrowth_UB(x^+ ∪ {j}, H, J_max) : Boolean
15     foreach H ∈ H do
16         if (UB_J(x^+ ∪ {j}, H) > J_max) then
17             return false
18     return true
```

Next, each item $i \in x^* \setminus \{x^{es}\}$ may lose some support, but no more than $|\mathcal{V}_\mathcal{D}(x^+)| - |\mathcal{V}_\mathcal{D}(x^+ \cup \{x^{es}\})|$, since some removed transactions may not contain i (line 5). Using this upper bound (denoted by *diff*), the estimated frequency of i is updated and set to $eSup_\mathcal{D}(i, x^+) - \mathit{diff}$ (lines 6-9). As indicated above, if $eSup_\mathcal{D}(i, x^+) \geq \theta$ then $|\mathcal{V}_\mathcal{D}(x^+ \cup \{i\})| \geq \theta$. Otherwise, we have to compute the right support by performing the intersection between covers (line 9). It is important to stress that the branching variable x^{es} will be returned (line 13) only if no positive witness is found (lines 10-12). Finally, the function $\mathcal{P}Growth_{UB}(x^+ \cup \{i\}, \mathcal{H}, J_{max})$ allows to test whether the current instantiation x^+ can be extended to a witness itemset using the free item $\{i\}$. It returns true if the upper bound of the current itemset x^+ when adding one item $\{i\}$ is less than J_{max} for all $h \in \mathcal{H}$ (lines 15-17). Here, the Jaccard constraint is fully satisfied and thus, we return the item $\{i\}$ with the witness flag set to *true*. This information will be supplied to the search engine (line 12) to accelerate solutions certification. We will denote by FIRSTWITCOV, our variable ordering heuristic that branches on the first free item satisfying the witness property.

Exploring the witness subtree. Let N be the node associated to the current itemset x^+ extended to a free item $\{i\}$. When the node N is detected as a positive witness during the branching, all supersets derived from N will also satisfy the Jaccard constraint. As these patterns are more likely to have similar covers, so a rather high Jaccard between them, we propose a simple strategy which avoids a complete exploration of the witness sub-tree rooted at N. Thus, we generate the first closed diverse itemset from N, add it to the current history and continue the exploration of the remaining search space using the new history. With a such strategy we have no guarantee that the closed itemset added to the history have the best Jaccard. But this strategy is fast.

5 Related work

The question of mining sets of diverse patterns has been addressed in the recent literature, both to offer more interesting results and to speed up the mining process. Van Leeuwen *et al.* propose populating the beam for subgroup discovery not purely with the *best* partial patterns to be extended but to take coverage overlap into account [20]. Beam search is heuristic, as opposed to our exhaustive approach and since they mine all patterns at the same time, diverse partial patterns can still lead to a less diverse final result. Dzyuba *et al.* propose using *XOR* constraints to partition the transaction set into disjoint subsets that are small enough to be efficiently mined using either a CP approach or a dedicated itemset miner [8]. Their focus is on efficiency, which they demonstrate by approximating the result set of an exhaustive operation. While they discuss pattern sets, they limit themselves to a strict non-overlap constraint on coverages. In [4], the authors propose using Monte Carlo Tree Search and upper confidence bounds to direct the search towards interesting regions in the lattice given the already explored space. While MCTS is necessarily randomized, it allows for anytime mining. The authors of [3] consider sets of subgroup descriptions as *disjunctions* of such patterns. Using a greedy algorithm exploiting upper bounds, the authors propose to iteratively extract up to k subgroup descriptions (similarly to our work). Notably, this approach requires a target attribute *and* a target value to focus on while our approach allows for unsupervised mining.

Earlier work has treated reducing redundancy as a post-processing step, e.g. [12] where a number of redundancy measures such as entropy are exploited in exhaustive search and the number of patterns in the set limited, [7] where the constraint-based itemset mining constraint is adapted to the pattern set settings, [5], which exploit bounds on predicting the presence of patterns from the patterns already included in $\mathcal{H}$ in a heuristic algorithm, or [21], which exploits the MDL principle to minimize redundancy among itemsets (and, in later work, sequential patterns). All of those methods require a potentially rather costly first mining step, and none exploits the Jaccard measure. As discussed in Section 2.1, there exist a number of constraint properties that allow for pruning, and Kifer *et al.*'s witness concept unifies them and discusses how to deal with constraints that do not have monotonicity properties [11]. The way to proceed in such a case is establishing *positive* and *negative* witnesses for the constraint, something we have done for the maximum pairwise Jaccard constraint. A rarely discussed aspect is that witnesses are closely related to CP since every witness enforces/forbids the inclusion of certain domain values.

6 Experiments and Results

The experimental evaluation is designed to address the following questions: (1) How (in terms of CPU-times and # of patterns) does our global constraint (denoted ClosedDiv) compare to the ClosedPatterns global constraint (denoted ClosedP) and the approach of Dzyuba et al. [8] (denoted Flexics)? (2) How do the resulting diverse FCIs compare qualitatively with those resulting from ClosedP and Flexics? (3) How far is the distance between the Jaccard index and the upper/lower bounds.

Dataset $\lvert\mathcal{I}\rvert \times \lvert\mathcal{T}\rvert$ $\rho(\%)$	$\theta(\%)$	#Patterns (1)	#Patterns (2)	Time (s) (1)	Time (s) (2)	#Nodes (2)	#Nodes (2)
CHESS	20	22,808,625	**96**	2838.30	**5.87**	45,617,249	**436**
75 × 3196	15	50,723,131	**393**	5666.03	**75.40**	101,446,261	**1,855**
49.33%	10	OOM	**4,204**	OOM	**3825.29**	OOM	**18,270**
HEPATITIS	30	83,048	**12**	9.64	**0.09**	166,095	**29**
68 × 137	20	410,318	**57**	42.00	**0.57**	820,635	**162**
50.00%	10	1,827,264	**2,270**	**169.59**	**76.91**	3,654,527	**5,256**
KR-VS-KP	30	5,219,727	**17**	682.94	**0.74**	10,439,453	**82**
73 × 3196	20	21,676,719	**96**	2100.79	**5.64**	43,353,437	**448**
49.32%	10	OOM	**4,120**	OOM	**3035.49**	OOM	**17,861**
CONNECT	30	460,357	**18**	1666.14	**14.81**	920,713	**77**
129 × 67557	18	2,005,476	**197**	5975.44	**573.66**	4,010,951	**900**
33.33%	15	3,254,780	**509**	9534.07	**1989.35**	6,509,559	**2,188**
HEART-CLEVELAND	10	12,774,456	**3,496**	1308.63	**257.39**	25,548,911	**7,977**
95 × 296	8	23,278,687	**12,842**	**2278.97**	2527.38	46,557,373	**28,221**
47.37%	6	43,588,346	58,240	**4126.84**	46163.06	87,176,691	124,705
SPLICE1	10	1,606	**422**	**6.55**	25.25	3,211	**843**
287 × 3190	5	31,441	**8,781**	**117.15**	5616.47	62,881	**17,594**
20.91%	2	589,588	-	**1179.55**	-	1,179,175	-

Dataset $\lvert\mathcal{I}\rvert \times \lvert\mathcal{T}\rvert$ $\rho(\%)$	$\theta(\%)$	#Patterns (1)	#Patterns (2)	Time (s) (1)	Time (s) (2)	#Nodes (2)	#Nodes (2)
MUSHROOM	5	8,977	**727**	**10.02**	60.70	17,953	**1,704**
112 × 8124	1	40,368	**12,139**	**34.76**	12532.95	80,735	**25,154**
18.75%	0.5	62,334	**27,768**	**50.05**	64829.06	124,667	**56,873**
T40I10D100K	8	138	127	**75.91**	447.20	275	253
942 × 100000	5	317	288	**331.47**	1561.34	633	575
4.20%	1	65,237	7,402	**5574.31**	58613.88	130,473	14,887
PUMSB	40	-	**4**	-	**57.33**	-	**16**
2113 × 49046	30	-	**15**	-	**267.72**	-	**64**
3.50%	20	-	**52**	-	**852.39**	-	**250**
T10I4D100K	5	11	11	**1.73**	6.31	21	21
870 × 100000	1	386	**361**	**434.25**	3125.06	771	722
1.16%	0.5	1,074	**617**	**881.31**	7078.90	2,147	**1,257**
BMS1	0.15	1,426	**609**	**11362.71**	68312.38	2,851	**1,220**
497 × 59602	0.14	1,683	**668**	**11464.93**	68049.00	3,365	**1,339**
0.51%	0.12	2,374	**823**	**13255.79**	79704.88	4,747	**1,651**
RETAIL	5	17	**13**	**10.74**	33.44	33	25
16470 × 88162	1	160	**111**	**297.21**	1625.73	319	227
0.06%	0.4	832	**528**	**6073.53**	31353.23	1,663	**1,093**

Table 1: CLOSEDDIV ($J_{max} = 0.05$) vs CLOSEDP. For columns #Patterns and #Nodes, the values in bold indicate a reduction more than 20% of the total number of patterns and nodes.“ – ” is shown when time limit is exceeded. OOM : Out Of Memory. (1): CLOSEDP (2): CLOSEDDIV

Experimental protocol. Experiments were carried out on classic UCI data sets, available at the FIMI repository (fimi.ua.ac.be/data). We selected several real-world data sets, their characteristics (name, number of items $|\mathcal{I}|$, number of transactions $|\mathcal{T}|$, density ρ) are shown in the first column of Table 1. We selected data sets of various size and density. Some data sets, such as Hepatitis and Chess, are very dense (resp. 50% and 49%). Others, such as T10 and Retail, are very sparse (resp. 1% and 0.06%). The implementation of the different global constraints and their constraint propagators were carried out in the `Choco` solver [17] version $4.0.3$, a Java library for constraint programming. The source code is publicly available.[5] Experiments were conducted on AMD Opteron 6174, 2.2 GHz with a RAM of 256 GB and a time limit of 24 hours. The default maximum heap size allowed by the JVM is 30 GB. We have selected for every data set frequency thresholds to have different numbers of frequent closed itemsets ($|Th(c)| \leq 15000$, $30000 \leq |Th(c)| \leq 10^6$, and $|Th(c)| > 10^6$). The only exception are the very large and sparse data sets Retail and Pumsb, where we do not find a large number of solutions. We used the CLOSEDP CP model as a baseline to determine suitable thresholds used with the CLOSEDDIV CP model. To evaluate the quality of a set of patterns in terms of diversity, we measured the average ratio of exclusive pattern coverages: $ECR(P_1, ..., P_k) = avg_{1 \leq i \leq k}(\frac{sup_{\mathcal{D}}(P_i) - |\mathcal{V}_{\mathcal{D}}(P_i) \cap \bigcup_{j \neq i} \mathcal{V}_{\mathcal{D}}(P_j)|}{sup_{\mathcal{D}}(P_i)})$.

(a) Comparing CLOSEDDIV with CLOSEDP and FLEXICS. Table 1 compares the performance of the two CP models for various values of θ on different data sets. Here, we report the CPU time (in seconds), the number of extracted patterns, and the number of nodes explored during search. This enables to evaluate the amount of inconsistent values pruned by each approach (filtering algorithm). We use MINCOV as variable ordering heuristic. The maximum diversity threshold J_{max} is set to 0.05. First, the results highlight the great discrepancy between the two models with a distinctly lower number of patterns generated by CLOSEDDIV (in the thousands) in comparison to CLOSEDP (in the millions). On dense and moderately dense data sets (from CHESS to MUSHROOM),

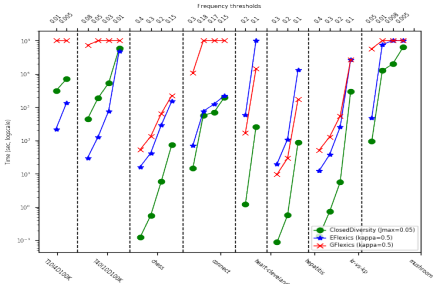
[5] https://github.com/lobnury/ClosedDiversity

the discrepancy is greatly amplified, especially for small values of θ. For instance, on CHESS, the number of patterns for CLOSEDDIV is reduced by 99% (from $\sim 50 \cdot 10^6$ solutions to 393) for θ equal to 15%. The density of the data sets provides an appropriate explanation for the good performance of CLOSEDDIV. As the number of closed patterns increases with the density, redundancy among these patterns increases as well. On very sparse data sets, CLOSEDDIV still outputs fewer solutions than CLOSEDP but the difference is less pronounced. This is explainable by the fact that on these data sets, where we have few solutions, almost all patterns are diverse.

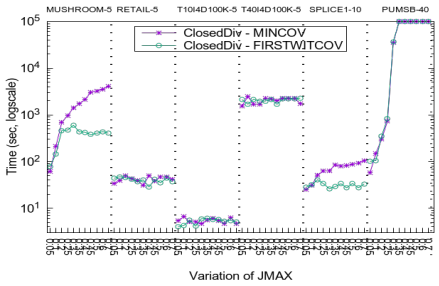
Second, regarding runtime, CLOSEDDIV exhibits different behaviours. On dense data sets ($\rho \geq 30\%$), CLOSEDDIV is more efficient than CLOSEDP and up to an order of magnitude faster. On CHESS (resp. CONNECT), the speed-up is 1455 (resp. 112) for $\theta = 30\%$. For instances resulting in between 500 and 5000 diverse FCIs, the speed-up is up to 5. This good performance of CLOSEDDIV is mainly due to the strength of the LB filtering rule that provides the CP solving process with more propagation to remove more inconsistent values in the search space. In addition, the number of nodes explored by CLOSEDDIV is always small comparing to CLOSEDP. These results support our previous observations. The only exception is HEART-CLEVELAND for which CLOSEDDIV is slower (especially for values of $\theta \leq 8\%$). This is mainly due to the relative large number of diverse patterns (≥ 12000), which induces higher lower bound computational overhead. We observe the same behaviour on the two moderately dense data sets SPLICE1 and MUSHROOM. On sparse data sets, CLOSEDDIV can take significantly more time to extract all diverse FCIs. This can be explained by the fact that on these instances almost all FCIs are diverse w.r.t. lower bound (on average about 70% for RETAIL and 39% for BMS1, see Table 1). Thus, non-solutions are rarely filtered, while the lower bound overhead greatly penalizes the CP solving process. On the very large PUMSB data set, finally, our approach is very efficient while CLOSEDP fails to complete the extraction.

Finally, Figure 2a compares CLOSEDDIV with FLEXICS (two variants) for various values of θ on different data sets: GFLEXICS, which uses CP4IM [6] as an oracle to enumerate the solutions, and EFLEXICS, a specialized variant, based on ECLAT [23]. We run WEIGHTGEN with values of $\kappa \in \{0.1, 0.5, 0.9\}$ [9]. For each instance, we fixed the number of samples to the number of solutions returned by CLOSEDDIV. We report results corresponding to the best setting of parameter κ. First, CLOSEDDIV largely dominates GFLEXICS, being more than an order of magnitude faster. Second, while EFLEXICS is faster than GFLEXICS, our approach is almost always ranked first, illustrating its usefulness for mining diverse patterns in an anytime manner.

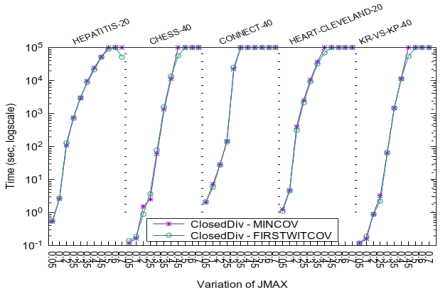
(b) Impact of varying J_{max}. We varied J_{max} from 0.1 to 0.7. The minimum support θ is fixed for each data set (indicated after '-'). Figure 2 shows detailed results. As expected, the greater J_{max}, the longer the CPU time. In fact, the size of the history $\mathcal{H}$ grows rapidly with the increase of J_{max}. This induces significant additional costs in the lower and upper bound computations. Moreover, when J_{max} becomes sufficiently large, the LB filtering of CLOSEDDIV occurs rarely since the lower bound is almost always below the J_{max} value (see Figure 3). Despite the hardness of some instances (i.e., $J_{max} \geq 0.35$), our CP approach is able to complete the extraction for almost all values of J_{max}. The only exception are the large and dense data sets, where CLOSEDDIV fails



(a) CPU-time MINCOV and FLEXICS



(b) Moderately dense and sparse dat sets.



(c) Dense data sets.

Dataset	θ(%)	Exclusive Coverage Ratio (ECR)		
		CLOSEDP	CLOSEDDIV–MINCOV	EFLEXICS
T40I10D100K	8	1.6E-01	**4.4E-01**	1.29E-01
	1	1.9E-01	**9.2E-01**	2.90E-01
SPLICE1	10	4.0E-02	**1.6E-01**	2.98E-02
	5	4.0E-02	**2.5E-01**	1.79E-01
CONNECT SPLICE1	18	0.00	**1.0E-01**	4.31E-02
	10	4E-02	**1.6E-01**	2.98E-02
	5	4E-2	**2.5E-01**	1.79E-01
CONNECT	18	1.25E-6	**1.0E-01**	4.31E-02
	15	0.00	**1.7E-01**	4.16E-02
MUSHROOM	5	3.3E-4	**5.6E-01**	2.63E-01
	1	3.73E-4	**4.3E-01**	4.2E-01
HEPATITIS	10	0.00	**2.5E-01**	1.0E-01
HEPATITIS	10	1.50E-3	**2.5E-01**	1.0E-01
T10I4D100K	1	5.9E-01	**9.3E-01**	5.72E-01
	0.5	4.8E-01	**9.6E-01**	4.63E-01

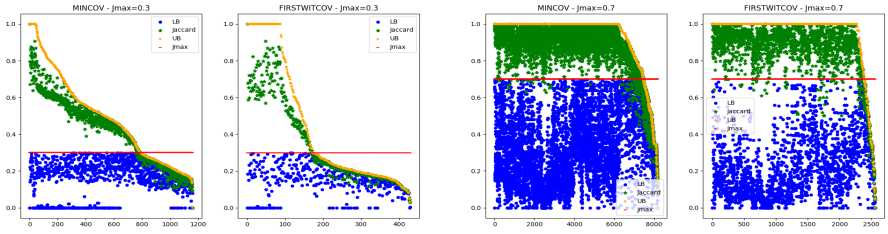
(d) Patterns ECR.

Fig. 2: CPU-time analysis (MINCOV vs FIRSTWITCOV and CLOSEDDIV vs FLEXICS) and patterns discrepancy analysis.

to complete the extraction within the time limit for $J_{max} \geq 0.45$. However, in practice, the user will only be interested in small values of J_{max} because the diversity of patterns is maximal and the number of patterns returned becomes manageable.

Figures 2b and 2c also compare the resolution time of our CP model using the two variable ordering heuristics MINCOV and FIRSTWITCOV. First, on dense data sets, both heuristics perform similarly, with a slight advantage for FIRSTWITCOV. On these data sets, the number of witness patterns mined remains very low ($\leq$ 100), thus the benefits of FIRSTWITCOV is limited (see Supp. material). On moderately dense data sets (MUSHROOM and SPLICE1), FIRSTWITCOV is very effective; on MUSHROOM it is up 10 times faster than MINCOV for J_{max} equal to 0.7. On these data sets, the number of witness patterns extracted is relatively high compared to dense ones. In this case, FIRSTWITCOV enables to guide the search to find diverse patterns more quickly. On sparse data sets, no heuristic clearly dominates the other. When regarding the number of diverse patterns generated (see Supp. material), we observe that FIRSTWITCOV returns less patterns on moderately dense and sparse data sets, while on dense data sets the number of diverse patterns extracted remains comparable.

(c) Qualitative analysis of the proposed relaxation. In this section, we shed light on the quality of the relaxation of the Jaccard constraint. Figure 3a shows, for a particular instance SPLICE1 with $J_{max} = 0.3$, the evolution of the LB_J and UB_J of the solutions



(a) SPLICE1 ($\theta = 10\%, J_{max} = 0.3$) (b) MUSHROOM ($\theta = 5\%, J_{max} = 0.7$)

Fig. 3: Qualitative analysis of the LB and UB relaxations.

found during search. Here, the solutions are sorted according to their UB_J. Concerning the lower bound, one can observe that the LB_J values are always below the J_{max} value. This shows how frequently the LB filtering rule of CLOSEDDIV occurs. This also supports the suitability of the LB filtering rule for pruning non-diverse FCIs. With regard to the upper bound, it is interesting to see that it gets very close to the Jaccard value, meaning that our Jaccard upper bounding provides a tight relaxation. Moreover, a large number of solutions have UB_J values either below or very close to J_{max}. This is indicative of the quality of the patterns found in terms of diversity. We recall that when $UB_J < J_{max}$, all partial assignments can immediately be extended to diverse itemsets, thanks to the anti-monotonicity property of our UB (see Proposition 5). We observe the same behaviour on MUSHROOM with $J_{max} = 0.7$ (see Figure. 3b). Finally, we can see that FIRSTWITCOV allows to quickly discover solutions of better quality in terms of UB_J and Jaccard values compared to MINCOV. This demonstrates the interest and the strength of our UB_J branching rule to get diverse patterns.

(d) Qualitative analysis of patterns. Figure 2d compares CLOSEDDIV with CLOSEDP and EFLEXICS in terms of the ECR measure, which should be as high as possible. Due to the huge number of patterns generated by CLOSEDP, a random sample of $k = 10$ solutions of all patterns is considered. Reported values are the average over 100 trials. ECR penalises overlap, and thus having two similar patterns is undesirable. According to ECR, leveraging Jaccard in CLOSEDDIV clearly leads to pattern sets with more diversity among the patterns. This is indicative of patterns whose coverage are (approximately) mutually exclusive. This should be desirable for an end-user tasked with exploring and interpreting the set of returned patterns.

7 Conclusions

In this paper, we showed that mining diverse patterns using a maximum Jaccard constraint cannot be modeled using an anti-monotonic constraint. Thus, we have proposed (anti-)monotonic lower and upper bound relaxations, which allow to make pruning effective, with an efficient branching rule, boosting the whole search process. The proposed approach is introduced as a global constraint called CLOSEDDIV where diversity is controlled through a threshold on the Jaccard similarity of pattern occurrences. Experimental results on UCI datasets demonstrate that our approach significantly reduces

the number of patterns, the set of patterns is diverse and the computation time is lower compared to CLOSEDP global constraint, particularly on dense data sets.

References

1. Supplementary Material (June 2020), https://github.com/lobnury/ClosedDiversity.
2. Belaid, M., Bessiere, C., Lazaar, N.: Constraint programming for mining borders of frequent itemsets. In: Proceedings of IJCAI 2019, Macao, China. pp. 1064–1070 (2019)
3. Belfodil, A., Belfodil, A., Bendimerad, A., Lamarre, P., Robardet, C., Kaytoue, M., Plantevit, M.: Fssd-a fast and efficient algorithm for subgroup set discovery. In: Proceedings of DSAA. pp. 91–99 (2019)
4. Bosc, G., Boulicaut, J.F., Raïssi, C., Kaytoue, M.: Anytime discovery of a diverse set of patterns with monte carlo tree search. Data mining and knowledge discovery **32**(3), 604–650 (2018)
5. Bringmann, B., Zimmermann, A.: The chosen few: On identifying valuable patterns. In: Proceedings of ICDM 2007. pp. 63–72
6. De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: 14th ACM SIGKDD. pp. 204–212 (2008)
7. De Raedt, L., Zimmermann, A.: Constraint-based pattern set mining. In: 7th SIAM SDM. pp. 237–248. SIAM (2007)
8. Dzyuba, V., van Leeuwen, M., De Raedt, L.: Flexible constrained sampling with guarantees for pattern mining. Data Mining and Knowledge Discovery **31**(5), 1266–1293 (2017)
9. Dzyuba, V., van Leeuwen, M.: Interactive discovery of interesting subgroup sets. In: International Symposium on Intelligent Data Analysis. pp. 150–161. Springer (2013)
10. Hoeve, W., Katriel, I.: Global constraints. In: Handbook of Constraint Programming, pp. 169–208. Elsevier Science Inc. (2006)
11. Kifer, D., Gehrke, J., Bucila, C., White, W.: How to quickly find a witness. In: Constraint-Based Mining and Inductive Databases. pp. 216–242. Springer Berlin Heidelberg (2006)
12. Knobbe, A.J., Ho, E.K.: Pattern teams. In: Proceedings of ECML-PKDD. pp. 577–584. Springer (2006)
13. Lazaar, N., Lebbah, Y., Loudni, S., Maamar, M., Lemière, V., Bessiere, C., Boizumault, P.: A global constraint for closed frequent pattern mining. In: Proceedings of the 22nd CP. pp. 333–349 (2016)
14. van Leeuwen, M.: Interactive Data Exploration Using Pattern Mining, pp. 169–182. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
15. Ng, R.T., Lakshmanan, L.V.S., Han, J., Pang, A.: Exploratory mining and pruning optimizations of constrained association rules. In: Proceedings of ACM SIGMOD. pp. 13–24 (1998)
16. Pei, J., Han, J., Lakshmanan, L.V.S.: Mining frequent item sets with convertible constraints. In: Proceedings of ICDE. pp. 433–442 (2001)
17. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Solver Documentation (2016)
18. Puolamäki, K., Kang, B., Lijffijt, J., De Bie, T.: Interactive visual data exploration with subjective feedback. In: Proceedings of ECML PKDD. pp. 214–229. Springer (2016)
19. Schaus, P., Aoga, J.O.R., Guns, T.: Coversize: A global constraint for frequency-based itemset mining. In: Proceedings of the 23rd CP 2017. pp. 529–546 (2017)
20. Van Leeuwen, M., Knobbe, A.: Diverse subgroup set discovery. Data Mining and Knowledge Discovery **25**(2), 208–242 (2012)
21. Vreeken, J., Van Leeuwen, M., Siebes, A.: Krimp: mining itemsets that compress. Data Mining and Knowledge Discovery **23**(1), 169–214 (2011)

22. Wang, J., Han, J., Pei, J.: CLOSET+: searching for the best strategies for mining frequent closed itemsets. In: Proceedings of the Ninth KDD. pp. 236–245. ACM (2003)
23. Zaki, M., Parthasarathy, S., Ogihara, M., Li, W.: New algorithms for fast discovery of association rules. In: Proceedings of KDD 1997, Newport Beach, California, USA, August 14-17, 1997. pp. 283–286. AAAI Press (1997)