Doctoral Program Proceedings

The 24th International Conference on Principles and Practice of Constraint Programming

August 27, 2018

Lille, France

Nadjib Lazaar and Anastasia Paparrizou

Doctoral Program Chairs of CP 2018

Program Committee

Andre Augusto Cire	University of Toronto
Catherine Dubois	ENSIIE-Samovar
Arnaud Gotlieb	SIMULA Research Laboratory
Serdar Kadioglu	Oracle Corporation
Zeynep Kiziltan	University of Bologna
Lars Kotthoff	University of Wyoming
Arnaud Lallouet	Huawei Technologies Ltd
Xavier Lorca	IMT Mines Albi
Ciaran McCreesh	University of Glasgow
Peter Nightingale	University of St Andrews
Marie Pelleau	University of Côte d'Azur
Justyna Petke	University College London
Andrea Rendl	University of Klagenfurt
Pierre Schaus	UCLouvain
Christian Schulte	KTH Royal Institute of Technology
Mohamed Siala	Insight, University College Cork
Laurent Simon	Labri, Bordeaux Institute of Technology
Guido Tack	Monash University
Mohamed Wahbi	Insight, University College Cork

DP Participants

Saad Attieh	University of St Andrews, UK
Mohamed-Bachir Belaid	University of Montpellier, France
Massimo Bono	University of Brescia, Italy
Maxime Chabert	Université de Lyon, Infologic, France
Mathieu Collet	University of Stavanger, Simula Research Laboratory, ABB Robotics, Norway
Jip Dekker	Monash University, Australia
Rohan Fossé	LABRI, France
Rémy Garcia	University of Côte d'Azur, France
Arthur Godet	TASC, IMT Atlantique, France
Shan He	Monash University, Australia
Khoi Hoang	Washington University in St. Louis, USA
Linnea Ingmar	Uppsala University, Sweden
Dimitri Justeau-Allaire	CIRAD / IAC, France
Michał Karpiński	Institute of Computer Science, University of Wroclaw, Poland
Gökberk Koçak	University of St Andrews, UK
Fanghui Liu	University of Connecticut, USA
Giovanni Lo Bianco	IMT Atlantique, France
Aurèlie Massart	UCLouvain/ICTEAM, Belgium
Anthony Palmieri	Huawei Technologies Ltd, France
Tomáš Peitl	Vienna University of Technology, Austria
Aditya Shrotri	Rice University, USA
Patrick Spracklen	University of St Andrews, UK
James Trimble	University of Glasgow, Scotland
Dimosthenis Tsouros	University of Western Macedonia, Greece
Kiana Zeighami	Monash University, Australia
Ghiles Ziat	LIP6, France
Heytem Zitoun	University of Côte d'Azur, France

DP Mentors

Christian Bessiere	CNRS, University of Montpellier
Michael Codish	Ben-Gurion University of the Negev
Maria Garcia de la Banda	Monash University
Tias Guns	Vrije Universiteit Brussel
Emmanuel Hebrard	LAAS-CNRS
George Katsirelos	INRA
Arnaud Lallouet	Huawei Technologies Ltd
Jimmy Lee	Chinese University of Hong Kong
Arnaud Malapert	University Nice Sophia Antipolis
Laurent Michel	University of Connecticut
Ian Miguel	University of St Andrews
Peter Nightingale	University of St Andrews
Marie Pelleau	University of Côte d'Azur
Gilles Pesant	Ecole Polytechnique Montreal
Claude-Guy Quimper	University of Laval
Andrea Rendl	University of Klagenfurt
Michel Rueher	University of Côte d'Azur
Lakhdar Sais	University of Artois
Pierre Schaus	UCLouvain
Christian Schulte	KTH Royal Institute of Technology
Meinolf Sellmann	GE Global Research
Helmut Simonis	Insight, University College Cork
Peter Stuckey	University of Melbourne
Guido Tack	Monash University
Charlotte Truchet	University of Nantes
Phillippe Vismara	Montpellier SupAgro
William Yeoh	Washington University in St. Louis

Contents

Deriving filtering algorithms from dedicated algorithms: zoom on the Bin Packing problem

Arthur Godet[(1)], Xavier Lorca[(2)], and Gilles Simonin[(1)]

(1) TASC, IMT Atlantique, LS2N UMR CNRS 6004
FR-44307 Nantes Cedex 3, France
arthur.godet@imt-atlantique.net & gilles.simonin@imt-atlantique.fr
(2) ORKID, Centre de Génie Industriel, IMT Mines Albi-Carmaux
Campus Jarlard, 81013 Albi cedex 09, France
xavier.lorca@mines-albi.fr

Abstract. Solving $\mathcal{NP}$-complete problems can be tough because of the combinatorics. Constraint Programming and Approximation algorithms can be used to solve these problems. In this paper, we explore how to automatically derive filtering algorithms from a dedicated algorithm solving the Bin Packing problem. To this end, we automatically derive a filtering algorithm from the Best-Fit algorithm. We empirically show that our filtering algorithm BF-Prop is experimentally strictly more efficient in terms of filtering than Shaw's state-of-the-art global constraint.

Keywords: constraint programming, approximation algorithms, filtering algorithms, bin packing problem

1 Introduction

Industrial systems have become more and more complex, making it harder for humans to take informed decisions. The need to mathematically model these systems, and more particularly the problems they include, rapidly arose. Artificial Intelligence fields arose as well in order to build backgrounds and theories for scientifically solving these problems.

Constraint Programming suffers from a lack of precision to define consistencies, i.e. the efficiency with which filtering algorithms remove values from variables' domains. Indeed, there exist only a few consistencies, which form a discontinuous hierarchy. On the other hand, Approximation algorithms are also used to solve Optimisation problems, but benefit from a very precise measure of their efficiency. More accurately, the different classes of Approximation algorithms form a continuum.

In this paper, we explore the possibility to automatically build a filtering algorithm from a dedicated algorithm for a well-known problem which has many applications, either pure or as a constraint in a more complex problem: the Bin Packing problem. The Bin-Packing satisfaction problem is known to be $\mathcal{NP}$-complete, while the optimisation one is $\mathcal{NP}$-hard and benefits from strong results in Approximation [6] as well as the existence of a global constraint [3].

In section 2, we define the notions and mechanisms of Constraint Programming and Approximation that we use in the following sections. Third section presents the Bin Packing problem as well as state of the art results in both fields for this particular problem. In section 4, we explain how we automatically derive a filtering algorithm from an approximation one. Section 5 describes the experimentations we ran as well as the results we obtained. Finally, section 6 concludes by presenting further research in this area.

2 Constraint Programming and Approximation Definitions

Constraint programming material. Resolving *Constraint Optimisation Problems* (COP) is challenging for the Constraint Programming community. This is true particularly when considering an industrial point of view for which finding a good trade-off between time complexity and the solution quality is essential. As described by Christian Bessière in [2], a *Constraint Satisfaction Problem* (CSP) is composed of a finite sequence of integer *variables*, denoted V, each one having a *domain* in $\mathbb{Z}$, denoted D, and of a set of *constraints*, denoted C. A constraint is a relation defined on a set of variables that is satisfied by a set of combinations, called *tuples*. That is to say a vector of $\mathbb{Z}^n$ (n being the number of variables in the scope of the constraint), each variable assigned to a value of its domain. For instance, the constraint $2 \leq x + y + z$, with x, y and z taking their value in $\{0, 1\}$, is satisfied by the tuples $(1, 1, 0), (1, 0, 1), (0, 1, 1)$ and $(1, 1, 1)$. A *solution* for a given CSP is an instantiation of all the variables to the values such that all the constraints are simultaneously satisfied. In this context, a COP is a natural extension of a CSP by adding an objective function which is a specific constraint. This constraint has a form $z = f(A)$ such that $A \subseteq V$ and z denotes the objective variable. The aim of the COP is to find a solution optimising (*e.g.*, minimising or maximising) the value affected to the variable z.

Once a CSP (resp. COP) is expressed, a specific algorithm, called a *solver*, will try to solve it by interlacing *constraint propagation* with a *backtracking algorithm*. In the following, *current state* of the problem refers to the domains of the variables after the constraints have removed some values due to former decisions. If the domain of a variable is reduced to one value, this variable is said to be *instantiated*. The set of all instantiated variables is called a *partial instantiation* if not all the variables of the problem are instantiated, and a *solution* if no constraint is violated. During the solving phase, the solver will call *propagators*, also called *filtering algorithms*, in order to remove inconsistent values from variables' domain, i.e. values that can not lead to a solution from the current state. Most of the time there is a one-to-one correspondence between constraints and propagators. However, in some cases, several propagators can be associated with one constraint.

Each time a value is removed from the domain of a variable, all the constraints that have this variable in their scope are called by the solver, i.e. all the corresponding filtering algorithms will run in order to remove values that can

not lead to a solution from the new current state according to these filtering algorithms. This process will be applied until no more deductions can be made by the filtering algorithms, i.e. no more values can be logically removed from the domains of the variables with respect to the efficiency of the filtering algorithms. This particular situation is called *fixpoint*.

There exist several *consistencies* in the literature that represent the efficiency of the propagators. In [2], Christian Bessiére defines and classes more than a dozen consistencies, from the *Path Consistency* to the *Forward Checking* (respectively the strongest and the weakest of the consistencies), passing by the *Generalised Arc Consistency* (GAC), which is considered as a reference when comparing different consistencies as it is a well-known and well-studied consistency. A consistency is stronger than another one if it filters at least all the values filtered by the second one.

The *Generalised Arc Consistency* (GAC) guarantees that every value still in the variables' domain is present in one tuple, this for every constraint. Nevertheless, the GAC does not guarantee that the tuples are globally possible from the current state. The *Generalised Arc Consistency* can be time-costly to guarantee, so one can prefer to apply the *Bounds Consistency* (BC) which consists in guaranteeing the GAC only on the bounds of the domain of the variables instead of the entire domain. Thus, the BC is weaker than the GAC.
For a given constraint, there can exist several filtering algorithms, each one having its own consistency. Generally, there does not exist a filtering algorithm for every consistency. But when this is the case, it offers the user a trade-off between time-complexity and efficiency of filtering, as stronger consistencies generally have higher time-complexities.

Approximation material. On the other hand, Approximation algorithms are polynomial-time algorithms designed to solve specific problems. Thus, for a given $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problem, an approximation algorithm will return a solution whose distance to the optimal can be mathematically measured. If the solutions returned by the algorithm can be bounded by a ratio ρ for a given measure, then the algorithm is said to be a *ρ-approximation algorithm*. There exist several classification of approximation algorithms depending on the used measure and the efficiency of the algorithms. For instance, an approximation algorithm is a *polynomial-time approximation scheme* (PTAS) if, for a parameter $\epsilon > 0$, the approximation algorithm's ratio is $1 + \epsilon$.

The low time complexity of these algorithms can be very useful. It is interesting to explore the potential learnings one can make from approximation theory to generate new efficient filtering rules for Constraint Programming. In the new section, we will present a classic problem from literature on which we can develop this approach.

3 A practical case: the Bin Packing Problem

The Bin Packing (BP) problem is composed of a fixed number of objects with different sizes and a set of bins. The satisfaction Bin Packing problem consists

in checking if it is possible to store all the objects in k bins. The optimisation Bin Packing problem searches the smallest number of bins that contain all the objects. This problem has been highly studied in the literature, and is a specification of the Generalised Assignment Problem (GAP). In [7] it is shown that the satisfaction BP problem is $\mathcal{NP}$-complete . The optimisation version of the problem is known to be $\mathcal{NP}$-hard . Using binary variables, a first mathematical model can be given:

$$opt = min \sum_{i=1}^{n} y_i \tag{1}$$

$$\sum_{j=1}^{n} a_j x_{ij} \leq B y_i, \ i \in [1, n] \tag{2}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \ j \in [1, n] \tag{3}$$

$$y_i \in 0, 1, \ i \in [1, n] \tag{4}$$

$$x_{ij} \in 0, 1, \ i \in [1, n], \ j \in [1, n] \tag{5}$$

with $1 \leq B$ being the capacity of the bins, $y_i = 1$ if the bin i is used (0 otherwise) and $x_{ij} = 1$ if the object j is in the bin i (0 otherwise), and a_j is the weight of the object j.

3.1 Results from Constraints Programming

The high difficulty of the BP problem leads to few approaches in Constraints. However, in [3] Paul Shaw presents a filtering algorithm $\mathcal{A}$ for the Bin Packing problem.

Let $\mathcal{O}$ (resp. $\mathcal{B}$) be the set of objects (resp. bins) in BP. We use one integer variable x_i per object o_i from $\mathcal{O}$. The domain of each x_i represents all the potential bins to store o_i. We also use one integer variable l_j per bin b_j from $\mathcal{B}$ representing the load of the bin b_j. The filtering algorithm $\mathcal{A}$ filters values in the domains of the variables x_i and l_j that lead to a certain fail. Indeed, as Shaw explained in [3], guaranteeing that the filtering algorithm is GAC is already a $\mathcal{NP}$-hard problem in the case of the Bin Packing problem. Thus Shaw's filtering algorithm can not guarantee the GAC. Nevertheless, Shaw's filtering algorithm does better than the Bounds Consistency.

3.2 Results from Approximation

Although the optimisation version of the Bin Packing problem is $\mathcal{NP}$-hard , this problem was well studied in the last decades in approximation theory. For instance there exist algorithms in PTAS/DPTAS approximation classes solving the Bin Packing optimisation problem [4]. We decided to focus our study on two well-know algorithms giving a good approximation ratio for the BP problem: the **First Fit** algorithm and the **Best Fit** algorithm.

The **First Fit** algorithm consists in assigning the first object of the list not already placed into the first bin that can contain it.

The **Best Fit** algorithm consists in assigning the first object of the list not already placed into the less empty bin that can still contain it.

4 A new propagator: BF-Prop

4.1 Presentation of the propagator

Our general methodology consists in selecting a dedicated algorithm solving the problem. Then we create a propagator whose filtering algorithm filters impossible values for the partial instantiation corresponding to the current node in the search tree, which is the tree of decisions took by the solver during the resolution. This filtering algorithm will generate **nogood**, which is a set of assignments that is not consistent with any solution [2], for $x_i = j$ choices leading to a worse solution than the one got only from the partial instantiation.

Example 1. We consider objects of size $1000, 500, 350, 200$ and 100 and that the bins are of size 1100. We also consider the Best Fit algorithm. For the partial instantiation $\{ x_1 = 1, x_2 = 2 \}$, our propagator's filtering algorithm will remove 1 from the domains of x_3 and x_4, because the objects x_3 and x_4 are too heavy to be put with the first object, represented by x_1. Moreover, the choices $x_5 = i$ for $i \geq 2$ will each generate a *nogood* for the current partial instantiation because they will lead to solutions with 3 bins where the current partial instantiation leads to a solution with 2 bins when applying the Best Fit algorithm. This signifies that for each partial instantiation with $x_1 = 1$ and $x_2 = 2$, x_5 can not take values that are greater or equal than 2. Thus x_5 will be instantiated to the value 1.

In fact, our propagator generates new constraints that are based on *nogoods*, i.e. that cut down the exploration of the branch of the search tree if it corresponds to a nogood. This can have impacts at several moments in the search tree.

Our approach currently has an obvious high computing cost because it applies the polynomial approximation algorithm several time at each node of the tree.

Proposition 1. *We note n the number of integer variables x_i, d the size of the largest domain and $f(n)$ the time complexity of the approximation algorithm. Our propagator's filtering algorithm complexity is then in $O(f(n)nd)$.*

As the Best Fit Decreasing algorithm's time complexity is $O(nlog(n))$, with n the number of objects to pack, our derived propagator BF-Prop is in $O(n^2log(n)d)$.

In the rest of this paper, we call **BF-Prop** the propagator deduced from the Best-Fit algorithm.

4.2 Generating Nogood for the Bin Packing problem

We note s the value of the solution returned by the approximation algorithm considering only the partial instantiation.

Our intuition is that for each possible instantiation $x_i = j$ added to the partial instantiation, if the solution s' returned by the approximation algorithm is worse than s, then we know that the sequence of instantiations made by the approximation algorithm, that lead to a greater number of bins s' than s, cannot exist in any optimal solution:

Proposition 2. *We note PI the Partial Instantiation and $\mathcal{S}$ the set of variables that were in the sequence of instantiations made by the approximation algorithm that lead to a greater number of bins s' than s. The generated nogood will be the negation of all the instantiations of PI and of $\mathcal{S}$ as well as the tried choice:*

$$\neg[(\bigwedge_{w \in PI} w = val_{PI}(w)) \wedge x_i = j \wedge (\bigwedge_{v \in \mathcal{S}} v = val_{\mathcal{S}}(v))] \tag{6}$$

Proof. As the partial instantiation and the choice $x_i = j$ lead to use more bins than the current best known solution, we know that any optimal solution cannot be built with these exact instantiations (the partial instantiation, the choice and the algorithm assignment decisions). So a nogood being the negation of all these instantiations can be generated and added to the set of constraints. □

It is interesting to notice that the sequence $\mathcal{S}$ doesn't need to contain all the variables of the problem, but only the ones that were instantiated at the moment when the approximation algorithm needs to use more bins than the bound s. For example, we consider bins of size 1000 and objects of size 600, 600, 500, 500, 400 and 400. If the partial instantiation is $x_1 = 1$ and $x_2 = 2$, the Best-Fit algorithm will say that it needs 3 bins. So if we try the instantiation $x_5 = 3$, the algorithm will use a 4^{th} bin when trying to put the second object that weight 500. So we can stop the algorithm and generate a nogood without having to pack the last object because we already know that the solution will be worse than the one we had. Indeed, the shorter the nogood clause, the better the performance of the deduced constraint.

5 Experimentations and results

The instances we worked on where created following a Weibull distribution, as described in [1]. We generated 100 instances of Bin Packing with a scale parameter of 1000, of size 10, of shape parameter between $0,5$ et $18,0$ and of size of bin between $1,0$ and $2,0$ times the biggest generated object. While these instances are randomly generated, Castineiras et al. show that real-world instances can be generated thanks to a Weibull distribution [1].

5.1 Experimentations

Our experimentations were run on Choco solver 4.0 [8] on a 2.4 GHz Intel i5-4210U processor. We consider the solver to fail in finding and proving the optimal solution if it takes more than 10 minutes.

Our protocol consists in several experimentations aiming to compare the possible benefits of our new propagator, based on a dedicated algorithm and which generates NOGOOD, with the results got by the model based on Paul Shaw's constraint and by an Oracle. For that, we will use the set of instances that we generated using a Weibull distribution.

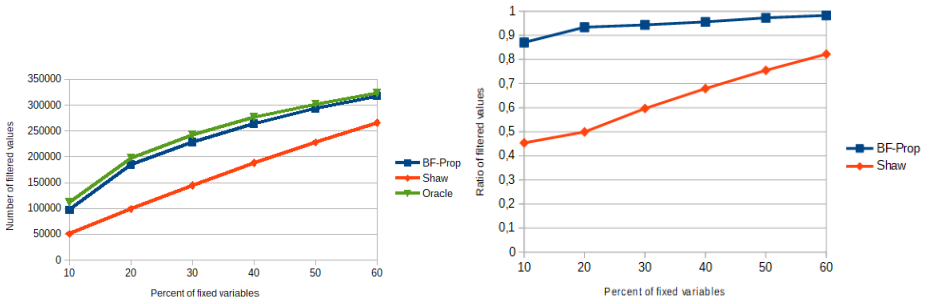
Every model will be compared on its solving capabilities when using the same variable/value selection heuristic. For our experimentations, we consider two heuristics: MINDOM-MINVALUE because it is very well-known, and DOM-WDEG [5] because it is the default heuristic used in Choco solver. For each instance, each model and each heuristic, we will compare the number of nodes needed and the time spent to find the first optimal solution (and to prove the optimality).

However what really interests us is the filtering capacity of our new propagator. So we will also do the following experimentation: for each instance and each model, we will record the number of filtered values in order to compare the filtering capacities of each model. We will also use an Oracle, representing the *Generalised-Arc-Consistency* for the optimisation Bin Packing problem, in order to have a GAC-base to compare all the models. For each instance, we will compare the values filtered by each propagator for a given state of the variables. More especially, for a given state of the variables, the Oracle will remove all the values from the variables' domains that are in none of the optimal solutions.

5.2 Results

For the research of the optimal solution of a Bin Packing problem of size 10, we obtained the following results. For non trivial instances (that is to say the ones for which the optimal is different than the number of objects), BF-PROP solved all the instances in 17.4 seconds, the integer model based on Shaw's constraint in 503.6 seconds and the binary model in 1640.5 seconds. To this end, BF-PROP used 177417 nodes, the integer model 12147504 and the binary model 14047374 nodes. These results are indicated in the array in figure 1.

Moreover, BF-PROP also showed very interesting results concerning the filtering process. For each of the 100 instances of size 10, we randomly fixed 10%, 20%, 30%, 40%, 50% and 60% of the variables to one value of their domain. After applying the two propagators (BF-PROP and Shaw's propagator), we compare how many values they have filtered from the variables' domain. The process was repeated 30 times for each instance. The two graphics in figure 1 shows the total number of filtered values over all the instances and the 30 times the process was repeated. The first graphic shows the raw numbers of filtered values for each propagator, while the second one shows the ratio of filtered values for BF-PROP and for Shaw's propagator compared to the Oracle.



(a) (b)

	Total time of solving (seconds)	Total used nodes	Total encountered fails
BF-Prop	17.36	177417	177340
Shaw	503.6	12147504	12147425
Binary model	379.5	14047374	14047295

(c)

Fig. 1. Experimentations' results

For instance, for 10% fixed variables, BF-Prop succeeds in filtering 87% of the values that the Oracle has filtered, compared to the 45% got by Shaw's propagator.

First thing we remark is that the more variables are instantiated at the start of the filtering process, the more efficient are the two propagators, closing the gap with the Oracle. What is interesting to note is that even for a few percentage of fixed variables BF-Prop filters almost as much as the Oracle. From 2 fixed variables on the 10, BF-Prop filters over 90% of what the Oracle filters !

Finally, for every treated instance and for every percentage of instantiated variables, Shaw's propagator filters 67% of what the Oracle filters whereas BF-Prop filters 95% of what the Oracle filters.

6 Conclusion and future work

This paper has introduced a new filtering algorithm, based on the Best-Fit approximation algorithm, for the optimisation Bin-Packing problem. We empirically showed that this propagator can be of great efficiency and even be better than the known state-of-the-art global constraint.

Future work will consist in studying the potential extensibility of automatically deriving filtering algorithms from dedicated ones to other $\mathcal{NP}$-hard problems, still using *nogoods* at the base of the filtering process. We will also work on finding a measure of filtering efficiency in order to have a better hierarchy of filtering algorithms, currently formed of discontinuous class of consistencies, using approximation measures.

References

1. Castiñeiras, I., De Cauwer, M., O'Sullivan, B.: Weibull-Based Benchmarks for Bin Packing. Principles and Practice of Constraint Programming: 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings, 207–222 (2012)
2. Rossi, F., Van Beek, P., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc. New York, NY, USA (2006)
3. Shaw P.: A constraint for bin packing. In Mark Wallace, editor, CP, volume 3258 of Lecture Notes in Computer Science. Springer, 648–662 (2004)
4. Demange, M., Monnot, J., Paschos, V. T.: Bridging gap between standard and differential polynomial approximation: the case of bin-packing. Appl. Math. Lett., vol.12, 127–133 (1999)
5. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting Systematic Search by Weighting Constraints. ECAI2004, Proceedings of the 16th European Conference on Artificial Intelligence, IOS Press, Amsterdam, 146–150 (2004)
6. Coffman,Jr., E. G., Garey, M. R., Johnson, D. S.: Approximation Algorithms for Bin Packing: A Survey. In Approximation Algorithms for NP-hard Problems, PWS Publishing Co., Boston, MA, USA, 46–93 (1997)
7. Garey, Michael R. and Johnson, David S.: Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
8. Prud'homme, C.,Fages, J., Lorca, X.: Choco Documentation (2017) - http://www.choco-solver.org

A Global Constraint for the Exact Cover Problem: Application to Conceptual Clustering

Maxime Chabert[1,2] and Christine Solnon[1]

[1] Université de Lyon, INSA Lyon, LIRIS, F-69622, France
[2] Infologic, France

Abstract. We introduce a global constraint dedicated to the exact cover problem: given a set S of elements and a set P of subsets of S, *exactCover* ensures that a set variable C is assigned to a subset of P which forms a partition of S. We introduce a new propagation algorithm for *exactCover* that uses Dancing Links to efficiently maintain the data structure that links elements and subsets. We compare this algorithm with different decompositions of *exactCover*, and show that it is an order of magnitude faster. We also introduce an extension of *exactCover* for the case where costs are associated with subsets in P, and minimal and maximal costs of selected subsets are bounded by integer variables. We experimentally evaluate the relevance of *exactCover* on conceptual clustering problems, and show that it scales better than recent ILP and CP models.

1 Introduction

Given a set S of elements and a set $P \subseteq \mathcal{P}(S)$ of subsets of S, an exact cover is a subset $C \subseteq P$ which is a partition of S, *i.e.*, for each element $a \in S$, there is exactly one subset $u \in C$ such that $a \in u$. Throughout this paper, elements of P (*i.e.*, subsets) are denoted u or v, and elements of S are denoted a or b. For each $u \in P$, we denote $u = \{a \in S | a \in u\}$ the elements of u, and for each $a \in S$, we denote $cover(a) = \{u \in P | a \in u\}$ the set of subsets that contain a.

The exact cover problem is $\mathcal{NP}$-complete [13]. Examples of applications of this problem are tiling problems (each square of a figure must be covered by exactly one polyomino, where each polyomino is composed of several adjacent squares) [15], and instruction selection problems (each instruction of a function must be covered by exactly one processor instruction, where each processor instruction is composed of several instructions) [5,11]. Our interest for this problem comes from a conceptual clustering application which is described in Section 6.

Overview of the paper. In Section 2, we define the *exactCover* global constraint, and we describe three decompositions of *exactCover*. In Section 3, we introduce a filtering algorithm for *exactCover* that ensures the same level of consistency as decompositions, but at a lower cost thanks to Knuth's Dancing Links [15]. In Section 4, we study how to add constraints on the number of selected subsets. In Section 5, we study how to add constraints on the minimal and maximal costs of selected subsets, when costs are associated with each subset of P. In Section 6, we evaluate the relevance of *exactCover* on a conceptual clustering application.

2 Definition and decompositions of ExactCover

Given a set S, a set $P \subseteq \mathcal{P}(\mathcal{S})$, and a set variable C, the global constraint $exactCover_{S,P}(C)$ is satisfied iff C is assigned to a subset of P which is an exact cover of S. Enforcing Generalized Arc-Consistency (GAC) of *exactCover* is $\mathcal{NP}$-complete as the exact cover problem is $\mathcal{NP}$-complete.

We describe below three decompositions of *exactCover*. In all cases, for each element $a \in S$, an integer variable $coveredBy[a]$ is used to decide which subset of P covers a, and its domain is $D(coveredBy[a]) = cover(a)$.

Boolean decomposition (BoolDec). This decomposition is used in [11] to solve an instruction selection problem. It uses Boolean variables instead of a set variable C to represent the selected subsets: for each subset $u \in P$, a Boolean variable $isSelected[u]$ indicates if u is selected and it is channeled to the *coveredBy* variables by the constraint: $\forall a \in u, coveredBy[a] = u \Leftrightarrow isSelected[u]$.

Set decomposition (SetDec). This decomposition is used in [1] to solve a conceptual clustering problem. For each element $a \in S$, it ensures that $coveredBy[a]$ belongs to C by posting the constraint $coveredBy[a] \in C$, and that a is covered by exactly one subset in C by posting the constraint $\#(cover(a) \cap C) = 1$.

Global cardinality decomposition (GccDec). This decomposition is used in [5] to solve an instruction selection problem. It exploits the fact that, for each subset $u \in P$, the number of *coveredBy* variables assigned to u is either equal to 0 (if u is not selected), or to $\#u$ (if u is selected). Hence, to model the exact cover problem, an Integer variable $nb[u]$ is associated with every subset $u \in P$: it represents the number of times u is assigned to a *coveredBy* variable and its domain is $D(nb[u]) = \{0, \#u\}$. ExactCover is equivalent to $gcc(coveredBy, P, nb)$.

Consistencies ensured by these decompositions. Enforcing Arc Consistency (AC) on BoolDec or SetDec ensures the same filtering level: it ensures that, for each pair of elements $\{a, b\} \subseteq S$ such that $D(coveredBy[a]) = \{u\}$, if $b \in u$ then $D(coveredBy[b]) = \{u\}$; otherwise, $\forall v \in D(coveredBy[b])$, $v \cap u = \emptyset$ (domains of *isSelected* variables (for BoolDec) or of C (for SetDec) are filtered consequently). For GccDec, as nb domains are not intervals, enforcing GAC is $\mathcal{NP}$-complete, and solvers usually enforce weaker consistencies [22,17].

Variable ordering heuristic. As pointed out by Knuth [15], the order used to select subsets has a great impact on performance, and better results are usually obtained by first selecting a subset that covers an element that belongs to the smallest number of possible subsets. When searching for exact covers with BoolDec, SetDec, or GccDec, this amounts to branching on *coveredBy* variables and using the *minDom* heuristic to select the next *coveredBy* variable to assign.

3 Propagation algorithm for exactCover

We introduce a propagation algorithm which enforces the same level of consistency as AC on BoolDec or SetDec. More precisely, it ensures the following

property: $\forall u \in C.lb, \forall v \in C.ub, u \neq v \Rightarrow u \cap v = \emptyset$ (where $C.lb$ and $C.ub$ denote the kernel and the envelope of the domain of C, respectively, *i.e.*, the domain of C contains every set X such that $C.lb \subseteq X \subseteq C.ub$).

To enforce this consistency, each time a subset u is added to $C.lb$, we remove from $C.ub$ every subset v such that $u \cap v \neq \emptyset$. To do this efficiently, we use a backtrackable data-structure introduced by Knuth for the exact cover problem, *i.e.*, *Dancing Links* [15]. Let us denote $uncov = \{a \in S | \forall u \in C.lb, a \notin u\}$ the set of elements that are not covered by a subset in the kernel of C. The idea is to maintain two collections of doubly linked circular lists: for each $a \in uncov$, the list L_a contains all the subsets that belong to $cover(a) \cap C.ub$; and for each $u \in P$, the list L_u contains all elements that belong to u. We also maintain a doubly linked circular list L_{uncov} which contains all elements of $uncov$. The relevance of using doubly linked circular lists is that removing a cell x may be undone (when backtracking) without knowing anything else but x. More precisely, let $prev[x]$ (resp. $next[x]$) be the cell before (resp. after) x. Removing x is done by performing $prev[next[x]] \leftarrow prev[x]$ and $next[prev[x]] \leftarrow next[x]$ (without modifying $prev[x]$ or $next[x]$), while restoring x is done by performing $prev[next[x]] \leftarrow x$ and $next[prev[x]] \leftarrow x$ (see [15] for details).

When a subset u is added to $C.lb$, we traverse L_u (using *next* links) and perform the following operations for each element a in L_u:

- Remove a from L_{uncov}
- Traverse L_a (using *next* links) and for each subset v in L_a:
  - Remove v from $C.ub$
  - Traverse L_v (using *next* links) and for each b in L_v: remove v from L_b

When backtracking, we perform the inverse operations: elements are restored instead of being removed and lists are traversed in reverse order by using *prev* links instead of *next* links. We only need to memorize the initial decision, *i.e.*, the addition of u to $C.lb$.

In both cases, the complexity is in $\mathcal{O}(d_P^2 \cdot d_S)$ where $d_P = \max_{u \in P} \#u$ and $d_S = \max_{a \in S} \#cover(a)$. This is an upper bound: in practice, for each element $a \in S$, the list L_a may contain fewer cells than $\#cover(a)$ as cells are removed each time a subset node is added to $C.lb$.

Ordering heuristic. To implement the same search as BoolDec, SetDec, and GccDec, at each search node, we traverse the list L_{uncov} to search for the smallest list L_a (list sizes are incrementally maintained), we select a subset u in L_a, and we branch on adding u to $C.lb$.

Experimental comparison. BoolDec, SetDec, GccDec, and our new global constraint (called *EC*) are all implemented in Choco v.4.0.3 [21]. All experiments were conducted on Intel(R) Core(TM) i7-6700 and 65GB of RAM. We consider the problem of enumerating all solutions of instances built from the instance ERP1 described in [1]. ERP1 has $\#S = 50$ elements and $\#P = 1579$ subsets. As it has a huge number of solutions, we consider instances obtained from ERP1 by selecting $x\%$ of its subsets in P, with $x \in [10, 30]$. For each value of x, we have randomly generated 10 instances. We consider the same ordering heuristics for all models (as described previously), and we break ties by fixing an order on

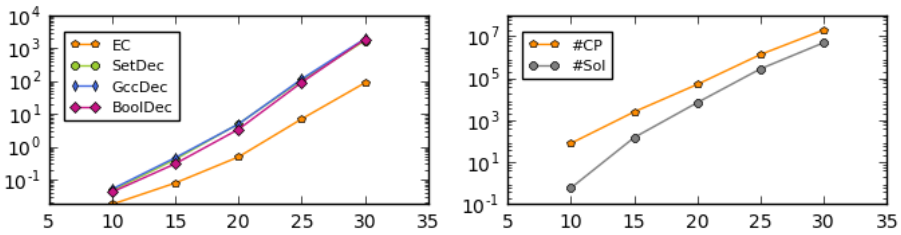


Fig. 1: Left: Evolution of the time (in seconds) spent by EC, BoolDec, SetDec, and GccDec to enumerate all solutions when varying the percentage x of subsets for ERP1 (average on 10 instances per value of x). Right: Evolution of the number of choice points (#CP) and solutions (#Sol) when varying x.

subsets and elements. Fig. 1 shows us that all models explore the same number of choice points. This was expected for SetDec, BoolDec, and EC. For GccDec, the consistency ensured by Choco is not formally defined. In practice, it explores exactly the same number of choice points as SetDec, BoolDec, and EC. The three decompositions have very similar times while EC is an order of magnitude faster. For example, when $x = 30\%$, EC needs 88s, on average, to enumerate all solutions whereas BoolDec, SetDec, and GccDec need 1861s, 1696s, and 1936s, respectively.

4 Constraining the cardinality of C

In some cases, the number of selected subsets must be constrained. In this case, we declare an integer variable K and constrain it to be equal to the cardinality of C. To do this, a first possibility (called *EC-C*) is to post the constraint $\#C = K$. A second possibility (called *EC-NV*) is to use *nvalues* [19]: we declare, for each element $a \in S$, an integer variable $coveredBy[a]$ which is linked with C by adding the constraint $coveredBy[a] \in C$, and we constrain K to be equal to the cardinality of C by posting the constraint $nvalues(coveredBy, K)$. Finally, a third possibility is to extend *exactCover* by adding K to its arguments: we define the global constraint $exactCoverK_{S,P}(C, K)$ which is satisfied iff C is an exact cover of S and the cardinality of C is equal to K. This constraint is propagated like *exactCover*, but we also ensure that $\#C.lb \leq K.lb \leq K.ub \leq \min\{\#C.ub, \#C.lb + \#uncov\}$ (where $K.lb$ and $K.ub$ denote the smallest and largest value in the domain of K). Furthermore, when there are at most two subsets that can still be added to C (*i.e.*, when $K.ub - \#C.lb \leq 2$), we remove from $C.ub$ every subset u such that $u \neq uncov$ and $\forall v \in C.ub \setminus C.lb, \{u, v\}$ is not a partition of $uncov$.

In Fig. 2, we compare EC-C, EC-NV, and *exactCoverK* (EC-K) for enumerating all solutions of an instance obtained from ERP1 by selecting 25% of the subsets in P, when constraining the cardinality of C to be equal to k (with

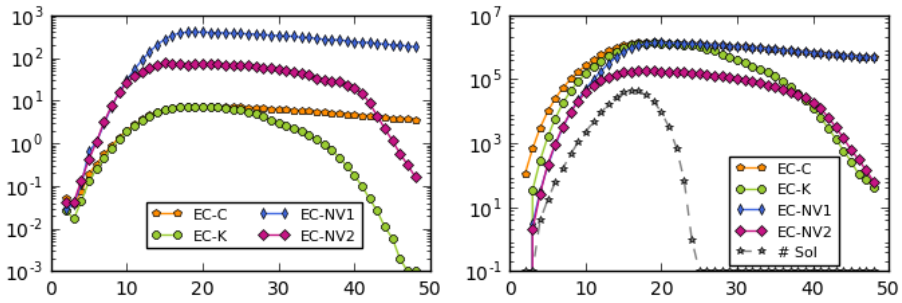


Fig. 2: Left: Evolution of the time (in seconds) spent by EC-K, EC-C, EC-NV with weak propagation (EC-NV1) and strong propagation (EC-NV2) to enumerate all solutions when varying k from 2 to 49, and constraining the cardinality of C to be equal to k, for an instance obtained from ERP1 by selecting 25% of P. Right: Evolution of the number of choice points and solutions.

$k \in [2, \#S - 1]$). For EC-NV, we consider two filtering levels: *nvalues* is implemented in Choco by combining *atMostNvalues* and *atLeastNvalues*, and each of these constraints has two filtering levels (weak and strong). We denote EC-NV1 (resp. EC-NV2) when weak (resp. strong) filtering is used for both constraints. As expected, EC-NV2 explores fewer choice points than other variants, except when $k \geq 39$, where EC-K explores slightly fewer choice points. However, this strong propagation is expensive and EC-K is always faster than other variants: it is similar to EC-C when $k < 30$ and it is clearly more efficient for larger values. Indeed, when k becomes large, EC-K detects efficiently that the current assignment is not consistent as soon as the number of uncovered elements becomes lower than the number of subsets that still must be added to $C.lb$.

5 Constraining cost bounds

In some applications, costs are associated with subsets of P, and the costs of the selected subsets must range between some given bounds. Let n be the number of costs, $c : [1, n] \times P \to \mathbb{R}$ be a function such that $c(i, u)$ is the i^{th} cost associated with subset u, and Min and Max be two arrays of n integer variables. The constraint $exactCoverCost_{S,P,c}(C, Min, Max)$ is satisfied iff C is an exact cover of S such that for each $i \in [1, n]$, $Min[i] = \min_{u \in C} c(i, u)$ and $Max[i] = \max_{u \in C} c(i, u)$. This constraint is propagated like *exactCover*, but we also ensure that, for each cost $i \in [1, n]$, $\forall u \in C.lb, Min[i].ub \leq c(i, u) \leq Max[i].lb$ and $\forall u \in C.ub, Min[i].lb \leq c(i, u) \leq Max[i].ub$. When the cardinality of C must also be constrained, we define the constraint $exactCoverCostK_{S,P,c}(C, Min, Max, K)$ that also performs filterings on K as defined in Section 4. In the next section, we use *exactCoverCostK* to solve conceptual clustering problems.

6 Application to Conceptual Clustering Problems

Clustering aims at grouping objects described by attributes into homogeneous clusters. The key idea of conceptual clustering is to provide a description of clusters. Usually, each cluster corresponds to a formal concept, *i.e.*, sets of objects that share a same subset of attributes. More formally, let $\mathcal{O}$ be a set of objects, and for each object $o \in \mathcal{O}$, let $attr(o)$ be the set of attributes that describes o. The intent of a subset of objects $O_i \subseteq \mathcal{O}$ is the set of attributes common to all objects in O_i, *i.e.*, $intent(O_i) = \cap_{o \in O_i} attr(o)$. O_i is a formal concept if $intent(O_i) \neq \emptyset$ and if $O_i = \{o \in \mathcal{O} : intent(O_i) \subseteq attr(o)\}$. A conceptual clustering is a partition of $\mathcal{O}$ in k subsets $O_1, \ldots, O_k$ such that, for each $i \in [1, k]$, O_i is a formal concept.

Computation of formal concepts. Formal concepts correspond to *closed itemsets* [20] and the set of all closed itemsets may be computed by using algorithms dedicated to the enumeration of frequent closed itemsets. In particular, LCM [26] extracts all formal concepts in linear time with respect to the number of formal concepts. Constraint Programming (CP) has been widely used to model and solve itemset search problems [23,14,8,7,16,24,25]. Indeed, CP allows one to easily model various constraints on the searched itemsets, and these constraints are used to reduce the search space.

CP for Conceptual Clustering. Conceptual clustering is a special case of k-pattern set mining, as introduced in [9]: conceptual clustering is defined by combining a cover and a non-overlapping constraint, and a CP model is proposed to solve this problem. [2] describes a CP model for clustering problems where a dissimilarity measure between objects is provided, and this CP model has been extended to conceptual clustering in [3]. Experimental results reported in [3] show that this model outperforms the binary model of [7]. [1] introduces another CP model, which improves the model of [3] and has been shown to scale better.

Conceptual Clustering as an Exact Cover Problem. There exist very efficient tools (*e.g.*, LCM [26]) for computing the set $\mathcal{F}$ of all formal concepts. Given this set $\mathcal{F}$, a conceptual clustering problem may be seen as an exact cover problem: the goal is to find a subset of $\mathcal{F}$ that covers every object exactly once. This exact cover problem may be solved by using Integer Linear Programming (ILP). In particular, [18] proposes an ILP model to solve this exact cover problem. This exact cover problem may also be solved by using CP, and [1] introduces a CP model which corresponds to the set decomposition described in Section 2.

Test instances. We describe in Table 1 six classical machine learning instances, coming from the UCI database, and six ERP instances coming from [1] (we do not consider ERP1 which is trivial).

Experimental results when optimizing a single criterion. We consider the problem of finding a conceptual clustering that optimizes a cost associated with formal concepts. We consider four different costs: the size (resp. frequency, diameter, and split) of a formal concept O_i is $\#intent(O_i)$ (resp. $\#O_i$, $\max_{o,o' \in O_i} d(o, o')$, and $\min_{o \in O_i, o' \in \mathcal{O} \setminus O_i} d(o, o')$, where $d(o, o') = 1 - \frac{\#(attr(o) \cap attr(o'))}{\#(attr(o) \cup attr(o'))}$ is the Jaccard

Table 1: Test instances: for each instance, $\#S$ gives the number of objects, $\#A$ gives the number of attributes, $\#P$ gives the number of formal concepts, and T gives the time (in seconds) spent by LCM to extract all formal concepts.

Name	$\#S$	$\#A$	$\#P$	T
ERP 2	47	47	8 1337	0.03
ERP 3	75	36	10 835	0.03
ERP 4	84	42	14 305	0.05
ERP 5	94	53	63 633	0.28
ERP 6	95	61	71 918	0.45
ERP 7	160	66	728 537	5.31

Name	$\#S$	$\#A$	$\#P$	T
UCI 1 (zoo)	101	36	4 567	0.01
UCI 2 (soybean)	630	50	31 759	0.10
UCI 3 (primary-tumor)	336	31	87 230	0.28
UCI 4 (lymph)	148	68	154 220	0.52
UCI 5 (vote)	435	48	227 031	0.68
UCI 6 (hepatitis)	137	68	3 788 341	13.9

Table 2: Time (in seconds) spent by ILP, HCP, FCP and EC to find a conceptual clustering that maximizes the minimal size (resp., maximizes the minimal frequency, minimizes the maximal diameter, and maximizes the minimal split). We report '-' when time exceeds 1000s.

	Max(SizeMin)				Max(FreqMin)				Min(DiamMax)				Max(SplitMin)			
	ILP	HCP	FCP	EC	ILP	HCP	FCP	EC	ILP	HCP	FCP	EC	ILP	HCP	FCP	EC
ERP 2	1.0	0.1	0.1	**0.0**	1.7	0.4	0.3	**0.1**	1.0	0.1	0.1	**0.0**	0.6	0.1	0.5	**0.0**
ERP 3	4.4	0.1	0.1	**0.0**	2.7	0.5	0.6	**0.1**	1.3	0.1	2.6	**0.0**	0.8	0.1	2.3	**0.0**
ERP 4	4.1	0.7	1.4	**0.1**	21.1	0.6	0.6	**0.1**	8.7	0.7	1.6	**0.1**	1.5	**0.1**	2.6	**0.1**
ERP 5	26.7	0.8	**0.2**	0.4	27.3	6.4	**1.0**	2.5	15.4	1.1	0.6	**0.4**	6.6	1.2	3.2	**0.4**
ERP 6	21.8	4.5	2.7	**0.7**	357.7	6.6	**1.5**	2.2	124.0	4.9	1.4	**0.6**	14.1	1.7	4.6	**0.7**
ERP 7	-	132.3	**6.5**	8.0	-	-	**9.6**	861.9	-	144.7	**3.9**	19.7	-	12.4	25.2	**6.8**
UCI 1	3.1	0.2	0.5	**0.1**	2.0	0.9	1.8	**0.1**	2.9	0.1	0.6	**0.0**	0.9	0.1	12.0	**0.0**
UCI 2	15.5	4.8	-	**0.2**	-	31.7	131.7	**1.0**	15.0	4.4	-	**0.2**	15.7	3.3	620.2	**0.2**
UCI 3	-	4.6	272.5	**0.4**	-	62.1	24.7	**0.8**	-	4.3	268.1	**0.4**	-	313.0	60.4	**0.5**
UCI 4	-	29.4	40.9	**1.1**	-	19.0	5.1	**1.8**	-	3.5	4.0	**1.0**	52.7	3.2	4.9	**0.6**
UCI 5	84.7	19.7	186.5	**0.9**	-	83.6	-	**1.1**	83.2	18.7	190.0	**0.9**	-	-	203.3	**1.4**
UCI 6	-	104.6	**1.1**	16.4	-	-	**2.5**	139.4	-	115.2	**1.8**	27.5	-	195.3	**9.0**	17.6

distance). The goal is to maximize the minimal cost of selected formal concepts for size, frequency and split, and to minimize the maximal cost for diameter. Table 2 reports results when optimizing a single criterion at a time, and when the number of selected formal concepts is constrained to belong to $[2, \#\mathcal{O} - 1]$. This problem is solved with *exactCoverCostK* (where the number of costs $n = 1$) while refining the ordering heuristic described in Section 3 as follows: when choosing the subset $u \in L_a$ to be added to $C.lb$, we choose the subset with the best cost. This model, called EC, is compared with the ILP model of [18] (denoted ILP), the full CP model of [1] (denoted FCP), and the hybrid CP model of [1] (denoted HCP). For EC, ILP, and HCP, we first compute the set of all formal concepts with LCM before solving an exact cover problem, and times reported in Table 2 include the time spent by LCM. EC is always much faster than ILP and HCP. EC is often faster than FCP, but for some instances FCP is faster than EC.

Experimental results when computing Pareto Fronts. Finally, we consider the problem of computing the Pareto front of non-dominated conceptual clusterings

	Frequency/Size				Split/Diameter			
	$\#s$	ILP	HCP	EC	$\#s$	ILP	HCP	EC
ERP 2	9	3.8	30.7	**1.4**	5	1.4	34.9	**0.9**
ERP 3	10	5.9	295.9	**5.1**	2	**1.4**	79.7	1.5
ERP 4	13	39.0	949.4	**18.7**	2	19.9	588.4	**5.7**
ERP 5	13	**88.3**	-	142.4	3	**27.2**	-	141.0
ERP 6	15	**445.0**	-	837.4	3	268.2	-	**147.5**

	Frequency/Size				Split/Diameter			
	$\#s$	ILP	HCP	EC	$\#s$	ILP	HCP	EC
UCI1	13	8.0	87.3	**3.5**	3	2.5	14.5	**0.1**
UCI2	-	-	-	-	3	242.2	-	**0.3**
UCI3	-	-	-	-	1	-	-	**0.3**
UCI4	-	-	-	-	5	-	-	**405.5**
UCI5	-	-	-	-	2	-	-	**0.6**

Table 3: Time (in seconds) spent by ILP, HCP, and EC to compute the set of non-dominated solutions for Frequency/Size and Split/Diameter. $\#s$ gives the number of non-dominated solutions. We report '-' when time exceeds 3600.

for two pairs of conflicting criteria (*i.e.*, Frequency/Size and Split/Diameter) when the number of selected formal concepts is constrained to belong to $[2, \#\mathcal{O}-1]$. This problem is solved with *exactCoverCostK* (where the number of costs $n = 2$), using the same dynamic approach as [6,10,25,1] to compute the Pareto front: we search for all solutions and, each time a solution *sol* is found, we dynamically post a constraint that prevents the search from finding solutions dominated by *sol*. Also, we refine the ordering heuristic described in Section 3 as follows: when choosing the subset $u \in L_a$ to be added to $C.lb$, we choose the non-dominated subset with the best cost on one of the two criteria. This model, called EC, is compared with the hybrid CP model of [1] (denoted HCP), using the same dynamic approach to compute the Pareto front. We also compare EC with the ILP model of [18], using the approach proposed in [2] to compute the Pareto front: we iteratively solve single criterion optimization problems while alternating between the two criteria. We do not compare with the Full CP model of [1] as it is not able to solve this problem within a reasonable amount of time.

Table 3 shows us that EC is much faster than HCP. EC is faster than ILP on UCI instances, and it is competitive on ERP instances. For all approaches, the Pareto front is smaller and also easier to compute when considering the Split/Diameter criterion, instead of the Frequency/Size criterion. This may come from the fact that Frequency and Size criterion are very conflicting criteria (formal concepts with large frequencies usually have small sizes, and vice versa).

7 Conclusion

We have experimentally shown that using Dancing Links to propagate an exact cover constraint significantly speeds up the filtering process, compared to classical decompositions. This allows CP to efficiently solve conceptual clustering problems. As further works, we plan to compare *exactCover* with the SAT models introduced in [12]. We also plan to extend *exactCover* to a soft version, such that a limited number of elements may be either not covered, or covered by more than one subset: this could be used to solve soft clustering problems and we plan to compare CP with the ILP approach of [18]. Finally, we plan to investigate the relevance of more advanced filterings that integrate, for example, the simplifications proposed in [4] for the hitting set problem.

References

1. Maxime Chabert and Christine Solnon. Constraint programming for multi-criteria conceptual clustering. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 460–476. Springer, 2017.
2. Thi-Bich-Hanh Dao, Khanh-Chuong Duong, and Christel Vrain. Constrained Clustering by Constraint Programming. *Artificial Intelligence*, pages –, June 2015.
3. Thi-Bich-Hanh Dao, Willy Lesaint, and Christel Vrain. Clustering conceptuel et relationnel en programmation par contraintes. In *JFPC 2015*, Bordeaux, France, June 2015.
4. Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Principles and Practice of Constraint Programming - 17th International Conference, CP. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2011.
5. Antoine Floch, Christophe Wolinski, and Krzysztof Kuchcinski. Combined scheduling and instruction selection for processors with reconfigurable cell fabric. In *21st IEEE International Conference on Application-specific Systems Architectures and Processors, ASAP 2010*, pages 167–174, 2010.
6. Marco Gavanelli. An algorithm for multi-criteria optimization in csps. In *Proceedings of the 15th European Conference on Artificial Intelligence*, ECAI'02, pages 136–140, Amsterdam, The Netherlands, The Netherlands, 2002. IOS Press.
7. Tias Guns. Declarative pattern mining using constraint programming. *Constraints*, 20(4):492–493, 2015.
8. Tias Guns, Siegfried Nijssen, and Luc De Raedt. Itemset mining: A constraint programming perspective. *Artif. Intell.*, 175(12-13):1951–1983, 2011.
9. Tias Guns, Siegfried Nijssen, and Luc De Raedt. k-pattern set mining under constraints. *IEEE Trans. Knowl. Data Eng.*, 25(2):402–418, 2013.
10. Renaud Hartert and Pierre Schaus. A support-based algorithm for the bi-objective pareto constraint. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 2674–2679. AAAI Press, 2014.
11. Gabriel Hjort Blindell. *Universal Instruction Selection*. PhD thesis, 2018.
12. Tommi Junttila and Petteri Kaski. Exact cover via satisfiability: An empirical study. In *Principles and Practice of Constraint Programming – CP 2010*, pages 297–304. Springer, 2010.
13. Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer, 1972.
14. Mehdi Khiari, Patrice Boizumault, and Bruno Crémilleux. Constraint programming for mining n-ary patterns. In *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, pages 552–567, 2010.
15. Donald E Knuth. Dancing links. *Millenial Perspectives in Computer Science*, 18:4, Sep 2009.
16. Nadjib Lazaar, Yahia Lebbah, Samir Loudni, Mehdi Maamar, Valentin Lemière, Christian Bessiere, and Patrice Boizumault. A global constraint for closed frequent pattern mining. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, pages 333–349, 2016.
17. Peter Nightingale. The extended global cardinality constraint: An empirical survey. *Artificial Intelligence*, 175(2):586 – 614, 2011.

18. Abdelkader Ouali, Samir Loudni, Yahia Lebbah, Patrice Boizumault, Albrecht Zimmermann, and Lakhdar Loukil. Efficiently finding conceptual clustering models with integer linear programming. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI 2016, New York, NY, USA, 9-15 July 2016*, pages 647–654, 2016.
19. François Pachet and Pierre Roy. Automatic generation of music programs. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99*, pages 331–345, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
20. Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lotfi Lakhal. Discovering frequent closed itemsets for association rules. In *Database Theory - ICDT '99, 7th International Conference*, pages 398–416, 1999.
21. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2016.
22. Claude-Guy Quimper, Alejandro López-Ortiz, Peter van Beek, and Alexander Golynski. Improved algorithms for the global cardinality constraint. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, pages 542–556. Springer, 2004.
23. Luc De Raedt, Tias Guns, and Siegfried Nijssen. Constraint programming for itemset mining. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Las Vegas, Nevada, USA, August 24-27, 2008*, pages 204–212, 2008.
24. Pierre Schaus, John O. R. Aoga, and Tias Guns. Coversize: A global constraint for frequency-based itemset mining. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 529–546. Springer, 2017.
25. Willy Ugarte, Patrice Boizumault, Bruno Crémilleux, Alban Lepailleur, Samir Loudni, Marc Plantevit, Chedy Raïssi, and Arnaud Soulet. Skypattern mining: From pattern condensed representations to dynamic constraint satisfaction problems. *Artif. Intell.*, 244:48–69, 2017.
26. Takeaki Uno, Tatsuya Asai, Yuzo Uchida, and Hiroki Arimura. *An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases*, pages 16–31. Springer Berlin Heidelberg, 2004.

Testing Global Constraints

Aurélie Massart, Valentin Rombouts, and Pierre Schaus

UCLouvain/ICTEAM Belgium

Abstract. Every Constraint Programming (CP) solver exposes a library of constraints for solving combinatorial problems. In order to be useful, CP solvers need to be bug-free. Therefore the testing of the solver is crucial to make developers and users confident. We present a Java library allowing any JVM based solver to test that the implementations of the individual constraints are correct. The library can be used in a test suite executed in a continuous integration tool or it can also be used to discover minimalist instances violating some properties (arc-consistency, etc) in order to help the developer to identify the origin of the problem using standard debuggers.

Keywords: Constraint Programming · Testing · Filtering

1 Introduction

The filtering algorithms inside constraint programming solvers ([1,2,3,4] etc.) are mainly tested using test suites implemented manually. Creating such unit tests is a significant workload for the developers and is also error prone.

The most elementary yet important test to achieve for a constraint is that no feasible solution is removed. One can always implement a checker verifying the feasibility of the constraint when all the variables are bound. By comparing the number of solutions generated with both the checker and the tested filtering algorithm, one can be confident that no solution is removed. This procedure can be repeated for many (small) instances (possibly randomly generated). Alternatively, one can compare with a decomposition of the constraint into (more) elementary ones. This latter approach can improve the coverage of the test suite.

Those unit tests verifying the non removal of feasible solutions do not verify other properties of constraints generally more difficult to test. For instance, the domain-consistency property is rarely tested outside some hard-coded small test examples.

We introduce CPChecker as a tool to ease the solver developer's life by automating the testing of properties of filtering algorithms. For instance, *algorithm A should filter more than algorithm B* or *Algorithm A should achieve arc or bound-consistency*, etc. The tool does not ensure that the tested filtering does not contain any bug - as it is impossible to test all possible input domains - but it can reveal the presence of one, if a test fails. The large variety of input domains pseudo-randomly generated should make the user confident that the tool would allow to detect most of the bugs.

Many constraint implementations are stateful and maintain some reversible data structures. Indeed, global constraints' filtering algorithms often maintain an internal state in order to be more efficient than their decomposition. This reversible state is also a frequent source of bugs. CPChecker includes the trail-based operations when testing constraints such that any bug due to the state management of the constraint also has a high chance to be detected. CPChecker is generic and can be interfaced with any JVM trailed based solvers. CPChecker is able to generate detailed explanations by generating minimal domain examples on which the user's filtering has failed, if any.

Related work In [5,6,7], the authors introduce tools to debug models. Some researches have also been done to help programmers while debugging codes for constraint programming [8]. To the best of our knowledge, these tools, unlike CPChecker, do not focus on the filtering properties of individual constraints.

In the next sections, we first detail how to test static filtering algorithms before explaining the testing of stateful filtering algorithms for trailed based solvers. Finally we introduce how CPChecker can be integrated into a test suite.

2 Testing Static Filtering Algorithms

CPChecker is able to test any static filtering algorithm acting over integer domains. Therefore, the user needs to implement a function taking some domains (array of set of ints) as input and returning the filtered domains[1]:

```
abstract class Filter {
 def filter(variables: Array[Set[Int]]): Array[Set[Int]]
}
```

CPChecker also needs a trusted filtering algorithm serving as reference with the same signature. The least effort for a user is to implement a checker for the constraint under the form of a predicate that specifies the semantic of the constraint. For instance a checker for the constraint $\sum_i x_i = 15$ can be defined as

```
 def sumChecker(x: Array[Int]): Boolean = x.sum == 15
```

One can create with CPChecker an Arc/Bound-Z/Bound-D/Range Consistent filtering algorithm by providing in argument to the corresponding constructor the implementation of the checker. For instance

```
class ArcFiltering(checker: Array[Int] => Boolean) extends
    Filter
val trustedArcSumFiltering = new ArcFiltering(sumChecker)
```

[1] Most of the code fragments presented are in Scala for the sake of conciseness but the library is compatible with any JVM-based language.

This class implements the `filter` function as a trusted filtering algorithm reaching the arc consistency by 1) computing the Cartesian product of the domains, 2) filtering with the checker the non solutions and 3) creating the filtered domains as the the union of the values. Similar filtering algorithms' (Bound-Z, Bound-D and Range) have been implemented from a checker.

Finally the `check` and `stronger` functions permit to respectively check that two compared filtering algorithms are the same or that the tested filtering is stronger than the trusted one.

```
def check/stronger(trustedFiltering: Filter, testedFiltering:
    Filter) : Boolean
```

The testing involves the following steps:

1. Random Domains generation [2].
2. Execution of the tested and trusted filtering algorithms (from CPChecker's filterings or another trusted one) to these random domains.
3. Comparison of the domains returned by the two filtering algorithms.

This process is repeated by default 100 times although all the parameters can be overridden for the creation of random domains, number of tests, etc.

2.1 Generation of Random Test Instances

In order to test a filtering implementation, CPChecker relies on a *property based testing* library called *ScalaCheck*[9][3]. This library includes support for the creation of random generators and for launching multiple test cases given those. CPChecker also relies on the ability of *ScalaCheck* of reducing the instance to discover a smaller test instance over which the error occurs.

2.2 Example

Here is an example for testing with CPChecker the arc-consistent *AllDifferent* constraint's in ***OscaR*** [2] solver :

```
object ACAllDiffTest extends App {
  def allDiffChecker(x: Array[Int]): Boolean = x.toSet.size ==
    x.length
  val trustedACAllDiff: Filter = new ArcFiltering(
   allDiffChecker)
  val oscarACAllDiff: Filter = new Filter {
    override def filter(variables: Array[Set[Int]]): Array[Set
   [Int]] = {
      val cp: CPSolver = CPSolver()
      val vars = variables.map(x => CPIntVar(x)(cp))
      val constraint = new AllDiffAC(vars)
      try {
```

[2] A seed can be set to reproduce the same tests.

[3] Similar libraries exist for most programming languages, all inspired by QuickCheck for Haskell.

```
10          cp.post(constraint)
11        } catch {
12          case _: Inconsistency => throw new NoSolutionException
13        }
14        vars.map(x => x.toArray.toSet)
15      }
16    }
17    check(trustedACAllDiff, oscarACAllDiff)
18  }
```

The trusted filtering algorithm is created thanks to the `ArcFiltering` class at line 3. The checker for AllDifferent simply verifies that the union of the values in the array has a cardinality equal to the size of the array, as defined at line 2. The tested filtering implements the `filter` function using ***OscaR***'s filtering. It first transforms the variables into ***OscaR***'s variables (line 7) then creates the constraint over them (line 8). It is then posted to the solver which filters the domains until fix-point before returning them.

3 Testing stateful constraints

Incremental Filtering Algorithms usually maintain some form of state in the constraints. It can for instance be reversible data-structures for trailed-based solvers. CPChecker allows to test a stateful filtering algorithm by testing it during a search while checking the state restoration. In terms of implementation, the incremental `check` and `stronger` functions compare `FilterWithState` objects that must implement two functions. The *setup* function reaches the fix-point while setting up the solver used for the search. The *branchAndFilter* function applies a branching operation on the current state of the solver and reaches a new fix-point for the constraint. The branching operations represent standard branching constraints such as $=, \neq, <, >$ and the `push/pop` operations on the trail allowing to implement the backtracking mechanism (see [10] for further details on this mechanism).

```
1  abstract class FilterWithState {
2    def setup(variables: Array[Set[Int]]): Array[Set[Int]]
3
4    def branchAndFilter(branching: BranchOp): Array[Set[Int]]
5  }
```

The process of testing an incremental/stateful filtering algorithm is divided into four consecutive steps :

1. Domains generation
2. Application of the *setup* function of the tested and trusted filtering algorithms.
3. Comparing the filtered domains returned at step 2.
4. Execution of a number of fixed number dives as explained next based on the application of *branchAndFilter* function.

3.1 Dives

A dive is performed by successively interleaving a push of the state and a domain restriction operation. When a leaf is reached (no or one solution remaining) the dive is finished and a random number of states are popped to start a new dive as detailed in the algorithm 1.

Algorithm 1: Algorithm performing dives

```
Dives (root, trail, nbDives)
    dives ← 0
    currentDomains ← root
    while dives < nbDives do
        while !currentDomains.isLeaf do
            trail.push(currentDomains)
            restriction ← new RandomRestrictDomain(currentDomains)
            currentDomains ← branchAndFilter(currentDomains, restriction)
        dives ← dives + 1
        for i ← 1 to Random(1,trail.size-1) do
            trail.pop()
```

3.2 Illustration over an Example

The next example illustrates CPChecker to test the ***OscaR***[2]'s filtering for the constraint $\sum_i x_i = 15$. It should reach Bound-Z consistency.

```
1  object SumBCIncrTest extends App {
2
3    def sumChecker(x: Array[Int]): Boolean = x.sum == 15
4    val trusted = new IncrementalFiltering(new BoundZFiltering(
       sumChecker))
5    val tested = new FilterWithState {
6      val cp: CPSolver = CPSolver()
7      var currentVars: Array[CPIntVar] = _
8
9      override def branchAndFilter(branching: BranchOp): Array[
       Set[Int]] ={
10       branching match {
11         case _: Push => cp.pushState()
12         case _: Pop => cp.pop()
13         case r: RestrictDomain => try {
14             r.op match {
15               case "=" => cp.post(currentVars(r.index) == r.
       constant)
16               ...}
17           } catch {
18             case _: Exception => throw new NoSolutionException
19           }
20       }
21       currentVars.map(x => x.toArray.toSet)
22     }
23
24     override def setup(variables: Array[Set[Int]]): Array[Set[
       Int]] = {
25       currentVars = variables.map(x => CPIntVar(x))
26       try {
27         solver.post(sum(currentVars) == 15)
28       } catch {
29         case _: Exception => throw new NoSolutionException
30       }
31       currentVars.map(x => x.toArray.toSet)
32     }
33   }
34   check(trusted, tested)
35 }
```

In this example, two `FilterWithState` are compared with the `check` function.

In CPChecker, the `IncrementalFiltering` class implements the `FilterWithState` abstract class for any `Filter` object. Therefore, the `IncrementalFiltering` created with a `BoundZFiltering` object is used as the trusted filtering (line 4) which it-self relies on the very simple `sumChecker` function provided by the user and assumed to be bug-free.

4 Custom Assertions

To ease the integration into a JUnit like test suite, CPChecker has custom assertions extending the *AssertJ*[11] library. The classes `FilterAssert` and `FilterWithStateAssert` follow the conventions of the library with the `filterAs` and `weakerThan` functions to respectively test a filtering algorithm, as in the `check` and `stronger` functions. An example of assertion is:

```
1 assertThat(tested).filterAs(trusted1).weakerThan(trusted2)
```

5 Code Source

CPChecker's code source is publicly available in the *Github* repository[4]. This repository also contains several examples of usage of CPChecker with both *Scala* solver and *Java* solvers, namely ***OscaR***[2], ***Choco***[1] and ***Jacop***[3]. From those examples, *CPChecker* detected that the arc consistent filtering of the *Global Cardinality* constraint of ***OscaR*** was not arc consistent for all the variables (the cardinality variables). This shows the genericity of *CPChecker* and that it can be useful to test and debug filtering algorithms with only a small workload for the user. Further details on the architecture and implementation of CPChecker can be found in the Master Thesis document available at the github repository[4].

6 Conclusion and Future Work

This article presented CPChecker, a tool to test filtering algorithms implemented in any JVM-based programming language based on the JVM. Filtering algorithms are tested over domains randomly generated which is efficient to find unexpected bugs. Principally written in *Scala*, CPChecker can be used to test simple and stateful filtering algorithms. It also contains its own assertions system to be directly integrated into test suites. As future work, we would like to integrate into CPChecker properties of scheduling filtering algorithms [12] such as edge-finder, not-first not-last, time-table consistency, energy filtering, etc. for testing the most recent implementation of scheduling algorithms [13,14,15,16,17,18].

[4] https://github.com/vrombouts/Generic-checker-for-CP-Solver-s-constraints

References

1. Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.
2. OscaR Team. OscaR: Scala in OR, 2012. Available from `https://bitbucket.org/oscarlib/oscar`.
3. Krzysztof Kuchcinski, Radoslaw Szymanek and contributors. Jacop solver. Available from `https://osolpro.atlassian.net/wiki/spaces/JACOP/`.
4. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
5. Micha Meier. Debugging constraint programs. In *International Conference on Principles and Practice of Constraint Programming*, pages 204–221. Springer, 1995.
6. Peter J. Stuckey Carleton Coffrin, Siqi Liu and Guido Tack. Solution checking with minizinc. In *ModeRef2017, The Sixteenth International Workshop on Constraint Modelling and Reformulation*, 2017.
7. Nadjib Lazaar, Arnaud Gotlieb, and Yahia Lebbah. A cp framework for testing cp. *Constraints*, 17(2):123–147, 2012.
8. Frédéric Goualard and Frédéric Benhamou. *Debugging Constraint Programs by Store Inspection*, pages 273–297. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.
9. ScalaCheck. `http://scalacheck.org`.
10. Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. Minicp: A minimalist open-source solver to teach constraint programming. *Technical Report*, 2018.
11. AssertJ Library. `http://joel-costigliola.github.io/assertj/index.html`.
12. Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science, 201.
13. Steven Gay, Renaud Hartert, and Pierre Schaus. Time-table disjunctive reasoning for the cumulative constraint. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 157–172. Springer, 2015.
14. Cyrille Dejemeppe, Sascha Van Cauwelaert, and Pierre Schaus. The unary resource with transition times. In *International conference on principles and practice of constraint programming*, pages 89–104. Springer, 2015.
15. Hamed Fahimi and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2637–2643. AAAI Press, 2014.
16. Petr Vilím. Global constraints in scheduling. 2007.
17. Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 230–245. Springer, 2011.
18. Alexander Tesch. A nearly exact propagation algorithm for energetic reasoning in o (n2logn). In *International Conference on Principles and Practice of Constraint Programming*, pages 493–519. Springer, 2016.

Constraint-based Generation of Trajectories for Single-Arm Robots

Mathieu Collet[1,2,3], Arnaud Gotlieb[2] and Morten Mossige[1,3]

[1] University of Stavanger, 4036 Stavanger, Norway
[2] Simula Research Laboratory, Lysaker, Norway
[3] ABB Robotics, 4349 Bryne, Norway

Abstract. Developing reliable software for Single-Arm Robots, SAR, is a challenging task. A typical system involves complex motion control systems and anti-collision systems, which are difficult to specify and to implement. Moreover, developing convincing test scenarios, which place the system into extreme cases, is extremely complicated due to the variety and complexity of possible robots' configurations. To overcome these challenges, this paper introduces Robtest, a new method to test SAR. This method presents how such systems can be tested in a more systematic and automated way by proposing a ***constraint based*** method that is able to automatically generate test scenarios. The automation of this process bring some diverse problems. Building on top of existing continuous integration processes and an existing constraint-based frameworks used at ABB Robotics, the main challenge is to design constraint models for SAR that can be used to automatically generate test scenarios for testing ABB's SAR. The method is implemented using continuous constraint solver, which can resolve systems with non-linear constraints. This property is interesting in this domain since robot can perform other motions than a linear trajectory. The solver is used to define the space configuration and to obtain all the trajectories of the system. Based on these trajectories and considering some parameters, (e.g. Number of point, the cost of the test), some test scenarios are generated.

Keywords: constraint based trajectories, single-arm robot, test case generation and continuous solver

1 Introduction

Software testing is an important activity used during software development and after the release for quality assurance. A test case used during testing includes input parameters for stimulating the system under test and the expected behaviour in terms of outputs. Engineers design test case scripts and execute them to validate the system under test. The execution of the test produces a test result which is compared to the expected result as stated in the test cases. The result of executing a test suite is either a pass or a fail verdict.

In the software industry today, despite all labor-intensive techniques to reduce the number of software errors in SAR, errors may be detected late in the design process, often very close to release date. In addition, detecting complex errors such as those due

to the interaction between multi-arm movements requires to extend the testing phase up to a level where its costs prevail on its benefice.

Traditionally, human engineers write test script and an automatic process able to run test cases supports their execution. The creation of a test case is highly dependent on the robot configuration under test. The engineer writes a program in which some instructions allow the robot to move into its space configuration. To test different parts of the robot, the tester should write different test cases which is a time-consuming and error-prone process.

To overcome the writing part of the test case, we propose Robtest, a method to generate test cases automatically, using constraint solving over continuous domains. The generation is realized by a toolbox, which contains several components. The toolbox takes three parameters as input and one as output, the test case.

This paper aims at presenting an innovative methodology leveraging a constraints solver over continuous domain for automatically generating robot test scenarios. The approach represents a novelty since, the method uses constraints programming to the generation problem of robot trajectories tests. The method improves the test toolchain for robot testing and includes more diversity between test cases executed by a robot under test. The system is deployed to automatically generate test scenarios and test them on a single arm robot system.

The remainder of this paper is organized in four sections. The next part deals with the background of the approach, presenting the robot space configuration and the required notions for a 6-axis robot. The following section presents the contribution of the new method by presenting the modelling and the usage of constraints in our method. Then, the preliminary experimental setup is presented as well as the first evaluation of our method. This paper is concluded by a discussion on the results and future work.

2 Background

Testing is an important part of software development and lifetime of the product. As exhaustive testing is costly, due to the time of writing a test by a tester, there is a need for automatic generation methods. Current best validation practices used to reduce the number of software errors are to apply techniques such as code review, manual unit/integration and system testing. However, these techniques are labor-intensive, and despite applying such techniques, software errors may still be detected late in the design process, often very close to release date. This is problematic because, late error detection is much more costly than uncovering errors early in the design process. In addition, detecting complex errors such as those due to the interaction between arm movements requires to extend the testing phase up to a level where its costs prevail on its benefice.

2.1 Robot Space

The goal of the introduced method is to generate test trajectory scenarios for single arm robots. The trajectory, which is a motion performed by the robot under certain speed, orientation and others constraints parameters, should stay inside the space configuration of the robot itself. The robot configuration can be defined by the space in which operations are allowed.

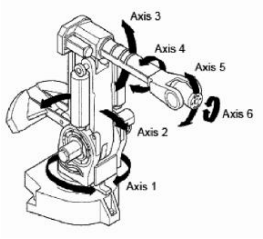


Fig. 1. DOF for a robot

A robot is a system with complex axis motion system. It can have several Degrees Of Freedom (DOF) regarding the number of axes that the robot contains. A robot with six different axis is called a 6DOF robot (see **Fig. 1**), while a 7DOF robot is a robot with seven axis. Concerning the space configuration of the robot, the resulting notion of DOF brings different dimensions spaces. Indeed, a 6DOF robot can move in a six dimensions space configuration. The three first dimensions are the Cartesians coordinates while the three last dimensions are responsible of the orientation inside the Cartesian space. For the sake of simplicity, the drawing inside the paper will be in 2D, but the proposed method tackles three dimensions space configuration.

In every robot configuration, different zones split the area into two distinct spaces. The first zone is the **forbidden zone** (FZ), which is a zone in which the robot cannot move due to diverse reasons. The first is a mechanical constraints of the robot itself. The second is due to some objects, human, etc. placed around the robot and where the robot's arm cannot move without provoking a dangerous and harmful collision. The remaining space around the robot forms the second zone named **working zone** (WZ). In this one, the robot can perform its tasks and freely move. A valid SAR test scenario is formed by points which defined a trajectory which go through WZ and never cross FZ. Generating such a test scenario is hard because of these constraints. To facilitate the handling, basic geometric shapes can be used (square, circle, etc.), **Fig. 2**.

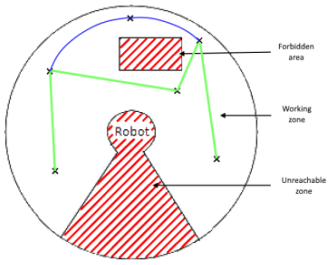


Fig. 2. Robot space configuration

2.2 Possible robot motions

Given an initial goal, a robot can move in many different ways. The three most common motions are presented here. The first one is the straight-line motion (SLM), called "moveL", in which the robot move from the origin point to the destination point linearly. The second motion is the one where the robot starts from an origin point and reaches the destination describing a circular motion (CM), going through a third point. This motion is named "moveC". The last motion is the "moveJ", where the robot move quickly from an origin point to a destination point. The characteristic of this motion is that it does not have to be in a straight line. This motion is a non-linear joint motion (JM), this means that all axis of the robot move in the same time to reach the destination point.

Since some motions are not linear (cf. CM and JM) and the robot's motions are continuous, the usage of a continuous constraint solver is justified as it can handle all these cases since the solver is working with nonlinear constraints.

2.3 Trajectory test case

To define a trajectory test case, at least two points must be defined, the starting point and the ending point. It is not possible to pass several time by the starting and ending points unlike the others. This is a constraint and a condition to satisfy, for a path, to be an acceptable trajectory. All these points are placed in the working zone. A trajectory is defined by nodes and edges which are added together:

$$Traj = N_{start} + \sum N, E + N_{end}$$

Where N are nodes and E are edges, N_{start} the starting point and N_{end} the ending point.

The trajectory formed by all the nodes and edges should respect the constraint of the space configuration, avoiding FZ.

3 Contribution

The goal of the proposed approach is to automatically generate trajectories having minimal costs. The notion of cost is explained in the following part. However, with our method, reaching a path with minimal cost is not the only purpose, we propose an "any cost" approach which generates all paths having a cost less than a given cost.

The notion of cost is one of the important part of the trajectory. Indeed, the cost is a function including different parameters, like the distance between nodes, the speed, the orientation of the tools mounted on the wrist of the robot, etc. The cost of an edge is defined by the following function:

$$C_{edg} = f(d_{edg}, s, o, ..)$$

Where d_{edg} is the distance between two nodes, s the speed on the edge and o the orientation.

The cost is calculated for each edges that satisfied the space configuration constraints. To obtain the cost of a trajectory, a sum of the individual cost set on each edge of the trajectory is applied:

$$C_{tr} = \sum C_{edg}$$

Another parameter of the trajectory is the number of time that a node can be visited. The simplest is to visit the nodes of the trajectory once. Nevertheless, the method allows to visit nodes more than once, therefore, allowing loops. This parameter is an input of the method, defined by the user.

To generate test trajectories, the user set three input parameters that are the maximal cost of the trajectory, C_{max}, the number of loop allowed on each point, N_{loop}, and the number of points in the initial cloud, N_{points}. This parameter does not include the starting point and ending point. From these parameters, the output of the method is a test case in which some motion instructions are defined.

To generate the trajectories test case, the method models the single-arm robot space configuration and applies some constraints technics in order to obtain a solution, a test case.

The modelling part is performed in a three-dimension space. To set up the problem, we consider N points placed in the working zone. **Fig. 3** shows an example with five points and the cost of three distinct trajectories. Once a trajectory is selected, the method checks if the cost and loop constraints are satisfied. **Fig. 4** shows an example of trajectory test case which is the output of the method.

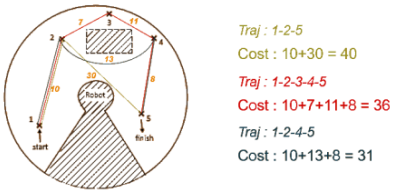


Fig. 3. Space configuration modelling

```
MODULE trajectories_test

    ! robtarget start here
    VAR robtarget p10 := [ [-300, -100, 300], ...
    VAR robtarget p20 := [ [-200, 200, 300], ...
    VAR robtarget p30 := [ [50, 300, 300], ...
    VAR robtarget p40 := [ [300, 200, 300], ...
    VAR robtarget p50 := [ [200, -75, 300], ...

    PROC main()
            ConfL \Off;

            !Enter move-instructions here
            MoveJ p10,v1000,fine,tool0;
            MoveL p20,v500,fine,tool0;
            MoveC p40,v500,fine,tool0;
            MoveL p50,v500,fine,tool0;

    ENDPROC
ENDMODULE
```

Fig. 4. Trajectory test case

To obtain a trajectory that meets all constraints defined by the user and the space configuration, the method proceeds into two steps. The first step is to define all edges allowed inside the space configuration and the associated cost on it. This part is done by the continuous constraint solver. At this stage, the method creates the model of the robot and runs the solver in order to define the possible link between nodes.

Once the first step is done, by using a systematic exploratory approach, our method generates trajectory test case. The cost of the generated trajectory should satisfy the input constraint defined by the user (C_{max}). To perform this, the approach is to check the cost of the trajectory each time an edge is added.

In the same time, since loops on nodes are allowed, the method checks before choosing a node if this one is already visited more than the input parameter defined by the user, N_{loop}. Based on the same code than the cost function, the checking is performed on each node of the trajectory.

4 Experimental setup and evaluation

This part presents the implementation of our method which includes two distinct components. The first is the **Continuous Constraint Based Trajectories Generation**, named CCB Generation and the second is the **Trajectories Generation**, called RT Generation. Both are executed for the automatic generation of trajectories test case.

On one hand, the CCB Generation component computes a graph where the nodes represent points inside the space configuration. On the other hand, the RT Generation component generates the trajectories in order to run a test.

4.1 CCB Generation component

This component aims at computing the cost of all edges inside the robot space configuration. It also defines *the accessibility matrix*, which is a table in which the cost of all edges is computed.

To implement this first part of the method, we use a continuous domains solver, which is a part of constraint programming. In such a solver, variables take their values in a continuous domain with floating-point bounds. As the geometric shapes of the distinct zones and various individual moves of the robot are composed of non-linear inequations, we argue that a constraint solving over continuous domains is the appropriate tool to address the problem. For the implementation of our approach, we use RealPaver *[1]*, which is one of the several existing solvers on continuous domains, such as IBEX *[2]*, NUMERICA *[3] or*, INTERLOG *[4]* just to name a few. We chose this solver for the experimentation because of its correctness to provide under- and over- approximation of the solution set and, its availability and simplicity of usage. RealPaver is used to check whether two points are reachable inside the robot configuration space and also to create the accessibility matrix.

Based on the configuration space and considering two points, the continuous constraint solver is able to know if these points are reachable. Constraint solving over continuous domains works by successively pruning the domains of each variable by using each constraint as a filter. By decomposing constraints in individual projectors and iterating the application of the projectors over the domains using a specific filtering consistency, the approach is able to reach a fixpoint where no more pruning can be performed. The termination of the algorithm is guaranteed by the usage of abstract numerical values which are computed over floating-point values. As the number of floating-point values is finite (if standard fixed-size representation is used over 32, 64 or 128 bits), and as only domain shrinking is possible, the algorithm necessarily terminates and reached a fixpoint.

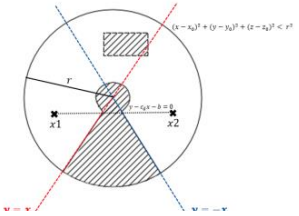


Fig. 5. Using of RealPaver

We model the reachability problem by checking the non-existence of intersection between the line passing by the two points and the various forbidden areas. If the solver proves the absence of solutions (i.e., unsatifiability), then it means that the points are reachable, as there is no intersection between the move and the forbidden zones. If the solver returns an under-approximation of the solution set (with a guarantee of the existence of at least one solution *[1]*), it means that there exists an intersection between the line and a forbidden zone, and thus the points are not reachable. **Fig. 5** shows the robot configuration space on an example.

Extending this computation to other moves than moveL (direct line between two points) is possible but requires more in-depth examination. In fact, extending the model to moveC (circle arc between three points) requires to solve a variation problem when only two points are initially given. We need to generalize the previous principle for all possible triples where only two points are known. This is interesting but outside the scope of this paper.

To compute the accessibility matrix, we run the solver on each pair of points of the space configuration. For each pair of points, a cost is computed based on some input parameters. For the sake of simplicity, in this paper, only the Euclidean distance (i.e.,

d_{ij}) between the points is integrated into the cost function, but, more advanced parameters can easily be inserted into this cost function. The matrix can be defined by the following rules:

$$[i][j] = \begin{cases} if\ E_{ij}\ exists, & C_{ij} = d_{ij} \\ otherwise\ , & C_{ij} = \infty \end{cases}$$

With i and j the two points under test, E_{ij} the edge formed by the nodes I and j, C_{ij} the cost and d_{ij} the distance between nodes i and j.

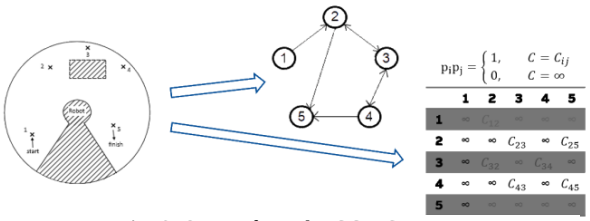


Fig. 6. Output from the CCB Generation

For a simpler understanding, the following presentation only deals with straight line motion. An example is given with N = 3.

Once CCB Generation has calculated all the possibilities, the second component uses the matrix of cost and the graph to generate trajectories.

4.2 RT Generation component

The RT Generation component is responsible of the trajectories test case generation. This component is based on the Depth First Search (DFS) algorithm with some variations to take into account the two above-mentioned parameters, C_{max} and N_{loop} and the ending point. The search is executed on the generated graph and the matrix from the CCB Generation. Using DFS algorithm, all path with a cost lower than the input parameter C_{max} is selected. All this paths create the test scenario. The search is going from node to node and at each iteration, the total cost of the trajectory is calculated.

Once all paths are found, the test case is generated with all trajectories found by RT Generation. An example is to execute this method on an IRB1200 ABB robot, which is a 6DOF single-arm robot. The test case resulting of our method, for this specific robot, is a RAPID program describing motion instruction. This language is specific to this robot, but it is based on a template and can easily be converted to another language given the template.

5 Evaluation

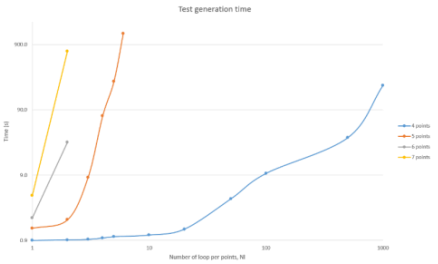
Our first approach of the method was done in two-dimension space to prove the concept but the aim is to deal with six-dimension solution for industrial robots. The integration to a three-dimension space is relatively simple, as we change the equation inside the solver. However, to handle a six-dimension, it can be interesting to use a different space configuration, like the joint space.

The proposed method has some limits that we discuss here. The first point, is the scalability of the method. The more points or loops are added, the more solutions are obtained with a fixed cost. This increases the time generation and all the generation process. To improve this, a solution is to take one trajectory only and generate it for a specific test scenario.

Based on the experimental setup, in the worst cost case ($C_{max} = \infty$), the following graph illustrates the evolution of the time execution in function of the input parameters, N_{loop} and N_{points}:

Computer configuration:
- Core i-7, 3,6 GHz
- 64 GB RAM

Parameter configuration:
- C_{max} = 100000



The goal of the following work is to find a trade-off between the three input parameters in order to integrate this process in a continuous integration.

Another limitation of the methodology, is the problem of singularity points. A singularity is a specific configuration of the robot in which there are several solutions to reach the next point. Let us take an example with a robot's wrist. In the figure presented here, if the robot wants to turn the tool mounted on the wrist, there are two solutions, the rotation "J4" or the rotation "J6". In this case more than one solution is possible, the robot is not able to choose one of them and the singularity error is raised.

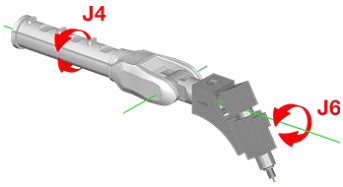


Fig. 7. Wrist singularity

With our approach, it is impossible to detect the error before the simulation of the test.

One option is to change the orientation of the tool. This solution will be implemented when the method will operate on a six-dimension space. Another solution consists to use the joint space *[5]*, *[6]* and calculates all axis to know if an axis will raise a singularity.

The presented methodology deals with single-arm robot, but the goal is to extend the approach in order to tackle configuration with multi robot system software, like a multi-arm robot or a multi robot cell. This approach will be studied in the future.

6 Conclusion

The paper contributes to the definition of a new methodology to automatically generate a robot's trajectories test scenario. The paper shows experimentally how to implement the method and how it can be used. The method has three distinct inputs and a trajectories test case as an output. With the methodology that we propose, some intelligence is added to the current process of test case generation and more diversity is included.

In the next step, the experimentation will be continued and compared with other solutions and the using of other solvers will be done. Moreover, some improvements should be added to the cost calculation. At this step, only the distance is considered, but the implementation should include all parameters described inside the contribution part.

References

1. L. Granvilliers and F. Benhamou. Algorithm 852: Realpaver: an Interval Solver using Constraint Satisfaction Techniques. ACM TOMS, 32(1):138-156, 2006.
2. Gilles Chabert. IBEX. Ibex (Interval-Based EXplorer) is a C++ library for solving nonlinear constraints over real numbe.. 2007, http://www.ibex-lib.org/.
3. P. Van Hentenryck et al, A gentle introduction to NUMERICA, Artificial Intelligence 103 (1–2) (1998) 209–235.
4. O. Lhomme. Consistency techniques for numeric CSPs. In R. Bajcsy, editor, Proceedings of the 13th IJCAI, (1993). pages 232–238.
5. Mike Stilman. Global Manipulation Planning in Robot Joint Space With Task Constraints. (2010)
6. Faria et al. Position-based kinematics for 7-dof serial manipulators with global configuration control, joint limit and singularity avoidance. Mechanism and Machine Theory, 121:317–334, 2018.

A Probabilistic Model to Count Solutions on the `alldifferent` Constraint

Giovanni Lo Bianco[1], Charlotte Truchet[2], Xavier Lorca[3], and Vlady Ravelomanana[4]

[1] IMT Atlantique, Nantes, France
giovanni.lo-bianco@imt-atlantique.fr
[2] University of Nantes, Nantes, France
[3] IMT Mines Albi-Carmaux, Albi, France
[4] University of Paris VII, Paris, France

Abstract. This article presents two probabilistic models for the `alldifferent` constraint. We are interested in studying the combinatorial structure of the `alldifferent` constraint. More specifically, we are interested in the existence of solutions for an instance of `alldifferent` and to count them. Such informations can be used in search heuristics. From this probabilistic approach, we show how to compute some estimators of the number of solutions and we use them in a search heuristics inspired by Counting-Based Search heuristics.

1 Introduction

In this article, we are interested in studying the combinatorial structure of some problems in order to be more efficient when exploring the solution space and when propagating the constraints. More specifically, we are interested in cardinality constraints. Cardinality constraints concerns the number of occurrences of values in a solution. We propose to study the `alldifferent` constraint [9], which constrains every value to appear at most once in a solution. The models that we present aim to be adapted for other cardinality constraints, as they have similar structure.

These probabilistic models will allow us to estimate the number of solutions for an instance of `alldifferent`. Solutions counting, in the constraint programming field, have already been studied by Pesant et al. [7]. They presented a heuristic, called Counting-Based Search, which suggest to explore first the area where there are likely more solutions. Counting solutions is at least as hard as the problem is. Pesant et al. therefore proposed an upper bound, less costly to compute, of the number of solutions for an instance of `alldifferent` and used it as an heuristic to explore the solution space.

These models are similar to the one proposed in [1]. It also presents a probabilistic model for `alldifferent`: the authors randomize the domain of the variables and estimate the probability that an instance of `alldifferent` is bound-consistent (BC) and show two asymptotic schemes depending on the ratio number of variables / number of values. The main difference with our model is that

Boisberranger et al. propose to randomize the domain of the variables, restricting those domains to be intervals. Also, they are interested in the probability that an instance of `alldifferent` is bound-consistent and show that it is not always necessary to apply the filtering algorithm, which is costly. In our study, we only focus on a probabilistic way to estimate the number of solutions. We give two new estimators to count the number of solutions for an instance of `alldifferent`. We compare the quality of these estimator to the bound proposed in [7]. Then, we compare the efficiency of those estimators within Counting-Based Search on small problems.

2 Preliminaries

Let $X = \{x_1, \ldots, x_n\}$, the set of variables. For each variable $x_i \in X$, we note D_i its domain and $Y = \bigcup_{i=1}^{n} D_i = \{y_1, \ldots, y_m\}$, the union of the domains. We now define formally the constraint `alldifferent` and the value graph of X.

2.1 The `alldifferent` constraint

Definition 1 (`alldifferent` [9]). *A constraint* $\texttt{alldifferent}(X)$ *is satisfied iff each variable* $x_i \in X$ *is instantiated to a value of its domain* D_i *and each value* $y_j \in Y$ *is chosen at most once. We define formally the set of allowed tuples:*

$$\mathcal{S}_X = \{(d_1, \ldots, d_n) \in D_1 \times \ldots \times D_n \mid \forall i, j \in \{1, ..., n\}, \\ i \neq j \Rightarrow d_i \neq d_j\}$$

Definition 2 (Value Graph). *Let* $G_X = G(X \cup Y, E)$*, the value graph of* X *with* $E = \{(x_i, y_j) \mid y_j \in D_i\}$*.* G_X *is a bipartite graph representing the domain of each variable. There is an edge between* x_i *and* y_j *iff* $y_j \in D_i$

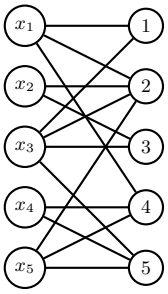
Example 1. Let $X = \{x_1, x_2, x_3, x_4, x_5\}$ with $D_1 = \{1, 2, 4\}$, $D_2 = \{2, 3\}$, $D_3 = \{1, 2, 3, 5\}$, $D_4 = \{4, 5\}$ et $D_5 = \{2, 4, 5\}$. We obtain the value graph G_X such as on Figure 1a. The tuples $\{1, 3, 5, 4, 2\}$ and $\{1, 2, 3, 5, 4\}$ are two solutions of $\texttt{alldifferent}(X)$.

2.2 Existence and solution counting

One property of the value graph, that is of great interest, is that a matching covering the variables corresponds to a solution of $\texttt{alldifferent}(X)$ [4].

Proposition 1. *The set of solutions of* $\texttt{alldifferent}(X)$ *is in bijection with the set of matchings of* G_X *covering* X*.*

To count the solutions of $\texttt{alldifferent}(X)$, we need to count the number of matchings covering X in G_X. We now introduce the biadjacency matrix and the permanent. The following definitions can be found in [6].



(a) Value Graph G_X of the Example 1

$$\mathcal{B}(G_X) = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

(b) The biadjacency matrix of the value graph

Fig. 1. Value graph and biadjacency matrix of Example 1.

Definition 3 (Biadjacency matrix). *Let $G(U \cup V, E)$, a bipartite graph. We define $\mathcal{B}(G) = (b_{ij})$, its biadjacency matrix, such as $\forall u_i \in U, \forall v_j \in V$, $b_{ij} = 1$, if $(u_i, v_j) \in E$ and $b_{ij} = 0$, otherwise.*

Example 2. Figure 1b represents the biadjacency matrix of the value graph of Example 1.

Definition 4 (Permanent). *Let $A = (a_{ij}) \in \mathbb{M}_{n,n}$, a square matrix and $\mathfrak{S}_n$, the group of permutations over $\{1, \ldots, n\}$. The permanent of A is defined as:*

$$Perm(A) = \sum_{\sigma \in \mathfrak{S}_n} \prod_{i=1}^{n} a_{i\sigma(i)} \tag{1}$$

A bipartite graph is called balanced if the two parts are the same size and a matching is called perfect if it covers every node of the graph. We can find a perfect matching in a bipartite graph only if it is balanced. Proposition 2 gives a way to count perfect matchings in such bipartite graphs.

Proposition 2. *Let G a balanced bipartite graph and $\mathcal{B}(G)$ its biadjacency matrix. We note $\#PM(G)$, the number of perfect matchings in G, then:*

$$\#PM(G) = Perm(\mathcal{B}(G)) \tag{2}$$

In our case, there can be less variables than values, then the value graph is not balanced. If the value graph G_X is unbalanced then $\#PM(G_X)$ is the number of matchings covering X. Pesant et al. [7] explains that we can compute the number of matchings covering X anyway by adding fake variables, that can be instantiated to every value, to balance the value graph. We call G'_X the modified value graph. They showed that:

$$\#PM(G_X) = \frac{\#PM(G'_X)}{(m-n)!}$$

With Proposition 1, we can conclude on Corollary 1:

Corollary 1. *The number of allowed tuples of* `alldifferent`(X) *is*

$$|S_X| = \frac{Perm(\mathcal{B}(G'_X))}{(m-n)!} \tag{3}$$

Example 3. The instance `alldifferent`(X) of Example 1 has

$$Perm\left(\mathcal{B}(G_X)\right) = 8 \text{ solutions.}$$

2.3 Upper bound on the number of solutions

The permanent is very costly to compute as it considers a sum over a group of permutations, which is of exponential size [10]. Therefore, Pesant et al. [7] propose to use an upper bound to estimate the number of solutions. They use the Brégman-Minc [2] upper bound and the Liang-Bai [5] upper bound:

Proposition 3 (Brégman-Minc Upper Bound). *We note d_i, the size of the domain D_i, then:*

$$|S_X| \leq UB^{BM}(X) = \prod (d_i!)^{\frac{1}{d_i}} \tag{4}$$

Proposition 4 (Liang-Bai Upper Bound). *We note d_i, the size of the domain D_i and $q_i = min(\lceil \frac{d_i+1}{2} \rceil, \lceil \frac{i}{2} \rceil)$, then:*

$$|S_X| \leq UB^{LB}(X) = \prod_{i=1}^{n} \sqrt{q_i(d_i - q_i + 1)} \tag{5}$$

As neither of these two bounds dominate the other, Pesant et al. choose to upper bound the number of solution this way:

$$|S_X| \leq UB^{PZQ}(X) = min(UB^{BM}(X), UB^{LB}(X)) \tag{6}$$

In next section, we present two new estimators of the number of solutions, based on a probabilistic approach. These estimators will be integrated in Counting Based Search heuristics in order to compare their efficiency with the bound proposed by Pesant et al.

3 A probabilistic approach for `alldifferent`

We present two probabilistic models for the `alldifferent` constraint. In both models, we propose to randomize the domain of the variables and then to study the expected number of solutions.

3.1 The Erdős-Renyi Model

Erdős and Renyi [3] introduced random graphs and studied the existence and the number of perfect matchings on these random structures. Applied directly to our problem, the idea is to randomize the domain of each variable such that : for all $x_i \in X$ and for all $y_j \in Y$, the event $\{y_j \in D_i\}$ happens with a predefined probability $p \in [0, 1]$ and all such events are **independent**:

$$\mathbb{P}(\{y_j \in D_i\}) = p \in [0, 1] \tag{7}$$

Proposition 5. *According to the Erdős-Renyi Model, the number of allowed tuples of* `alldifferent`*(X) is expected to be:*

$$\mathbb{E}^{ER}(|\mathcal{S}_X|) = \frac{m! \cdot p^n}{(m-n)!} \tag{8}$$

Proof. To simplify notation, we note $B = \mathcal{B}(G_X)$, the biadjacency matrix of the random value graph and, similarly, $B' = \mathcal{B}(G'_X)$. The matrix B can be seen as a random matrix, for which each element is a random 0-1 variable B_{ij} such that $P(\{B_{ij} = 1\}) = p$. According to Corollary 1, we have

$$\begin{aligned}\mathbb{E}(|S_X|) &= \mathbb{E}\left(\frac{Perm(B')}{(m-n)!}\right) \\ &= \mathbb{E}\left(\frac{1}{(m-n)!}\sum_{\sigma\in\mathfrak{S}_m}\prod_{i=1}^{m} B'_{i\sigma(i)}\right) = \mathbb{E}\left(\frac{1}{(m-n)!}\sum_{\sigma\in\mathfrak{S}_m}\prod_{i=1}^{n} B_{i\sigma(i)}\right)\end{aligned}$$

because $\forall i > n, B'_{ij} = 1$ and $\forall i \leq n, B'_{ij} = B_{ij}$. The operator expectancy is linear and $\forall \sigma \in \mathfrak{S}_m, \forall i \in \{1, \ldots, n\}$ the random variables $B_{i\sigma(i)}$ are independent, then

$$\begin{aligned}\mathbb{E}(|S_X|) &= \frac{1}{(m-n)!}\sum_{\sigma\in\mathfrak{S}_m}\prod_{i=1}^{n}\mathbb{E}\left(B_{i\sigma(i)}\right) \\ &= \frac{1}{(m-n)!}\sum_{\sigma\in\mathfrak{S}_m}\prod_{i=1}^{n} p = \frac{m! \cdot p^n}{(m-n)!}\end{aligned}$$

□

When $n = m$, the expectancy of the number of solution is $n! \cdot p^n$. This result have been shown in [3]. We extended it here to unbalanced bipartite graph.

This model is quite simple as it considers only one parameter to modelize an instance of `alldiferent`. We have an estimator of the number of allowed tuples only based on the edge density in the graph. In Example 1, we have $n = m = 5$ and there are 14 edges, then the edge density is $\frac{14}{25} = 0,56$. We can expect $5! \cdot 0,56^5 = 6,61$ solutions (there are actually 8 solutions).

The next model considers the size of each domain, we expect the resulting estimator to be sharper.

3.2 Random instances with fixed domains size (FDS model)

Let $\{d_1, \ldots, d_n\}$, be the predefined size of each domain. In this model, we pick randomly uniformly an instance of `alldifferent` among those which, for each variable $x_i \in X$, the corresponding domain size is $|D_i| = d_i$. We define $\mathcal{E}_{(d_1,\ldots,d_n)}$, the set of such instances:

$$\mathcal{E}_{(d_1,\ldots,d_n)} = \{\texttt{alldifferent}(X) \mid \forall x_i \in X, |D_i| = d_i\}$$

If we pick randomly uniformly an instance from $\mathcal{E}_{(d_1,\ldots,d_n)}$, then we can evaluate the probability that a value y_j is in the domain D_i:

$$\begin{aligned}\mathbb{P}(\{y_j \in D_i\}) &= \frac{\#\text{number of configurations including } y_j}{\#\text{number of possible configurations}} \\ &= \frac{\binom{m-1}{d_i-1}}{\binom{m}{d_i}} = \frac{d_i}{m}\end{aligned}$$

It is important to notice that for two different values y_{j_1} and y_{j_2} and for a same domain D_i, the events $\{y_{j_1} \in D_i\}$ and $\{y_{j_2} \in D_i\}$ are not independent, unlike the Erdős-Renyi model. However, for two different domains D_{i_1} and D_{i_2} and for any pair of value y_{j_1} and y_{j_2} (possibly the same), the events $\{y_{j_1} \in D_{i_1}\}$ and $\{y_{j_2} \in D_{i_2}\}$ are independent.

Proposition 6. *According to the FDS Model, the number of allowed tuples of* `alldifferent`*(X) is expected to be:*

$$\mathbb{E}^{FDS}(|\mathcal{S}_X|) = \frac{m!}{(m-n)! \cdot m^n} \cdot \prod_{i=1}^{n} d_i \tag{9}$$

Proof. The proof is very similar to the proof of Proposition 5. □

From Example 1, we have $n = m = 5$ and the domain size are: $d_1 = 3, d_2 = 2, d_3 = 4, d_4 = 2$ and $d_5 = 3$. We can expect $\frac{5!}{5^5} \cdot 3 \cdot 2 \cdot 4 \cdot 2 \cdot 3 = 5,53$ solutions. On this example the expectancy with the Erdős-Renyi model is more accurate.

4 Experimental analysis

We first present a qualitative analysis of the different estimators. Then, we adapt the Counting-Based search strategy, presented in [7], so that it is guided by the estimators presented in Section 3 and not only the upper bound from Section 2.

4.1 Qualitative analysis on `alldifferent` instances

For this qualitative analysis, we have generated randomly and uniformly 10000 instances of `alldifferent`, which present at least one solution, with $n = 10$ variables and $m = 10$ values. We choose randomly a parameter p and we generate an instance according to the Erdős-Renyi model. For each of those instances, we have computed the three estimators of the number of solutions and we compare them to the real number of solutions. Figure 2 shows the percentage of instances for different ranges of relative gap between the value of the estimators and the real number of solutions. The relative gap is computed this way : $\Delta = \frac{|est-real|}{real}$ where est is the value of the estimator and $real$ is the value of the true number of solutions.

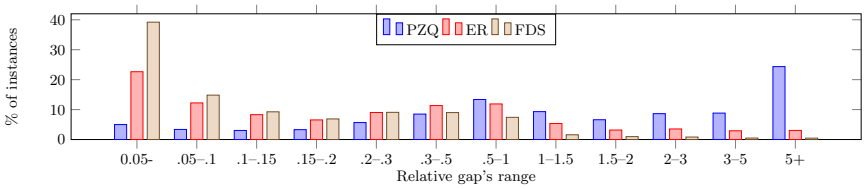


Fig. 2. Percentage of instances per relative gap for each estimator

We notice that the third estimator, the expectancy according to the FDS model, seems more accurate as, for about 40% of instances, the relative gap is less than or equal to 0.05 and, for less than 10% of instances, the relative gap is superior to 1. The expectancy according to the Erdős-Renyi model is less accurate. As for the upper bound UB^{PZQ}, it is very far from being accurate.

These results can be explained by the fact that the two last estimators correspond to the expected number of solutions according to Erdős-Renyi model and FDS model, whereas the first estimator is an upper bound. The FDS estimator is more accurate than the ER estimator, as it considers the distribution of domains size and not only the density of edges. For search strategies, the correlation between the estimator and the real number of solutions is more important than the quality of the estimator. In next subsection, we compare the efficiency of Counting-Based search for the three estimators.

4.2 Estimators' efficiency within Counting-Based Search

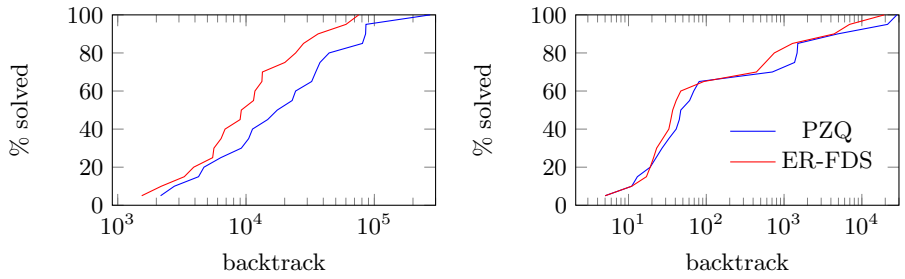
We have adapted the $maxSD$ heuristic presented in [7], originally designed with the PZQ estimator, such that the solution densities are computed from the ER estimator and the FDS estimator. We first wanted to compare the efficiency of the three estimators by running the three version of $maxSD$ on two different problems: the Quasigroup Completion problem with holes and the Magic Square problem. Both can be expressed with one or several `alldifferent` constraints. Some known hard instances of those problems have been generated by Pesant et

al. Unfortunately, our implementation of the three versions of $maxSD$ is still a bit naive and it takes several hours to run one of those instances. Therefore, we decided to generate randomly easier instances.

Concerning the Quasigroup Completion problem, we did not manage to generate easier non-trivial instances. The generated instances are solved (or prove to be unsatisfying) during the first or second propagation stage, which is not interesting when comparing search strategies.

As for the Magic Square problem, we randomize the instances this way: given n, the dimension of the problem and c the number of filled cases in the square at the beginning, we choose randomly uniformly c cases in the square, that we fill randomly uniformly among the number of possible arrangements $A_c^{n^2}$.

To solve the generated instances, we have implemented each version of $maxSD$ in the solver Choco v4.0.6 [8]. Figure 3 shows the evolution of the number of solved instances with the number of backtracks. For these plots, we have randomly generated 20 instances of Magic Squares with $n = 5$ and $c = 5$ and 20 instances for $n = 5$ and $c = 10$.



(a) n=5 and c=5 (b) n=5 and c=10

Fig. 3. % solved instances per number of backtracks for different parameters

We noticed that $maxSD$ with the ER estimator and the FDS estimator surprisingly behave in a very similar way. For this reason we plot one curve for those two estimators in both charts. We do not have any explanation for this phenomenon yet. Also, it appears like, for these generated instances, $maxSD$ with the two expectancy estimators performs better, especially for the hardest instances (Figure 3a).

5 Conclusion

In this paper, we have presented two probabilistic models for `alldifferent` and two estimators of the number of solutions. We have adapted the Counting-Based search strategy for those new estimators. We still need to work on their implementation so we can run them on bigger and harder instances. Yet, the results so far are encouraging. We also think to adapt these probabilistic models to other cardinality constraints.

References

1. Boisberranger, J.D., Gardy, D., Lorca, X., Truchet, C.: When is it worthwhile to propagate a constraint? A probabilistic analysis of alldifferent (2013)
2. Bregman, L.M.: Some properties of nonnegative matrices and their permanents. Soviet Math. Dokl (1973)
3. Erdos, P., Renyi, A.: On random matrices. Publication of the Mathematical Institute of the Hungarian Academy of Science (1963)
4. van Hoeve, W.J.: The alldifferent constraint: A survey. CoRR **cs.PL/0105015** (2001)
5. Liang, H., Bai, F.: An upper bound for the permanent of (0,1- matrices). Linear Algebra and its Applications (2004)
6. Lovasz, L., Plummer, M.D.: Matching Theory. American Mathematical Society (2009)
7. Pesant, G., Quimper, C., Zanarini, A.: Counting-based search: Branching heuristics for constraint satisfaction problems. J. Artif. Intell. Res. **43**, 173–210 (2012). https://doi.org/10.1613/jair.3463, https://doi.org/10.1613/jair.3463
8. 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
9. Régin, J.: A filtering algorithm for constraints of difference in csps. pp. 362–367 (1994)
10. Valiant, L.G.: The complexity of computing the permanent. Theor. Comput. Sci. **8**, 189–201 (1979)

Three New Approaches for the Maximum Common Edge Subgraph Problem*

James Trimble[1], Ciaran McCreesh[1], and Patrick Prosser[1]

University of Glasgow, Glasgow, Scotland
j.trimble.1@research.gla.ac.uk

Abstract. In the maximum common edge subgraph problem, the objective is to find a graph with as many edges as possible that is simultaneously a non-induced subgraph of each of two input graphs. We report our results from adapting a recent algorithm for maximum common *induced* subgraph, using a well-known reduction using line graphs. We also introduce three new direct constraint program encodings for the problem, and describe a new dedicated solver.

Keywords: Maximum common subgraph · Line graph · Constraint programming

1 Introduction

This paper considers the problem of finding a subgraph that appears in each of two input graphs and has as many edges as possible. This problem has numerous applications in biology and chemistry [12], and has also been applied in computer science to a problem of mapping tasks to processors [3]. All of the graphs we consider are undirected, unlabelled, and without loops.

Given two input graphs P and T (for "pattern" and "target"), the *maximum common edge subgraph problem (MCES)* is to find a graph with as many edges as possible that is isomorphic to a subgraph of P and to a subgraph of T simultaneously. These subgraphs are not required to be induced; that is, there can be edges present in either input graph that are not present in the subgraph.

MCES is NP-hard by a simple reduction from Hamiltonian Cycle: the graph in the Hamiltonian Cycle instance becomes the target graph in the MCES instance, and a cycle graph with the same number of vertices becomes the pattern graph. However, a variant of MCES in which the found subgraph is required to be connected can be solved in polynomial time on outerplanar graphs of bounded degree [1].

The *maximum common induced subgraph (MCIS)* problem—which we will take advantage of to solve MCES—is defined similarly, but the objective is to find a subgraph with as many vertices as possible, and the subgraphs must be induced. The two graphs on the left of fig. 1 show an example of an MCES instance; the two graphs on the right show an MCIS instance. In each case, an optimal solution is highlighted.

* Supported by EPSRC.

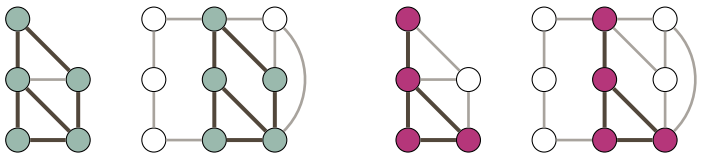


Fig. 1. The two graphs on the left are the pattern and target of a MCES instance, with an optimal solution (6 edges) highlighted. The two graphs on the right are the pattern and target of an MCIS instance, with an optimal solution (4 vertices) highlighted.

In this paper, we give preliminary results comparing three new approaches to solving MCES. The first of these approaches is to use constraint program encodings with an off-the-shelf CP solver. The second uses a modified version of a recent MCIS solver, McSplit, on the line graphs of the two input graphs. The third solves MCES directly, using a data structure and a propagation algorithm similar to those in McSplit. We find that the dedicated algorithms are orders of magnitude faster than our CP models.

Section 2 outlines related prior work. Section 3 describes our three new methods for solving the problem. Section 4 presents experimental results. Section 5 concludes and gives directions for further work.

2 Related work

McGregor [9] gives an early forward-checking algorithm in which decisions are made by mapping a vertex in P to a vertex in T. A matrix—which has a row for each edge in P and a column for each edge in T—keeps track of the remaining feasible edge assignments. The upper bound used is simply the number of rows in the matrix containing at least one non-zero value.

The *line graph* of a graph $G = (V, E)$ is a graph with a vertex for each element of E, such that two vertices are adjacent if and only if their corresponding edges in G share an endpoint. By a result due to Whitney [14], we can find the maximum common edge subgraph of two graphs by searching for a maximum common *induced* subgraph on their line graphs. Several algorithms for MCES, such as RASCAL [11], use this method. This approach has a single pitfall: the triangle graph K_3 and the claw graph $K_{1,3}$ (fig. 2) both have K_3 as their line graph, and it is possible that identical induced subgraphs of the line graphs correspond to subgraphs of the original graph with a triangle-graph connected component of one graph replaced by a claw-graph connected component of the other. Due to the shapes of the graphs, this is known as a ΔY exchange. It is straightforward to check for this occurrence during search, and to backtrack when it is detected. RASCAL does this by comparing the degree sequences of the subgraphs of the original graphs.

Marenco [7] and Bahense et al. [2] formulate MCES as an integer program, and solve it using techniques including branch-and-cut.

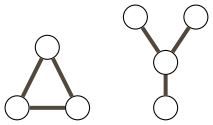


Fig. 2. The graphs K_3 and $K_{1,3}$.

3 Our Methods

In this section, we describe the methods we have used to solve maximum common edge subgraph: a set of three CP models implemented in MiniZinc, and two dedicated solvers inspired by CP techniques.

3.1 CP Models

We have implemented three constraint program models for MCES in the MiniZinc language. In each case, we assume without loss of generality that the pattern graph has no more vertices than the target graph. The first (and simplest) model (fig. 3) has one variable for each vertex in the pattern graph, and one value for each vertex in the target graph. An alldifferent constraint over the set of variables ensures that no target vertex is used twice. The objective function counts the number of adjacent pairs of pattern vertices that are mapped to adjacent target vertices.

```
int: np;                % order of pattern graph
int: nt;                % order of target graph

set of int: VP = 1..np;  % pattern vertices
set of int: VT = 1..nt;  % target vertices

array[VP, VP] of int: P; % adjacency matrix of pattern graph
array[VT, VT] of int: T; % adjacency matrix of target graph

array[VP] of var VT: m;  % pattern vertex -> target vertex mappings

var 1..np*(np-1): objval;

constraint objval = sum (v in VP, w in VP where w > v /\ P[v, w]==1)
        (T[m[v], m[w]]);

constraint alldifferent(m);

solve :: int_search(m, first_fail, indomain_split, complete)
        maximize objval;
```

Fig. 3. The first MiniZinc model

A second MiniZinc model could construct a mapping from edges to edges. However, doing so would require a large number of disjunctive constraints to

enforce incidence. We solve this problem by instead mapping pattern edges to *oriented* target edges, having two values per target edge in each variable. We also have an additional value $\perp$ signifying that the pattern edge is not used. Constraints ensure that if two pattern edges are assigned to target edges, then they have a shared endpoint if and only if their assigned target edges share the corresponding endpoint. An alldifferent-except-$\perp$ constraint ensures that each edge value is used only once. Finally, a set of constraints ensures that a single target edge may not be used in both orientations. The objective value is the count of variables that take non-$\perp$ values. This model is intricate, and we do not list it here for space reasons.

Our third model (fig. 4) includes the vertex variables and alldifferent constraint of model 1, and the edge variables and objective function of model 2. A set of constraints ensures that an edge variable takes a particular edge value if and only if the vertex variables corresponding to the pattern edge's endpoints take the values corresponding to the target edge's endpoints.

```
int: np;                % order of pattern graph
int: nt;                % order of target graph

int: mp;                % size of pattern graph
int: mt;                % size of target graph * 2
                        %   (i.e. number of oriented edges)

set of int: VP = 1..np;  % pattern vertices
set of int: VT = 1..nt;  % target vertices

array[VP, VP] of int: P; % adjacency matrix of pattern graph
array[VT, VT] of int: T; % adjacency matrix of target graph

array[1..mp, 1..2] of int: PE; % edge list of pattern graph
array[1..mt, 1..2] of int: TE; % oriented edge list of target graph

array[VP] of var VT: m;  % pattern vertex -> target vertex mappings
array[1..mp] of var 0..mt: m_edge;  % pattern edge index -> target
                                    %   edge index mappings. 0 means _|_

var 1..mp: objval;

constraint forall (i in 1..mp, j in 1..mt)
        (m_edge[i]==j <-> (m[PE[i,1]]==TE[j,1] /\ m[PE[i,2]]==TE[j,2]));
constraint objval = sum (a in m_edge) (a != 0);
constraint alldifferent(m);

solve :: int_search(m_edge, first_fail, indomain_split, complete)
        maximize objval;
```

Fig. 4. The third MiniZinc model

3.2 McSplit

Our first dedicated solver is based on McSplit, a recent algorithm for the MCIS problem [8]. McSplit is a forward-checking algorithm, with a variable for each vertex in the pattern graph and a value for each vertex in the target graph, along with a dummy value $\perp$ representing that the pattern vertex is not used. During search, any two domains are either disjoint (if we ignore $\perp$) or identical; this special structure of the problem allows domains to be stored in a compact data structure in which variables with identical domains share a single representation of their domain in memory. McSplit implements the soft all-different bound [10] in linear time by exploiting this special structure to avoid having to perform a matching, and uses variable-ordering heuristics inspired by the dual viewpoint [5].

We do not call McSplit directly on the pattern and target graphs, but rather on their line graphs, using the method described in section 2. To detect ΔY exchanges, we maintain a counter for each vertex in the *original* pattern and target graphs, which records how many incident edges (represented by vertices in the line graphs) are currently being used. If the number of non-zero counters for the pattern graph differs from the number of non-zero counters for the target graph, a ΔY exchange has occurred and it is necessary to backtrack.

3.3 SplitP

Our second dedicated solver, SplitP, shares the compact data structures and the forward-checking of McSplit, and uses a similar partitioning algorithm to filter domains. However, SplitP does not use line graphs, but rather models the problem directly in style of our MiniZinc models 2 and 3, with variables for pattern edges and values for *oriented* target edges.

We use the example graphs in fig. 5 to illustrate SplitP's data structures.

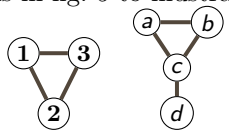


Fig. 5. Example graphs P and T

Initially, each pattern edge (which we denote by its endpoints; for example $(1, 2)$) has all four target edges in its domain. Rather than storing the four domains separately, we store the same information using two arrays of edges. In addition, we store the integer 2 to signify that either orientation of a target edge may be chosen (for example, edge $(1, 2)$ may be mapped to either (a, b) or (b, a)). Thus, before any tentative edge assignments are made, the domains are stored as:

$$[(1,2),(1,3),(2,3)] \qquad [(a,b),(a,c),(b,c),(c,d)] \qquad 2$$

Now, suppose that the algorithm has made the tentative assignment of pattern edge $(1, 2)$ to target edge (a, c). Edge $(1, 3)$ has (a, b) and $\perp$ in its domain, while

edge $(2, 3)$ has (c, b), (c, d) and $\perp$ in its domain. These domains are stored as follows (with the 1 at the end of each line signifying that only the shown orientation of each target edge is permitted).

$$\begin{array}{lll} [(1,3)] & [(a,b)] & 1 \\ [(2,3)] & [(c,b),(c,d)] & 1 \end{array}$$

We can view the McSplit approach (where values correspond to vertices in the target line graph, and thus effectively correspond to unoriented edges in the original target graph) as delaying the decision of which way to orient the target edges until after finding an optimal solution. SplitP, by contrast, makes orientation decisions as early as possible.

3.4 "Down" variants

The branch-and-bound solvers McSplit and SplitP attempt to find increasingly large incumbent subgraphs. In addition, we have implemented variants of these two algorithms that take the opposite approach, solving a sequence of decision problems where we first ask whether we can find a subgraph that uses all of the edges in the smaller graph, then all edges but one, and so on. These are named McSplit↓ and SplitP↓. (A version of McSplit↓ for MCIS was introduced in [8]; this used the approach of Hoffmann et al. [6].)

4 Experiments

For our experiments, we used a database of randomly-generated pairs of graph [13, 4]. To keep the total run time manageable, we selected the first ten instances from each family with no more than 35 vertices, giving a total of 2750 instances.

We used a cluster of machines with Intel Xeon E5-2697A v4 CPUs. For the CP models, MiniZinc 2.1.7 and the Gecode solver were used. We used the `first_fail` and `indomain_split` MiniZinc search annotations for these models.

Our experiments cover only the new approaches introduced in this paper, and do not provide a comparison with existing state-of-the-art algorithms. We intend to provide a fuller comparison—using more solvers and more families of instances—in a future version of this paper.

Figure 6 shows a cumulative plot of run times. Each point on a curve indicates the number of instances that were, individually, solved in less than a time shown on the horizontal axis. For example, approximately 200 of the instances could be solved by the MiniZinc 1 model in 1000 ms or less per instance.

The dedicated solvers are more three orders of magnitude faster than the fastest MiniZinc model (model 3). The ↓ variants are slightly faster than the branch-and-bound variants, and each version of SplitP is around twice as fast as its corresponding McSplit program.

The first plot of Figure 7 is a scatter plot of run times for SplitP↓ and McSplit↓, with one point per instance. The second plot shows search node counts.

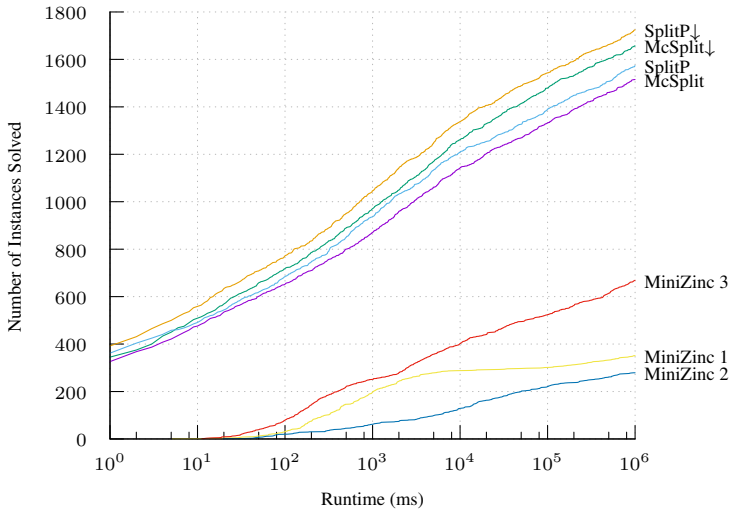


Fig. 6. Cumulative plot of run times

Instances where either algorithm timed out are not shown on the second plot. On most instances, McSplit↓ takes slightly more time and search nodes than SplitP↓. This difference is perhaps down to small differences in the choices made for variable and value ordering heuristics. While both McSplit↓ and SplitP↓ use a smallest-domain-first variable ordering, McSplit↓ breaks ties on degree in the linegraph, while SplitP↓ uses dynamic heuristics. We plan to carry out further investigations into tie-breaking rules in the future.

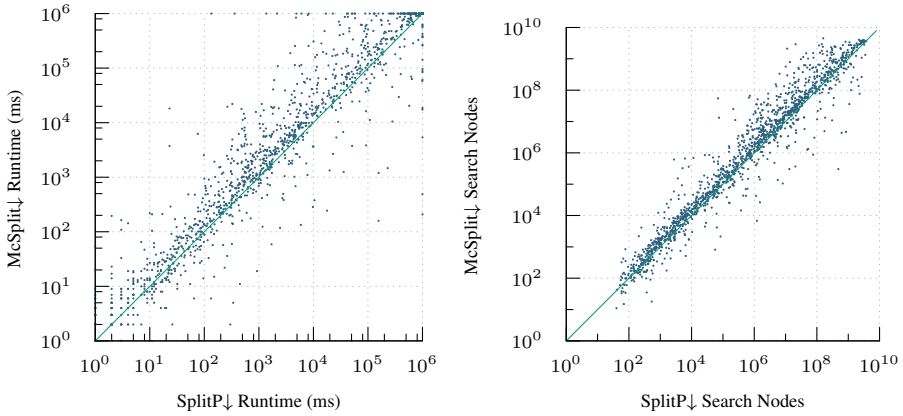


Fig. 7. Scatter plots of run times and search nodes, SplitP↓ versus McSplit↓. Each point represents one instance.

Figure 8 shows run times and search nodes for SplitP and SplitP↓. SplitP↓ is seldom more than three times slower than the branch-and-bound variant, and is often orders of magnitude faster.

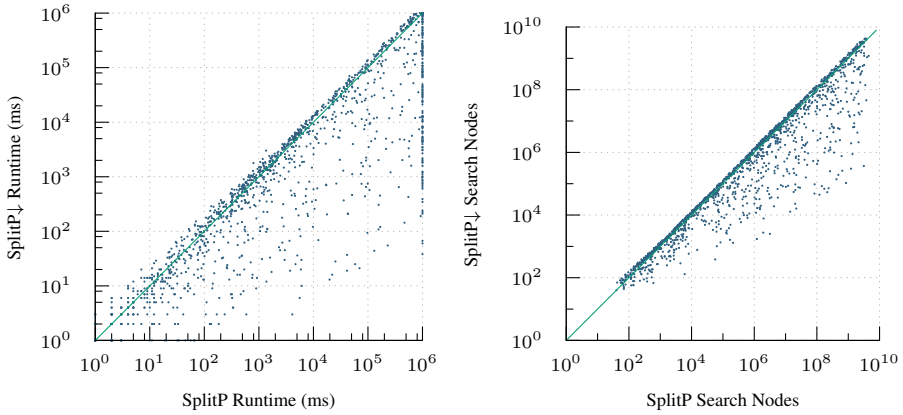


Fig. 8. Scatter plots of run times and search nodes, SplitP versus SplitP↓. Each point represents one instance.

5 Conclusion

We have implemented three constraint program models for maximum common edge subgraph, and two dedicated solvers inspired by constraint programming techniques. Of the CP models, we found the most effective to be a model containing variables for both vertices and edges. The dedicated solvers were orders of magnitude faster than the CP models; of these, the SplitP↓ and McSplit↓ variants were faster than their branch-and-bound counterparts. SplitP was somewhat faster than McSplit, but this may be simply due to the choices of variable and value ordering heuristics.

Although the CP models ran much more slowly than the dedicated algorithms, they have the advantage of flexibility: it would be straightforward to add additional constraints to the model if required. By contrast, it is unlikely that additional constraints could easily be used with the compact data structures of McSplit and SplitP.

In the future, we plan to extend our programs to handle labels on vertices and edges, and directed edges. We will also investigate a weighted version of the problem, where each mapping between an edge in the pattern graph and an edge in the target graph has an associated weight. We believe that a reduction to maximum weight clique using an association graph encoding could be used to solve this problem.

We also intend to run a fuller set of experiments, comparing against existing state-of-the-art algorithms and studying the effect of heuristics on solver performance.

References

1. Akutsu, T., Tamura, T.: A polynomial-time algorithm for computing the maximum common connected edge subgraph of outerplanar graphs of bounded degree. Algorithms **6**(1), 119–135 (2013). https://doi.org/10.3390/a6010119, https://doi.org/10.3390/a6010119
2. Bahiense, L., Manic, G., Piva, B., de Souza, C.C.: The maximum common edge subgraph problem: A polyhedral investigation. Discrete Applied Mathematics **160**(18), 2523–2541 (2012). https://doi.org/10.1016/j.dam.2012.01.026, https://doi.org/10.1016/j.dam.2012.01.026
3. Bokhari, S.H.: On the mapping problem. IEEE Trans. Computers **30**(3), 207–214 (1981). https://doi.org/10.1109/TC.1981.1675756, https://doi.org/10.1109/TC.1981.1675756
4. Conte, D., Foggia, P., Vento, M.: Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. J. Graph Algorithms Appl. **11**(1), 99–143 (2007), http://jgaa.info/accepted/2007/ConteFoggiaVento2007.11.1.pdf
5. Geelen, P.A.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: ECAI. pp. 31–35 (1992)
6. Hoffmann, R., McCreesh, C., Reilly, C.: Between subgraph isomorphism and maximum common subgraph. In: Singh, S.P., Markovitch, S. (eds.) Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA. pp. 3907–3914. AAAI Press (2017), http://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14948
7. Marenco, J.: Un algoritmo branch-and-cut para el problema de mapping. Ph.D. thesis, Masters thesis, Universidad de Buenos Aires, 1999. (1999)
8. McCreesh, C., Prosser, P., Trimble, J.: A partitioning algorithm for maximum common subgraph problems. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017. pp. 712–719 (2017). https://doi.org/10.24963/ijcai.2017/99, https://doi.org/10.24963/ijcai.2017/99
9. McGregor, J.J.: Backtrack search algorithms and the maximal common subgraph problem. Softw., Pract. Exper. **12**(1), 23–34 (1982). https://doi.org/10.1002/spe.4380120103, https://doi.org/10.1002/spe.4380120103
10. Petit, T., Régin, J., Bessière, C.: Specific filtering algorithms for over-constrained problems. In: Walsh, T. (ed.) Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2239, pp. 451–463. Springer (2001). https://doi.org/10.1007/3-540-45578-7_31, https://doi.org/10.1007/3-540-45578-7_31
11. Raymond, J.W., Gardiner, E.J., Willett, P.: RASCAL: calculation of graph similarity using maximum common edge subgraphs. Comput. J. **45**(6), 631–644 (2002). https://doi.org/10.1093/comjnl/45.6.631, https://doi.org/10.1093/comjnl/45.6.631
12. Raymond, J.W., Willett, P.: Maximum common subgraph isomorphism algorithms for the matching of chemical structures. Journal of Computer-Aided Molecular Design **16**(7), 521–533 (2002). https://doi.org/10.1023/A:1021271615909, http://dx.doi.org/10.1023/A:1021271615909
13. Santo, M.D., Foggia, P., Sansone, C., Vento, M.: A large database of graphs and its use for benchmarking graph isomorphism algorithms. Pattern Recognition Letters **24**(8), 1067–1079 (2003). https://doi.org/10.1016/S0167-8655(02)00253-2, http://dx.doi.org/10.1016/S0167-8655(02)00253-2

14. Whitney, H.: Congruent graphs and the connectivity of graphs. American Journal of Mathematics **54**(1), 150–168 (1932)

Towards a constraint system for round-off error analysis of floating-point computation

Rémy Garcia (student), Claude Michel (advisor), Marie Pelleau, and Michel Rueher (co-authors)

Université Côte d'Azur, CNRS, I3S, France
firstname.lastname@i3s.unice.fr

Abstract. In this paper, we introduce a new constraint solver aimed at analyzing the round-off errors that occur in floating-point computations. Such a solver allows reasoning on round-off errors by means of constraints on ranges of error values. This new solver is built by incorporating in a solver for constraints over the floating-point numbers the domain of errors which is dual to the domain of values. Both domains, the domain of values and the domain of errors, are associated with each variable of the problem. Additionally, we introduce projection functions that filter these domains as well as the mechanisms required for the analysis of errors. Preliminary experiments are encouraging.
Numerous works, which are based on an overestimation of actual errors, try to address similar issues. However, they do not provide critical information to reason on those errors, for example, by computing input values that exercise a given error.
To our knowledge, our solver is the first constraint solver with such reasoning capabilities over round-off errors.

Keywords: floating-point numbers · round-off error · constraints over floating-point numbers · domain of errors

1 Introduction

Floating-point computations induce errors due to rounding operations required to close the set of floating-point numbers. These errors are symptomatic of the distance between the computation over the floats and the computation over the reals. Moreover, they are behind many problems, such as the precision or the numerical stability of floating-point computation. Especially when users omit to take into account the nature of floating-point arithmetic and use it directly like real number arithmetic. Identifying, quantifying and localizing those errors are tedious tasks that are difficult to achieve without tools automating it. A well-known example of computations deviation due to errors on floating-point numbers is Rump's polynomial [15]:

$$333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

where $a = 77617$ and $b = 33096$. The exact value of this expression, computed using the GMP library, is $-\frac{54767}{66192} \approx -0.827396056$.

However, when this expression is evaluated on simple floats with a rounding mode set to the nearest even, the computed result is $\approx -6.3382530011411 \times 10^{29}$, which is far apart from the real value. The difference between these two results, about $-6.3382530011411 \times 10^{29}$, emphasizes the need for round-off error analysis tools.

Floating-point computation errors have been the subject of many works based on an overestimation of actual errors. Let us mention the abstract interpreter Fluctuat [6, 5] that combines affine arithmetic and zonotopes to analyze the robustness of programs over floating-point numbers. Another more recent work, PRECiSA [17, 13], relies on static analysis to evaluate round-off errors in a program. Nasrine Damouche [2] and Eva Darulova [3] have developed techniques for the automatic enhancement of numerical code. Their approach is based on an evaluation of round-off errors to estimate the distance between the expression over floats and the expression over reals. These approaches compute an error estimation which can be refined by splitting the search space into subdomains. However, it is not possible to directly reason on those errors, for example, by computing input values that exercise a given error. In order to overcome this lack of reasoning capabilities and to enhance the analysis of errors, we propose to incorporate in a constraint solver over floats [18, 10, 1, 11, 12], the domain of errors which is dual to the domain of values. Both domains are associated with each variable of the problem. Additionally, we introduce projection functions that filter those domains as well as mechanisms required for the analysis of errors. More precisely, we focus on the analysis of deviation between computations over the floats and computations over the reals.

Our approach is based on interval arithmetic for approximating the domains of errors. In addition, a search applied on reduced domains computes input values that satisfy constraints on errors. We deliberately ignore the possibility of an initial observational error on input data even though initial computational errors are handled. Thus input data are assumed with an initial error of zero. For the sake of simplicity, we restrain ourselves to the four classical arithmetic operations. This simplification permits exact computation of values over reals[1] for both values of expressions and computation of errors. Finally, the rounding mode is left to default, i.e. a rounding to the nearest even.

2 Notation and definitions

2.1 Floating-point numbers

The set of floating-point numbers is a finite subset of the rationals that has been introduced to approximate real numbers on a computer. The IEEE standard for

[1] By using a rational arithmetic library for computation on the reals and down to the memory limit.

floating-point arithmetic [9] defines the format of the different types of floating-point numbers as well as the behavior of arithmetic operations on these floating-point numbers. In the sequel, floating-point numbers are restricted to the more common one i.e. simple binary floating-point numbers represented using 32 bits and double binary floating-point numbers represented using 64 bits.

A binary floating-point number v is represented by a triple (s, e, m) where s is the sign of v, e, its exponent and m, its mantissa. When $e > 0$, v is normalized and its value is given by :

$$(-1)^s \times 1.m \times 2^{e-bias}$$

where the *bias* allows us to represent negative values of the exponent. For instance, for 32 bits floating-point numbers, the size of s is 1 bit, the size of e is 8 bits, the size of m is 23 bits and the *bias* is equal to 127.

x^+ denotes the smallest floating-point number strictly larger than x while x^- denotes the largest floating-point number strictly smaller than x. In other words, x^+ is the successor of x while x^- is its predecessor.

An *Ulp*, which stands for *unit in the last place*, is the distance which separate two consecutive floating-point numbers. However, this definition is ambiguous for floats that are a power of 2 like 1.0: in such a case, and if $x > 0$, then $x^+ - x = 2 * (x - x^-)$. To make this point clear, an explicit formulation of this distance is used whenever required.

3 Quantification of computation deviations

Computation on floating-point numbers is different from computation over real numbers due to rounding operations. Since the set of floating-point numbers is a finite subset of the real, in general, the result of an operation on the floats is not a float. In order to close the set of floating-point numbers for those operations, the result should be rounded to the nearest float according to a direction chosen beforehand.

The IEEE 754 norm [9] defines the behavior of floating-point arithmetic. For the four basic operations, it requires correct rounding, i.e. the result of an operation over the floats must be equal to the rounding of the result of the equivalent operation over the reals. More formally, $z = x \odot y = round(x \cdot y)$ where z, x and y are floating-point numbers, $\odot$ is one of the four basic arithmetic operations on the floats, namely, $\oplus$, $\ominus$, $\otimes$, and $\oslash$, $\cdot$ being the equivalent operation on the reals, and *round* being the rounding function. This property bounds the error introduced by an operation over floats to $\pm\frac{1}{2}ulp(z)$ for correctly rounded operations with a rounding mode set to round to the nearest even float, which is the most frequent rounding mode.

When the result of an operation on the floats is rounded, it is different from the one expected on the reals. Moreover, each operation that belongs to a complex expression is likely to introduce a difference between the expected result on the reals and the one computed on the floats. Whereas for a given operation, the computed float is optimal in terms of rounding, the accumulation of these

approximations can lead to significant deviations, like the one observed with Rump's polynomial.

Compared to its equivalent over the reals, the deviation of a computation over the floats takes root in each elementary operation. Therefore, it is possible to rebuild it from the composition of each elementary operation behavior. Input variables can come with errors attached due to previous computations. For example, for the variable x, the deviation on the computation of x, e_x, is given by $e_x = x_{\mathbb{R}} - x_{\mathbb{F}}$ where $x_{\mathbb{R}}$ and $x_{\mathbb{F}}$ denote the expected results on the reals and on the floats respectively. Contrary to an observational error, e_x is signed. This choice is required to capture correctly specific behaviors of floating-point computations, such as error compensations.

As such, computation deviation due to a subtraction can be formulated as follows: for $z = x \ominus y$, the error on z, e_z, is equal to $(x_{\mathbb{R}} - y_{\mathbb{R}}) - (x_{\mathbb{F}} \ominus y_{\mathbb{F}})$. As $e_x = x_{\mathbb{R}} - x_{\mathbb{F}}$ and $e_y = y_{\mathbb{R}} - y_{\mathbb{F}}$, we have

$$e_z = ((x_{\mathbb{F}} + e_x) - (y_{\mathbb{F}} + e_y)) - (x_{\mathbb{F}} \ominus y_{\mathbb{F}})$$

The deviation between the result on the reals and the result on the floats for a subtraction can then be computed by the following formula:

$$e_z = e_x - e_y + ((x_{\mathbb{F}} - y_{\mathbb{F}}) - (x_{\mathbb{F}} \ominus y_{\mathbb{F}}))$$

In this formula, the last term, $((x_{\mathbb{F}} - y_{\mathbb{F}}) - (x_{\mathbb{F}} \ominus y_{\mathbb{F}}))$, characterizes the error produced by the subtraction operation itself. Let $e_\ominus$ denotes this subtraction operation error. The formula can then be simplified to:

$$e_z = e_x - e_y + e_\ominus$$

The formula comprises two elements: firstly the combination of deviations from input values and secondly, the deviation introduced by the elementary operation.

$$\begin{aligned}
\mathit{Addition} &: z = x \oplus y \rightarrow e_z = e_x + e_y + e_\oplus \\
\mathit{Subtraction} &: z = x \ominus y \rightarrow e_z = e_x - e_y + e_\ominus \\
\mathit{Multiplication} &: z = x \otimes y \rightarrow e_z = x_{\mathbb{F}} e_y + y_{\mathbb{F}} e_x + e_x e_y + e_\otimes \\
\mathit{Division} &: z = x \oslash y \rightarrow e_z = \frac{y_{\mathbb{F}} e_x - x_{\mathbb{F}} e_y}{y_{\mathbb{F}}(y_{\mathbb{F}} + e_y)} + e_\oslash
\end{aligned}$$

Fig. 1. Computation of deviation for basic operations

Figure 1 formulates computation deviations for all four basic operations. For each of these formulae, the error computation combines deviations from input values and the error introduced by the current operation. Notice that, for multiplication and division, this deviation is proportional to input values.

All these formulae compute the difference between the expected result on the reals and the actual one on the floats for a basic operation. Our constraint solver over the errors on the floats relies on these formulae.

4 Domain of errors

In a classical *CSP*, to each variable x is associated $\mathbf{x}$ its domain of values. It denotes the set of possible values that this variable can take. When the variable takes values in $\mathbb{F}$, its domain of values is represented by an interval of floats:

$$\mathbf{x}_{\mathbb{F}} = [\underline{x}_{\mathbb{F}}, \overline{x}_{\mathbb{F}}] = \{x_{\mathbb{F}} \in \mathbb{F}, \underline{x}_{\mathbb{F}} \leq x_{\mathbb{F}} \leq \overline{x}_{\mathbb{F}}\}$$

where $\underline{x}_{\mathbb{F}} \in \mathbb{F}$ and $\overline{x}_{\mathbb{F}} \in \mathbb{F}$.

Computation errors form a new dimension to consider. They require a specific domain in view of the distinct nature of elements to represent, but also, due to the possible values of errors which belong to the set of reals. Therefore, we introduce a domain of errors, which is associated with each variable of a problem. Since all arithmetic constraints processed here are reduced to the four basic operations, and since those four operations are applied over floats, i.e. a finite subset of rationals, this domain can be defined as an interval of rationals with bounds in $\mathbb{Q}$:

$$\mathbf{e}_x = [\underline{e}_x, \overline{e}_x] = \{e_x \in \mathbb{Q}, \underline{e}_x \leq e_x \leq \overline{e}_x\}$$

where $\underline{e}_x \in \mathbb{Q}$ and $\overline{e}_x \in \mathbb{Q}$.

Another domain of errors is required for the smooth running of our system: it is the domain of errors on operations, denoted by $e_\odot$, that appears in the computation of the deviations (see Figure 1). Contrary to previous domains, it is not attached to each variable of a problem but to each *instance* of an arithmetic operation of a problem.

Like the domain of errors attached to a variable, it takes values in the set of rationals. Thus, we have:

$$\mathbf{e}_\odot = [\underline{e}_\odot, \overline{e}_\odot] = \{e_\odot \in \mathbb{Q}, \underline{e}_\odot \leq e_\odot \leq \overline{e}_\odot\}$$

where $\underline{e}_\odot \in \mathbb{Q}$ and $\overline{e}_\odot \in \mathbb{Q}$.

This triple, composed of the domain of values, the domain of errors, and the domain of errors on operations, is required to represent the set of phenomena due firstly to possible values on variables and secondly, to different errors that come into play in computation over floats.

5 Projection functions

The filtering process of our solver is based on classical projection functions to reduce the domains of variables. Domains of values can be computed by projection functions defined in [11] and extended in [1] and [10] but new ones are required for the domains of errors.

Those projections on the domains of errors are made through an extension over intervals of formulae from Figure 1. Since these formulae are written over reals, they can naturally be extended to intervals. For example, in the case of the subtraction, we get the four projection functions below:

$$\mathbf{e}_z \leftarrow \mathbf{e}_z \cap (\mathbf{e}_x - \mathbf{e}_y + \mathbf{e}_\ominus)$$
$$\mathbf{e}_x \leftarrow \mathbf{e}_x \cap (\mathbf{e}_z + \mathbf{e}_y - \mathbf{e}_\ominus)$$
$$\mathbf{e}_y \leftarrow \mathbf{e}_y \cap (-\mathbf{e}_z + \mathbf{e}_x + \mathbf{e}_\ominus)$$
$$\mathbf{e}_\ominus \leftarrow \mathbf{e}_\ominus \cap (\mathbf{e}_z - \mathbf{e}_x + \mathbf{e}_y)$$

where $\mathbf{e}_x$, $\mathbf{e}_y$, and $\mathbf{e}_z$ are the domains of errors of variables x, y, and z respectively and $\mathbf{e}_\ominus$ is the domain of errors on the subtraction.

Figure 2 gives projection functions for the three other arithmetic operations. Note that the projection on $y_\mathbb{F}$ for the division requires solving a quadratic equation and requires the computation of a square root. Thanks to outward roundings, correct computation of such a square root on rationals is obtained using floating-point square root at the price of an over-approximation. Note also that these projections handle the general case. For the sake of simplicity, special cases like a division by zero are not exposed here.

Addition :

$$\mathbf{e}_z \leftarrow \mathbf{e}_z \cap (\mathbf{e}_x + \mathbf{e}_y + \mathbf{e}_\oplus)$$
$$\mathbf{e}_x \leftarrow \mathbf{e}_x \cap (\mathbf{e}_z - \mathbf{e}_y - \mathbf{e}_\oplus)$$
$$\mathbf{e}_y \leftarrow \mathbf{e}_y \cap (\mathbf{e}_z - \mathbf{e}_x - \mathbf{e}_\oplus)$$
$$\mathbf{e}_\oplus \leftarrow \mathbf{e}_\oplus \cap (\mathbf{e}_z - \mathbf{e}_x - \mathbf{e}_y)$$

Multiplication :

$$\mathbf{e}_z \leftarrow \mathbf{e}_z \cap (\mathbf{x}_\mathbb{F}\mathbf{e}_y + \mathbf{y}_\mathbb{F}\mathbf{e}_x + \mathbf{e}_x\mathbf{e}_y + \mathbf{e}_\otimes)$$
$$\mathbf{e}_x \leftarrow \mathbf{e}_x \cap \left(\frac{\mathbf{e}_z - \mathbf{x}_\mathbb{F}\mathbf{e}_y - \mathbf{e}_\otimes}{\mathbf{y}_\mathbb{F} + \mathbf{e}_y}\right)$$
$$\mathbf{e}_y \leftarrow \mathbf{e}_y \cap \left(\frac{\mathbf{e}_z - \mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{e}_\otimes}{\mathbf{x}_\mathbb{F} + \mathbf{e}_x}\right)$$
$$\mathbf{e}_\otimes \leftarrow \mathbf{e}_\otimes \cap (\mathbf{e}_z - \mathbf{x}_\mathbb{F}\mathbf{e}_y - \mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{e}_x\mathbf{e}_y)$$
$$\mathbf{x}_\mathbb{F} \leftarrow \mathbf{x}_\mathbb{F} \cap \left(\frac{\mathbf{e}_z - \mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{e}_x\mathbf{e}_y - \mathbf{e}_\otimes}{\mathbf{e}_y}\right)$$
$$\mathbf{y}_\mathbb{F} \leftarrow \mathbf{y}_\mathbb{F} \cap \left(\frac{\mathbf{e}_z - \mathbf{x}_\mathbb{F}\mathbf{e}_y - \mathbf{e}_x\mathbf{e}_y - \mathbf{e}_\otimes}{\mathbf{e}_x}\right)$$

Division :

$$\mathbf{e}_z \leftarrow \mathbf{e}_z \cap \left(\frac{\mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{x}_\mathbb{F}\mathbf{e}_y}{\mathbf{y}_\mathbb{F}(\mathbf{y}_\mathbb{F} + \mathbf{e}_y)} + \mathbf{e}_\oslash\right)$$
$$\mathbf{e}_x \leftarrow \mathbf{e}_x \cap \left((\mathbf{e}_z - \mathbf{e}_\oslash)(\mathbf{y}_\mathbb{F} + \mathbf{e}_y) + \frac{\mathbf{x}_\mathbb{F}\mathbf{e}_y}{\mathbf{y}_\mathbb{F}}\right)$$
$$\mathbf{e}_y \leftarrow \mathbf{e}_y \cap \left(\frac{\mathbf{e}_x - \mathbf{e}_z\mathbf{y}_\mathbb{F} + \mathbf{e}_\oslash\mathbf{y}_\mathbb{F}}{\mathbf{e}_z - \mathbf{e}_\oslash + \frac{\mathbf{x}_\mathbb{F}}{\mathbf{y}_\mathbb{F}}}\right)$$
$$\mathbf{e}_\oslash \leftarrow \mathbf{e}_\oslash \cap \left(\mathbf{e}_z - \frac{\mathbf{y}_\mathbb{F}\mathbf{e}_x - \mathbf{x}_\mathbb{F}\mathbf{e}_y}{\mathbf{y}_\mathbb{F}(\mathbf{y}_\mathbb{F} + \mathbf{e}_y)}\right)$$
$$\mathbf{x}_\mathbb{F} \leftarrow \mathbf{x}_\mathbb{F} \cap \left(\frac{(\mathbf{e}_\oslash - \mathbf{e}_z)\mathbf{y}_\mathbb{F}(\mathbf{y}_\mathbb{F} + \mathbf{e}_y) + \mathbf{y}_\mathbb{F}\mathbf{e}_x}{\mathbf{e}_y}\right)$$
$$\mathbf{y}_\mathbb{F} \leftarrow \mathbf{y}_\mathbb{F} \cap [min(\underline{\delta}_1, \underline{\delta}_2), max(\overline{\delta}_1, \overline{\delta}_2)]$$

with

$$\delta_1 \leftarrow \frac{\mathbf{e}_x - (\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y - \sqrt{\mathbf{\Delta}}}{2(\mathbf{e}_z - \mathbf{e}_\oslash)}$$
$$\delta_2 \leftarrow \frac{\mathbf{e}_x - (\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y + \sqrt{\mathbf{\Delta}}}{2(\mathbf{e}_z - \mathbf{e}_\oslash)}$$
$$\mathbf{\Delta} \leftarrow [0, +\infty) \cap ((\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y - \mathbf{e}_x)^2 + 4(\mathbf{e}_z - \mathbf{e}_\oslash)\mathbf{e}_y\mathbf{x}_\mathbb{F}$$

Fig. 2. Projection functions of arithmetic operation

Projection functions, on the domain of errors, support only arithmetic operations and assignment, where the computation error from the expression is

transmitted to the assigned variable. Since the error is not involved in comparison operators, their projection functions only manage domains of values.

The set of those projection functions is used to reduce all variables' domains until a fixed point is reached. For the sake of efficiency, but also to get around potential slow convergence, the fixed point computation is stopped when no domain reduction is greater than 5%.

6 Links between the domain of values and the domain of errors

In order to take advantage of domain reductions of one domain in another domain, clear and strong links must be established between the domain of values and the domain of errors. What naturally occurs for domains of values thanks to constraints on values requires more attention when it comes to the relations between dual domains.

A first relation between the domain of values and the domain of errors on operations is based upon the IEEE 754 norm, which guarantees that basic arithmetic operations are correctly rounded. Since the four basic operations are correctly rounded to the nearest even float, we have

$$(x \odot y) - \frac{(x \odot y) - (x \odot y)^-}{2} \leq (x \cdot y) \leq (x \odot y) + \frac{(x \odot y)^+ - (x \odot y)}{2}$$

where x^- and x^+ denote respectively, the greatest floating-point number strictly smaller than x and the smallest floating-point number strictly larger than x. In other words, the result over floats is, at a half-ulp, the distance between two successive floats from the result over reals. Thus, the error on an operation is contained in this ulp:

$$-\frac{(x \odot y) - (x \odot y)^-}{2} \leq e_\odot \leq +\frac{(x \odot y)^+ - (x \odot y)}{2}$$

This equation sets a relation between the domain of values and the domain of errors on operations: operation errors can never be greater than the greatest half-ulp of the domain of values on the operation result. The projection function for the domain of errors on operations is obtained by extending this formula to intervals:

$$\mathbf{e}_\odot \leftarrow \mathbf{e}_\odot \cap \left[-\frac{min((\underline{z} - \underline{z}^-), (\overline{z} - \overline{z}^-))}{2}, +\frac{max((\underline{z}^+ - \underline{z}), (\overline{z}^+ - \overline{z}))}{2}\right]$$

Finally, these links are refined by means of other well-known properties of floating-point arithmetic like the Sterbenz property of the subtraction [16] or the Hauser property on the addition [7]. Both properties give conditions under which these operations produce exact results. As is the well-known property that $2^k * x$ is exactly computed provided that no overflow occurs.

7 Constraints over errors

Usually, constraints available in a solver establish relations between variables of a problem. The duality of domains available in our solver requires introducing a distinction between the domain of values and the domain of errors. In order to preserve the current semantic of expressions, variables keep on representing possible values. A dedicated function, $err(x)$, makes it possible to express constraints over errors. For example, $abs(err(x)) \geq \epsilon$, denote a constraint which demands that the error on variable x be, in absolute value, greater or equal to ϵ. It should be noted that since errors are taking their values in $\mathbb{Q}$, the constraint is over rationals.

When a constraint involves errors and variables, the latter, with domains over floats, are promoted to rationals. Therefore, the constraint is converted to a constraint over rationals.

8 Preliminary experiments

Projection functions and constraints over errors are being evaluated in a prototype based on Objective-CP [8], which already handles constraints over floats thanks to the projection functions of FPCS [12]. All experiments are carried out on a MacBook Pro i7 2,8GHz with 16GB of memory.

8.1 Predator prey

Predator prey [4] has been extracted from the FPBench test suite[2]:

```
double predatorPrey(double x) {
  double r = 4.0;
  double K = 1.11;
  double z = (((r * x) * x) / (1.0 + ((x / K) * (x / K))));
  return z;
}
```

When $x \in [\frac{1}{10}, \frac{3}{10}]$, Objective-CP using a simple and single filtering process reduces the domain of z to $[3.72770587e{-}02, 3.57101682e{-}01]$ and its error domain to $[-1.04160431e{-}16, 1.04160431e{-}16]$. Fluctuat reduces the domain of z to $[3.72770601e{-}02, 3.44238968e{-}01]$ and its error to $[-1.33729201e{-}16, 1.33729201e{-}16]$. Thus, while Fluctuat provides better bounds for the domain of values, Objective-CP provides better bounds for the domain of error. Moreover, our solver can use a search procedure to compute error values that are reachable. For example, we can search for input values such that the error on z will be strictly greater than zero.

With this constraint the solver output $z = 3.35493528698854015512864 6e{-}01$ with an error of $3.09661231429390630181643 2e{-}17$.

[2] See fpbench.org.

Since rational numbers are used for computations of errors it is crucial to take solving time into account. Table 1 shows times in seconds for the generation of round-off error bounds on some benchmarks from FPBench. Solving time for Objective-CP are correct, especially as a search procedure is done in addition of filtering. Those times comfort us in the use of rational numbers for representing errors.

	Gappa	Fluctuat	Real2Float	FPTaylor	PRECiSA	Objective-CP
carbonGas	0.152	**0.025**	0.815	1.209	3.830	*0.060*
verhulst	*0.034*	0.043	0.465	0.812	0.789	**0.032**
predPrey	0.052	**0.031**	0.735	0.916	0.477	*0.050*
turbine1	*0.165*	**0.028**	67.960	2.906	110.272	0.232

Table 1. Times in seconds for the generation of round-off error bounds. For Objective-CP a searh is also used. (bold indicates the best approximation and italic indicates the second best)

9 Conclusion

In this paper, we introduced a constraint solver capable of reasoning over computation errors on floating-point numbers. It is built over a system of dual domains, the first one characterizing possible values that a variable of the problem can take and the second one defining errors committed during computations. Moreover, there are particular domains, bind to instances of arithmetic operations in numerical expressions of the constraints, which represent errors in those operations. Our solver, enhanced with projection functions and constraints over errors, offers unique possibilities to reason on computation errors. Preliminary experiments are promising and will naturally be reinforced with more benchmarks.

Such a solver might appear limited by the use of rational numbers and multiprecision integers. However, a thorough examination of the solver behavior has shown that its main limit lies in its approximation of round-off errors. Firstly, round-off errors are not uniformly distributed across input values. As a result, finding input values that satisfy some error constraints has often to resort to an enumeration of possible values. Secondly, round-off error of operations are overestimated. Such an overestimation does not perform the fine domain reductions that would allow an efficient search. Therefore, a deeper understanding and a tighter representation of the round-off error on an operation basis is a must to actually improve the behavior of our solver.

Further work include extending support for a wider set of arithmetic functions, improving the search to quickly find solutions in the presence of constraints over errors and to add global optimization capabilities, for example, by using a branch-and-bound method. Then, by formulating a problem as an optimization problem, we should be in a position to determine for which input values the error is maximal.

Another direction of improvement is the combination of CSP with other tools dedicated to round-off error like abstract interpreter in an approach similar to what has already been done for domains of values [14].

References

1. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Software Testing, Verification and Reliability **16**(2), 97–121 (2006)
2. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. STTT **19**(4), 427–448 (2017)
3. Darulova, E., Kuncak, V.: Sound compilation of reals. In: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 235–248. ACM (2014)
4. Darulova, E., Kuncak, V.: Sound compilation of reals. pp. 235–248. POPL '14 (2014)
5. Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19,. LNCS, vol. 6174, pp. 212–226 (2010)
6. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4134, pp. 18–34 (2006)
7. Hauser, J.R.: Handling floating-point exceptions in numeric programs. ACM Trans. Program. Lang. Syst. **18**(2), 139–174 (Mar 1996)
8. Hentenryck, P.V., Michel, L.: The objective-cp optimization system. In: 19th International Conference on Principles and Practice of Constraint Programming (CP 2013). pp. 8–29 (2013)
9. IEEE: 754-2008 - IEEE Standard for floating point arithmethic (2008)
10. Marre, B., Michel, C.: Improving the floating point addition and subtraction constraints. In: Proceedings of the 16th international conference on Principles and practice of constraint programming (CP'10). pp. 360–367. LNCS 6308, St. Andrews, Scotland (6–10th Sep 2010)
11. Michel, C.: Exact projection functions for floating point number constraints. In: AI&M 1-2002, Seventh international symposium on Artificial Intelligence and Mathematics (7th ISAIM). Fort Lauderdale, Floride (US) (2–4th Jan 2002)
12. Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: 7th International Conference on Principles and Practice of Constraint Programming (CP 2001). pp. 524–538 (2001)
13. Moscato, M.M., Titolo, L., Dutle, A., Muñoz, C.A.: Automatic estimation of verified floating-point round-off errors via static analysis. In: Computer Safety, Reliability, and Security - 36th International Conference, SAFECOMP 2017, Trento, Italy, September 13-15. pp. 213–229 (2017)
14. Ponsini, O., Michel, C., Rueher, M.: Verifying floating-point programs with constraint programming and abstract interpretation techniques. Automated Software Engineering **23**(2), 191–217 (Jun 2016)
15. Rump, S.: Algorithms for verified inclusions: Theory and practice. In: Moore, R.E. (ed.) Reliability in Computing: The Role of Interval Methods in Scientific Computing, pp. 109–126. Academic Press Professional, Inc., San Diego, CA, USA (1988)
16. Sterbenz, P.H.: Floating Point Computation. Prentice-Hall (1974)
17. Titolo, L., Feliú, M.A., Moscato, M.M., Muñoz, C.A.: An abstract interpretation framework for the round-off error analysis of floating-point programs. In: Verification, Model Checking, and Abstract Interpretation - 19th International Conference, VMCAI 2018, Los Angeles, CA, USA, January 7-9. pp. 516–537 (2018)
18. Zitoun, H., Michel, C., Rueher, M., Michel, L.: Search strategies for floating point constraint systems. In: 23rd International Conference on Principles and Practice of Constraint Programming, CP 2017. pp. 707–722 (2017)

Towards solving ESSENCE, a proof of concept using multi sets and sets

Saad Attieh and Christopher Jefferson

School of Computer Science, University of St Andrews, St Andrews, UK
{sa74,caj21}@st-andrews.ac.uk

Abstract. We propose a local search solver that operates on the high level structures found in the ESSENCE abstract constraint specification language. High quality neighbourhoods are automatically derived from the structured variable types such as set of partition, sequence of function. The solver we present, ATHANOR, is distinguished from other local search solvers as it can operate directly on the high level types in ESSENCE without refining such types into low level representations. This provides a major scalability advantage for problems with nested structures such as set of set, since ATHANOR dynamically adds and deletes constraints as the sizes of these structures vary during search. The ESSENCE language contains many abstract variable types. In this paper, we present an implementation which supports multi sets and sets as a proof of concept. We outline the framework required to perform local search on ESSENCE expressions, covering incremental evaluation, dynamic unrolling and neighbourhood construction. The solver is benchmarked against other constraint programming and local search solvers with Sonet, a problem which makes use of the nested variable type multi set of set. Future work will focus on broadening the range of types supported by ATHANOR.

1 Introduction

Constraint modelling languages, such as MiniZinc [16] or ESSENCE [8,9,10] offer to users a convenient means of expressing a constraint problem without concerning themselves with the specific details of a particular constraint solver. We focus herein on the ESSENCE language, which is characterised by its support for abstract type such as set, multiset, function and partition, and particularly by its support for nesting of these types, such as set of multisets, or multiset of functions. To illustrate, consider the ESSENCE specification of the Synchronous Optical Networking Problem (Sonet, problem 56 at www.csplib.org. See also [11,22]) in Figure 1. An ESSENCE specification identifies: the input parameters of the problem class (`given`), whose values define an instance; the combinatorial objects to be found (`find`); the constraints the objects must satisfy (`such that`); identifiers declared (`letting`); and an (optional) objective function (`min/maximising`). In this example, the single abstract decision variable `network` is a multiset of sets, representing the rings on which communicating nodes are installed.

```
given nnodes, nrings, capacity : int(1..)
letting Nodes be domain int(1..nnodes)
$ connections that must be achieved between the nodes
given demand : set of set (size 2) of Nodes

find network :
  mset (size nrings) of set (maxSize capacity) of Nodes

such that
$All connections between nodes are achieved
forAll pair in demand .
    exists ring in network .
        pair subsetEq ring

$ objective: minimise total number of connections to rings.
$ i.e. minimise sum of the size of each ring.
minimising sum ring in network . |ring|
```

Fig. 1: Essence specification of the Synchronous Optical Networking problem. A set of rings is used to facilitate to communication between nodes. A node may be installed onto multiple rings. The task is to ensure that all pairs of nodes that require to communicate, given in `demand`, are able to do so while minimising the total number of installations onto the rings.

The Conjure automated constraint modelling system [1,2,4,5] refines an Essence specification into a solver-independent constraint model in the Essence Prime modelling language [18], where the abstract decision variables are represented as constrained collections of primitive variables, such as integer or Boolean variables. The Savile Row system [17,18,19] then transforms and prepares the Essence Prime model for input to a particular constraint solver, such as Minion [12], or SAT.

Refinement obscures the abstract structure apparent in the original Essence specification, which is a particular problem for constructing neighbourhoods for local search. In recent work [3], we presented a method for generating neighbourhoods in Essence (i.e. pre-refinement) and then refining them along with the problem specification. In this paper, we describe a proof-of-concept solver Athanor that takes an alternative approach: operating directly on an Essence specification, performing local search on the abstract variables. We motivate this approach below.

1.1 Why operate directly on Essence specifications

There are several benefits to operating directly over the high level variables. Firstly, with all the type information retained, Athanor is able to automatically construct neighbourhoods that are useful for solving the problem. Such

neighbourhoods would be hard to recover by analysing the model after it has been refined for traditional CP or local search solvers. For example, by identifying that a variable has the set type, Athanor can construct neighbourhoods that add to the set, remove from the set or swap one value for another. However, if a variable has the sequence type, neighbourhoods which take into account the order of elements would be added, for example, reversing a contiguous subsequence of elements.

Another benefit is that the only constraints that Athanor must consider are those that describe the problem and its objective. On the other hand, the equivalent low level representations of the abstract variable types in Essence, that is, representations which are accepted by traditional CP or local search solvers, must be composed of only integer and boolean variables. This requires that additional constraints be posted which maintain the chosen representations/encodings of the abstract variables. For example, insuring that a collection of integers form a valid partition. This is not necessary with a solver that understands the higher level types and their properties that must be maintained.

Finally, there is a scalability advantage whereby the size of high level variables can vary during search. For example, given a set of items, during search, the problem constraints must only be posted on the items in the set. As items are added, new constraints can be posted dynamically and can be removed if those items are deleted from the set. In comparison, if refined to a representation composed of integers and booleans, sufficiently many variables and constraints must be added to the model to handle the possibility of the set being its maximum size, even though this may be far larger than the size that optimises the objective.

This paper presents the framework that has been constructed to support solving Essence specifications directly. Though we make mention of several Essence types, the framework currently supports sets and multi sets which can be arbitrarily nested. The same methodology that is described in the following sections will be applied to the rest of the Essence types in future work. Section 2 describes the implementation of an incremental evaluator for constraints. Section 3 briefly discusses constraint violations and how they have been integrated with high level structured types. Section 4 describes the method of dynamically adding and removing constraints as items are added and removed from set variables. Section 5 outlines how neighbourhoods are automatically derived in the context of the Sonet problem and closes with some benchmark results comparing Athanor with other local search and CP solvers.

2 Incremental Evaluation

Athanor represents an Essence specification as a pair of abstract syntax trees (ASTs), one representing the constraints in the specification, the other representing the objective function. The leaves (which represent the variables in the problem) are assigned values during search. The solver incrementally updates the AST to reflect changes to the set of variable assignments. Incremental eval-

uation takes advantage of the fact that if a leaf of the tree (variable) is assigned to a new value, only its ancestors [1] must be reevaluated.

At the start of search, ATHANOR begins by assigning a random value to each of the variables followed by full evaluation of the AST. Afterwords, in order to facilitate incremental evaluation, every node in the AST attaches a trigger to each of its children, a call back which is invoked notifying the parent of changes to the child nodes that might affect the value yielded by the parent. Every type of node (integer returning, set returning, etc.) can trigger at least two types of events.

- `possibleValueChange()`, notifying the parent that the value yielded by the child may change. This allows the parent to record any properties of the child before the child's value is altered. This event must precede any change to the child but it is legal for no change to actually take place.
- `valueChanged()`, notifying the parent that the value yielded by the child has changed.

However, for higher level types such as set and multi set, simply indicating that a value has changed can greatly hinder incremental evaluation since these types are composed of many elements. Such an event gives no indication as to how many of the elements in the set need reevaluating. Therefore, high level types can make use of more descriptive events. For example, a set also has:

- `valueAdded()`,
- `valueRemoved()`,
- `possibleMemberValueChange()`,
- `memberValueChange()`.

Of course, `memberValueChange()` can be formed by composing `valueRemoved()` and `valueAdded()` but as shown in the next section, it can be useful to consider these two events as one. Also note that a constraint may choose to attach triggers to a set as a whole or trigger on the items within.

Figure 2 gives an example AST state during incremental evaluation, using the expression describing the objective of the Sonet problem in Figure 1.

The process that takes place when adding the number 3 to the set `ring2` is as follows:

- `ring2` triggers the event `possibleValueChange()` and this is echoed all the way to the root.
- When `sum` receives the `possibleValueChange()` event, it caches the value of the operand that triggered the event, `size:v=1`.
- The integer 3 is then added to `ring2`. Its value is now `ring2:v={1,3}`,
- the event `valueAdded()` is sent to the parent.
- `size` updates its value to `size:v=2`, the event `valueChanged()` is sent to its parent.

[1] the nodes on the path from the leaf to the root

```
1 sum ring in network .
2         |ring|
```

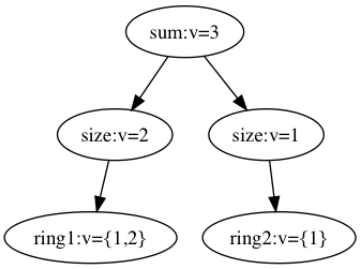


Fig. 2: Incremental evaluation. $|x|$ means size of x

- `sum` updates its value by subtracting the old value `size:v=1` and adds the new value `size:v=2`. The old value was cached when the `sum` received the `possibleValueChange()` event at the start.
- The node now has the value `sum:v=4`, the event `valueChanged()` is forwarded to its parent...

3 Violation counts

In Figure 2 we showed how the values of integers and sets are incrementally updated. A similar procedure is used for Boolean expressions, which have been extended to store a violation count. A violation count is an integer, a heuristic that gives an indication as to the magnitude of the change necessary to the set of assignments in order that all constraints become satisfied. Methods of calculating violations have been inspired fromvan Hentenryck and Michel [14]. For example, given two integers x and y and the constraint $c(x = y)$, the violation on c $v(c) = |x - y|$. A violation count is also attributed to the variables under a constraint. These variable violation counts are a heuristic used to give an indication as to what extent each variable contributes towards the violating constraints. The violation count on a variable u is the sum of violations attributed to u by the constraints posted on u. This helps to guide the solver when selecting which variables to modify when searching for a feasible solution. However, previous work on violation counts have only made reference to variables with integer or boolean types. We must examine how to extend the method to variables with nested types. Consider the example shown in Figure 3 in which a constraint is posted on to the integers contained within a set.

```
1 find s : set of int...
2 such that forAll i in s . c(i)
```

Fig. 3: Constraint on a nested type, $c(i)$ is some arbitrary constraint on i

We must consider how violations are attributed when elements in$s$ violate the constraint c. As mentioned, the violation counts on variables are supposed to guide the solver towards identifying the cause of violating constraints. Hence, the rules for attributing violations to nested types is as follows:

- When a violation is attributed to an element i of a structure s such as a set or sequence, the same violation is added to s and successively the structure that contains s and so on to the most outer structure.
- However, if i is itself a containing structure, the violation is not attributed to any of the elements in i. Neither is the violation propagated to any of the siblings of i, that is, other elements contained in s.
- Hence, the violation on any containing structure s is the sum of violations attributed to s plus the sum of violations attributed to the elements in s.

Consider the constraint $|s| = 1$ (the size of s is 1). If this constraint is violated, all the elements in s are all equally to blame. Therefore, there bares no benefit in attributing any violations to the elements in s, only s as a whole is assigned a violation. However, if the constraint is like that shown in Figure 3, it makes sense to assign a violation to only those elements in s that are causing the violation, so that the solver may be biased towards altering their value. s itself inherits the violation of its elements so that it may be distinguished from other variables; it is natural to consider a set with two violating elements to have a larger violation than a set with one violating element.

4 Dynamic unrolling of quantifiers

Although an Essence specification has a fixed number of abstract variables, these variables are usually of container types (set, sequence, multi set, and so on). The values of such variables can vary considerably in size and hence new elements can be introduced or deleted during search. Essence also allows the quantification over such containers attaching a constraint to each of the elements within. Therefore, it must be possible to add and delete constraints in accordance with the changes in size of the values of the variables being quantified over.

```
1 find s : set of int(1..5)
2 such that
3 forAll i in s. i % 2 = 0
```

Fig. 4: Quantifying over a set

Consider Figure 4, for example. In the AST representation, s is an operand of the `forAll` node. The `forAll` node also has one operand for each item in s. The `forAll` node also stores the expression (`i%2=0`) that is to be applied

to each element in the set. This expression is a template, meaning that it is represented by an incomplete AST. The AST is incomplete as i in the expression does not refer to one variable instance. Rather, it is used to refer to each value in the set. We call this the iterator. When s changes from being empty to having one element, an operand is added to the `forAll` node, the expression template is copied in and made complete by assigning the iterator to the newly added element. The AST subtree representing the copied expression is than fully evaluated much like the evaluation of the entire AST at the start of search. The nodes in the subtree then begin triggering on their children just as described in Section 2.

However, as more elements are added, rather than copying the unevaluated expression template, the expression subtree most recently added to the `forAll` node is copied. This is because the expression would have already been evaluated. It is only necessary to assign the iterator to point to a new element and a `valueChanged()` event passed up the subtree. As might be expected, as elements are deleted from the set, their corresponding subtrees are also removed.

5 Experiments with the sonet problem

As mentioned previously, the strength of operating directly on Essence types is that the Essence type constructors can convey information on the structure of the problem being solved. Referring to the Sonet problem presented in Figure 1, notice that we only have one abstract variable.

```
1 find network :
2   mset (size nrings) of set (maxSize capacity) of Nodes
```

The `size` attribute on the most outer type (multi set) forces the multi set to have a fixed size. This allows Athanor to drop all neighbourhoods that would attempt to change the size of the multi set. It therefore generates neighbourhoods that manipulate its elements. Since the inner type is a set of variable size, the neighbourhoods `setAdd`, `setRemove` are added. Along with this are `setSwap`, exchange one element in a set for another and `setAssignRandom`, assign the entire set to a new value. The `setSwap` and `setRemove` neighbourhoods are biased towards changing the sets that are most violating and furthermore, selecting the most violating elements within the set.

A simple search strategy is built upon these neighbourhoods. The solver runs in a loop: for each iteration, a random neighbourhood is executed and the change to the total constraint violation and objective are examined. If the violation count is decreased or if the objective is improved, the new solution is accepted. Otherwise, the change is reversed. This means that until the set of assignments form a feasible solution, the solver will accept any change to the objective (better or worse) provided that the set of assignments is brought closer to a feasible solution. As mentioned previously, the distance from a feasible solution is a heuristic. It is the total constraint violation count.

Once a feasible solution is found (the violation count reaches 0), the solver may only make progress by making changes that do not worsen the objective. If after a number of iterations [2] no improvement is observed, the solver relaxes the restriction on the violation count; accepting solutions if they improve the objective and allowing some constraints to be violated. In general, this procedure can be considered as being similar to hill climbing search procedures commonly found in local search solvers. Future work would focus on extending the set of search strategies to include other popular methods such as Simulated Annealing [15] or Tabu Search [13].

We performed experiments on a range of Sonet instances, comparing the performance of Athanor against other local search solvers that also automatically derive neighbourhoods. These were two variants of large neighbourhood search (LNS), propagation guided [20] and explanation guided [21], implemented with the Choco 4.0.6 solver as well as OscaR/CBLS [7,23] which derives neighbourhoods from constraints. The input to Oscar-CBLS was produced by using Conjure and Savile Row to refine the models to MiniZinc [16] and the MiniZinc instances were subsequently specialised for Oscar-CBLS's Minizinc backend [6] using MiniZinc 2.1.7. Care was taken to match the Choco and OscaR models as closely as possible.

Our results are summarised in table 1, the objective achieved after ten seconds and after ten minutes is shown, these values are the median of 10 runs. As can be seen, our solver always achieves the best performance both after ten seconds and after ten minutes.

Table 1: Minimising objective, after 10 seconds and 600 seconds. Best results for each time period are given in bold.

instance	Athanor		lns-eb		lns-pg		oscar	
	10s	600s	10s	600s	10s	600s	10s	600s
sonet1	**66.5**	**62.5**	72.5	65.0	85.0	72.5	75.5	72.5
sonet2	**182.5**	**117.0**	216.5	143.5	281.0	123.5	546.0	157.5
sonet3	**115.5**	**91.5**	132.5	104.5	153.0	101.0	256.5	121.5
sonet4	**184.0**	132.0	234.0	148.5	283.5	133.5	531.5	167.0
sonet5	**258.5**	**166.5**	394.5	199.0	474.5	176.0	843.0	227.5
sonet6	**294.0**	**178.0**	391.5	227.0	490.0	188.0	829.5	259.5
sonet7	**370.5**	**212.0**	534.0	285.5	956.0	253.5	1152.5	319.0
sonet8	**355.5**	**197.0**	704.5	261.5	1236.5	236.0	1379.0	295.0

[2] this is a tunable parameter to the solver

6 Conclusion

We have presented the benefits for having a solver operate directly on the abstract variables present in an Essence specification. The solver is able to utilise the type system to construct more intelligent neighbourhoods and is able to model the abstract types directly without having to resort to possibly expensive encodings. We have shown a framework for incremental evaluation of Essence ASTs as the abstract variables are manipulated during search. This includes the dynamic unrolling of quantifiers as variables are introduced during search. A brief discussion of how violations are typically calculated for boolean constraints is presented and how this has been extended to support variables with arbitrarily nested types. Finally, we benchmark a proof of concept solver, operating on the Sonet problem which uses the Essence types multi set and set. The solver outperforms other local search solvers (LNS and oscar-CBLS) who derive neighbourhoods from lower level encodings.

References

1. Akgün, Ö.: Extensible automated constraint modelling via refinement of abstract problem specifications. Ph.D. thesis, University of St Andrews (2014)
2. Akgün, Ö., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in Conjure. In: International Conference on Principles and Practice of Constraint Programming. pp. 107–116. Springer (2013)
3. Akgün, O., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P., Salamon, A., Spracklen, P.: A framework for constraint based local search using Essence. In: IJCAI (to appear) (2018)
4. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Breaking conditional symmetry in automated constraint modelling with Conjure. In: ECAI. pp. 3–8 (2014)
5. Akgün, Ö., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence. pp. 4–11. AAAI Press (2011)
6. Björdal, G., Monette, J.N., Flener, P., Pearson, J.: A constraint-based local search backend for MiniZinc. Constraints **20**(3), 325–345 (2015). https://doi.org/10.1007/s10601-015-9184-z
7. Codognet, P., Diaz, D.: Yet another local search method for constraint solving. In: SAGA. pp. 73–90. LNCS 2264, Springer (2001). https://doi.org/10.1007/3-540-45322-9_5
8. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The Essence of Essence. Modelling and Reformulating Constraint Satisfaction Problems pp. 73–88 (2005)
9. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of Essence: A constraint language for specifying combinatorial problems. In: IJCAI. pp. 80–87 (2007)
10. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)

11. Frisch, A.M., Hnich, B., Miguel, I., Smith, B.M., Walsh, T.: Transforming and refining abstract constraint specifications. In: International Symposium on Abstraction, Reformulation, and Approximation. pp. 76–91. Springer (2005)
12. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: ECAI. vol. 141, pp. 98–102 (2006)
13. Glover, F., Laguna, M.: Tabu search. In: Handbook of combinatorial optimization, pp. 2093–2229. Springer (1998)
14. Hentenryck, P.V., Michel, L.: Constraint-based local search. The MIT press (2009)
15. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. science **220**(4598), 671–680 (1983)
16. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: CP. pp. 529–543. LNCS 4741, Springer (2007)
17. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: CP. pp. 590–605. LNCS 8656, Springer (2014)
18. Nightingale, P., Akgün, O., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (2017). https://doi.org/10.1016/j.artint.2017.07.001
19. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In: CP. pp. 330–340. LNCS 9255, Springer (2015)
20. Perron, L., Shaw, P., Furnon, V.: Propagation guided large neighborhood search. In: CP. pp. 468–481. LNCS 3258, Springer (2004). https://doi.org/10.1007/978-3-540-30201-8_35, https://doi.org/10.1007/978-3-540-30201-8_35
21. Prud'homme, C., Lorca, X., Jussien, N.: Explanation-based large neighborhood search. Constraints **19**(4), 339–379 (2014). https://doi.org/10.1007/s10601-014-9166-6, https://doi.org/10.1007/s10601-014-9166-6
22. Smith, B.M.: Symmetry and search in a network design problem. In: International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming. pp. 336–350. Springer (2005)
23. Van Hentenryck, P., Michel, L.: Constraint-based local search. MIT Press (2005)

Maximal Frequent Itemset Mining with Non-monotonic Side Constraints

Gökberk Koçak, Özgür Akgün, Ian Miguel, and Peter Nightingale

School of Computer Science, University of St Andrews, St Andrews, UK
{gk34, ozgur.akgun, ijm, pwn1}@st-andrews.ac.uk

Abstract. Frequent itemset mining (FIM) is a method for finding regularities in transaction databases. It has several application areas, such as market basket analysis, genome analysis, and drug design. Finding frequent itemsets allows further analysis to focus on a small subset of the data. For large datasets the number of frequent itemsets can also be very large, defeating their purpose. Therefore, several extensions to FIM have been proposed in the literature, such as adding high-utility (or low-cost) constraints and only finding maximal frequent itemsets. In this paper we present a constraint programming based approach that combines arbitrary side constraints with maximal frequent itemset mining. We compare our approach with state-of-the-art algorithms via the MiningZinc system (where possible) and show significant contributions in terms of performance and applicability.

Keywords: Data mining · Pattern mining · Frequent itemset mining · Maximal frequent itemset mining · Constraint Modelling

1 Introduction

Frequent itemset mining (FIM) is a method for finding regularities in transaction databases. It has numerous application areas, such as market basket analysis, genome analysis, and drug design [1,17]. Finding frequent itemsets allows further analysis and human inspection to focus on a small subset of the data[10].

FIM is performed on transaction databases, where each transaction is a set of items. For a subset of items S, we define $support(S)$ to represent the number of transactions that have S as a subset. A frequent itemset is any S with $support(S) \geq t$, where t is the threshold of frequency. This threshold is often given as a percentage of the total number of transactions in the dataset.

The number of transactions and the number of items in a single transaction may vary greatly between application domains. For example, among the sixteen datasets listed on the CP4IM website[1] and hosted on the UCI Machine Learning Repository[8], the number of transactions range between 101 and 8124, and the number of items ranges between 27 and 287.

Lymphography [22] is a medium size dataset with 148 transactions and 68 items. Despite its relatively small size, there are nearly ten million frequent

[1] `https://dtai.cs.kuleuven.be/CP4IM/datasets/`

```
1  language Essence 1.3
2  letting ITEM be domain int(...)
3  given db : mset of set of ITEM
4  given min_support : int
5  given current_size : int
6  given solutions_so_far : set of (set of ITEM)
7  find fis : (set (size current_size) of ITEM)
8  such that
9    (sum entry in db . toInt(fis subsetEq entry)) >= min_support,
10   C(fis),
11   forAll sol in solutions_so_far .
12     !(fis subset sol)
```

Fig. 1. Essence specification for Maximal and Constrained Frequent Itemset Mining.

itemsets in this dataset (with $t = 10\%$). Extensions to FIM were proposed to reduce the number of frequent itemsets and to produce more focused results. There are broadly two categories of these extensions in the literature:

1. application-specific side constraints on the frequent itemsets
2. constraints between solutions, such as maximality or closedness.

The combination of these two kinds of constraints was explored in [6], where the employed algorithm needs to be carefully configured depending on the properties of the side constraints. In this paper, we present a constraint programming based declarative approach that works with arbitrary side constraints completely automatically. We focus on maximal frequent itemset mining since it is a significantly more difficult problem to solve. Our approach can be applied to closed frequent itemset mining with minor modifications.

Figure 1 presents an Essence [11,13,12] specification for maximal and frequent itemset mining, which we will use throughout this paper. Essence is a constraint specification language whose key feature is support for abstract decision variables, such as multisets, sets, functions, and relations, as well as nested types, such as multisets of relations, or the set of tuples present in the figure. Essence specifications are solved via a toolchain comprising the Conjure [3,4,5] and Savile Row [18] automated constraint modelling tools. Savile Row has multiple backends to accommodate solutions via constraint or SAT solvers. We used these backends and generated Minion and SAT models to solve. Our initial experiments show that generated SAT model on SAT solvers give better performance. Therefore, we employ the SAT solver `nbc_minisat_all`[20] in our experiments.

Contributions. In this paper we give a high-level declarative problem specification in Essence for the maximal frequent itemset mining problem and a

method that supports non-monotonic side constraints. Our method is completely declarative and automated, it does not require the user to make any decisions about which mining algorithm to employ. We present an exhaustive empirical study where we compare our method to several execution plans offered by MiningZinc [7,15]. We also construct non-trivial instances for the high-utility and low-cost maximal itemset mining problems at varying levels of frequency thresholds.

2 Extensions to Frequent Itemset Mining

Plain frequent itemset mining is concerned only with finding subsets of items that occur together in a transaction database. There are typically a large number of frequent itemsets and algorithms like Apriori, LCM and Eclat provide very efficient ways of enumerating them. For example, for the small dataset in Figure 2 with four items (in $\mathbb{I}$) and three transactions (in $\mathbb{T}$), there are ten frequent itemsets if the minimum support is 2.

Side constraints are often added to an itemset mining problem to provide focus on some itemsets. Some of these constraints are simple by enforcing or forbidding certain itemsets or putting lower/upper bounds on the cardinality of itemsets, etc. Constraints like these have been incorporated into existing frequent itemset solvers in the past. For example, LCMv2 [21] does not support itemset cardinality constraints, however a later version (LCMv5) added support for them. Adding support for new side constraints inside a dedicated itemset solver requires non-trivial reasoning in terms of its integration to the existing algorithm.

Constraints among solutions. Another class of extensions to frequent itemset mining is in the form of constraints among solutions. Maximality is a well-known constraint in this class[14]. A frequent itemset is maximal only if none of its supersets are frequent. This condition intuitively follows from the observation that if an itemset is frequent all of its subsets will also be frequent, and hence including them in the result set does not add value. In our running example with `min_support` 2, the maximal itemsets are $\{\{3,4\},\{1,2,4\}\}$.

Maximal and constrained itemset mining. Combining problem specific side constraints with constraints between solutions is appealing since this combination would provide the benefits of both classes of extensions. There is some ambiguity about what this combination might mean and this was one of the motivations of the method we develop here.

$$\mathbb{I} = \{1,2,3,4\}$$
$$\mathbb{T} = \{\{1,2,4\},\{1,2,3,4\},\{3,4\}\}$$
$$FIS = \{\{\},\{1\},\{2\},\{3\},\{4\},\{1,2\},\{1,4\},\{2,4\},\{3,4\},\{1,2,4\}\}$$

Fig. 2. A small database of transactions

There are two possible definitions for the combined problem: (1) all maximal frequent itemsets that also satisfy the side constraint (2) all frequent itemsets that satisfy the side constraint and are maximal within this solution set.

The former is strictly less useful since when we remove a maximal frequent itemset from the solution set due to the side constraint, we might also remove several of its subsets which are actually support the side constraint and doesn't have any supersets which does the same. We demonstrate the difference between the two definitions in Figure 3 using our running example with a minimum support value of 2, and a minimum cost threshold of 3. Starting from the same database, the two methods reach different sets of solutions. In the first table, the problem arises from removing the set $\{1, 2, 4\}$ and consequently losing all of its subsets from the solution set. This produces an incomplete set of solutions.

(Step 1) Database	(Step 2) Maximal Itemsets	(Step 3) Maximal and Low-Cost
$\{\{1,2,4\}, \{1,2,3,4\}, \{3,4\}\}$	$\{\{3,4\}, \{1,2,4\}\}$	$\{\{3,4\}\}$

(Step 1) Database	(Step 2) Frequent Itemsets	(Step 3) Low-Cost	(Step 4) Low-Cost and Maximal
$\{\{1,2,4\}, \{1,2,3,4\}, \{3,4\}\}$	$\{\{\}, \{1\}, \{2\}, \{3\}, \{4\}, \{1,2\}, \{1,4\}, \{2,4\}, \{3,4\}, \{1,2,4\}\}$	$\{\{\}, \{1\}, \{2\}, \{3\}, \{4\}, \{1,2\}, \{1,4\}, \{2,4\}, \{3,4\}\}$	$\{\{1,2\}, \{1,4\}, \{2,4\}, \{3,4\}\}$

Fig. 3. The difference between the order of application of side constraints and the maximality constraint.

This problem only occurs when the side constraint is non-monotonic, as explained in [6]. A side constraint C is monotone when for any two frequent itemsets a and b with $a \subset b$, $C(a) \implies C(b)$. The two side constraints that we consider in this paper are high-utility (monotone) and low-cost (not monotone).

The combination of these two kinds of constraints was explored in [6], where the employed algorithm needs to be carefully configured depending on properties of the side constraints. Hence, using a frequent itemset mining algorithm like LCM or Eclat to find all maximal frequent itemsets followed by a post process to filter those that don't satisfy the side constraint will lose solutions. In addition to MiningZinc and specialised algorithms, there are two recent studies [16] and [19] which propose global constraints for the closure constraint. However, these papers also propagate for the constraint between solutions first to lose some information by applying side constraints on post-processing.

In this paper, we present a constraint programming based declarative approach that works with non-monotonic side constraints completely automatically.

3 Maximal FIM with non-monotonic side constraints

Our approach to maximal itemset mining (with or without side constraints) is iterative. We first find frequent itemsets of the largest cardinality possible, that of the largest transaction. For the largest cardinality, all frequent itemsets are guaranteed to be also maximal frequent itemsets, since they have no frequent supersets that can rule them outside of the maximal set. We then iteratively decrement the cardinality by one and solve for all frequent itemsets. At each iteration we also produce an *exclusion constraint* (Lines 11–12 of Figure 1), which ensures that all solutions found at a certain iteration are maximal with respect to solutions found at previous iterations. This approach is sound, i.e. we do not produce any frequent itemsets that are not maximal, since the maximal property of an itemset only depends on larger frequent itemsets. The approach is also complete since we enumerate all solutions for every possible itemset cardinality.

Algorithm 1 Iterative Maximal and Constraint Itemset Mining with CP

```
1: procedure IterativeMiner(D)
2:     solutions_so_far ← {}
3:     size ← ub
4:     while size ≥ 0 do
5:         solutions ← Solve(D, solutions_so_far, size)
6:         solutions_so_far ← solutions_so_far ∪ solutions
7:         size ← size − 1
8:     end while
9:     return solutions_so_far
10: end procedure
```

For a given problem class Conjure is called once to refine the Essence specification in Figure 1 into a constraint model in Essence Prime. Conjure produces an Occurrence model [2] for the set variable, which uses an array of Boolean decision variables for each item. SavileRow employs the Minion constraint solver as a preprocessing step that achieves singleton arc consistency on the lower and upper bounds of decision variables. This step has the potential to reduce the number of iterations considerably by reducing the range of itemset cardinalities that need to be considered. Thereafter, each iteration is performed as described in Algorithm 1.

For the upper bound *ub*, we use the cardinality of the largest maximal frequent itemset as our starting point if it is obvious. Otherwise, we start from the cardinality of the widest transaction in the database.

Inside the `Solve` procedure (Line 6) SavileRow translates the model into SAT. We selected a SAT solver due to the large number of Boolean variables in the model and the relatively simple constraints. We use `nbc_minisat_all` [20], which is an AllSAT solver based on MiniSAT [9]. SavileRow runs the solver at each level, collects the results, and modifies the SAT encoding by adding new exclusion constraints.

```
1 given cost_values : matrix indexed by [ITEM] of int(0..5)
2 given max_cost : int
3 given utility_values : matrix indexed by [ITEM] of int(0..5)
4 given min_utility : int
5 such that
6  (sum item in fis . cost_values[item]) <= max_cost
7  (sum item in fis . utility_values[item]) >= min_cost
```

Fig. 4. Essence specification for High Utility and Low-Cost constraint.

4 Empirical evaluation

Our experiment uses a low-cost constraint in addition to the high-utility constraint to have a non-monotonic constrained model in the end. The low-cost constraint can be written in the MiningZinc language in a similar way to the high-utility constraint in the model of Figure 8 of [15]. MiningZinc produces a number of execution plans when provided with this model. However, all of these execution plans produce faulty answers. The low-cost constraint is not monotonic, so the MiningZinc execution plans suffer from the problem we describe in Section 2.

Experiments were performed with 16 processes in parallel on a 32-core AMD Opteron 6272 at 2.1 GHz with 256 GB RAM. We modified the MiningZinc source code to use a different temporary directory for each of its invocation. By default MiningZinc uses a fixed directory for its temporary files, which precludes us from running multiple MiningZinc processes at the same time.

All of these execution plans either calculate maximal itemsets first and then apply side constraints, or contain the maximality constraint inside a constraint model and apply it simultaneously with the side constraints.

For sixteen datasets and five different frequency levels we construct 80 instances. With a 3-hour time limit, We also run all execution plans produced by MiningZinc, and 12 plans out of 18 produce an incorrect number of solutions for at least one instance even without maximality constraint for the reasons are unknown to us. We exclude these execution plans from our comparison.

4.1 Maximal High-Utility and Low-Cost Itemset Mining

Table 1 gives the runtimes of several MiningZinc execution plans and our method (Essence Mining). In square brackets we indicate the Gecode option used in a given execution plan generated by MiningZinc, where F represents a model rewriting called `freq_items`, and R represents another called `reify`. FI indicates a preprocessing step called `FreqItems` that runs before Gecode.

The Essence Mining column contains time taken by our method, including modelling overhead. We compare this runtime against the runtimes of the MiningZinc execution plans and indicate the winner using a bold font. In addition, we provide the time taken by only the AllSAT solver for our method in the Essence Mining (Solver Time) column. In general, the modelling overhead is small. However in some cases (for example in the audiology dataset) there is a significant difference between the total time and the solver time. On the harder instances of the `hypothyroid` dataset our method is the only one that finishes before the time limit.

We experiment with a combined high-utility and low-cost itemset mining problem using the frequency thresholds (10%, 20%, 30%, 40% and 50%). The cost/utility values are uniformly randomly chosen to a value between 0 and 5. A cost threshold and a utility threshold is chosen to limit the number of maximal frequent itemsets to at most tens of thousands of maximal frequent itemsets. These thresholds are chosen independently of the frequency value. We add four more parameters and arithmetic constraints as listed in Figure 4.

We extended the MiningZinc model (from Figure 8 of [15]). Due to the issue that was explained in Section 2 the extended model gives fewer results: it misses a large number of solutions. Hence, comparing our performance to this model is not sensible.

Following the analysis of [6] we decided to relax the maximality condition for MiningZinc and find all frequent itemsets that satisfy the side constraints. In other words, we use MiningZinc to perform the first three steps in the second table of Figure 3. To achieve the same results as our method another procedure would also be needed. We verified the output of MiningZinc experiments by expanding our maximal frequent itemsets to full frequent itemsets.

5 Conclusion

In this paper we have presented a high-level declarative problem specification in Essence for the maximal frequent itemset mining problem and a method that supports non-monotonic side constraints. Our method is completely declarative and automated, it does not require the user to make any decisions about which mining algorithm to employ. To the best of our knowledge this is the first declarative method of performing the maximal frequent itemset mining task with arbitrary side constraints (whether they are monotone or not).

We tested our method against all execution plans offered by MiningZinc for two different kinds of side constraints: high utility and low cost. Combining both of these constraints is not monotonic and our approach proves to be very effective in handling it.

Instance	Gecode	Gecode [F]	Gecode [FR]	Gecode [R]	FI + Gecode [F]	FI + Gecode [FR]	Essence Mining	Essence Mining (Solver Time)	Number of Solutions
lymph-50	15	**5**	10	10	**5**	9	13	0	477
lymph-40	33	**8**	23	23	9	20	15	0	351
lymph-30	103	29	93	91	32	80	**15**	2	485
lymph-20	194	60	202	203	64	188	**19**	4	865
lymph-10	920	526	987	1,263	**447**	1,002	1,088	24	3,590
krvskp-50	*	3,896	282	281	3,911	**274**	517	441	136
krvskp-40	*	6,774	794	792	6,901	**761**	1,121	1,029	287
krvskp-30	*	*	3,057	3,057	*	**2,971**	4,652	2,278	885
krvskp-20	*	*	*	*	*	*	**9,414**	5,344	4,057
krvskp-10	*	*	*	*	*	*	*	*	*
hypo-50	*	*	*	*	*	*	**4,424**	1,792	2,823
hypo-40	*	*	*	*	*	*	**1,481**	1,251	254
hypo-30	*	*	*	*	*	*	**4,374**	1,921	415
hypo-20	*	*	*	*	*	*	**6,154**	2,860	7
hypo-10	*	*	*	*	*	*	**5,726**	2,325	90
hepatisis-50	15	**9**	15	15	**9**	15	16	3	1,351
hepatisis-40	67	37	72	73	39	68	**30**	11	3,919
hepatisis-30	322	195	256	254	200	237	**51**	27	5,866
hepatisis-20	1,862	1,402	1,192	1,191	1,452	1,119	**88**	61	7,344
hepatisis-10	10,368	8,614	3,283	3,356	9,457	2,962	**209**	165	8,969
heart-50	55	**16**	18	17	**16**	**16**	27	9	353
heart-40	254	107	122	120	110	113	**69**	45	1,915
heart-30	1,469	760	521	516	774	485	**269**	241	3,390
heart-20	*	8,219	7,561	7,585	8,082	6,564	**2,089**	1,350	21,941
heart-10	*	*	*	*	*	*	**4,178**	1,844	7,812
german-50	98	36	**23**	**23**	35	**23**	59	28	146
german-40	354	136	127	128	140	120	**107**	70	655
german-30	1,479	662	893	883	671	844	**286**	238	2,466
german-20	7,049	3,532	6,371	6,369	3,618	5,683	**889**	806	9,680
german-10	*	*	6,780	7,288	*	6,485	**5,344**	2,192	2,198
australian-50	479	200	182	177	204	177	**140**	108	1,040
australian-40	2,860	1,688	1,182	1,174	1,764	1,094	**404**	365	2,253
australian-30	*	*	10,672	10,647	*	10,272	**3,049**	1,833	7,228
australian-20	*	*	*	*	*	*	**8,634**	5,365	10,722
australian-10	*	*	*	*	*	*	*	*	*
audiology-50	*	*	1,123	1,115	*	1,025	**62**	9	2,052
audiology-40	*	*	436	433	*	370	**59**	5	2,235
audiology-30	*	4,688	423	417	4,751	349	**56**	5	1,217
audiology-20	*	*	1,940	1,950	*	1,647	**34**	6	1,021
audiology-10	*	*	5,081	5,113	*	4,591	**38**	9	1,098
anneal-50	*	*	*	*	*	*	**105**	69	1,598
anneal-40	*	*	*	*	*	*	**188**	147	2,792
anneal-30	*	*	*	*	*	*	**155**	130	2,069
anneal-20	*	*	*	*	*	*	**303**	275	2,483
anneal-10	*	*	*	*	*	*	**452**	421	3,430

Table 1. Maximal High-Utility and Low-Cost Itemset Mining on 9 datasets. Times are in seconds (* indicates a 3-hour timeout).

References

1. Agrawal, R., Imieliński, T., Swami, A.: Mining association rules between sets of items in large databases. In: Acm sigmod record. vol. 22, pp. 207–216. ACM (1993)
2. Akgün, Ö.: Extensible automated constraint modelling via refinement of abstract problem specifications. Ph.D. thesis, University of St Andrews (2014)
3. Akgün, Ö., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in conjure. In: International Conference on Principles and Practice of Constraint Programming. pp. 107–116. Springer (2013)
4. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Breaking conditional symmetry in automated constraint modelling with conjure. In: ECAI. pp. 3–8 (2014)
5. Akgün, Ö., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence. pp. 4–11. AAAI Press (2011)
6. Bonchi, F., Lucchese, C.: On closed constrained frequent pattern mining. In: Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on. pp. 35–42. IEEE (2004)
7. De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 204–212. ACM (2008)
8. Dheeru, D., Karra Taniskidou, E.: UCI machine learning repository (2017), `http://archive.ics.uci.edu/ml`
9. Een, N.: Minisat: A sat solver with conflict-clause minimization. In: Proc. SAT-05: 8th Int. Conf. on Theory and Applications of Satisfiability Testing. pp. 502–518 (2005)
10. Fournier-Viger, P., Lin, J.C.W., Vo, B., Chi, T.T., Zhang, J., Le, H.B.: A survey of itemset mining. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery **7**(4) (2017)
11. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The essence of essence. Modelling and Reformulating Constraint Satisfaction Problems p. 73 (2005)
12. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of essence: A constraint language for specifying combinatorial problems. In: IJCAI. vol. 7, pp. 80–87 (2007)
13. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)
14. Gouda, K., Zaki, M.J.: Efficiently mining maximal frequent itemsets. In: Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on. pp. 163–170. IEEE (2001)
15. Guns, T., Dries, A., Nijssen, S., Tack, G., De Raedt, L.: Miningzinc: A declarative framework for constraint-based mining. Artificial Intelligence **244**, 6–29 (2017)
16. Lazaar, N., Lebbah, Y., Loudni, S., Maamar, M., Lemière, V., Bessiere, C., Boizumault, P.: A global constraint for closed frequent pattern mining. In: International Conference on Principles and Practice of Constraint Programming. pp. 333–349. Springer (2016)
17. Naulaerts, S., Meysman, P., Bittremieux, W., Vu, T.N., Vanden Berghe, W., Goethals, B., Laukens, K.: A primer to frequent itemset mining for bioinformatics. Briefings in bioinformatics **16**(2), 216–231 (2013)

18. Nightingale, P., Akgün, O., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (2017). https://doi.org/http://dx.doi.org/10.1016/j.artint.2017.07.001
19. 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)
20. Toda, T., Soh, T.: Implementing efficient all solutions sat solvers. Journal of Experimental Algorithmics (JEA) **21**, 1–12 (2016)
21. Uno, T., Kiyomi, M., Arimura, H.: Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In: Fimi. vol. 126 (2004)
22. Zwitter, M., Soklic, M.: Lymphography domain. University Medical Center, Institute of Oncology, Ljubljana, Yugoslavia (1988)

A Global Constraint for Mining Generator Itemsets

Mohamed-Bachir Belaid[1], Christian Bessiere[1], Nadjib Lazaar[1]

[1] LIRMM, University of Montpellier, CNRS, Montpellier, France
{belaid, bessiere, lazaar}@lirmm.fr

Abstract. In itemset mining, the notion of *generator* is present in several tasks such as mining frequent itemsets, association rules, etc. It has recently been shown that constraint programming is a flexible way to tackle data mining tasks. In this paper we propose a global constraint Generator for mining generator itemsets. We provide a polynomial complete propagator for Generator proving that it achieves domain consistency.

1 Introduction

In itemset mining, generator itemsets are used to capture the minimality property. An itemset is a generator if there does not exists any subset with the same frequency. Generator itemset were first introduced in [1] for efficiently mining frequent itemsets. In [7] frequent generators are used for mining the *minimal non-redundant association rules.* For efficiently mining *minimal rare itemsets* in [8] the notion of generator is used.

In a recent line of work [2–4, 6], constraint programming (CP) has been used as a declarative way to solve some data mining tasks, such as itemset mining or sequence mining. Such an approach can not yet compete the state of the art data mining algorithms in terms of CPU time for standard data mining queries but the CP approach is competitive as soon as we need to add user's constraints. In addition, adding constraints is easily done by specifying the constraints directly in the model without the need to revise the solving process.

In this paper we propose a new global constraint, called Generator, for mining itemsets that are generators. We propose a polynomial filtering algorithm for Generator. We prove that this algorithm achieves domain consistency. The paper is organized as follows. Section 2 gives some background material. In section 3 we present our new global constraint Generator. We conclude in section 4.

2 Background

2.1 Itemsets

Let $\mathcal{I} = \{1, \ldots, n\}$ be a set of n *item* indices and $\mathcal{T} = \{1, \ldots, m\}$ a set of m *transaction* indices. An itemset P is a subset of $\mathcal{I}$. $\mathcal{D}$ is the transactional

Table 1: Transaction dataset example with six items and five transactions.

trans.	Items
t_1	A B
t_2	A C D E
t_3	B C D F
t_4	A B C D
t_5	A B C F

dataset, where, $\mathcal{D} \subseteq \mathcal{I} \times \mathcal{T}$. The cover of an itemset P, denoted by $cover(P)$, is the set of transactions containing P. The (relative) frequency of an itemset P is $freq(P) = \frac{|cover(P)|}{|\mathcal{T}|}$.

Definition 1 (Generator [1]) *A* generator *is an itemset P such that there does not exist any itemset $Q \subsetneq P$ such that $freq(Q) = freq(P)$.*

Example 1. Consider the transaction dataset presented in Table 1. The itemset AC is a generator because $freq(AC) = 60\%$ and none of its subsets ($\emptyset$, A, C) have the same frequency ($freq(\emptyset) = 100\%$, $freq(A) = freq(C) = 80\%$). The itemset CD is not a generator because it has the same frequency as one of its subsets: $freq(CD) = freq(D) = 60\%$.

2.2 Constraint Programming

A constraint programming model specifies a set of variables $X = \{x_1, \ldots, x_n\}$, a set of domains $dom = \{dom(x_1), \ldots, dom(x_n)\}$, where $dom(x_i)$ is the finite set of possible values for x_i, and a set of constraints $\mathcal{C}$ on X. A constraint $c_j \in \mathcal{C}$ is a relation that specifies the allowed combinations of values for its variables $var(c_j)$. An assignment on a set $Y \subseteq X$ of variables is a mapping from variables in Y to values, and a valid assignment is an assignment where all values belong to the domain of their variable. A solution is an assignment on X satisfying all constraints. The constraint satisfaction problem (CSP) consists in deciding whether an instance of a CP model has solutions (or in finding a solution). Constraint programming is the art of writing problems as CP models and solving them by finding solutions. Constraint solvers typically use backtracking search to explore the search space of partial assignments. At each assignment, constraint propagation algorithms prune the search space by enforcing local consistency properties such as domain consistency.

Domain Consistency (DC). A constraint c on $X(c)$ is domain consistent, if and only if, for every $x_i \in X(c)$ and every $d_j \in dom(x_i)$, there is a valid assignment satisfying c such that $x_i = d_j$.

Global constraints are constraints defined by a relation on any number of variables. These constraints allow the solver to better capture the structure of the problem. Examples of global constraints are AllDifferent, Regular, Among, etc. (see [5]). Some global constraints (such as Regular) allow a decomposition

preserving DC. For the other global constraints, it is not possible to efficiently propagate them by generic DC algorithms because these algorithms are exponential in the number of variables of the constraint. Fortunately, for many global constraints (such as Alldifferent), dedicated propagation algorithms can be designed to achieve DC in time polynomial in the size of the input, that is, the domains and the required extra parameters.

3 The Global Constraint Generator

In this section we introduce Generator a new global constraint for mining itemsets that are generators (see Definition 1).

We introduce a vector x of n Boolean variables, where x_i represents the presence of item i in the itemset. In the rest of the paper we will use the following notations:

- $x^{-1}(1) = \{i \in \mathcal{I} \mid x_i = 1\}$
- $x^{-1}(0) = \{i \in \mathcal{I} \mid x_i = 0\}$

Definition 2 (Generator constraint) *Let x be a vector of Boolean variables and $\mathcal{D}$ be a dataset. The global constraint* Generator$_{\mathcal{D}}(x)$ *holds if and only if $x^{-1}(1)$ is a generator.*

The propagator we propose for the Generator constraint is based on the following property of generators.

Proposition 1 *Given two itemsets P and Q, if P is not a generator and $P \subsetneq Q$, then Q is not a generator.*

Proof. Derived from Theorem 2 in [1], where we set frequency to 0. □

Algorithm. The propagator Filter-Generator for the global constraint Generator is presented in Algorithm 1. Filter-Generator takes as input the variables x. Filter-Generator starts by computing the cover of the itemset $x^{-1}(1)$ and stores it in `cover` (line 3). Then, for each item $j \in x^{-1}(1)$, Filter-Generator computes the cover of the subset $x^{-1}(1)\backslash\{j\}$, and stores it in `cov[j]` (lines 4-5). Filter-Generator can then remove items i that cannot belong to a generator containing $x^{-1}(1)$. To do that, for every item j in $x^{-1}(1) \cup \{i\}$, we compare the cover of $x^{-1}(1) \cup \{i\}$ (i.e., `cover` $\cap$ $cover(i)$) to the cover of $x^{-1}(1) \cup \{i\} \setminus \{j\}$ (i.e., `cov[j]` $\cap$ $cover(i)$) (line 8). If they have equal size (i.e., same frequency), we remove i from the possible items, that is, we remove 1 from $dom(x_i)$ and break the loop (line 9).

Theorem 1. *The propagator* Filter-Generator$_{\mathcal{D}}$ *enforces domain consistency.*

Proof. We first prove that the value 0 for a variable x_i such that $i \notin (x^{-1}(1) \cup x^{-1}(0))$ always belongs to a solution of the constraint Generator, and so cannot be pruned by domain consistency. Suppose $i \notin x^{-1}(1) \cup x^{-1}(0)$. If $x^{-1}(1)$ is

Algorithm 1: FILTER-GENERATOR$_{\mathcal{D}}$ (x)

```
1 InOut: x = {x_1 ... x_n}: Boolean item variables;
2 begin
3     cover ← cover(x^{-1}(1));
4     foreach j ∈ x^{-1}(1) do
5         cov[j] ← cover(x^{-1}(1) \ {j});
6     foreach i ∉ x^{-1}(1) ∪ x^{-1}(0) do
7         foreach j ∈ x^{-1}(1) ∪ {i} do
8             if |cover ∩ cover(i)| = |cov[j] ∩ cover(i)| then
9                 dom(x_i) ← dom(x_i) \ {1}; break;
```

a generator, removing the value 0 from $dom(x_i)$ increases $x^{-1}(1)$ to $x^{-1}(1) \cup \{i\}$, and then $x^{-1}(1)$ cannot be returned as a generator, which contradicts the hypothesis. Suppose now that $x^{-1}(1)$ is not a generator. We know from Proposition 1 that for any $Q \supsetneq x^{-1}(1)$, Q is not a generator. Thus, $x^{-1}(1) \cup \{i\}$ cannot belong to any generator, and value 0 cannot be pruned from $dom(x_i)$.

We now prove that FILTER-GENERATOR prunes value 1 from $dom(x_i)$ exactly when i cannot belong to a generator containing $x^{-1}(1)$. Suppose value 1 of x_i is pruned by FILTER-GENERATOR. This means that the test in line 8 was true, that is, there exists a sub-itemset of $x^{-1}(1) \cup \{i\}$ with the same frequency as $x^{-1}(1) \cup \{i\}$. Thus, by definition, $x^{-1}(1) \cup \{i\}$ does not belong to any generator. Suppose now that value 1 of x_i is not pruned. From line 8, we deduce that there does not exist any subset of $x^{-1}(1) \cup \{i\}$ with the same frequency as $x^{-1}(1) \cup \{i\}$. Thus $x^{-1}(1) \cup \{i\}$ is a generator and value 1 of x_i is domain consistent. □

Theorem 2. *Given a transaction dataset $\mathcal{D}$ of n items and m transactions, the algorithm* FILTER-GENERATOR$_{\mathcal{D}}$ *has an $O(n^2 \times m)$ time complexity.*

Proof. Computing the size of the cover of an itemset is in $O(n \times m)$. Line 5 is called at most n times, leading to a time complexity of $O(n^2 \times m)$. The test at line 8 is done at most n^2 times. The covers of $x^{-1}(1) \cup \{i\}$ and $x^{-1}(1) \cup \{i\} \setminus \{j\}$ at line 8 are computed in $O(m)$ thanks to the `cover` and `cov` data structures. Thus, the time complexity of lines 6-9 is bounded above by $O(n^2 \times m)$. As a result, FILTER-GENERATOR has an $O(n^2 \times m)$ time complexity. □

Note that without the use of the `cov` structure (that is, by recomputing $cover(x^{-1}(1) \setminus \{j\})$ at each execution of the loop at line 6), the time complexity becomes $O(n^3 \times m)$. However, this version is less memory consuming and can be more efficient in practice. It is also important to stress that domain consistency on GENERATOR does not depend on $x^{-1}(0)$. Thus, FILTER-GENERATOR is not called during the resolution when a variable is instantiated to zero.

4 Conclusion

We have introduced in this paper a new global constraint for mining generator itemsets. For that we have proposed a polynomial propagator achieving domain consistency. The global constraint can easily be used for modelling many data mining problems.

References

1. Bastide, Y., Taouil, R., Pasquier, N., Stumme, G., Lakhal, L.: Mining frequent patterns with counting inference. ACM SIGKDD Explorations Newsletter **2**(2), 66–75 (2000)
2. De Raedt, L., Guns, T., Nijssen, S.: Constraint programming for itemset mining. In: Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. pp. 204–212. ACM (2008)
3. Khiari, M., Boizumault, P., Crémilleux, B.: Constraint programming for mining n-ary patterns. In: International Conference on Principles and Practice of Constraint Programming. pp. 552–567. Springer (2010)
4. Lazaar, N., Lebbah, Y., Loudni, S., Maamar, M., Lemière, V., Bessiere, C., Boizumault, P.: A global constraint for closed frequent pattern mining. In: International Conference on Principles and Practice of Constraint Programming. pp. 333–349. Springer (2016)
5. Rossi, F., Van Beek, P., Walsh, T.: Handbook of constraint programming. Elsevier (2006)
6. 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)
7. Szathmary, L., Napoli, A., Kuznetsov, S.O.: Zart: A multifunctional itemset mining algorithm. In: 5th International Conference on Concept Lattices and Their Applications (CLA'07). pp. 26–37 (2007)
8. Szathmary, L., Napoli, A., Valtchev, P.: Towards rare itemset mining. In: Tools with Artificial Intelligence, 2007. ICTAI 2007. 19th IEEE International Conference on. vol. 1, pp. 305–312. IEEE (2007)

Sub-domain Selection Strategies For Floating Point Constraint Systems *

Heytem Zitoun[1], Claude Michel[1], Michel Rueher[1], and Laurent Michel[2]

[1] Université Côte d'Azur, CNRS, I3S, France
firstname.lastname@i3s.unice.fr
[2] University of Connecticut, Storrs, CT 06269-2155
ldm@engr.uconn.edu

Abstract. Program verification is a key issue for critical applications such as aviation, aerospace, or embedded systems. Bounded model checking (BMC) and constraint programming (CBMC, CBPV, ...) approaches are based on counter-examples that violate a property of the program to verify. Searching for such counter-examples can be very long and costly when the programs to check contains floating point computations. This stems from the fact that existing search strategies have been designed for discrete domains and, to a lesser extent, continuous domains. In [12], we have introduced a set of variable choice strategies that take advantages of the specificities of the floats, e.g., domain density, cancellation and absorption phenomena. In this paper we introduce new sub-domain selection strategies targeting domains involved in absorption and using techniques derived from higher order consistencies. Preliminary experiments on a significant set of benchmarks are very promising.

1 Introduction

Programs with floating-point computations control complex and critical systems in numerous domains, including cars and other transportation systems, nuclear energy plants, or medical devices. Floating-point computations are derived from mathematical models on real numbers [8], but computations on floating-point numbers are different from computations on real numbers. For instance, with binary floating-point numbers, some real numbers cannot be represented (e.g., 0.1 does not have an exact representation). Floating point arithmetic operators are neither associative nor distributive, and may be subject to phenomena such as absorption and cancellation. Furthermore, the behavior of programs containing floating-point computations varies with the programming language, the compiler, the operating system, or the hardware architecture.

Figure 1 illustrates how the flow of a very simple program over the floats ($\mathbb{F}$) can differs from the expected flow over the reals ($\mathbb{R}$). When interpreting the program over reals, the instruction `doThenPart` should be executed. However, an absorption on the floats (the value 1 is absorbed by `1e8f`[3]) leads the program through the `else` branch.

* This work was partially supported by ANR COVERIF (ANR-15-CE25-0002).

[3] On simple precision and with rounding set to "to the nearest even".

```
void foo(){
  float a = 1e8f;
  float b = 1.0f;
  float c = -1e8f;
  float r = a + b + c;
  if(r >= 1.0f){
    doThenPart();
  } else {
    doElsePart();
  }
}
```

Fig. 1. Motivation example

In [12], we have introduced a set of *variable selection strategies* based on specific properties of floats like domain density, cancellation and absorption phenomena. The resulting search strategies are much more efficient but do not really scale for harder and more realistic benchmarks. Indeed, like in other applications of constraint techniques, efficient solvers not only requires appropriate variable selection strategies but also need relevant value selection strategies. So, this paper focuses on *value selection strategies* for floating-point constraint solvers dedicated to the search of counter-examples in program verification applications.

Standard value selection strategies over the floats are derived from *sub-domain selection strategies* used over the reals; sub-domains being generated by using various splitting techniques, eg, $x \leq v$ or $x > v$ with $v = \frac{x+\overline{x}}{2}$.

In this paper we introduces four new sub-domain selection strategies. The first one, exploits absorption phenomena, the second one embraces ideas derived from strong consistency and the two last ones extend strategies introduced in [12]. We have evaluated these new sub-domain selection strategies on a significant set of benchmarks originate with program verification. We implemented a set of over 300 search strategies that are combinations of variable selection strategies previously introduced, sub-domain selection strategies presented in the following pages and variations of different criteria like filtering. All strategies were implemented in Objective-CP, the optimization tool introduced in [11].

In summary, the contributions are *new sub-domain selection strategies* dedicated to float system.

The rest of this article is organized as follows. Section 2 presents some notations, and definitions necessary for understanding this document. Section 3 provides a brief reminder of the strategies presented in [12]. Section 4 explains the new splitting strategies we propose. Section 5 is devoted to an analysis of the experimental results. Finally, Section 6 discusses the work in progress and the perspectives.

2 Notations and definitions

2.1 Floating point numbers

Floating point numbers approximate real numbers. The IEEE754-2008 standard for floating point numbers [9] sets floating point formats, as well as, some floating point arithmetic properties. The two most common formats defined in the IEEE754 standard are *simple* and *double* floating point number precision which, respectively, use 32 bits and 64 bits. A floating point number is a triple (s, m, e) where $s \in \{0, 1\}$ represents the sign, the mantissa m (also called significant), which is p bits long, and, e the exponent [8]. A *normalized* floating point number is defined by:

$$(-1)^s 1.m \times 2^e$$

To allow gradual underflow, IEEE754 introduces denormalized numbers whose value is given by:

$$(-1)^s 0.m \times 2^0$$

Note that simple precision are represented with 32 bits and a 23 bits mantissa ($p = 23$) while doubles use 64 bits and a 52 bits mantissa ($p = 52$).

2.2 Absorption

Absorption occurs when adding two floating point numbers with different order of magnitude. The result of such an addition is the furthest from zero. For instance, in C, using simple floating point numbers with a rounding mode set "to the nearest even", $10^8 + 1.0$ evaluates to 10^8. Thus, 1.0 is absorbed by 10^8.

2.3 Notations

In the sequel, x, y and z denote variables and $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$, their respective domains. When required, $x_{\mathbb{F}}$, $y_{\mathbb{F}}$ and $z_{\mathbb{F}}$ denote variables over $\mathbb{F}$ and $\mathbf{x}_{\mathbb{F}}$, $\mathbf{y}_{\mathbb{F}}$ and $\mathbf{z}_{\mathbb{F}}$, their respective domains while $x_{\mathbb{R}}$, $y_{\mathbb{R}}$ and $z_{\mathbb{R}}$ denote variables over $\mathbb{R}$ and $\mathbf{x}_{\mathbb{R}}$, $\mathbf{y}_{\mathbb{R}}$ and $\mathbf{z}_{\mathbb{R}}$, their respective domains. Note that $\mathbf{x}_{\mathbb{F}} = [\underline{x}_{\mathbb{F}}, \overline{x}_{\mathbb{F}}] = \{x_{\mathbb{F}} \in \mathbb{F}, \underline{x}_{\mathbb{F}} \leq x_{\mathbb{F}} \leq \overline{x}_{\mathbb{F}}\}$ with $\underline{x}_{\mathbb{F}} \in \mathbb{F}$ and $\overline{x}_{\mathbb{F}} \in \mathbb{F}$. Likewise, $\mathbf{x}_{\mathbb{R}} = [\underline{x}_{\mathbb{R}}, \overline{x}_{\mathbb{R}}] = \{x_{\mathbb{R}} \in \mathbb{R}, \underline{x}_{\mathbb{R}} \leq x_{\mathbb{R}} \leq \overline{x}_{\mathbb{R}}\}$ with $\underline{x}_{\mathbb{R}} \in \mathbb{F}$ and $\overline{x}_{\mathbb{R}} \in \mathbb{F}$. Let $x_{\mathbb{F}} \in \mathbb{F}$, then $x_{\mathbb{F}}^+$ is the smallest floating point number strictly superior to $x_{\mathbb{F}}$ and $x_{\mathbb{F}}^-$ is the biggest floating point number strictly inferior to $x_{\mathbb{F}}$. In a similar way, $x_{\mathbb{F}}^{+[N]}$ is the N^{th} floating point strictly superior to $x_{\mathbb{F}}$ and $x_{\mathbb{F}}^{-[N]}$ is the N^{th} floating point strictly inferior to $x_{\mathbb{F}}$. In addition, given a constraint c, $vars(c)$ denotes the set of floating point variables appearing in c. Finally, given a set s, $|s|$ denotes the cardinality of s.

3 Search strategies based on floating-point properties

In [12], we have introduced a set of *variable selection strategies* based on specific properties of floats like domain density, cancellation and absorption phenomena. The resulting search strategies are much more efficient but do not really scale for harder and more realistic benchmarks.

Fig. 2. Sub-domains generated by *splitAbs* ($\underline{x} > 0$ and $\underline{y} > 0$)

4 Sub-domain selection strategies

In this section we introduce four new sub-domain selection strategies. The first one takes advantage of absorption, the second is derived from strong consistency filtering techniques and tries to reduce the domain at a limited cost.The two last ones generalize sub-domain selection strategies presented in [26].

4.1 Absorption-based strategy

Let's recall that absorption occurs when adding two floating point numbers with different order of magnitude. The result of such an addition is the number the furthest from zero.

The objective of the *splitAbs* strategy is to concentrate on the most relevant absorption phenomena, in other words, giving priority to the sub-domains of x and y most likely to lead to an absorption.

Before going into the details of this absorption-based sub-domain selection strategy, let us recall the key points of $MaxAbs$, the variable strategy introduced in [12]. $MaxAbs$ is a variable selection strategy that picks the variable "absorbing the most". More precisely, this variable selection strategy needs to branch on two variables involved in absorption. The first variable –represented by `x` in Figure 2– must have the highest absorption rate. After selection of variable `x`, strategy $MaxAbs$ examines addition and subtraction constraints to select a variable `y`, the values of which are most absorbed by the values of by `x`. Coordinated branching on these variables is performed to exploit the latent absorption.

Sub-domain selection strategy : *splitAbs* Once these two variables are selected, the sub-domain selection heuristic will perform at most three splits on each variable. Figure 2 illustrates an instance where two sub-domains are generated (match with the case where domains are positive). It's easy to extend it to the others cases by symmetry. In this Figure, most interesting sub-domains are for $x : [2^{e_{\overline{x}}}, \overline{x}]$ and for $y : y \cap [0, 2^{e_{\overline{x}}-p-1}]$ (where p is the mantissa size). The first represents the sub-domain of x values absorbing y values. The second represents the sub-domain of y totally absorbed by x.

Example 1. Consider the function in Figure 3 and assume that inputs are coming from sensors, and their ranges are $[0.0, 1e{+}04]$ for `x`, and $[-16.0, 4.0]$ for `y`. The `else` branch corresponds to an unstable state of the system. Determining if this state is reachable and from which input values is a legitimate question. This problem is reduced to identifying if z can be equal to x which corresponds to absorption. Figure 4 shows resulting sub-domains. This strategy focuses on $[8.1920009765625000e{+}03, 1.0000000000000000e{+}04]$ for `x` and

```
void foo(float x, float y){
    float z = x + 2 * y;
    if(z != x)
      systemOK();
    else
      systemNOK();
}
```

Fig. 3. A program with absorptions

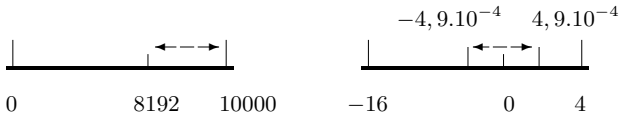


Fig. 4. Resulting sub-domains of unstable example

$[-4.8828122089616954e{-}04, 4.8828122089616954e{-}04]$ for **y** which correspond to domains involved in absorption. No solutions involving a value belonging to another sub-domain exist and the initial filtering has no impact on those domains. The unstable state is reachable with, for instance, values $1e{+}04$ for x and $2.44140625e{-}04$ for y.

This strategy can also be combined with another strategy, which is called whenever x has no values that absorb y.

4.2 Splitting strategy inspired by 3B-consistency : 3BSplit

Our next sub-domain splitting strategy, called `3Bsplit`, is inspired by a higher consistency named 3B-consistency [10]. 3B-Consistency is a relaxation on continuous domains of path consistency, a higher order extension of arc-consistency. Roughly speaking, 3B-Consistency checks whether 2B-consistency (or Hull consistency) can be enforced when the domain of a variable is reduced to the value of one of its bounds in the whole system [3]. 3B-consistency is in practice very effective on problems with multiple occurrence variables. However, insuring such consistency could be costly: the 3B-algorithm might attempt many times to unsuccessfully refute sub-domains near the bounds of the initial domain by means of a 2B-consistency.

The 3Bsplit sub-domain selection also attempts to enforce the consistency at the bounds of a domain at a single variable level. A key observation here is that if a small sub-domain at the bounds of the initial domain, e.g. $sd = [\underline{x}, \underline{x} + \delta]$, is immediately refuted in the next search node, then the resulting domain, $[\underline{x}+\delta, \overline{x}]$, offers a better lower bound than the one of the initial domain. Moreover, there is probably room to still improve this bound if the same step is reiterated using a wider sub-domain like $[\underline{x}, \underline{x} + 2\delta]$. Indeed, such a process is similar to a *shaving* applied to the initial domain but without requiring additional domain state management: in the case of a `3Bsplit`, the capability to return to the initial state is naturally supported by the search.

A sub-domain split might not be refuted immediately in the next search node. It might be refuted either after exploring a deeper search sub-tree or it

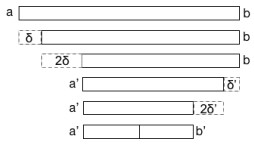


Fig. 5. `Illustration of 3BSplit`

might provide one or more solutions. Both cases underline some difficulties to improve the bound under examination. The next step of the splitting strategy thus switch to the next bound or, if both bounds have been checked, to another search node or strategy. Figure 5 illustrates `3Bsplit` behavior.

To summarize, `3Bsplit` exploits information on the sub-tree to decide whether enforcing the current bound has a chance to be done effortlessly or if it would be wiser to go to the next step. As a result, this strategy dynamically splits the current domain according to the behavior of the search in sub-trees. Note also that choosing an initial small sub-domain at the bounds of the domain is similar to the next strategy (Section 4.3), i.e., `3Bsplit` also provides opportunities to find solutions in the neighbourhood of the current bounds.

4.3 Mixing sub-domain and enumeration

This sub-domain selection strategy extends strategies from [4]. We propose two ways to extend these strategies. The first one `Enum-N`, enumerates `N` values of both bound and one in the middle before considering the rest of the domain. The second one, `Delta-N`, instead of enumerating each `N` values of the bounds, builds the sub-domains implied by `N` floating point numbers. Due to the huge number of evaluated strategies, the value of N is arbitrarily limited to 5 in the experiment part.

Enum-N
This sub-domain selection strategy is a direct generalization of [4]. Generally, the filtering process tightens the bounds until a support is found. This sub-domain selection strategy is optimistic and hope that filtering will lead to find a solution close to the bounds. To achieve this goal, it enumerates few values at the bounds. Figure 6a illustrates this strategy. The domains is split in $2 * N + 3$ sub domains. For instance, with $N = 5$, the 13 following domains will be generated :

- $[min, min]$
- $\cdots$
- $[min^{+[N]}, min^{+[N]}]$
- $[min^{+[N+1]}, mid^{-}]$
- $[mid, mid]$
- $[mid^{+}, max^{-[N+1]}]$
- $[max^{-[N]}, max^{-[N]}]$
- $\cdots$
- $[max, max]$

If the cardinality of the domain is lower than $2 * N + 3$, all values will be enumerated.

Delta-N The first objective of this strategy is to find a solution at the bounds. Enumerating can lead the search to explore a deep sub-tree before finding a

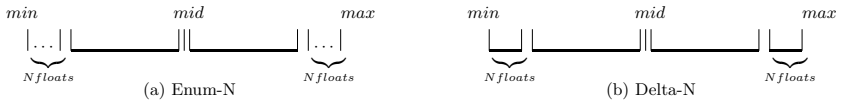


Fig. 6. Illustration of new strategies

solution or performing reduction. Here, considering the sub-domain implied by N floating-point numbers instead of a single value, gives opportunities to the filtering process to operate some pruning through propagation. It also improves the chance to find a solution or to remove all N values without enumerating any of them. This sub-domain selection strategy is very flexible. Indeed, by adapting the value of N, the whole strategy behavior change. For instance, if $N = |\alpha|$, with $\alpha = [\underline{x}, \frac{\underline{x}+\overline{x}}{2}]$, this sub-domain strategy becomes a classic bisection. Finally, a dynamic modification of the value of N during the search can be interesting. Starting the search by a classic bisection ($N = |\alpha|$) and reducing its value might be a good idea. Figure 6b illustrates this strategy. Regardless the value of N and the cardinality of the domain, at most the 5 following sub-domains will be generated :

- $[min, min^{+[N]}]$
- $[min^{+[N+1]}, mid^{-}]$
- $[mid, mid]$
- $[mid^{+}, max^{-[N+1]}]$
- $[max^{-[N]}, max]$

If the cardinality of the domain is lower than $2 * N + 3$, the last two sub-domains will not be generated, and the first two will be balanced with respect to the middle of the domain.

5 Experimental Evaluation

The experiments combine different variable selection strategies with sub-domain selection strategies on a set of 49 benchmarks. They also consider variations on the type of consistency for the strategies, sub-cuts, and the reselection (or not) of the same variable at the next node. Sub-cut corresponds to an alternative sub-domain selection strategy and is relevant for *splitAbs* and $3Bsplit$. For *splitAbs*, sub-cut is called when no absorptions occur. For $3Bsplit$, it is called to refute small sub-domains. In the state of the art, the standard strategy is based on lexicographic variable ordering, and a bisection based on 2B-consistency. These options result in no less than 325 unique strategies evaluated on all 49 benchmarks.

All experiments were performed on a Linux system, with an Intel Xeon processor running at 2.40GHz and with 12GB of memory. All strategies have been implemented into the Objective-CP solver. All the floating point computations are performed in simple precision and with a rounding mode “to the nearest even”.

5.1 Benchmarks

The benchmarks used in these experiments come from test and program verification. SMTLib [1], FPBench [6], and CBMC [2] (but also [5, 4, 7]) are the main

sources. In each case, the goal is to find a counter-example, hence the majority of instances are *satisfiable*. The number of constraints and variables varies from 2 to about 3000. Table summarizes thoses results of all the strategies on the satisfiable instances can be found at `www.i3s.unice.fr/~hzitoun/dp18/benchmark.html`.

5.2 Analysis

In results Tables, the standard strategy (lexicographic order with bisection, 2B filtering at 5% and reselection allowed) is at the 194^{th} position on 325 strategies. So, 193 strategies among those introduced, are clearly more efficient than the standard strategy for solving this kind of problem. The `Virtual Best Strategy` is 120 times faster than the reference strategy. The best strategy (column 1) is 4 times faster than the reference strategy. Using the specificities of floats to guide the search has a clear impact on the resolution time. Among the strategies that are efficient on this set of benchmarks, the variable selection strategies are based on lexicographic order, number of occurrences, density or absorption. While strategies based on width, a conventional variable selection strategy for integer domains, struggle to solve problems as soon as it becomes a bit realistic. Strategies based on cardinality are also in the same cases. The best search based on `MaxCard` and `MaxWidth` are at 160^{th} and 161^{th} positions.

Sub-domain selection strategies introduced in this article are working well. Indeed, the faster strategy exploiting `Delta-N` is in second position, whereas `Enum-N` is at 6^{th} position. The best $3BSplit$ is placed at 11^{th}. $SplitAbs$, for its part, is at 112^{th} position. $SplitAbs$ performance are clearly related to the percentage of absorption of the problem. Indeed, as shown in the online tables, the set of benchmarks limited to those with at least 5% of absorption are resolved without timeout. All these strategies perform better than the standard one.

Eight of the 10 best strategies prohibit the repeated selection of a variable at subsequent search nodes. Among the 10 worst, only one of them prohibits reselection. It appears that "reselection" impacts the ability to deliver solutions faster.

6 Conclusion

A previous article proposed a set of variable selection strategies using the specificities of floats to guide the search. These variable selection strategies improve the search of a counter-example outlining a property violation in a program to verify, but aren't sufficient to scale for harder and more realistic benchmarks. Dedicated sub-domain selection strategies for floats are needed. Contributions of this article are a set of sub-domain selection strategies over floats. The first one, exploits absorption phenomena, the second one embraces ideas derived from strong consistency and the two last ones extend strategies introduced in [4]. These strategies are compared on a set of satisfiable benchmarks. Several strategies presented, perform well, and obtain much better results than the standard strategy used to solve this kind of problem.

References

1. Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.
2. Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ansi-c programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.
3. Hélène Collavizza, François Delobel, and Michel Rueher. Comparing partial consistencies. *Reliable Computing*, pages 213–228, 1999.
4. Hélène Collavizza, Claude Michel, and Michel Rueher. Searching critical values for floating-point programs. In *Testing Software and Systems - 28th IFIP WG 6.1 International Conference, ICTSS 2016, Graz, Austria, October 17-19, 2016, Proceedings*, pages 209–217, 2016.
5. Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck. Cpbpv: A constraint-programming framework for bounded program verification. *Constraints*, 15(2):238–264, 2010.
6. Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Chen Qiu, Alexander Sanchez-Stern, and Zachary Tatlock. Toward a standard benchmark format and suite for floating-point analysis. In Sergiy Bogomolov, Matthieu Martel, and Pavithra Prabhakar, editors, *Numerical Software Verification*, pages 63–77, Cham, 2017. Springer International Publishing.
7. Vijay D'Silva, Leopold Haller, Daniel Kroening, and Michael Tautschnig. Numeric bounds analysis with conflict-driven learning. In Cormac Flanagan and Barbara König, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 48–63, 2012.
8. David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.
9. IEEE. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard*, 754, 2008.
10. Olivier Lhomme. Consistency techniques for numeric csps. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'93, pages 232–238, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.
11. Pascal Van Hentenryck and Laurent Michel. The objective-cp optimization system. In *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, pages 8–29, Berlin, Heidelberg, 2013.
12. Heytem Zitoun, Claude Michel, Michel Rueher, and Laurent Michel. Search strategies for floating point constraint systems. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, pages 707–722, 2017.