LNCS 6140

Andrea Lodi
Michela Milano
Paolo Toth (Eds.)

Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems

7th International Conference, CPAIOR 2010
Bologna, Italy, June 2010
Proceedings

Springer

Andrea Lodi   Michela Milano
Paolo Toth (Eds.)

Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems

7th International Conference, CPAIOR 2010
Bologna, Italy, June 14-18, 2010
Proceedings

Springer

Volume Editors

Andrea Lodi
Michela Milano
Paolo Toth
DEIS
University of Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
E-mail: {andrea.lodi,michela.milano,paolo.toth}@unibo.it

Library of Congress Control Number: 2010927799

CR Subject Classification (1998): G.1.6, G.1, G.2.1, F.2.2, I.2, J.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN	0302-9743
ISBN-10	3-642-13519-6 Springer Berlin Heidelberg New York
ISBN-13	978-3-642-13519-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

The 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010) was held in Bologna, Italy, June 16-18, 2010.

The conference is intended primarily as a forum to focus on the integration and hybridization of the approaches of constraint programming (CP), artificial intelligence (AI), and operations research (OR) technologies for solving large-scale and complex real-life combinatorial optimization problems. CPAIOR is focused on both theoretical and practical, application-oriented contributions.

The interest of the research community in this conference is witnessed by the high number of high-quality submissions received this year, reaching 39 long and 33 short papers. From these submissions, we chose 18 long and 17 short papers to be published in full in the proceedings.

This volume includes extended abstracts of the invited talks given at CPAIOR. Namely, one by Matteo Fischetti (University of Padova) on cutting planes and their use within search methods; another by Carla Gomes (Cornell University) on the recently funded NSF "Expedition in Computing" grant on the topic of computational sustainability and on the potential application of hybrid optimization approaches to this area; a third by Peter Stuckey (University of Melbourne) on the integration of SATisfiability solvers within constraint programming and integer programming solvers.

Two days before CPAIOR, a Master Class was organized by John Hooker on "Experimental Study of Algorithms and Benchmarking". The Master Class was composed of two parts: in the first, two leading researchers gave overview talks in the area. Catherine McGeoch (Amherst College) discussed statistical methods, and Carla Gomes (Cornell University) discussed the scientific use of experimentation. In the second part of the Master Class, software vendors described how they do benchmarking. The Master Class was intended for PhD students, researchers, and practitioners. We are very grateful to John, who brought this excellent program together. Finally, a rich program of one-day workshops was organized on June 15.

We warmly thank Zeynep Kiziltan for her work as Publicity Chair, Fabio Parisini for managing the conference website, and Enrico Malaguti for the management of the EasyChair System. We are very grateful to Meinolf Sellmann, who acted as Workshop Chair and put together an exciting program with five half-day workshops. Managing submissions and conference proceedings by means of the EasyChair System made our work a lot easier and we warmly thank EasyChair for this.

Many thanks to the members of the Program Committee, who reviewed all the submissions in detail and discussed conflicting papers deeply. We warmly thank the external reviewers as well.

Special thanks go to Marco Gavanelli and Andrea Roli, the Conference Chairs who took care of the many details concerning the organization, and to Vanessa Grotti (Planning Congressi), for her work on budgeting, planning and booking.

Finally, we would like to thank the sponsors who made it possible to organize this conference: the ARTIST Design, Network of Excellence, the Institute for Computational Sustainability (ICS), the Cork Constraint Computation Center, the Association for Constraint Programming (ACP), the Optimization for Sustainable Development (OSD) Chair, IBM and FICO.

A special mention should be made of FONDAZIONE DEL MONTE - 1473 for its generous support of the publication of these proceedings and of ALMA MATER STUDIORUM - Università di Bologna for the continuous help and support of the organization of CPAIOR 2010.

June 2010

Andrea Lodi
Michela Milano
Paolo Toth

Conference Organization

Program Chairs

Andrea Lodi	DEIS, University of Bologna, Italy
Michela Milano	DEIS, University of Bologna, Italy
Paolo Toth	DEIS, University of Bologna, Italy

Program Committee

Philippe Baptiste	École Polytechnique, France
Roman Barták	Charles University, Czech Republic
Christopher Beck	University of Toronto, Canada
Andrew Davenport	IBM, USA
Matteo Fischetti	University of Padova, Italy
Bernard Gendron	University of Montréal, Canada
Youssef Hamadi	Microsoft Research, United Kingdom
Susanne Heipcke	Xpress Team, FICO, France
Holger Hoos	University of British Columbia, Canada
Narendra Jussien	École des Mines de Nantes, France
Thorsten Koch	ZIB, Germany
Laurent Michel	University of Connecticut, USA
Barry O'Sullivan	University College Cork, Ireland
Laurent Perron	Google, France
Gilles Pesant	École Polytechnique de Montréal, Canada
Jean-Charles Régin	University of Nice-Sophia Antipolis, France
Louis-Martin Rousseau	École Polytechnique de Montréal, Canada
Meinolf Sellmann	Brown University, USA
Paul Shaw	IBM, France
Helmut Simonis	4C, Ireland
Michael Trick	Carnegie Mellon University, USA
Pascal Van Hentenryck	Brown University, USA
Willem-Jan van Hoeve	Carnegie Mellon University, USA
Petr Vilim	IBM, Czech Republic
Mark Wallace	Monash University, Australia
Tallys Yunes	University of Miami, USA

Local Organization

Marco Gavanelli	University of Ferrara, Italy
Andrea Roli	DEIS, University of Bologna, Italy
Enrico Malaguti	DEIS, University of Bologna, Italy
Fabio Parisini	DEIS, University of Bologna, Italy

External Reviewers

Table of Contents

Towards a MIP-Cut Metascheme

Matteo Fischetti

DEI, University of Padova, Italy
matteo.fischetti@unipd.it

Cutting planes (*cuts*) are very popular in the OR community, where they are used to strengthen the Linear Programming (LP) relaxation of Mixed-Integer Programs (MIPs) in the hope of improving the performance of an exact LP-based solver. In particular, an intense research effort has been devoted to the study of families of *general* cuts, whose validity does not require the presence of a specific MIP structure—as opposed to problem-specific cuts such as, e.g., subtour elimination or comb inequalities for the traveling salesman problem.

Among general cuts, Gomory's Mixed-Integer Cuts (GMICs) play a central role both in theory and in practice. These cuts have been introduced by Ralph Gomory about 50 years ago in his seminal paper [1]. Though elegant and computationally cheap, they were soon abandoned because they were considered of little practical use [2]. The situation changed radically more than 30 years later, when Balas, Ceria, Cornuéjols and Natraj [3] found how to take advantage of exactly the same cuts but in an different framework. In our view, this is a good example of the importance of a sound framework for MIP cuts.

Even today, MIP solvers are quite conservative in the use of general cuts, and in particular of GMICs, because of known issues due to the iterative accumulation of the cuts in the optimal LP basis. This leads to numerical instability because of a typically exponential growth of the determinant of the LP basis.

Following our recent joint work with Balas and Zanette [4,5], in this talk we argue that the known issues with cutting plane methods are largely due to the overall *framework* where the cuts are used, rather than to the cuts themselves. This is because the two main cutting plane modules (the LP solver and the cut generator) form a closed-loop system that is intrinsically prone to instability. Hence a kind of "decoupling filter" needs to be introduced in the loop if one wants to exploit the full power of a given family of cuts.

A main goal of the talk is to refocus part of the current research effort from the definition of new cut families to the way the cuts are actually used. In fact, cutting planes still miss an overall "meta-scheme" to control cut generation and to escape local optima by means of diversification phases—very well in the spirit of Tabu or Variable Neighborhood Search meta-schemes for primal heuristics. The development of sound meta-schemes on top of a basic separation tool is therefore an interesting new avenue for future research, with contributions expected from all the three CP/AI/OR communities. The relax-and-cut framework for GMICs recently proposed in the joint work with Salvagnin [6] can be viewed as a first step in this direction.

References

1. Gomory, R.E.: An algorithm for the mixed integer problem. Technical Report RM-2597, The RAND Cooperation (1960)
2. Cornuéjols, G.: Revival of the Gomory cuts in the 1990's. Annals of Operations Research 149(1), 63–66 (2006)
3. Balas, E., Ceria, S., Cornuéjols, G., Natraj, N.: Gomory cuts revisited. Operations Research Letters 19, 1–9 (1996)
4. Zanette, A., Fischetti, M., Balas, E.: Lexicography and degeneracy: can a pure cutting plane algorithm work? Mathematical Programming (2009), doi:10.1007/s10107-009-0300-y
5. Balas, E., Fischetti, M., Zanette, A.: On the enumerative nature of Gomory's dual cutting plane method. Mathematical Programming B (to appear, 2010)
6. Fischetti, M., Salvagnin, D.: A relax-and-cut framework for Gomory's mixed-integer cuts. In: CPAIOR 2010 Proceedings (2010)

Challenges for CPAIOR in Computational Sustainability

Carla P. Gomes

Cornell University
Ithaca, NY, USA
gomes@cs.cornell.edu

The notions of sustainability and sustainable development were first introduced in the seminal report of the United Nations World Commission on Environment and Development, known as the Brundtland report or Our Common Future [3]. Sustainable development is "development that meets the needs of the present without compromising the ability of future generations to meet their needs." Sustainability and sustainable development concern balancing environmental, economic, and societal needs for a sustainable future.

The development of policies for sustainable development often involves decision making and policy making problems concerning the management of our natural resources involving significant computational challenges that fall into the realm of computing and information science and related disciplines (e.g., operations research, applied mathematics, and statistics).

Computational Sustainability is a new emerging field that aims to apply techniques from computer science and related disciplines to help manage the balance of environmental, economic, and societal needs for sustainable development[1]. The focus of Computational Sustainability is on developing computational and mathematical models, methods, and tools for a broad range of sustainability related applications: from decision making and policy analysis concerning the management and allocation of resources to the design of new sustainable techniques, practices and products. The range of problems that fall under Computational Sustainability is therefore rather wide, encompassing computational challenges in disciplines as diverse as environmental sciences, economics, sociology, and biological and environmental engineering.

In this talk I will provide examples of computational sustainability challenge domains ranging from wildlife preservation and biodiversity, to balancing socio-economic needs and the environment, to large-scale deployment and management of renewable energy sources. I will discuss how computational sustainability problems offer challenges but also opportunities for the advancement of the state of the art of computing and information science and related fields, highlighting some overarching computational themes in constraint reasoning and optimization, machine learning, and dynamical systems. I will also discuss the need for a new approach to study such challenging problems in which computational problems are viewed as "natural" phenomena, amenable to a scientific methodology in which principled experimentation, to explore problem parameter spaces and hidden problem structure, plays as prominent a role as formal analysis [2]. Such an approach differs from the traditional computer science approach, based on abstract mathematical models, mainly driven by worst-case analyzes. While formulations of real-world computational tasks lead frequently to worst-case intractable problems, often such real world tasks contain hidden structure enabling scalable methods. It is therefore important to develop new approaches to identify and exploit real-world structure, combining principled experimentation with mathematical modeling, that will lead to scalable and practically effective solutions.

In summary, the new field of Computational Sustainability brings together computer scientists, operation researchers, applied mathematicians, biologists, environmental scientists, and economists, to join forces to study and provide solutions to computational problems concerning sustainable development, offering challenges but also opportunities for the advancement of the state of the art of computing and information science and related fields.

Acknowledgments

The author is the lead Principal Investigator of an NSF Expedition in Computing grant on Computational Sustainability (Award Number: 0832782). The author thanks the NSF Expeditions in Computing grant team members for their many contributions towards the development of a vision for Computational Sustainability, in particular, Chris Barrett, Antonio Bento, Jon Conrad, Tom Dietterich, John Gunckenheimer, John Hopcroft, Ashish Sabharwhal, Bart Selman, David Shmoys, Steve Strogatz, and Mary Lou Zeeman.

References

[1] Gomes, C.: Computational sustainability: Computational methods for a sustainable environment, economy, and society. The Bridge, National Academy of Engineering 39(4) (Winter 2009)

[2] Gomes, C., Selman, B.: The science of constraints. Constraint Programming Letters 1(1) (2007)

[3] UNEP. Our common future. Published as annex to the General Assembly document A/42/427, Development and International Cooperation: Environment. Technical report, United Nations Environment Programme (UNEP) (1987)

Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving

Peter J. Stuckey

NICTA Victoria Laboratory
Department of Computer Science and Software Engineering
University of Melbourne, 3010 Australia
pjs@cs.mu.oz.au

Abstract. Finite domain propagation solving, the basis of constraint programming (CP) solvers, allows building very high-level models of problems, and using highly specific inference encapsulated in complex global constraints, as well as programming the search for solutions to take into account problem structure. Boolean satisfiability (SAT) solving allows the construction of a graph of inferences made in order to determine and record effective nogoods which prevent the searching of similar parts of the problem, as well as the determination of those variables which form a tightly connected hard part of the problem, thus allowing highly effective automatic search strategies concentrating on these hard parts. Lazy clause generation is a hybrid of CP and SAT solving that combines the strengths of the two approaches. It provides state-of-the-art solutions for a number of hard combinatorial optimization and satisfaction problems. In this invited talk we explain lazy clause generation, and explore some of the many design choices in building such a hybrid system, we also discuss how to further incorporate mixed integer programming (MIP) solving to see if we can also inherit its advantages in combinatorial optimization.

1 Introduction

Propagation is an essential aspect of finite domain constraint solving which tackles hard combinatorial problems by interleaving search and restriction of the possible values of variables (propagation). The propagators that make up the core of a finite domain propagation engine represent trade-offs between the speed of inference of information versus the strength of the information inferred. Good propagators represent a good trade-off at least for some problem classes. The success of finite domain propagation in solving hard combinatorial problems arises from these good trade-offs, and programmable search, and has defined the success of constraint programming (CP).

Boolean Satisfiability (SAT) solvers have recently become remarkably powerful principally through the combination of: efficient engineering techniques for implementing inference (unit propagation) using watched literals, effective methods for generating and recording nogoods which prevent making a set of decisions which has already proven to be unhelpful (in particular 1UIP nogoods), and efficient search heuristics which concentrate on the hard parts of the problem combined with restarting to escape from early commitment to choices. These changes, all effectively captured in Chaff [1], have made SAT solvers able to solve problems orders of magnitude larger than previously possible.

Can we combine these two techniques in a way that inherits the strengths of each, and avoids their weaknesses. *Lazy clause generation* [2,3] is a hybridization of the two approaches that attempts to do this. The core of lazy clause generation is simple enough, we examine a propagation based solver and understand its actions as applying to an underlying set of Boolean variables representing the integer (and set of integer) variables of the CP model.

In this invited talk we will first introduce the basic theoretical concepts that underlie lazy clause generation. We discuss the relationship of lazy clause generation to SAT modulo theories [4]. We then explore the difficulties that arise in the simple theoretical hybrid, and examine design choices that ameliorate some of these difficulties. We discuss how complex global constraints interact with lazy clause generation. We then examine some of the remaining challenges for lazy clause generation: incorporating the advantages of mixed integer programming (MIP) solving, and building hybrid adaptive search strategies. The remainder of this short paper will simply introduce the basic concepts of lazy clause generation.

2 Lazy Clause Generation by Example

The core of lazy clause generation is fairly straightforward to explain. An integer variable x with initial domain $[\,l \,..\, u\,]$ is represented by two sets of Boolean variables $[\![x \leq d]\!], l \leq d < u$ and $[\![x = d]\!], l \leq d \leq u$. The meaning of each Boolean variable $[\![c]\!]$ is just the condition c. In order to prevent meaningless assignments to these Boolean variables we add clauses that define the conditions that relate them.

$$\begin{aligned}
[\![x \leq d]\!] &\rightarrow [\![x \leq d+1]\!], l \leq d \leq u-2 \\
[\![x = d]\!] &\rightarrow [\![x \leq d]\!], l \leq d \leq u-1 \\
[\![x = d]\!] &\rightarrow \neg[\![x \leq d-1]\!], l < d \leq u \\
[\![x \leq d]\!] \wedge \neg[\![x \leq d-1]\!] &\rightarrow [\![x = d]\!], l < d \leq u-1 \\
[\![x \leq l]\!] &\rightarrow [\![x = l]\!] \\
\neg[\![x \leq u-1]\!] &\rightarrow [\![x = u]\!]
\end{aligned}$$

Each Boolean variable encodes a domain change on the variable x. Setting $[\![x = d]\!]$ true sets variable x to d. Setting $[\![x = d]\!]$ false excludes the value d from the domain of x. Setting $[\![x \leq d]\!]$ true creates an upper bound d on the variable x. Setting $[\![x \leq d]\!]$ false creates a lower bound $d + 1$ on the variable x. We can hence mimic all domain changes using the Boolean variables. More importantly we can record the *behaviour* of a finite domain propagator using clauses over these variables.

Consider the usual bounds propagator for the constraint $x = y \times z$ (see e.g. [5]). Suppose the domain of x is $[\,-10\,..\,10\,]$, y is $[\,2\,..\,10\,]$ and z is $[\,3\,..\,10\,]$. The bounds propagator determines that the lower bound of x should be 6. In doing so it only made use of the lower bounds of y and z. We can record this as a clause c_1

$$(c_1) : \neg[\![y \leq 1]\!] \wedge \neg[\![z \leq 2]\!] \rightarrow \neg[\![x \leq 5]\!]$$

It also determines the upper bound of z is 5 using the upper bound of x and the lower bound of y, and similarly the upper bound of y is 3. These can be recorded as

$$(c_2) : [\![x \leq 10]\!] \wedge \neg[\![y \leq 1]\!] \rightarrow [\![z \leq 5]\!]$$
$$(c_3) : [\![x \leq 10]\!] \wedge \neg[\![z \leq 2]\!] \rightarrow [\![y \leq 3]\!]$$

Similarly if the domain of x is $[\,-10\,..\,10\,]$, y is $[\,-2\,..\,3\,]$ and z is $[\,-3\,..\,3\,]$, the bounds propagator determines that the upper bound of x is 9. In doing so it made use of both the upper and lower bounds of y and z. We can record this as a clause

$$\neg[\![y \leq -3]\!] \wedge [\![y \leq 3]\!] \wedge \neg[\![z \leq -4]\!] \wedge [\![z \leq 3]\!] \rightarrow [\![x \leq 9]\!]$$

In fact we could strengthen this explanation since the upper bound of x will remain 4 even if the lower bound of z was -4, or if the lower bound of y were -3. So we could validly record a stronger explanation of the propagation as

$$\neg[\![y \leq -3]\!] \wedge [\![y \leq 3]\!] \wedge \neg[\![z \leq -5]\!] \wedge [\![z \leq 3]\!] \rightarrow [\![x \leq 9]\!]$$

or

$$\neg[\![y \leq -4]\!] \wedge [\![y \leq 3]\!] \wedge \neg[\![z \leq -4]\!] \wedge [\![z \leq 3]\!] \rightarrow [\![x \leq 9]\!]$$

In a lazy clause generation system every time a propagator determines a domain change of a variable it records a clause that *explains* the domain change. We can understand this process as *lazily* creating a clausal representation of the information encapsulated in the propagator. Recording the clausal reasons for domain changes creates an implication graph of domain changes. When conflict is detected (an unsatisfiable constraint) we can construct a reason for the conflict, just as in a SAT (or SMT solver).

Suppose the domain of x is $[\,6\,..\,20\,]$, domain of y is $[\,2\,..\,20\,]$, z is $[\,3\,..\,10\,]$ and t is $[\,0\,..\,20\,]$ and we have constraints $x \leq t$, $x = y \times z$ and $y \geq 4 \vee z \geq 7$. Suppose search adds the new constraint $t \leq 10$ (represented by $[\![t \leq 10]\!]$). The inequality changes the upper bounds of x to 10 with explanation $(c_4) : [\![t \leq 10]\!] \rightarrow [\![x \leq 10]\!]$. The multiplication changes the upper bounds of z to 5 ($[\![z \leq 5]\!]$), and y to 3 ($[\![y \leq 3]\!]$) with the explanations c_2 and c_3 above, and the disjunctive constraint (which is equivalent to $(c_5) : \neg[\![y \leq 3]\!] \vee \neg[\![z \leq 6]\!]$) makes $\neg[\![z \leq 6]\!]$ true which by the domain constraints makes $(c_6) : [\![z \leq 5]\!] \rightarrow [\![z \leq 6]\!]$ unsatisfiable. The implication graph is illustrated in Figure 1.

We can explain the conflict by any cut that separates the conflict node from the earlier parts of the graph. The first unique implication point (1UIP) cut chooses the closest literal to the conflict where all paths from the last decision

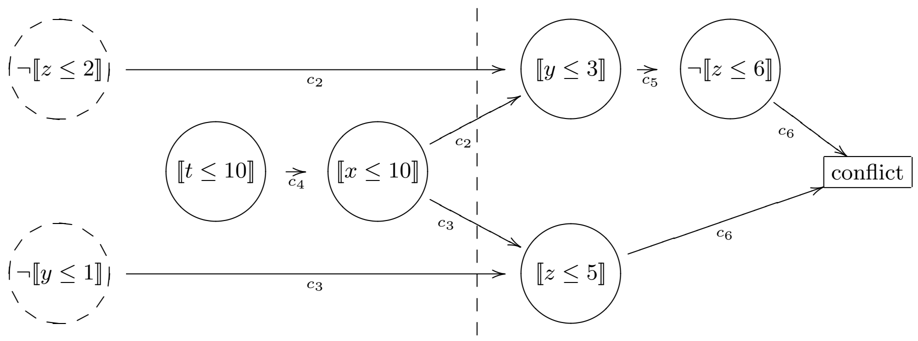


Fig. 1. Implication graph of propagation

to the conflict flow through that literal, and draws the cut just after this literal. The 1UIP cut for Figure 1 is shown as the dashed line. The resulting nogood is

$$\neg[\![z \le 2]\!] \wedge \neg[\![y \le 1]\!] \wedge [\![x \le 10]\!] \rightarrow \mathit{false}$$

Note that if we ever reach a situation in the future where the lower bound of y is at least 2, and the lower bound of z is at least 3, then the lower bound of x will become at least 11 using this clause.

Since we are explaining conflicts completely analogously to a SAT (or SMT) solver we can attach activities to the Boolean variables representing the integer original variables. Each Boolean variable examined during the creation of the explanation (including those appearing in the final nogood) has their activity bumped. Every once in a while all activities counts are decreased, so that more recent activity counts for more. This allows us to implement activity based VSIDS search heuristic for the hybrid solver. We can also attach activity counters to clauses, which are bumped when they are involved in the explanation process.

Since all of the clauses generated are redundant information we can at any stage remove any of the generated clauses. This gives us the opportunity to control the size of the clausal representation of the problem. Just as in a SAT solver we can use clausal activities to decide which generated clauses are most worthwhile retaining.

3 Concluding Remarks

The simple description of lazy clause generation in the previous section does not lead to an efficient lazy clause generation solver, except for some simple kinds of examples. In practice we need to also lazily generate the Boolean variables required to represent the original integer (and set of integer) variables. We may also choose to either eagerly generate the explanation clauses as we execute forward propagation, or lazily generate explanations on demand during the process of explaining a conflict. For each propagator we have to determine how to efficiently determine explanations of each propagation, and which form the explanation should take. In particular for global constraints many choices arise. Lazy clause generation also seems to reduce the need for global constraints, since in some cases decomposition of the global constraint, together with conflict learning, seems to recapture the additional propagation that the global constraint has over its decomposition. Decompositions of global constraints may also be more incremental that the global, and learn more reusable nogoods. In short, lazy clause generation requires us to revisit much of the perceived wisdom for creating finite domain propagation solvers, and indeed leads to many open questions on the right design for a lazy clause generation solver. Experiments have shown that for some classes of problem, such as resource constrained project scheduling problems [6] and set constraint solving [7], lazy clause generation provides state-of-the-art solutions.

References

1. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of 38th Conference on Design Automation (DAC 2001), pp. 530–535 (2001)
2. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
3. Feydy, T., Stuckey, P.: Lazy clause generation reengineered. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 352–366. Springer, Heidelberg (2009)
4. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and abstract DPLL modulo theories. In: Baader, F., Voronkov, A. (eds.) LPAR 2004. LNCS (LNAI), vol. 3452, pp. 36–50. Springer, Heidelberg (2005)
5. Marriott, K., Stuckey, P.: Programming with Constraints: an Introduction. MIT Press, Cambridge (1998)
6. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Why cumulative decomposition is not as bad as it sounds. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 746–761. Springer, Heidelberg (2009)
7. Gange, G., Stuckey, P., Lagoon, V.: Fast set bounds propagation using a BDD-SAT hybrid. Journal of Artificial Intelligence Research (to appear, 2010)

On Matrices, Automata, and Double Counting

Nicolas Beldiceanu[1], Mats Carlsson[2], Pierre Flener[3], and Justin Pearson[3]

[1] Mines de Nantes, LINA UMR CNRS 6241, FR-44307 Nantes, France
Nicolas.Beldiceanu@emn.fr
[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
Mats.Carlsson@sics.se
[3] Uppsala University, Department of Information Technology, Box 337, SE-751 05 Sweden
Pierre.Flener@it.uu.se, Justin.Pearson@it.uu.se

Abstract. Matrix models are ubiquitous for constraint problems. Many such problems have a matrix of variables $\mathcal{M}$, with the same constraint defined by a finite-state automaton $\mathcal{A}$ on each row of $\mathcal{M}$ and a global cardinality constraint *gcc* on each column of $\mathcal{M}$. We give two methods for deriving, by double counting, necessary conditions on the cardinality variables of the *gcc* constraints from the automaton $\mathcal{A}$. The first method yields linear necessary conditions and simple arithmetic constraints. The second method introduces the *cardinality automaton*, which abstracts the overall behaviour of all the row automata and can be encoded by a set of linear constraints. We evaluate the impact of our methods on a large set of nurse rostering problem instances.

1 Introduction

Several authors have shown that matrix models are ubiquitous for constraint problems. Despite this fact, only a few constraints that consider a matrix and some of its constraints as a whole have been considered: the $allperm$ [8] and $lex2$ [7] constraints were introduced for breaking symmetries in a matrix, while the $colored\_matrix$ constraint [13] was introduced for handling a conjunction of *gcc* constraints on the rows and columns of a matrix. We focus on another recurring pattern, especially in the context of personnel rostering, which can be described in the following way.

Given three positive integers R, K, and V, we have an $R \times K$ matrix $\mathcal{M}$ of decision variables that take their values within the finite set of values $\{0, 1, \ldots, V-1\}$, as well as a $V \times K$ matrix $\mathcal{M}^{\#}$ of cardinality variables that take their values within the finite set of values $\{0, 1, \ldots, R\}$. Each row r (with $0 \leq r < R$) of $\mathcal{M}$ is subject to a constraint defined by a finite-state automaton $\mathcal{A}$ [2,12]. For simplicity, we assume that each row is subject to the same constraint. Each column k (with $0 \leq k < K$) of $\mathcal{M}$ is subject to a *gcc* constraint that restricts the number of occurrences of the values according to column k of $\mathcal{M}^{\#}$: let $\#_k^v$ denote the number of occurrences of value v (with $0 \leq v < V$) in column k of $\mathcal{M}$, that is, the cardinality variable in row v and column k of $\mathcal{M}^{\#}$. We call this pattern the *matrix-of-automata-and-gcc* pattern. In the context of personnel rostering, a possible interpretation of this pattern is:

- R, K, and V respectively correspond to the number of persons, days, and types of work (e.g., *morning shift*, *afternoon shift*, *night shift*, or *day off*) we consider.

- Each row r of $\mathcal{M}$ corresponds to the work of person r over K consecutive days.
- Each column k of $\mathcal{M}$ corresponds to the work by the R persons on day k.
- The automaton $\mathcal{A}$ on the rows of $\mathcal{M}$ encodes the rules of a valid schedule for a person; it can be the product of several automata defining different rules.
- The gcc constraint on column k represents the demand of services for day k. In this context, the cardinality associated with a given service can either be fixed or be specified to belong to a given range.

A typical problem with this kind of pattern is the lack of interaction between the row and column constraints. This is especially problematic when, on the one hand, the row constraint is a sliding constraint expressing a distribution rule on the work, and, on the other hand, the demand profile (expressed with the gcc constraints) varies drastically from one day to the next (e.g., during weekends and holidays in the context of personnel rostering). This issue is usually addressed by experienced constraint programmers by manually adding necessary conditions (implied constraints) that are most of the time based on some simple counting conditions depending on some specificity of the row constraints. Let us first introduce a toy example to illustrate this phenomenon.

Example 1. Take a 3×7 matrix $\mathcal{M}$ of 0/1 variables (i.e., $R = 3$, $K = 7$, $V = 2$), where on each row we have a $global\_contiguity$ constraint (all the occurrences of value 1 are contiguous) for which Figure 1 depicts a corresponding automaton (the reader can ignore the assignments to counters c and d at this moment). In addition, $\mathcal{M}^{\#}$ defines the following gcc constraints on the columns of $\mathcal{M}$:

- Columns 0, 2, 4, and 6 of $\mathcal{M}$ must each contain two 0s and a single 1.
- Columns 1, 3, and 5 of $\mathcal{M}$ must each contain two 1s and a single 0.

A simple double counting argument proves that there is no solution to this problem. Indeed, consider the sequence of numbers of occurrences of 1s on the seven columns of $\mathcal{M}$, that is 1, 2, 1, 2, 1, 2, 1. Each time there is an increase of the number of 1s between two adjacent columns, a new group of consecutive 1s starts on at least one row of the matrix. From this observation we can deduce that we have at least four groups of consecutive ones, namely one group starts at the first column (since implicitly before the first column we have zero occurrences of value 1) and three groups start at the columns

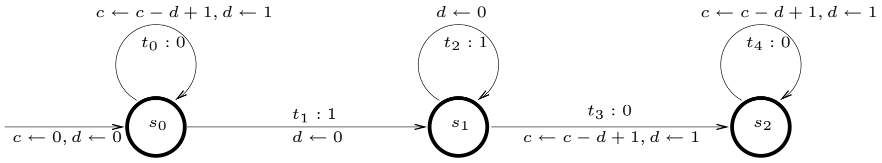

$c \leftarrow c - d + 1, d \leftarrow 1$
$t_0 : 0$
$d \leftarrow 0$
$t_2 : 1$
$c \leftarrow c - d + 1, d \leftarrow 1$
$t_4 : 0$
$c \leftarrow 0, d \leftarrow 0$
s_0
$t_1 : 1$
$d \leftarrow 0$
s_1
$t_3 : 0$
$c \leftarrow c - d + 1, d \leftarrow 1$
s_2

Fig. 1. Automaton associated with the $global\_contiguity$ constraint, with initial state s_0, final states s_0, s_1, s_2, and transitions t_0, t_1, t_2, t_3, t_4 labelled by values 0 or 1. The missing transition for value 1 from state s_2 is assumed to go to a dead state. The automaton has been annotated with counters [2]: the final value of counter c is the number of stretches of value 0, whereas d is an auxiliary counter.

containing two 1s. But since we have a $global\_contiguity$ constraint on each row of the matrix and since the matrix only has three rows, there is a contradiction.

The contributions of this paper include:

– Methods for deriving necessary conditions on the cardinality variables of the gcc constraints from string properties that hold for an automaton $\mathcal{A}$ (Sections 2.1 to 2.3).
– A method for annotating an automaton $\mathcal{A}$ with counter variables extracting string properties from $\mathcal{A}$ (Section 2.4).
– Another method for deriving necessary conditions on the cardinality variables, called the *cardinality automaton*, which simulates the overall behaviour of all the row automata (Section 3).
– An evaluation of the impact of our methods in terms of runtime and search effort on a large set of nurse rostering problem instances (Section 4).

Since our methods essentially generate linear constraints as necessary conditions, they may also be relevant in the context of linear programming.

2 Deriving Necessary Conditions from String Properties

We develop a first method for deriving necessary conditions for the *matrix-of-automata-and-gcc* pattern. The key idea is to approximate the set of solutions to the row constraint by string properties such as:

– Bounds on the number of letters, words, prefixes, or suffixes (see Section 2.1).
– Bounds on the number of stretches of a given value (see Section 2.2).
– Bounds on the lengths of stretches of a given value (see Section 2.3).

We first develop a set of formulae expressed in terms of simple arithmetic constraints for such string properties. Each formula gives a necessary condition for the *matrix-of-automata-and-gcc* pattern provided that the set of solutions of the row constraint satisfies a given string property. We then show how to extract automatically such string properties from an automaton (see Section 2.4) and outline a heuristic for selecting relevant string properties (see Section 2.5). String properties can also be seen as a communication channel for enhancing the propagation between row and column constraints.

In Sections 2.1 and 2.2, the derived constraints use the well-known combinatorial technique of *double counting* (see for example [9]). Here we use the two-dimensional structure of the matrix, counting along the rows and the columns. Some feature is considered, such as the number of appearances of a word or stretch, and the occurrences of that feature are counted for the rows and columns separately. When the counting is exact, these two values will coincide. In order to derive useful constraints that will propagate, we derive lower and upper bounds on the given feature occurring when counted columnwise. These are then combined into inequalities saying that the sum of these column-based lower bounds is *at most* the sum of given row-based upper bounds, or that the sum of these column-based upper bounds is *at least* the sum of given row-based lower bounds.

2.1 Constraining the Number of Occurrences of Words, Prefixes, and Suffixes

A *word* is a fixed sequence of values, seen as letters. Suppose we have the following bounds for each row on how many times a given word occurs (possibly in overlapping fashion) on that row, all numbering starting from zero:

- $LW_r(w)$ is the minimum number of times that the word w occurs on row r.
- $UW_r(w)$ is the maximum number of times that the word w occurs on row r.

Note that letters are just singleton words. It is not unusual that the $LW_r(w)$ (or $UW_r(w)$) are equal for all rows r for a given word w. From this information, we now infer by double counting two necessary conditions for each such word.

Necessary Conditions. Let $|w|$ denote the length of word w, and let w_j denote the j^{th} letter of word w. The following bounds

$$lw_k(w) = \max\left(\left(\sum_{j=0}^{|w|-1} \#_{k+j}^{w_j}\right) - (|w|-1)\cdot R, 0\right) \tag{1}$$

$$uw_k(w) = \min_{j=0}^{|w|-1} \#_{k+j}^{w_j} \tag{2}$$

correspond respectively to the minimum and maximum number of occurrences of word w that start at column $k \in [0, K-|w|]$. These bounds can be obtained as follows:

- Since the cardinality variables only count the number of times a value occurs in each column and does not constrain *where* it occurs, the lower bound (1) is the worst-case intersection of all column value occurrences.
- A word cannot occur more often than its minimally occurring letter, hence bound (2).

Note that if some cardinality variable is not fixed, then the expressions above should be interpreted as arithmetic constraints. We get the following necessary conditions:

$$\sum_{k=0}^{K-|w|} lw_k(w) \leq \sum_{r=0}^{R-1} UW_r(w) \quad \text{(3a)} \qquad \sum_{k=0}^{K-|w|} uw_k(w) \geq \sum_{r=0}^{R-1} LW_r(w) \quad \text{(3b)}$$

Note that (3b) trivially holds when all $LW_r(w)$ are zero.

Generalisation: Replacing Each Letter by a Set of Letters. In the previous paragraph, all letters of the word w were fixed. We now consider that each letter of a word can be replaced by a finite non-empty set of possible letters. For this purpose, let w_j now denote the j^{th} set of letters of word w. Hence the bounds $lw_k(w)$ and $uw_k(w)$ are now defined by aggregation as follows:

$$lw_k(w) = \max\left(\left(\sum_{j=0}^{|w|-1} \sum_{c\in w_j} \#_{k+j}^{c}\right) - (|w|-1)\cdot R, 0\right) \tag{4}$$

$$uw_k(w) = \min_{j=0}^{|w|-1} \left(\sum_{c\in w_j} \#_{k+j}^{c}\right) \tag{5}$$

We get the same necessary conditions as before. Note that (4) and (5) specialise respectively to (1) and (2) when all w_j are singleton sets.

Extension: Constraining Prefixes and Suffixes. We now consider constraints on a word occurring as a prefix (the first letter of the word is at the first position of the row) or suffix (the last letter of the word is at the last position of the row). Suppose we have the following bounds:

- $LWP_r(w)$ is the minimum number of times (0 or 1) word w is a prefix of row r.
- $UWP_r(w)$ is the maximum number of times (0 or 1) word w is a prefix of row r.
- $LWS_r(w)$ is the minimum number of times (0 or 1) word w is a suffix of row r.
- $UWS_r(w)$ is the maximum number of times (0 or 1) word w is a suffix of row r.

From these bounds, we get the following necessary conditions:

$$lw_0(w) \leq \sum_{r=0}^{R-1} UWP_r(w) \quad \text{(6a)} \qquad uw_0(w) \geq \sum_{r=0}^{R-1} LWP_r(w) \quad \text{(6b)}$$

$$lw_{K-|w|}(w) \leq \sum_{r=0}^{R-1} UWS_r(w) \quad \text{(7a)} \qquad uw_{K-|w|}(w) \geq \sum_{r=0}^{R-1} LWS_r(w) \quad \text{(7b)}$$

Note that (6b) trivially holds when all $LWP_r(w)$ are zero, and that (7b) trivially holds when all $LWS_r(w)$ are zero. Note that these necessary conditions also hold when each letter of a constrained prefix or suffix is replaced by a set of letters.

2.2 Constraining the Number of Occurrences of Stretches

Given a row r of fixed variables and a value v, a *stretch* of value v is a maximum sequence of values on row r that only consists of value v. Suppose now that we have bounds for each row on how many times a stretch of a given value v can occur on that row:

- $LS_r(v)$ is the minimum number of stretches of value v on row r.
- $US_r(v)$ is the maximum number of stretches of value v on row r.

It is not unusual that the $LS_r(v)$ (or $US_r(v)$) are equal for all rows r for a given value v.

Necessary Conditions. The following bounds (under the convention that $\#^v_{-1} = 0$ for each value v)

$$ls^+_k(v) = \max(0, \#^v_k - \#^v_{k-1}) \tag{8}$$

$$us^+_k(v) = \#^v_k - \max(0, \#^v_{k-1} + \#^v_k - R) \tag{9}$$

correspond respectively to the minimum and maximum number of stretches of value v that *start* at column k. Again, *if some cardinality variable is not fixed, then the expressions above should be interpreted as arithmetic constraints*. The intuitions behind these formulae are as follows:

- If the number of occurrences of value v on column k (i.e., $\#_k^v$) is strictly greater than the number of occurrences of value v on column $k-1$ (i.e., $\#_{k-1}^v$), then this means that at least $\#_k^v - \#_{k-1}^v$ new stretches of value v can start at column k.
- If the total of the number of occurrences of value v on column k (i.e., $\#_k^v$) and the number of occurrences of value v on column $k-1$ (i.e., $\#_{k-1}^v$) is strictly greater than the number of rows R, then the quantity $\#_{k-1}^v + \#_k^v - R$ represents the minimum number of stretches of value v that cover both column $k-1$ and column k. From this minimum intersection we get the maximum number of new stretches that can start at column k.

By aggregating these bounds for all the columns of the matrix, we get the following necessary conditions through double counting:

$$\sum_{k=0}^{K-1} ls_k^+(v) \le \sum_{r=0}^{R-1} US_r(v) \quad (10a) \qquad \sum_{k=0}^{K-1} us_k^+(v) \ge \sum_{r=0}^{R-1} LS_r(v) \quad (10b)$$

Similarly, the following bounds (under the convention that $\#_K^v = 0$ for each value v)

$$ls_k^-(v) = \max(0, \#_k^v - \#_{k+1}^v) \quad (11)$$
$$us_k^-(v) = \#_k^v - \max(0, \#_{k+1}^v + \#_k^v - R) \quad (12)$$

correspond respectively to the minimum and maximum number of stretches of value v that *end* at column k. We get similar necessary conditions:

$$\sum_{k=0}^{K-1} ls_k^-(v) \le \sum_{r=0}^{R-1} US_r(v) \quad (13a) \qquad \sum_{k=0}^{K-1} us_k^-(v) \ge \sum_{r=0}^{R-1} LS_r(v) \quad (13b)$$

Note that (10b) and (13b) trivially hold when all $LS_r(v)$ are zero.

Generalisation: Replacing the Value by a Set of Values. In the previous paragraph, the value v of a stretch was fixed. We now consider that a stretch may consist of a finite non-empty set, denoted by $\hat{v}$, of possible letters that are all considered equivalent. Let $\#_k^{\hat{v}}$ denote the quantity $\sum_{v\in\hat{v}}(\#_k^v)$, that is the total number of occurrences of the values of $\hat{v}$ in column k. The bounds (8), (9), (11), (12) are generalised as follows:

$$ls_k^+(\hat{v}) = \max(0, \#_k^{\hat{v}} - \#_{k-1}^{\hat{v}}) \quad (14)$$
$$us_k^+(\hat{v}) = \#_k^{\hat{v}} - \max(0, \#_{k-1}^{\hat{v}} + \#_k^{\hat{v}} - R) \quad (15)$$
$$ls_k^-(\hat{v}) = \max(0, \#_k^{\hat{v}} - \#_{k+1}^{\hat{v}}) \quad (16)$$
$$us_k^-(\hat{v}) = \#_k^{\hat{v}} - \max(0, \#_{k+1}^{\hat{v}} + \#_k^{\hat{v}} - R) \quad (17)$$

and we get the following necessary conditions:

$$\sum_{k=0}^{K-1} ls_k^+(\hat{v}) \le \sum_{v\in\hat{v}}\sum_{r=0}^{R-1} US_r(v) \quad (18a) \qquad \sum_{k=0}^{K-1} us_k^+(\hat{v}) \ge \sum_{v\in\hat{v}}\sum_{r=0}^{R-1} LS_r(v) \quad (18b)$$

$$\sum_{k=0}^{K-1} ls_k^-(\hat{v}) \leq \sum_{v\in\hat{v}} \sum_{r=0}^{R-1} US_r(v) \quad \text{(19a)} \qquad \sum_{k=0}^{K-1} us_k^-(\hat{v}) \geq \sum_{v\in\hat{v}} \sum_{r=0}^{R-1} LS_r(v) \quad \text{(19b)}$$

Note that (18a), (18b), (19a), and (19b) specialise respectively to (10a), (10b), (13a), and (13b) when $\hat{v} = \{v\}$.

2.3 Constraining the Minimum and Maximum Length of a Stretch

Suppose now that we have lower and upper bounds on the length of a stretch of a given value v for each row:

- $LLS(v)$ is the minimum length of a stretch of value v in every row.
- $ULS(v)$ is the maximum length of a stretch of value v in every row.

Necessary Conditions

$$\forall k \in [0, K-1] : \#_k^v \geq \sum_{j=\max(0,k-LLS(v)+1)}^{k} ls_j^+(v) \tag{20}$$

$$\forall k \in [0, K-1] : \#_k^v \geq \sum_{j=k}^{\min(K-1,k+LLS(v)-1)} ls_j^-(v) \tag{21}$$

The intuition behind (20) resp. (21) is that the stretches starting resp. ending at the considered columns j must overlap column k.

$$\begin{gathered}\forall k \in [0, K-1-ULS(v)] : \\ ls_k^+(v) + \sum_{j=LLS(v)}^{ULS(v)} \#_{k+j}^v - (ULS(v) - LLS(v) + 1) \cdot R \leq 0\end{gathered} \tag{22}$$

$$\begin{gathered}\forall k \in [ULS(v), K-1] : \\ ls_k^-(v) + \sum_{j=LLS(v)}^{ULS(v)} \#_{k-j}^v - (ULS(v) - LLS(v) + 1) \cdot R \leq 0\end{gathered} \tag{23}$$

The intuition behind (22) is as follows. Consider a stretch beginning at column k. Then there must be an element distinct from v in column $j \in [k + LLS(v), k + ULS(v)]$ of the same row. So at least one of the terms in the summation of (22) will get a zero contribution from the given row. The reasoning in (23) is similar but considers stretches ending at column k.

2.4 Extracting Occurrence, Word, and Stretch Constraints from an Automaton, or How to Annotate an Automaton with String Properties

Toward automatically inferring the constant bounds $LW_r(w)$, $LWP_r(w)$, $LWS_r(w)$, $LS_r(w)$, etc, of the previous sub-sections, we now describe how a given automaton

Table 1. For each annotation in the first column, the second column gives the number of new counters, the third column gives their initial values, and the fourth column shows the string property variable among the final counter values. In the first three rows, ℓ is the word length.

Annotation	Number of counters	Initial values	Final values
$wordocc(\hat{v}^+, n)$	ℓ	$[0, ..., 0]$	$[\_, ..., n]$
$wordprefix(\hat{v}^+, b)$	$\ell + 1$	$[1, 0, ..., 0]$	$[\_, ..., b]$
$wordsuffix(\hat{v}^+, b)$	ℓ	$[0, ..., 0]$	$[\_, ..., b]$
$stretchocc(\hat{v}, n)$	2	$[0, 0]$	$[n, \_]$
$stretchminlen(\hat{v}, n)$	3	$[+\infty, +\infty, 0]$	$[n, \_, \_]$
$stretchmaxlen(\hat{v}, n)$	2	$[0, 0]$	$[n, \_]$

$\mathcal{A}$ can be automatically annotated with counter variables constrained to reflect properties of the strings that the automaton recognises. This is especially useful if $\mathcal{A}$ is a product automaton for several constraints. For this purpose, we use the *automaton* constraint introduced in [2], which (unlike the *regular* constraint [12]) allows us to associate counters to a transition. Each string property requires (i) a counter variable whose final value reflects the value of that string property, (ii) possibly some auxiliary counter variables, (iii) initial values of the counter variables, and (iv) update formulae in the automaton transitions for the counter variables. We now give the details for some string properties.

In this context, n denotes an integer or decision variable, b denotes a 0/1 integer or decision variable, $\hat{v}$ denotes a set of letters, $\hat{v}^+$ denotes a nonempty sequence of letters in $\hat{v}$, and s_i denotes the i^{th} letter of word s. We describe the annotation for the following string properties for any given string:

- $wordocc(\hat{v}^+, n)$: Word $\hat{v}^+$ occurs n times.
- $wordprefix(\hat{v}^+, b)$: $b = 1$ iff word $\hat{v}^+$ is a prefix of the string.
- $wordsuffix(\hat{v}^+, b)$: $b = 1$ iff word $\hat{v}^+$ is a suffix of the string.
- $stretchocc(\hat{v}, n)$: Stretches of letters in set $\hat{v}$ occur n times.
- $stretchminlen(\hat{v}, n)$: If letters in set $\hat{v}$ occur, then n is the length of the shortest such stretch, otherwise $n = +\infty$.
- $stretchmaxlen(\hat{v}, n)$: If letters in set $\hat{v}$ occur, then n is the length of the longest such stretch, otherwise $n = 0$.

For a given annotation, Table 1 shows which counters it introduces, as well as their initial and final values, while Table 2 shows the formulae for counter updates to be used in the transitions. Figure 1 shows an automaton annotated for $stretchocc(\{0\}, n)$.

An automaton can be annotated with multiple string properties—annotations do not interfere with one another—and can be simplified in order to remove multiple occurrences of identical counters that come from different string properties.

It is worth noting that propagation is possible from the decision variables to the counter variables, and vice-versa.

Table 2. Given an annotation and a transition of the automaton reading letter u, the table gives the counter update formulae to be used in this transition. For each annotation in the first column, the second column shows the counter names, and the third column shows the update formulae. The fourth column shows the condition under which each formula is used. In the first three multirows, ℓ is the word length.

Annotation	Counter values	New counter values	Condition
$wordocc(\hat{v}^+, n)$	$[c_1, ..., c_\ell]$	$[1, ...]$	$u \in \hat{v}_1^+$
		$[..., c_{i-1}, ...]$	$1 < i < \ell \wedge u \in \hat{v}_i^+$
		$[..., c_\ell + c_{\ell-1}]$	$u \in \hat{v}_\ell^+$
		$[..., 0, ...]$	$0 < i < \ell \wedge u \notin \hat{v}_i^+$
		$[..., c_\ell]$	$u \notin \hat{v}_\ell^+$
$wordprefix(\hat{v}^+, b)$	$[c_0, c_1, ..., c_\ell]$	$[0, ..., c_{i-1}, ...]$	$0 < i < \ell \wedge u \in \hat{v}_i^+$
		$[0, ..., \max(c_\ell, c_{\ell-1})]$	$u \in \hat{v}_\ell^+$
		$[0, ..., 0, ...]$	$0 < i < \ell \wedge u \notin \hat{v}_i^+$
		$[0, ..., c_\ell]$	$u \notin \hat{v}_\ell^+$
$wordsuffix(\hat{v}^+, b)$	$[c_1, ..., c_\ell]$	$[1, ...]$	$u \in \hat{v}_1^+$
		$[..., c_{i-1}, ...]$	$1 < i < \ell \wedge u \in \hat{v}_i^+$
		$[..., c_{\ell-1}]$	$u \in \hat{v}_\ell^+$
		$[..., 0, ...]$	$0 < i < \ell \wedge u \notin \hat{v}_i^+$
		$[..., c_\ell]$	$u \notin \hat{v}_\ell^+$
$stretchocc(\hat{v}, n)$	$[c, d]$	$[c - d + 1, 1]$	$u \in \hat{v}$
		$[c, 0]$	$u \notin \hat{v}$
$stretchminlen(\hat{v}, n)$	$[c, d, e]$	$[\min(d, e + 1), d, e + 1]$	$u \in \hat{v}$
		$[c, c, 0]$	$u \notin \hat{v}$
$stretchmaxlen(\hat{v}, n)$	$[c, d]$	$[\max(c, d + 1), d + 1]$	$u \in \hat{v}$
		$[c, 0]$	$u \notin \hat{v}$

2.5 Heuristics for Selecting Relevant String Properties for an Automaton

In our experiments (see Section 4), we chose to look for the following string properties:

– For each letter, lower and upper bounds on the number of its occurrences.
– For each letter, lower and upper bounds on the number or length of its stretches.
– Each word of length at most 3 that cannot occur at all.
– Each word of length at most 3 that cannot occur as a prefix or suffix.

These properties are derived, one at a time, as follows. We annotate the automaton as described in the previous section by the candidate string property. Then we compute by labelling the feasible values of the counter variable reflecting the given property, giving up if the computation does not finish within 5 CPU seconds. Among the collected word, prefix, suffix, and stretch properties, some properties are subsumed by others and are thus filtered away. Other properties could certainly have been derived, e.g., not only forbidden words, but also bounds on the number of occurrences of words. Our choice was based on (a) which properties we are able to derive necessary conditions for, and (b) empirical observations of what actually pays off in our benchmarks.

3 The Cardinality Automaton of an Automaton

The previous section introduced different complementary ways of generating necessary conditions (expressed in terms of arithmetic constraints) from a given automaton for the row constraints of the matrix $\mathcal{M}$ when its columns are subject to *gcc* constraints. This section presents an orthogonal systematic approach, again based on double counting, that can handle a larger class of column constraints completely mechanically.

Consider an $R \times K$ matrix $\mathcal{M}$, where on each row we have the same constraint, represented by an automaton $\mathcal{A}$ of p states $s_0, \dots, s_{p-1}$, and on each column we have a *gcc* or linear (in)equality constraint where all the coefficients are the same. We will first construct an automaton that simulates the parallel running of the R copies of $\mathcal{A}$ and consumes entire columns of $\mathcal{M}$. Since this new automaton has p^R states, we then abstract it by just *counting* the automata that are in each state of $\mathcal{A}$. As even this abstracted automaton has a size exponential in p, we then use a linear-size encoding with linear constraints that allows us to consider also the column constraints on $\mathcal{M}$.

3.1 Necessary Row Constraints

The vector automaton $\overline{\mathcal{A}_R}$ consumes vectors of size R. Its states are sequences of R states of $\mathcal{A}$, where entry ℓ is the state of the automaton of row ℓ. There is a transition from state $\langle s_{i_0}, \dots, s_{i_{R-1}} \rangle$ to state $\langle s_{j_0}, \dots, s_{j_{R-1}} \rangle$ if and only if for each ℓ there is a transition in $\mathcal{A}$ from s_{i_ℓ} to s_{j_ℓ}. A state $\langle s_{i_0}, \dots, s_{i_{R-1}} \rangle$ is initial (resp. final) if each of the s_{i_ℓ} is the initial (resp. a final) state of $\mathcal{A}$.

The cardinality (vector) automaton $\#\left(\overline{\mathcal{A}_R}\right)$ is an abstraction of the vector automaton $\overline{\mathcal{A}_R}$ that also consumes vectors of size R. Its states are sequences of p numbers, whose sum is R, where entry i is the number of automata $\mathcal{A}$ in state s_i. There is a transition from state $\langle c_{i_0}, \dots, c_{i_{p-1}} \rangle$ to state $\langle c_{j_0}, \dots, c_{j_{p-1}} \rangle$ if and only if there exists a multiset of R transitions in $\mathcal{A}$ such that for each ℓ there are c_{i_ℓ} of these R transitions going out from s_ℓ, and for each m there are c_{j_m} of these R transitions arriving into s_m. A state $\langle c_{i_0}, \dots, c_{i_{p-1}} \rangle$ is initial (resp. final) if $c_{i_\ell} = 0$ whenever s_ℓ is not the initial (resp. a final) state of $\mathcal{A}$.

The number of states of $\#\left(\overline{\mathcal{A}_R}\right)$ is the number of ordered partitions of p, and thus exponential in p. However, it is possible to have a compact encoding via constraints. Toward this, we use $K+1$ sequences of p decision variables S_i^k in the domain $\{0, 1, \dots, R\}$ to encode the states of an arbitrary path of length K (the number of columns) in $\#\left(\overline{\mathcal{A}_R}\right)$. For $k \in \{1, \dots, K\}$, the sequence $\langle S_0^k, S_1^k, \dots, S_{p-1}^k \rangle$ has as possible values the states of $\#\left(\overline{\mathcal{A}_R}\right)$ after it has consumed column $k-1$; the sequence $\langle S_0^0, S_1^0, \dots, S_{p-1}^0 \rangle$ is fixed to $\langle R, 0, \dots, 0 \rangle$ when, without loss of generality, s_0 is the initial state of $\mathcal{A}$. We get the following constraints:

$$\forall k \in \{0, \dots, K\} : S_0^k + S_1^k + \dots + S_{p-1}^k = R \tag{24}$$

$$\forall i \in \{0, \dots, p-1\} : S_i^K = 0 \leftarrow s_i \text{ is not a final state of } \mathcal{A} \tag{25}$$

Assume that $\mathcal{A}$ has a set $\mathcal{T} = \{(a_0, \ell_0, b_0), (a_1, \ell_1, b_1), \dots, (a_{q-1}, \ell_{q-1}, b_{q-1})\}$ of q transitions, where transition (a_i, ℓ_i, b_i) goes from state $a_i \in \{s_0, s_1, \dots, s_{p-1}\}$ to state

$b_i \in \{s_0, s_1, \ldots, s_{p-1}\}$ upon reading letter $\ell_i \in \{0, 1, \ldots, V-1\}$. We use K sequences of q decision variables T_i^k in the domain $\{0, 1, \ldots, R\}$ to encode the transitions of an arbitrary path of length K in $\#\left(\overline{\mathcal{A}_R}\right)$. For $k \in \{0, \ldots, K-1\}$, the sequence $\langle T^k_{(a_0,\ell_0,b_0)}, T^k_{(a_1,\ell_1,b_1)}, \ldots, T^k_{(a_{q-1},\ell_{q-1},b_{q-1})}\rangle$ gives the numbers of automata $\mathcal{A}$ with transition $(a_0, \ell_0, b_0), (a_1, \ell_1, b_1), \ldots, (a_{q-1}, \ell_{q-1}, b_{q-1})$ upon reading the character of their row in column k. We get the following constraint for column k:

$$T^k_{(a_0,\ell_0,b_0)} + T^k_{(a_1,\ell_1,b_1)} + \cdots + T^k_{(a_{q-1},\ell_{q-1},b_{q-1})} = R \tag{26}$$

Consider two state encodings $\langle S_0^k, S_1^k, \ldots, S_{p-1}^k\rangle$ and $\langle S_0^{k+1}, S_1^{k+1}, \ldots, S_{p-1}^{k+1}\rangle$, and consider the transition encoding $\langle T^k_{(a_0,\ell_0,b_0)}, T^k_{(a_1,\ell_1,b_1)}, \ldots, T^k_{(a_{q-1},\ell_{q-1},b_{q-1})}\rangle$ between these two state encodings (with $0 \leq k < K$). To encode paths of length K in $\#\left(\overline{\mathcal{A}_R}\right)$, we introduce the following constraints. First, we constrain the number of automata $\mathcal{A}$ at any state s_j before reading column k to equal the number of firing transitions going out from s_j when reading column k:

$$\forall j \in \{0, \ldots, p-1\} : S_j^k = \sum_{(a_i,\ell_i,b_i)\in\mathcal{T} \,:\, a_i = s_j} T^k_{(a_i,\ell_i,b_i)} \tag{27}$$

Second, we constrain the number of automata $\mathcal{A}$ at state s_j after reading column k to equal the number of firing transitions coming into s_j when reading column k:

$$\forall j \in \{0, \ldots, p-1\} : S_j^{k+1} = \sum_{(a_i,\ell_i,b_i)\in\mathcal{T} \,:\, b_i = s_j} T^k_{(a_i,\ell_i,b_i)} \tag{28}$$

A reformulation with linear constraints when $R = 1$ and there are *no* column constraints is described in [6].

3.2 Necessary Column Constraints and Channelling Constraints

The necessary constraints above on the state and transition variables only handle the row constraints, but they can also be used to handle column constraints of the considered kinds. These necessary constraints can thus be seen as a communication channel for enhancing the propagation between row and column constraints.

If column k has a *gcc*, then we constrain the number of occurrences of value v in column k to equal the number of transitions on v when reading column k:

$$\forall v \in \{0, \ldots, V-1\} : \#_k^v = \sum_{(a_i,\ell_i,b_i)\in\mathcal{T} \,:\, \ell_i = v} T^k_{(a_i,\ell_i,b_i)} \tag{29}$$

If column k constrains the sum of the column, then we constrain that sum to equal the value-weighted number of transitions on v when reading column k:

$$\sum_{r=0}^{R-1} \mathcal{M}[r,k] = \sum_{v=0}^{V-1} v \cdot \left(\sum_{(a_i,\ell_i,b_i)\in\mathcal{T} \,:\, \ell_i = v} T^k_{(a_i,\ell_i,b_i)} \right) \tag{30}$$

Furthermore, for more propagation, we can link the variables S_i^k back to the state variables [2] of the R automata $\mathcal{A}$. For this purpose, let the variables $Q_i^0, Q_i^1, \ldots, Q_i^K$ (with $0 \leq i < R$) denote the $K+1$ states visited by automaton $\mathcal{A}$ on row i of length K. We get the following *gcc* necessary constraints:

$$\forall k \in \{0, \ldots, K\} : gcc(\langle Q_0^k, Q_1^k, \ldots, Q_{R-1}^k \rangle, \langle 0 : S_0^k, 1 : S_1^k, \ldots, p{-}1 : S_{p-1}^k \rangle) \quad (31)$$

Example 2. In the context of an $R = 4$ by $K = 6$ matrix with a *global_contiguity* constraint on each row and a *gcc* constraint on each column, we illustrate the set of linear constraints associated with column k (where $0 \leq k < 6$) of the matrix. An automaton $\mathcal{A}$ associated with the *global_contiguity* constraint was described by Figure 1 of Example 1. It has $p = 3$ states s_0, s_1, s_2 and $q = 5$ transitions $(s_0, 0, s_0)$, $(s_0, 1, s_1)$, $(s_1, 1, s_1)$, $(s_1, 0, s_2)$, $(s_2, 0, s_2)$ labelled by values 0 and 1. The encoding has $p \cdot (K+1) = 21$ variables S_i^k such that $S_0^k + S_1^k + S_2^k = 4$ for every k. Since s_0 is the initial state of $\mathcal{A}$, we require that $S_0^0 = 4$ since $S_1^0 = 0 = S_2^0$. Since $\mathcal{A}$ only has final states, no S_j^K is constrained to be zero. The encoding also has $q \cdot K = 30$ variables T_i^k such that $T_{(s_0,0,s_0)}^k + T_{(s_0,1,s_1)}^k + T_{(s_1,1,s_1)}^k + T_{(s_1,0,s_2)}^k + T_{(s_2,0,s_2)}^k = 4$ for every k. The following three sets of linear necessary constraints link the variables above for every k:

$$\begin{array}{ll}
S_0^k = T_{(s_0,0,s_0)}^k + T_{(s_0,1,s_1)}^k & (\textit{transitions that exit state } s_0) \\
S_1^k = T_{(s_1,1,s_1)}^k + T_{(s_1,0,s_2)}^k & (\textit{transitions that exit state } s_1) \\
S_2^k = T_{(s_2,0,s_2)}^k & (\textit{transitions that exit state } s_2) \\
\\
S_0^{k+1} = T_{(s_0,0,s_0)}^k & (\textit{transitions that enter state } s_0) \\
S_1^{k+1} = T_{(s_0,1,s_1)}^k + T_{(s_1,1,s_1)}^k & (\textit{transitions that enter state } s_1) \\
S_2^{k+1} = T_{(s_1,0,s_2)}^k + T_{(s_2,0,s_2)}^k & (\textit{transitions that enter state } s_2) \\
\\
\#_k^0 = T_{(s_0,0,s_0)}^k + T_{(s_1,0,s_2)}^k + T_{(s_2,0,s_2)}^k & (\textit{transitions labelled by value } 0) \\
\#_k^1 = T_{(s_0,1,s_1)}^k + T_{(s_1,1,s_1)}^k & (\textit{transitions labelled by value } 1)
\end{array}$$

4 Evaluation and Conclusion

NSPLib [14] is a very large repository of (artificially generated) instances of the *nurse scheduling problem* (NSP), which is about constructing a duty roster for nursing staff. Let N be the number of nurses, D the number of days of the scheduling horizon, and S the number of shifts. The objective is to construct an $N \times D$ matrix of values in the integer interval $[1, S]$, with value S representing the off-duty "shift".

In *instance files*, there are hard *coverage constraints* and soft preference constraints; we only use the former here: they give for each day d and shift s the lower bound on the number of nurses that must be assigned to shift s on day d, and can be modelled by a global cardinality constraint (*gcc*) on the columns. We stress that the *gcc* constraints on any two columns are in general *not* the same. There are instance files for $N \times 7$ rosters with $N \in \{25, 50, 75, 100\}$, and for $N \times 28$ rosters with $N \in \{30, 60\}$.

In *case files*, there are hard constraints on the rows. For each shift s, there are lower and upper bounds on the number of occurrences of s in any row (the daily assignment of some nurse): this can be modelled by gcc constraints on the rows. There are even lower and upper bounds on the cumulative number of occurrences of the working shifts $1, \ldots, S-1$ in any row: this can be modelled by gcc constraints on the off-duty value S and always gives tighter occurrence bounds on S than in the previous gcc constraints. For each shift s, there are also lower and upper bounds on the length of any stretch of value s in any row: this can be modelled by $stretch\_path$ constraints on the rows. Finally, there are lower and upper bounds on the length of any stretch of the working shifts $1, \ldots, S-1$ in any row: this can be modelled by generalised $stretch\_path\_partition$ constraints [3] on the rows. We stress that the constraints on any two rows are the *same*. There are 8 case files for the $N \times 7$ rosters, and another 8 case files for the $N \times 28$ rosters. We automatically generated (see [3] for details) deterministic finite automata (DFA) for all the row constraints of each case, but used their minimised product DFA instead (obtained through standard DFA algorithms), thereby getting domain consistency on the conjunction of all row constraints [2]. For each case, string properties were automatically selected off-line as described in Section 2.5, and cardinality automata were automatically constructed off-line as described in Section 3.

Under these choices, the NSPLib benchmark corresponds to the pattern studied in this paper. To reduce the risk of reporting improvements where another search procedure can achieve much of the same impact, we use a two-phase search that exploits the fact that there is a single domain-consistent constraint on each row and column:

- Phase 1 addresses the column (coverage) constraints only: it seeks to assign enough nurses to given shifts on given days to satisfy *all but one* coverage constraint. To break row symmetries, an equivalence relation is maintained: two rows (nurses) are in the same equivalence class while they are assigned to the same shifts and days.
- In Phase 2, one column constraint and all row constraints remain to be satisfied. But these constraints form a Berge-acyclic CSP [1], and so the remaining decision variables can be trivially labelled without search.

This search procedure is much more efficient than row-wise labelling under decreasing value ordering (value S always has the highest average number of occurrences per row) in the presence of a decreasing lexicographic ordering constraint on the rows.

The objective of our experiments is to measure the impact in runtime and backtracks when using either or both of our methods. The experiments were run under SICStus Prolog 4.1.1 and Mac OS X 10.6.2 on a 2.8 GHz Intel Core 2 Duo with a 4GB RAM. All runs were allocated 1 CPU minute. For each case and nurse count N, we used the *first* 10 instances for each configuration of the NSPLib coverage complexity indicators, that is instances 1–270 for the $N \times 7$ rosters and 1–120 for the $N \times 28$ rosters.

Table 3 summarises the running of these 3120 instances using neither, either, and both of our methods. Each row first indicates the number of known instances of some satisfiability status ('sat' for satisfiable, and 'unsat' for unsatisfiable) for a given case and nurse count N, and then the performance of each method to the first solution, namely the number of instances decided to be of that status without timing out, as well as the total runtime (in seconds) and the total number of backtracks on all instances where *none* of the four methods timed out (it is very important to note that this means that these totals are *comparable*, but also that they do not reveal any performance gains on instances where *at least one* of the methods timed out). Numbers in boldface indicate

Table 3. NSPlib benchmark results

				Neither			String Properties			Cardinality DFA			Both		
Case	N	Status	Known	#Inst	Time	#Bktk	#Inst	Time	#Bktk	#Inst	Time	#Bktk	#Inst	Time	#Bktk
7	25	sat	230	**230**	**16.7**	32099	**230**	42.6	13909	**230**	39.8	13813	**230**	74.8	**13781**
		unsat	38	37	51.9	113413	**38**	57.1	19491	**38**	**37.2**	21133	**38**	57.9	**12877**
7	50	sat	216	213	**9.5**	12165	**216**	24.0	**11055**	214	32.4	11077	**216**	49.8	11057
		unsat	43	40	**55.0**	79629	42	87.5	22082	**43**	107.5	61092	**43**	55.0	**10863**
7	75	sat	210	208	**13.0**	12709	209	22.1	628	**210**	48.8	12421	**210**	49.1	**340**
		unsat	48	**48**	78.5	155490	**48**	**36.3**	8860	**48**	45.3	12455	47	42.0	**8267**
7	100	sat	220	217	**9.0**	361	**219**	30.7	361	217	52.2	**355**	**219**	74.1	**355**
		unsat	26	22	26.3	8909	24	4.9	**452**	23	4.9	993	**25**	**2.8**	**452**
8	25	sat	263	**263**	**2.2**	282	**263**	10.3	282	**263**	14.4	**76**	**263**	22.6	**76**
		unsat	7	**7**	36.2	121367	**7**	**0.0**	**19**	**7**	0.2	**19**	**7**	0.2	**19**
8	50	sat	259	**259**	**4.5**	**136**	**259**	17.3	**136**	**259**	27.8	**136**	**259**	40.8	**136**
		unsat	11	10	28.0	49358	**11**	**3.2**	715	10	58.8	29784	**11**	4.0	**592**
8	75	sat	246	245	**7.2**	449	245	23.4	**230**	**246**	46.2	449	**246**	61.4	**230**
		unsat	22	21	54.4	112880	**22**	**0.1**	**21**	**22**	0.4	53	**22**	0.4	**21**
8	100	sat	262	261	**10.7**	**239**	**262**	32.5	**239**	261	65.5	**239**	**262**	87.9	**239**
		unsat	6	4	0.2	73	**6**	**0.0**	**4**	4	0.4	73	**6**	0.1	**4**
15	30	sat	87	84	**245.3**	**37**	86	257.3	**37**	86	1205.6	**37**	**87**	1219.5	**37**
		unsat	23	9	26.8	2513	**23**	**1.9**	**9**	18	17.9	83	**23**	6.0	**9**
15	60	sat	87	**87**	**361.8**	**131**	**87**	380.4	**131**	**87**	2108.2	**131**	**87**	2137.1	**131**
		unsat	13	8	32.8	1001	**13**	**2.9**	**8**	11	40.9	390	**13**	6.3	**8**
16	30	sat	100	**100**	**567.5**	**153**	**100**	578.6	**153**	**100**	2541.0	**153**	**100**	2557.8	**153**
		unsat	10	4	11.0	172	**10**	**1.4**	**4**	6	68.5	165	**10**	4.9	**4**
16	60	sat	105	**105**	**706.9**	**142**	**105**	722.0	**142**	88	3329.9	**142**	88	3350.2	**142**
		unsat	3	1	25.7	579	**3**	**0.0**	**1**	2	0.8	**1**	**3**	0.8	**1**

best performance in a row. It turned out that Cases 1–6, 9–10, 12–14 are very simple (in the absence of preference constraints), so that our methods only decrease backtracks on one of those 2220 instances, but increase runtime. It also turned out that Case 11 is very difficult (even in the absence of preference constraints), so that even our methods systematically time out, because the product automaton of all row constraints is very big; we could have overcome this obstacle by using the built-in *gcc* constraint and the product automaton of the remaining row constraints, but we wanted to compare all the cases under the same scenario. Hence we do not report any results on Cases 1–6, 9–14.

An analysis of Table 3 reveals that our methods decide more instances without timing out, and that they often drastically reduce the runtime and number of backtracks (by up to four orders of magnitude), especially on the shared unsatisfiable instances. However, runtimes are often increased (by up to one order of magnitude) on the shared satisfiable instances. String properties are only rarely defeated by the cardinality DFA on any of the three performance measures, but their combination is often the overall winner, though rarely by a large margin. A more fine-grained evaluation is necessary to understand when to use which string properties without increasing runtime on the satisfiable instances. The good performance of our methods on unsatisfiable instances is indicative of gains when exploring the whole search space, such as when solving an optimisation problem or using soft (preference) constraints.

With constraint programming, NSPLib instances (without the soft preference constraints) were also used in [4,5], but under row constraints that are different from those of the NSPLib case files that we used. NSP instances from a different repository were used in [11], though with soft global constraints: one of the insights reported there was the need for more interaction between the global constraints, and our paper shows steps that can be taken in that direction.

Since both our methods essentially generate linear constraints, they may also be relevant in the context of linear programming. Future work may also consider the integration of our techniques with the *multicost-regular* constraint [10], which allows the direct handling of a *gcc* constraint in the presence of automaton constraints (as on the rows of NSPLib instances) without explicitly computing the product automaton, which can be very big.

References

1. Beeri, C., Fagin, R., Maier, D., Yannakakis, M.: On the desirability of acyclic database schemes. Journal of the ACM 30, 479–513 (1983)
2. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 107–122. Springer, Heidelberg (2004)
3. Beldiceanu, N., Carlsson, M., Rampon, J.-X.: Global constraint catalog. Technical Report T2005-08, Swedish Institute of Computer Science (2005), The current working version is at: `www.emn.fr/x-info/sdemasse/gccat/doc/catalog.pdf`
4. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: SLIDE: A useful special case of the CARDPATH constraint. In: ECAI 2008, pp. 475–479. IOS Press, Amsterdam (2008)
5. Brand, S., Narodytska, N., Quimper, C.-G., Stuckey, P.J., Walsh, T.: Encodings of the *sequence* constraint. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 210–224. Springer, Heidelberg (2007)
6. Côté, M.-C., Gendron, B., Rousseau, L.-M.: Modeling the *regular* constraint with integer programming. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 29–43. Springer, Heidelberg (2007)
7. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 462–476. Springer, Heidelberg (2002)
8. Frisch, A.M., Jefferson, C., Miguel, I.: Constraints for breaking more row and column symmetries. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 318–332. Springer, Heidelberg (2003)
9. Jukna, S.: Extremal Combinatorics. Springer, Heidelberg (2001)
10. Menana, J., Demassey, S.: Sequencing and counting with the *multicost-regular* constraint. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 178–192. Springer, Heidelberg (2009)
11. Métivier, J.-P., Boizumault, P., Loudni, S.: Solving nurse rostering problems using soft global constraints. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 73–87. Springer, Heidelberg (2009)
12. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004)
13. Régin, J.-C., Gomes, C.: The *cardinality matrix* constraint. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 572–587. Springer, Heidelberg (2004)
14. Vanhoucke, M., Maenhout, B.: On the characterization and generation of nurse scheduling problem instances. European Journal of Operational Research 196(2), 457–467 (2009); NSPLib is at: `www.projectmanagement.ugent.be/nsp.php`

The Increasing Nvalue Constraint

Nicolas Beldiceanu, Fabien Hermenier, Xavier Lorca, and Thierry Petit

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

Abstract. This paper introduces the Increasing_Nvalue constraint, which restricts the number of distinct values assigned to a sequence of variables so that each variable in the sequence is less than or equal to its successor. This constraint is a specialization of the Nvalue constraint, motivated by symmetry breaking. Propagating the Nvalue constraint is known as an NP-hard problem. However, we show that the chain of non strict inequalities on the variables makes the problem polynomial. We propose an algorithm achieving generalized arc-consistency in $O(\Sigma_{\mathrm{Di}})$ time, where Σ_{Di} is the sum of domain sizes. This algorithm is an improvement of filtering algorithms obtained by the automaton-based or the Slide-based reformulations. We evaluate our constraint on a resource allocation problem.

1 Introduction

The Nvalue constraint was introduced by Pachet *et al.* in [10] to express a restriction on the number of distinct values assigned to a set of variables. Even if finding out whether a Nvalue constraint has a solution or not is NP-hard [6], a number of filtering algorithms were developed over the last years [4,3]. Motivated by symmetry breaking, this paper considers the conjunction of an Nvalue constraint with a chain of non strict inequalities constraints, that we call Increasing_Nvalue. We come up with a filtering algorithm that achieves general arc-consistency (GAC) for Increasing_Nvalue in $O(\Sigma_{\mathrm{Di}})$ time, where Σ_{Di} is the sum of domain sizes. This algorithm is more efficient than those obtained by using generic approaches such as encoding Increasing_Nvalue as a finite deterministic automaton [12] or as a Slide constraint [5], which respectively require $O(n(\cup_{Di})^3)$ and $O(nd^4)$ time complexities for achieving GAC, where n denotes the number of variables, $\cup_{Di}$ is the total number of potential values in the domains, and d the maximum size of a domain. Part of its efficiency relies on a specific data structure, i.e. a *matrix of ordered sparse arrays*, which allows multiple ordered queries (*i.e.*, Set and Get) to the columns of a sparse matrix.

Experiments proposed in this paper are based on a real-life resource allocation problem related to the management of clusters. Entropy is a Virtual Machine (VM) manager for clusters [8], which provides an autonomous and flexible engine to manipulate the state and the position of VMs on the different working nodes composing the cluster. The constraint programming part affects the VMs (the tasks) on a reduced number of nodes (the resources) in the cluster. It uses a Nvalue constraint maintaining the number of nodes required to host all the VMs. However, in practice, the resources consumed are most often equivalents from one VM to another. This leads to a limited number of equivalence classes among the VMs, and Increasing_Nvalue is used for breaking such symmetries.

Section 2 recalls some definitions and formally introduces the Increasing_Nvalue constraint. Next, Section 3 describes a necessary and sufficient condition for the feasibility of the Increasing_Nvalue constraint. Section 4 presents an algorithm enforcing GAC for Increasing_Nvalue(N,X). Section 5 evaluates the impact of our method on a resource allocation problem. Finally, Section 6 describes generic approaches for reformulating Increasing_Nvalue that are less efficient.

2 Preliminaries

Given a sequence of variables X, the *domain* $\mathrm{D}(x)$ of a *variable* $x \in X$ is the finite set of integer values that can be assigned to variable x. $\mathcal{D}$ is the union of all domains in X. We use the notations $\min(x)$ for the minimum value of $D(x)$ and $\max(x)$ for the maximum value of $D(x)$. The sum of domains sizes over $\mathcal{D}$ is $\Sigma_{\mathrm{Di}} = \sum_{x_i \in X} |D(x_i)|$. $A[X]$ denotes an assignment of values to variables in X. Given $x \in X$, $A[x]$ is the value of x in $A[X]$. $A[X]$ is *valid* iff $\forall x_i \in X$, $A[x_i] \in \mathrm{D}(x_i)$. An *instantiation* $I[X]$ is a valid assignment of X. Given $x \in X$, $I[x]$ is the value of x in $I[X]$. A *constraint* $C(X)$, specifies the allowed combinations of values for a set of variables X. It defines a subset $\mathcal{R}_C(\mathcal{D})$ of the cartesian product of the domains $\Pi_{x_i \in X} \mathrm{D}(x_i)$. A *feasible instantiation* of $C(X)$ is an instantiation which is in $\mathcal{R}_C(\mathcal{D})$. If $I[X]$ is a feasible instantiation of $C(X)$ then $I[X]$ *satisfies* $C(X)$. W.l.o.g., we consider that X contains at least two variables. Given $X = [x_0, x_1, \ldots, x_{n-1}]$ and i, j two integers such that $0 \leq i < j \leq n-1$, $I[x_i, \ldots, x_j]$ is the projection of $I[X]$ on the sequence $[x_i, \ldots, x_j]$.

Definition 1. *The constraint* Increasing_Nvalue(N, X) *is defined by a variable* N *and a sequence of* n *variables* $X = [x_0, x_1, \ldots, x_{n-1}]$. *Given an instantiation of* $[N, x_0, x_1, \ldots, x_{n-1}]$, Increasing_Nvalue($N, X$) *is satisfied iff:*

1. *N is equal to the number of distinct values assigned to the variables in X.*
2. $\forall i \in [0, n-2]$, $x_i \leq x_{i+1}$.

3 Feasibility of the Increasing Nvalue Constraint

This section presents a necessary and sufficient condition for the feasibility of the Increasing_Nvalue constraint. We first show that the number of distinct values of any instantiation $I[X]$ such that $\forall i \in [0, n-2]$, $I[x_i] \leq I[x_{i+1}]$, is equal to the number of stretches in $I[X]$. A *stretch* [11] is defined as a maximum length sequence of consecutive variables assigned to the same value. For any variable $x \in X$ and any value $v \in D(x)$, we compute the minimum and maximum number of stretches among all possible instantiations $I[X]$ such that $I[x] = v$.

Next, given a variable $x \in X$, we provide the properties linking the natural ordering of values in $D(x)$ and the minimum and maximum number of stretches that can be obtained by assigning a value to x. From these properties, we prove that there exists an instantiation satisfying the constraint for any value of $\mathrm{D}(N)$ between the minimum $\underline{s}(X)$ and the maximum $\overline{s}(X)$ of possible numbers of stretches. This leads to the main result of this section: Increasing_Nvalue(N, X) is feasible iff $\mathrm{D}(N) \cap [\underline{s}(X), \overline{s}(X)] \neq \emptyset$ (Proposition 3 in Section 3.3).

3.1 Estimating the Number of Stretches

Any feasible instantiation $I[X]$ of Increasing_Nvalue(N, X) satisfies $I[x_i] \leq I[x_j]$ for all $i < j$. In the following, an instantiation $I[x_0, x_1, \ldots, x_{n-1}]$ is said to be *well-ordered* iff for i and j s.t. $0 \leq i < j \leq n-1$, we have $I[x_i] \leq I[x_j]$. A value $v \in \mathrm{D}(x)$ is said to be *well-ordered* with respect to x iff it can be part of at least one well-ordered instantiation.

Lemma 1. *Let $I[X]$ be an instantiation. If $I[X]$ satisfies* Increasing_Nvalue(X, N) *then $I[X]$ is well-ordered.*

Proof. From Definition 1, if $I[X]$ satisfies the constraint then $\forall i \in [0, n-2]$, $I[x_i] \leq I[x_{i+1}]$. By transitivity of $\leq$, the Lemma holds. □

Definition 2 (stretch). *Let $I[x_0, x_1, \ldots, x_{n-1}]$ be an instantiation. Given i and j such that $0 \leq i \leq j \leq n-1$, a* stretch *of $I[X]$ is a sequence of consecutive variables $[x_i, \ldots, x_j]$ such that in $I[X]$: (1) $\forall k \in [i, j]$, $\forall \ell \in [i, j]$, $x_k = x_\ell$. (2) either $i = 0$ or $x_{i-1} \neq x_i$. (3) either $j = n-1$ or $x_j \neq x_{j+1}$.*

Lemma 2. *Given a well-ordered instantiation $I[X]$, the number of stretches in $I[X]$ is equal to the number of distinct values in $I[X]$.*

Proof. $I[X]$ is well-ordered then, for any i and j s.t. $0 \leq i < j \leq n-1$, we have $I[x_i] \leq I[x_j]$. Consequently, if x_i and x_j belong to two distinct stretches and $i < j$ then $I[x_i] < I[x_j]$. □

It is possible to evaluate for each value v in each domain $\mathrm{D}(x_i)$ the exact minimum and maximum number of stretches of well-ordered suffix instantiations $I[x_i, \ldots, x_n]$ such that $I[x_i] = v$, and similarly for prefix instantiations. This evaluation is performed w.r.t. the domains of variables x_j such that $j > i$.

Notation 1. *Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables and let v be a value of $\mathcal{D}$. The exact minimum number of stretches among all well-ordered instantiations $I[x_i, \ldots, x_{n-1}]$ such that $I[x_i] = v$ is denoted by $\underline{s}(x_i, v)$. By convention, if $v \notin \mathrm{D}(x_i)$ then $\underline{s}(x_i, v) = +\infty$. Similarly, the exact minimum number of stretches among all well-ordered instantiations $I[x_0, \ldots, x_i]$ such that $I[x_i] = v$ is denoted by $\underline{p}(x_i, v)$. By convention, if $v \notin \mathrm{D}(x_i)$ then $\underline{p}(x_i, v) = +\infty$.*

Lemma 3. *Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables. $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, $\underline{s}(x_i, v)$ can be computed as follows:*

1. *If* $i = n-1$*:* $\underline{s}(x_i, v) = 1$,
2. *If* $i < n-1$*:* $\underline{s}(x_i, v) = \min(\ \underline{s}(x_{i+1}, v),\ \ \min_{w>v}(\underline{s}(x_{i+1}, w)) + 1\ )$.

Proof. By induction. When $|X| = 1$ there is one stretch. Thus, if $i = n-1$, for any $v \in \mathrm{D}(x_i)$, we have $\underline{s}(x_i, v) = 1$. Consider now, a variable x_i, $i < n-1$, and a value $v \in \mathrm{D}(x_i)$. Instantiations s.t. $I[x_{i+1}] < v$ cannot be augmented with value v for x_i to form a well-ordered instantiation $I[x_i, \ldots, x_{n-1}]$. Thus, let $I[x_{i+1}, \ldots, x_{n-1}]$ be an instantiation s.t. $I[x_{i+1}] \geq v$, which minimizes the number of stretches in $[x_{i+1}, \ldots, x_{n-1}]$. Either $I[x_{i+1}] = v$ and $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$ since the first stretch of $I[x_{i+1}, \ldots, x_{n-1}]$ is extended when augmenting $I[x_{i+1}, \ldots, x_{n-1}]$ with value v for x_i, or $I[x_{i+1}] \neq v$ and $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, I[x_{i+1}]) + 1$ since value v creates a new stretch. By construction, instantiations of $[x_{i+1}, \ldots, x_{n-1}]$ that do not minimize the number of stretches cannot lead to a value $\underline{s}(x_i, v)$ strictly less than $\min(\underline{s}(x_{i+1}, w), w > v) + 1$, even if $I[x_{i+1}] = v$. □

Given a sequence of variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, computing $\underline{p}(x_i, v)$ is symmetrical: If $i = 0$: $\underline{p}(x_i, v) = 1$. If $i > 0$: $\underline{p}(x_i, v) = \min(\ \underline{p}(x_{i-1}, v),\ \ \min_{w<v}(\underline{p}(x_{i-1}, w)) + 1\ )$.

Moreover, for a given variable x_i, we evaluate for each value v the exact maximum number of stretches that may appear among all well-ordered instantiations $I[x_i, \ldots, x_{n-1}]$ with $I[x_i] = v$, and similarly for prefix instantiations.

Notation 2. *Let* $X = [x_0, x_1, \ldots, x_{n-1}]$ *be a sequence of variables and let* v *be a value of* $\mathcal{D}$*. The exact maximum number of stretches among all well-ordered instantiations* $I[x_i, \ldots, x_{n-1}]$ *with* $I[x_i] = v$ *is denoted by* $\overline{s}(x_i, v)$*. By convention, if* $v \notin \mathrm{D}(x_i)$ *then* $\overline{s}(x_i, v) = 0$*. Similarly, the exact maximum number of stretches among all well-ordered instantiations* $I[x_1, \ldots, x_i]$ *with* $I[x_i] = v$ *is denoted by* $\overline{p}(x_i, v)$*. By convention, if* $v \notin \mathrm{D}(x_i)$ *then* $\overline{p}(x_i, v) = 0$*.*

Lemma 4. *Let* $X = [x_0, x_1, \ldots, x_{n-1}]$ *be a sequence of variables.* $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, $\overline{s}(x_i, v)$ *can be computed as follows:*

1. *If* $i = n-1$*:* $\overline{s}(x_i, v) = 1$,
2. *If* $i < n-1$*:* $\overline{s}(x_i, v) = \max(\ \overline{s}(x_{i+1}, v),\ \ \max_{w>v}(\overline{s}(x_{i+1}, w)) + 1\ )$.

Proof. Similar to Lemma 3. □

Given a sequence of variables $X = [x_0, x_1, \ldots, x_{n-1}]$, $\forall x_i \in X$, $\forall v \in \mathrm{D}(x_i)$, computing $\overline{p}(x_i, v)$ is symmetrical: If $i = 0$: $\overline{p}(x_i, v) = 1$, If $i > 0$: $\overline{p}(x_i, v) = \max(\ \overline{p}(x_{i-1}, v),\ \ \max_{w<v}(\overline{s}(x_{i-1}, w)) + 1\ )$.

3.2 Properties on the Number of Stretches

This section enumerates the properties that link the natural ordering of values in a domain $D(x_i)$ with the minimum and maximum number of stretches that can be obtained in the sub-sequence $x_i, x_{i+1}, \ldots, x_{n-1}$. We consider only well-ordered values, which may be part of a feasible instantiation of Increasing_Nvalue.

Properties on a Single Value. The next three properties are directly deduced, by construction, from Lemmas 3 and 4.

Property 1. Any value $v \in \mathrm{D}(x_i)$ well-ordered w.r.t. x_i is such that $\underline{s}(x_i, v) \leq \overline{s}(x_i, v)$.

Property 2. Let $v \in \mathrm{D}(x_i)$ $(i < n-1)$ be a value well-ordered w.r.t. x_i. If $v \in \mathrm{D}(x_{i+1})$ and v is well-ordered w.r.t. x_{i+1} then $\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$.

Property 3. Let $v \in \mathrm{D}(x_i)$ $(i < n-1)$ be a value well-ordered w.r.t. x_i. If $v \in \mathrm{D}(x_{i+1})$ and v is well-ordered w.r.t. x_{i+1} then $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, v)$.

Proof. From Lemma 4, if there exists a value $w \in D(x_{i+1})$, $w > v$, which is well-ordered w.r.t. x_{i+1} and s.t. $\overline{s}(x_{i+1}, w) \geq \overline{s}(x_{i+1}, v)$ then $\overline{s}(x_i, v) > \overline{s}(x_{i+1}, v)$. Otherwise, $\overline{s}(x_i, v) = \overline{s}(x_{i+1}, v)$. □

Ordering on Values. The two following properties establish the links between the natural ordering of values in $D(x_i)$ and the minimum and maximum number of stretches in the sub-sequence starting from x_i.

Property 4. Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables and let $i \in [0, n-1]$ be an integer. $\forall v, w \in \mathrm{D}(x_i)$ two well-ordered values, $v \leq w \Rightarrow \underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$.

Proof. If $v = w$ the property holds. If $i = n-1$, by Lemma 3, $\underline{s}(x_{n-1}, v) = \underline{s}(x_{n-1}, w) = 1$. The property holds. Given $i < n-1$, let v', w' be two well-ordered values of $\mathrm{D}(x_{i+1})$ such that $v' \geq v$ and $w' \geq w$, which minimize the number of stretches starting at x_{i+1}: $\forall \alpha \geq v$, $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_{i+1}, \alpha)$ and $\forall \beta \geq w$, $\underline{s}(x_{i+1}, w') \leq \underline{s}(x_{i+1}, \beta)$. Such values exist because v and w are well-ordered values. Then, by construction we have $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_{i+1}, w')$, and, from Lemma 3, $\underline{s}(x_{i+1}, w') \leq \underline{s}(x_i, w)$, which leads to $\underline{s}(x_{i+1}, v') \leq \underline{s}(x_i, w)$. By Lemma 3, $\underline{s}(x_i, v) \leq \underline{s}(x_{i+1}, v') + 1$. Thus, $\underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$. □

A symmetrical property holds on the maximum number of stretches.

Property 5. Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables and $i \in [0, n-1]$ an integer. $\forall v, w \in \mathrm{D}(x_i)$ two well-ordered values, $v \leq w \Rightarrow \overline{s}(x_i, v) \geq \overline{s}(x_i, w)$.

Proof. If $v = w$ the property holds. If $i = n-1$, by Lemma 4, $\overline{s}(x_{n-1}, v) = \overline{s}(x_{n-1}, w) = 1$. The property holds. Given $i < n-1$, let $w' \in \mathrm{D}(x_{i+1})$ be well-ordered, s.t. $w' \geq w$, and maximizing the number of stretches starting at x_{i+1} ($\forall \beta \geq w$, $\overline{s}(x_{i+1}, w') \geq \overline{s}(x_{i+1}, \beta)$). By Lemma 4, $\overline{s}(x_i, w) \leq \overline{s}(x_{i+1}, w') + 1$. Since $v < w$ and thus $v < w'$, $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, w') + 1$. The property holds. □

Ordering on the Maximum Number of Stretches. The intuition of Property 6 stands from the fact that, the smaller a well-ordered value v w.r.t. a variable x_i is, the more stretches one can build on the sequence $[x_i, x_{i+1}, \ldots, x_{n-1}]$ with $x_i = v$.

Property 6. Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables and let i be an integer in interval $[0, n-1]$. $\forall v, w \in \mathrm{D}(x_i)$ two well-ordered values, $\overline{s}(x_i, w) < \overline{s}(x_i, v) \Rightarrow v < w$.

Proof. We show that if $v \geq w$ then, we have a contradiction with $\overline{s}(x_i, w) < \overline{s}(x_i, v)$. If $i = n-1$, Lemma 4 ensures $\overline{s}(x_{n-1}, w) = \overline{s}(x_{n-1}, v) = 1$, a contradiction. Now, let us consider the case where $i < n-1$. If $v = w$ then $\overline{s}(x_i, w) = \overline{s}(x_i, v)$, a contradiction. Otherwise ($v > w$), let v' be a value of $\mathrm{D}(x_{i+1})$ such that $v' \geq v$ which maximizes $\overline{s}(x_{i+1}, \alpha)$, $\alpha \geq v$. Such a value exists because v is well-ordered. By construction $w < v'$. By Lemma 4, $\overline{s}(x_i, w) \geq \overline{s}(x_{i+1}, v') + 1$ (1). By construction we have also $v \leq v'$, which implies $\overline{s}(x_{i+1}, v') + 1 \geq \overline{s}(x_i, v)$ (2). From (1) and (2) we have $\overline{s}(x_i, w) \geq \overline{s}(x_i, v)$, a contradiction. □

Ordering on the Minimum Number of Stretches. There is no implication from the minimum number of stretches to the ordering of values in domains. Let $X = [x_0, x_1, x_2]$ with $D(x_0)$=$D(x_1)$=$\{1, 2, 3\}$ and $D(x_2) = \{1, 2, 4\}$. $\underline{s}(x_0, 1) = 1$ and $\underline{s}(x_0, 3) = 2$, thus $\underline{s}(x_0, 1) < \underline{s}(x_0, 3)$ and $1 < 3$. Consider now that $D(x_2) = \{2, 3, 4\}$. $\underline{s}(x_0, 1) = 2$ and $\underline{s}(x_0, 3) = 1$, thus $\underline{s}(x_0, 3) < \underline{s}(x_0, 1)$ and $3 > 1$.

Summary. Next table summarizes the relations between well-ordered values v and w in $D(x_i)$ and the estimations of the minimum and maximum number of stretches among all instantiations starting from these values (that is, $I[x_i, \ldots, x_{n-1}]$ such that $I[x_i] = v$ or such that $I[x_i] = w$).

Precondition	Property	Proposition
$v \in D(x_i)$ is well-ordered	$\underline{s}(x_i, v) \leq \overline{s}(x_i, v)$	Prop. 1
$v \in D(x_i)$ is well-ordered, $i < n-1$ and $v \in D(x_{i+1})$	$\underline{s}(x_i, v) = \underline{s}(x_{i+1}, v)$ $\overline{s}(x_i, v) \geq \overline{s}(x_{i+1}, v)$	Prop. 2 Prop. 3
$v \in D(x_i)$, $w \in D(x_i)$ are well-ordered and $v \leq w$	$\underline{s}(x_i, v) \leq \underline{s}(x_i, w) + 1$ $\overline{s}(x_i, v) \geq \overline{s}(x_i, w)$	Prop. 4 Prop. 5
$v \in D(x_i)$, $w \in D(x_i)$ are well-ordered and $\overline{s}(x_i, w) < \overline{s}(x_i, v)$	$v < w$	Prop. 6

3.3 Necessary and Sufficient Condition for Feasibility

Notation 3. *Given a sequence of variables* $X = [x_0, x_1, \ldots, x_{n-1}]$*,* $\underline{s}(X)$ *is the minimum value of* $\underline{s}(x_0, v)$*,* $v \in \mathrm{D}(x_0)$*, and* $\overline{s}(X)$ *is the maximum value of* $\overline{s}(x_0, v)$*,* $v \in \mathrm{D}(x_0)$*.*

Proposition 1. *Given an* INCREASING_NVALUE(N, X) *constraint, if* $\underline{s}(X) > \max(\mathrm{D}(N))$ *then the constraint has no solution. Symmetrically, if* $\overline{s}(X) < \min(\mathrm{D}(N))$ *then the constraint has no solution.*

Proof. By construction from Lemmas 3 and 4. □

W.l.o.g., $\mathrm{D}(N)$ can be restricted to $[\underline{s}(X), \overline{s}(X)]$. However, observe that $\mathrm{D}(N)$ may have holes or may be strictly included in $[\underline{s}(X), \overline{s}(X)]$. We prove that for any value k in $[\underline{s}(X), \overline{s}(X)]$ there exists a value $v \in \mathrm{D}(x_0)$ such that $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$. Thus, any value in $D(N) \cap [\underline{s}(X), \overline{s}(X)]$ is feasible.

Proposition 2. *Let $X = [x_0, x_1, \ldots, x_{n-1}]$ be a sequence of variables. For any integer k in $[\underline{s}(X), \overline{s}(X)]$ there exists v in $\mathrm{D}(x_0)$ such that $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$.*

Proof. Let $k \in [\underline{s}(X), \overline{s}(X)]$. If $\exists v \in \mathrm{D}(x_0)$ s.t. $k = \underline{s}(x_0, v)$ or $k = \overline{s}(x_0, v)$ the property holds. Assume $\forall v \in \mathrm{D}(x_0)$, either $k > \overline{s}(x_0, v)$ or $k < \underline{s}(x_0, v)$. Let v', w' be the two values such that v' is the maximum value of $\mathrm{D}(x_0)$ such that $\overline{s}(x_0, v') < k$ and w' is the minimum value such that $k < \underline{s}(x_0, w')$. Then, we have $\overline{s}(x_0, v') < k < \underline{s}(x_0, w')$ (1). By Property 1, $\underline{s}(x_0, w') \leq \overline{s}(x_0, w')$. By Property 6, $\overline{s}(x_0, v') < \overline{s}(x_0, w') \Rightarrow w_0 < v'$. By Properties 4 and 1, $w_0 < v' \Rightarrow \underline{s}(x_0, w') \leq \overline{s}(x_0, v') + 1$, a contradiction with (1). □

Algorithm 1. Building a solution for INCREASING_NVALUE(k, X).

```
1 if $k \notin [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$ then return "no solution" ;
2 $v$ := a value $\in \mathrm{D}(x_0)$ s.t. $k \in [\underline{s}(x_0, v), \overline{s}(x_0, v)]$ ;
3 for $i := 0$ to $n - 2$ do
4     $I[x_i] := v$;
5     if $\forall v_{i+1} \in \mathrm{D}(x_{i+1})$ s.t. $v_{i+1} = v$, $k \notin [\underline{s}(x_{i+1}, v_{i+1}), \overline{s}(x_{i+1}, v_{i+1})]$ then
6         $v := v_{i+1}$ in $\mathrm{D}(x_{i+1})$ s.t. $v_{i+1} > v \wedge k - 1 \in [\underline{s}(x_{i+1}, v_{i+1}), \overline{s}(x_{i+1}, v_{i+1})]$;
7         $k := k - 1$ ;
8 $I[x_{n-1}] := v$; return $I[X]$;
```

Lemma 5. *Given an* INCREASING_NVALUE(N, X) *constraint and an integer k, if $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$ then* Algorithm 1 *returns a solution of* INCREASING_NVALUE(N, X) *with $N = k$. Otherwise,* Algorithm 1 *returns "no solution" since no solution exists with $N = k$.*

Proof. The first line of Algorithm 1 ensures that either $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$ and k belongs to $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N)$, or there is no solution (from Propositions 1 and 2). At each new iteration of the **for** loop, by Lemmas 3 and 4 and Proposition 2, either the condition (line 6) is satisfied and a new stretch begins at $i + 1$ with a greater value (which guarantees that $I[\{x_1, \ldots, x_{i+1}\}]$ is well-ordered) and k is decreased by 1, or it is possible to keep the current value v for $I[x_{i+1}]$. Therefore, at the start of a **for** loop (line 4), $\exists v \in \mathrm{D}(x_i)$ s.t. $k \in [\underline{s}(x_i, v), \overline{s}(x_i, v)]$. When $i = n - 1$, by construction $k = 1$ and $\forall v_{n-1} \in \mathrm{D}(x_{n-1})$, $\underline{s}(x_{n-1}, v_{n-1}) = \overline{s}(x_{n-1}, v_{n-1}) = 1$; $I[X]$ is well-ordered and contains k stretches. From Lemma 2, instantiation $I[\{N\} \cup X]$ with $I[N] = k$ is a solution of INCREASING_NVALUE(N, X) with k distinct values in X. □

Lemma 5 leads to a necessary and sufficient feasibility condition.

Proposition 3. *Given an* INCREASING_NVALUE(N, X) *constraint, the two following propositions are equivalent:*

1. INCREASING_NVALUE(N, X) *has a solution.*
2. $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$.

Proof. ($\Rightarrow$) Assume Increasing_Nvalue(N, X) has a solution. Let $I[\{N\} \cup X]$ be such a solution. By Lemma 2 the value k assigned to N is the number of stretches in $I[X]$. By construction (Lemmas 3 and 4) $k \in [\underline{s}(X), \overline{s}(X)]$. Thus, $[\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$. ($\Leftarrow$) Let $k \in [\underline{s}(X), \overline{s}(X)] \cap \mathrm{D}(N) \neq \emptyset$. From Lemma 5 it is possible to build a feasible solution for Increasing_Nvalue(N, X). □

4 GAC Filtering Algorithm for Increasing Nvalue

This section presents an algorithm enforcing GAC for Increasing_Nvalue(N, X) in $O(\Sigma_{\mathrm{Di}})$ time complexity, where Σ_{Di} is the sum of domain sizes of the variables in X. For a given variable $x_i \in X$ and a value $v \in D(x_i)$, the principle is to estimate the minimum and maximum number of stretches among all instantiations $I[X]$ with $I[x_i] = v$, to compare the interval derived from these two bounds and $D(N)$. In order to do so, w.l.o.g. we estimate the minimum and maximum number of stretches related to prefix instantiations $I[x_0, \ldots, x_i]$ and suffix instantiations $I[x_i, \ldots, x_{n-1}]$.

Definition 3 (GAC). *Let $C(X)$ be a constraint. A* **support** *on $C(X)$ is an instantiation $I[X]$ which satisfies $C(X)$. A domain $\mathrm{D}(x)$ is* **arc-consistent** *w.r.t. $C(X)$ iff $\forall v \in \mathrm{D}(x)$, v belongs to a support on $C(X)$. $C(X)$ is* **(generalized) arc-consistent** *(GAC) iff $\forall x_i \in X$, $\mathrm{D}(x_i)$ is arc-consistent.*

4.1 Necessary and Sufficient Condition for Filtering

From Lemma 5, values of $D(N)$ which are not in $[\underline{s}(X), \overline{s}(X)]$ can be removed from $D(N)$. By Proposition 3, all remaining values in $D(N)$ are feasible. We now give a necessary and sufficient condition to remove a value from $D(x_i)$, $x_i \in X$.

Proposition 4. *Consider an* Increasing_Nvalue(N, X) *constraint. Let $i \in [0, n-1]$ be an integer and v a value in $D(x_i)$. The two following propositions are equivalent:*

1. *$v \in D(x_i)$ is arc-consistent w.r.t.* Increasing_Nvalue
2. *v is well-ordered w.r.t. $D(x_i)$ and $[\underline{p}(x_i, v) + \underline{s}(x_i, v) - 1, \overline{p}(x_i, v) + \overline{s}(x_i, v) - 1] \cap \mathrm{D}(N) \neq \emptyset$.*

Proof. If v is not well-ordered then from Lemma 1, v is not arc-consistent w.r.t. Increasing_Nvalue. Otherwise, $\underline{p}(x_i, v)$ is the exact minimum number of stretches among well-ordered instantiations $I[x_0, \ldots, x_i]$ such that $I[x_i] = v$ and $\underline{s}(x_i, v)$ is the exact minimum number of stretches among well-ordered instantiations $I[x_i, \ldots, x_{n-1}]$ such that $I[x_i] = v$. Thus, by construction $\underline{p}(x_i, v) + \underline{s}(x_i, v) - 1$ is the exact minimum number of stretches among well-ordered instantiations $I[x_0, x_1, \ldots, x_{n-1}]$ such that $I[x_i] = v$. Let $\mathcal{D}_v \subseteq \mathcal{D}$ be the set of domains such that all domains in $\mathcal{D}_v$ are equal to domains in $\mathcal{D}$ except $D(x_i)$ which is reduced to $\{v\}$. We call X_v the set of variables associated with domains in $\mathcal{D}_v$. From Definition 3, $\underline{p}(x_i, v) + \underline{s}(x_i, v) - 1 = \underline{s}(X_v)$. By a symmetrical reasoning, $\overline{p}(x_i, v) + \overline{s}(x_i, v) - 1 = \overline{s}(X_v)$. By Proposition 3, the proposition holds. □

4.2 Algorithms

From Proposition 4, we derive a filtering algorithm achieving GAC in $O(\Sigma_{\mathrm{Di}})$. For a given variable x_i $(0 \leq i < n)$, we need to compute the prefix and suffix information $\underline{p}(x_i, v)$, $\overline{p}(x_i, v)$, $\underline{s}(x_i, v)$ and $\overline{s}(x_i, v)$, no matter whether value v belongs or not to the domain of x_i. To reach an overall complexity of $O(\Sigma_{\mathrm{Di}})$, we take advantage of two facts:

1. Within our algorithm we always iterate over $\underline{p}(x_i, v)$, $\overline{p}(x_i, v)$, $\underline{s}(x_i, v)$ and $\overline{s}(x_i, v)$ by scanning the value of $D(x_i)$ in increasing or decreasing order.
2. For a value v that does not belong to $D(x_i)$, 0 (resp. n) is the default value for $\overline{p}(x_i, v)$ and $\overline{s}(x_i, v)$ (resp. $\underline{p}(x_i, v)$ and $\underline{s}(x_i, v)$).

For this purpose we create a data structure for handling such sparse matrices for which write and read accesses are always done by iterating in increasing or decreasing order through the rows in a given column. The upper part of next table describes the three primitives on *ordered sparse matrices* as well as their time complexity. The lower part gives the primitives used for accessing or modifying the domain of a variable. Primitives which restrict the domain of a variable x return true if $D(x) \neq \emptyset$ after the operation, false otherwise.

Primitives (*access to matrices*)	**Description**	**Complexity**
SCANINIT($mats, i, dir$)	indicates that we will iterate through the i^{th} column of matrices in $mats$ in increasing order ($dir = \uparrow$) or decreasing order ($dir = \downarrow$)	$O(1)$
SET($mat, i, j, info$)	performs the assignment $mat[i, j] := info$	$O(1)$
GET(mat, i, j):int	returns the content of entry $mat[i, j]$ or the default value if such entry does not belong to the sparse matrix (a set of q consecutive calls to GET on the same column i and in increasing or decreasing row indexes is in $O(q)$)	amortized

Primitives (*access to variables*)	**Description**	**Complexity**
ADJUST_MIN(x, v):boolean	adjusts the minimum of var. x to value v	$O(1)$
ADJUST_MAX(x, v):boolean	adjusts the maximum of var. x to value v	$O(1)$
REMOVE_VAL(x, v):boolean	removes value v from domain $D(x)$	$O(1)$
INSTANTIATE(x, v):boolean	fix variable x to value v	$O(1)$
GET_PREV(x, v):int	returns the largest value w in $D(x)$ such that $w < v$ if it exists, returns v otherwise	$O(1)$
GET_NEXT(x, v):int	returns the smallest value w in $D(x)$ such that $w > v$ if it exists, returns v otherwise	$O(1)$

Algorithm 3 corresponds to the main filtering algorithm that implements Proposition 4. In a first phase it restricts the minimum and maximum values of variables $[x_0, x_1, \ldots, x_{n-1}]$ w.r.t. to all the inequalities constraints (i.e. it only keeps well-ordered values). In a second step, it computes the information related

Algorithm 2. BUILD_SUFFIX($[x_0, x_1, \dots, x_{n-1}], \underline{s}[][], \overline{s}[][]$).

```
ALLOCATE $mins, maxs$;
SCANINIT($\{\underline{s}, \overline{s}\}, n-1, \downarrow$); $v := \max(x_{n-1})$;
repeat
    SET($\underline{s}, n-1, v, 1$); SET($\overline{s}, n-1, v, 1$); $w := v$; $v :=$getPREV($x_{n-1}, v$);
until $w = v$ ;
for $i := n-2$ downto $0$ do
    SCANINIT($\{\underline{s}, \overline{s}, mins, maxs\}, i+1, \downarrow$); $v := \max(x_{i+1})$;
    repeat
        if $v < \max(x_{i+1})$ then
            SET($mins, i+1, v, \min($GET($mins, i+1, v+1$),GET($\underline{s}, i+1, v$)$)$);
            SET($maxs, i+1, v, \max($GET($maxs, i+1, v+1$),GET($\overline{s}, i+1, v$)$)$);
        else
            SET($mins, i+1, v,$GET($\underline{s}, i+1, v$));
            SET($maxs, i+1, v,$GET($\overline{s}, i+1, v$));
        $w := v$; $v :=$getPREV($x_{i+1}, v$);
    until $w = v$ ;
    SCANINIT($\{\underline{s}, \overline{s}\}, i, \downarrow$); SCANINIT($\{\underline{s}, \overline{s}, mins, maxs\}, i+1, \downarrow$); $v := \max(x_i)$;
    repeat
        if $v = \max(x_{i+1})$ then
            SET($\underline{s}, i, v,$GET($\underline{s}, i+1, v$)); SET($\overline{s}, i, v,$GET($\overline{s}, i+1, v$));
        else
            if $v \geq \min(x_{i+1})$ then
                SET($\underline{s}, i, v, \min($GET($\underline{s}, i+1, v$),GET($mins, i+1, v+1$)$+1)$);
                SET($\overline{s}, i, v, \max($GET($\overline{s}, i+1, v$),GET($maxs, i+1, v+1$)$+1)$);
            else
                SET($\underline{s}, i, v,$GET($mins, i+1, \min(x_{i+1})$)$+1$);
                SET($\overline{s}, i, v,$GET($maxs, i+1, \min(x_{i+1})$)$+1$);
        $w := v$; $v :=$getPREV($x_i, v$);
    until $w = v$ ;
```

to the minimum and maximum number of stretches on the prefix and suffix matrices $\underline{p}$, $\overline{p}$, $\underline{s}$, $\overline{s}$. Finally, based on this information, it adjusts the bounds of N and does the necessary pruning on each variable $x_0, x_1, \dots, x_{n-1}$. Using Lemmas 3 and 4, Algorithm 2 builds the suffix matrices $\underline{s}$ and $\overline{s}$ used in Algorithm 3 ($\underline{p}$ and $\overline{p}$ are constructed in a similar way):

1. In a first step, column $n-1$ of matrices $\underline{s}$ and $\overline{s}$ are initialised to 1 (i.e. see the first item of Lemmas 3 and 4).
2. In a second step, columns $n-2$ down to 0 are initialised (i.e. see the second item of Lemmas 3 and 4). In order to avoid recomputing from scratch the quantities $\min(\underline{s}(x_{i+1}, v), \min_{w>v}(\underline{s}(x_{i+1}, w)) + 1)$ and $\max(\overline{s}(x_{i+1}, v), \max_{w>v}(\overline{s}(x_{i+1}, w)) + 1)$ we introduce two sparse ordered matrices $mins[i,j]$ and $maxs[i,j]$. When initialising the i^{th} columns of matrices $\underline{s}$ and $\overline{s}$ we first compute the $i+1^{th}$ columns of matrices $mins$ and $maxs$ (i.e. see the first **repeat** of the **for** loop). Then, in the second **repeat**

of the **for** loop we initialise the i^{th} columns of $\underline{s}$ and $\overline{s}$. Observe that we scan columns $i+1$ of matrices *mins* and *maxs* in decreasing rows indices.

Consequently, Algorithm 2 takes $O(\Sigma_{\rm Di})$ time and Algorithm 3 prunes all the values that are not arc-consistent in INCREASING_NVALUE in $O(\Sigma_{\rm Di})$.[1]

Algorithm 3. INCREASING_NVALUE($N, [x_0, x_1, \ldots, x_{n-1}]$) : boolean.

```
1  if $n = 1$ then return INSTANTIATE($N, 1$);
2  for $i = 1$ to $n - 1$ do if $\neg$ADJUST_MIN($x_i, \min(x_{i-1})$) then return false;
3  for $i = n - 2$ downto $0$ do if $\neg$ADJUST_MAX($x_i, \max(x_{i+1})$) then return false;
4  ALLOCATE $\underline{p}, \overline{p}, \underline{s}, \overline{s}$;
5  BUILD_PREFIX $\underline{p}, \overline{p}$; BUILD_SUFFIX $\underline{s}, \overline{s}$;
6  SCANINIT($\{\underline{s}, \overline{s}\}, 0, \uparrow$);
7  if $\neg$ADJUST_MIN($N, \min_{v \in D(x_0)}$(GET($\underline{s}, 0, v$))) then return false;
8  if $\neg$ADJUST_MAX($N, \max_{v \in D(x_0)}$(GET($\overline{s}, 0, v$))) then return false;
9  for $i := 0$ to $n - 1$ do
10     SCANINIT($\{\underline{p}, \overline{p}, \underline{s}, \overline{s}\}, i, \uparrow$); $v := \min(x_i)$;
11     repeat
12         $\underline{N}_v$ :=GET($\underline{p}, i, v$)+GET($\underline{s}, i, v$) $- 1$; $\overline{N}_v$ :=GET($\overline{p}, i, v$)+GET($\overline{s}, i, v$) $- 1$;
13         if $[\underline{N}_v, \overline{N}_v] \cap D(N) = \emptyset$ and $\neg$REMOVE_VAL($x_i, v$) then return false;
14         $w := v$; $v :=$ GETNEXT($x_i, v$);
15     until $w = v$ ;
16 return true ;
```

5 Using Increasing Nvalue for Symmetry Breaking

This section provides a set of experiments for the INCREASING_NVALUE constraint. First, Section 5.1 presents a constraint programming reformulation of a NVALUE constraint into a INCREASING_NVALUE constraint to deal with symmetry breaking. Next, Section 5.2 evaluates the INCREASING_NVALUE on a real life application based on constraint programming technology. In the following, all experiments were performed with the Choco constraint programming system [1], on an Intel Core 2 Duo 2.4GHz with 4GB of RAM, and 128Mo allocated to the Java Virtual Machine.

5.1 Improving NVALUE Constraint Propagation

Enforcing GAC for a NVALUE constraint is a NP-Hard problem and existing filtering algorithms perform little propagation when domains of variables are sparse [3,4]. In our implementation, we use a representation of NVALUE which is based on occurrence constraints of Choco. We evaluate the effect of the INCREASING_NVALUE constraint when it is used as an implied constraint on equivalence

[1] The source code of the INCREASING_NVALUE constraint is available at `http://choco.emn.fr`

Table 1. Evaluation of the INCREASING_NVALUE constraint according the number of equivalence classes among the variables

Number of	NVALUE model				INCREASING_NVALUE model			
equivalences	nodes	failures	time(ms)	solved(%)	nodes	failures	time(ms)	solved(%)
1	2798	22683	6206	76	**28**	**0**	**51**	**100**
3	1005	12743	4008	76	**716**	**7143**	**3905**	**82**
5	1230	14058	**8077**	72	**1194**	**12067**	8653	72
7	850	18127	**6228**	64	**803**	**16384**	6488	**66**
10	387	3924	**2027**	58	387	**3864**	2201	58
15	1236	16033	**6518**	38	**1235**	**16005**	7930	38
20	379	7296	**5879**	58	379	7296	6130	58

classes, in addition to the NVALUE. Thus, given a set $\mathcal{E}(X)$ of equivalence classes among the variables in X, the pruning of the global constraint NVALUE(X, N) can be strengthened in the following way:

$$Nvalue(N, X) \tag{1}$$

$$\forall E \in \mathcal{E}(X), Increasing\_Nvalue(N_E, E) \tag{2}$$

$$\max_{E \in \mathcal{E}(X)} (N_E) \leq N \leq \sum_{E \in \mathcal{E}(X)} (N_E) \tag{3}$$

where N_E denotes the occurrence variable associated to the set of equivalent variables $E \in \mathcal{E}(X)$ and $E \subseteq X$.

Parameters recorded are the number of nodes in the tree search, the number of fails detected during the search and the solving time to reach a solution. Variables of our experiments are the maximum number of values in the variable domains, the percentage of holes in the variable domains and the number of equivalence classes among the variables. The behavior of our experiments is not related to the number of variables: sizes 20, 40 and 100 have been evaluated.

Tables 1 and 2 report the results of experiments for 40 variables and domains containing at most 80 values (size 20 and 40 are also tested). For Table 1, 50 instances are generated for each size of equivalence classes. For Table 2, 350 instances are generated for each density evaluated. A timeout on the solving time to a solution is fixed to 60 seconds. A recorded parameter is included in the average iff both approaches solve the instance. Then, two approaches are strictly comparable if the percentage of solved instances is equal. Otherwise, the recorded parameters can be compared for the instances solved by both approaches.

Table 1 illustrates that equivalence classes among the variables impact the performances of the INCREASING_NVALUE constraint model. We observe that the performances (particularly the solving time) are impacted by the number of equivalence classes. From one equivalence class to 7, the average number of variables involved in each equivalence class is sufficient to justify the solving time overhead which is balanced by the propagation efficiency. From 10 to 20, the size of each equivalence class is not significant (in the mean, from 4 to 2 variables involved in each INCREASING_NVALUE constraint). Thus, we show that

Table 2. Evaluation of the INCREASING_NVALUE constraint according the percentage of holes in the domains

holes(%)	NVALUE model				INCREASING_NVALUE model			
	nodes	failures	time(ms)	solved(%)	nodes	failures	time(ms)	solved(%)
25	1126.4	13552	5563.3	63.1	**677.4**	**8965.5**	**5051.1**	**67.7**
50	2867.1	16202.6	**4702.1**	50.8	**1956.4**	**12345**	4897.5	**54.9**
75	5103.7	16737.3	**3559.4**	**65.7**	**4698.7**	**15607.8**	4345.5	65.1

the propagation gain (in term of nodes and failures) is not significant while the solving time overhead could be important.

Unsurprisingly, Table 2 shows that the number of holes in the variable domains impact the performances of the INCREASING_NVALUE constraint model. However, we notice when the number of holes in the domains increases the number of solved instances decreases. Such a phenomenon are directly related with the fact that propagation of NVALUE is less efficient when there exist holes in the variable domains.

5.2 Integration in a Resource Scheduling Problem

Entropy[2] [8] provides an autonomous and flexible engine to manipulate the state and the position of VMs (hosting applications) on the different working nodes composing the cluster. This engine is based on Constraint Programing. It provides a core model dedicated to the assignment of VMs to nodes and some dedicated constraints to customize the assignment of the VMs regarding to some users and administrators requirements.

The core model denotes each node (the resources) by its CPU and memory capacity and each VM (the tasks) by its CPU and memory demands to run at a peak level. The constraint programming part aims at computing an assignment of each VM that (i) satisfies the resources demand (CPU and memory) of the VMs, and (ii) uses a minimum number of nodes. Finally, liberating nodes can allow more jobs to be accepted into the cluster, or can allow powering down unused nodes to save energy. In this problem two parts can be distinguished: (i) VMs assignment on nodes w.r.t. resource capacity: this is a bidimensional bin-packing problem. It is modeled by a set of knapsack constraints associated with each node. Propagation algorithm is based on COSTREGULAR propagator [7] to deal with the two dimensions of the resource; (ii) Restriction on the number of nodes used to assign all the VMs. VMs are ranked according to their CPU and memory consumption (this means there is equivalence classes among the VMs). NVALUE and INCREASING_NVALUE are used (Section 5.1) to model this part.

In practice, the results obtained by the INCREASING_NVALUE constraint evaluation, within the constraint programming module of Entropy, point out a short gain in term of solving time (3%), while the gain in term of nodes and failures is more significant (in the mean 35%). Such a gap is due to the tradeoff between the propagation gain (filtered values) and solving time induced by the algorithm.

[2] `http://entropy.gforge.inria.fr`

6 Related Work

GAC for the Increasing_Nvalue constraint can be also obtained by at least two different generic techniques, namely by using a finite deterministic automaton with a polynomial number of transitions or by using the Slide constraint.

Given a constraint C of arity k and a sequence X of n variables, the Slide(C,X) constraint [5] is a special case of the *cardpath* constraint. The *slide* constraint holds iff $C(X_i, X_{i+1}, \ldots, X_i + k - 1)$ holds for all $i \in [1, n - k + 1]$. The main result is that GAC can be enforced in $O(nd^k)$ time where d is the maximum domain size. An extension called $slide_j(C,X)$ holds iff $C(X_{ij+1}, X_{ij+2}, \ldots, X_{ij+k})$ holds for all $i \in [0, \frac{n-k}{j}]$. Given $X = \{x_i \mid i \in [1;n]\}$, the Increasing_Nvalue constraint can be encoded as $\text{Slide}_2(C,[x_i, c_i]_{i\in[1;n]})$ where (a) $c_1, c_2, \ldots, c_n$ are variables taking their value within $[1, n]$ with $c_1 = 1$ and $c_n = N$, and (b) $C(x_i, c_i, x_{i+1}, c_{i+1})$ is the constraint $b \Leftrightarrow x_i \neq x_{i+1} \wedge c_{i+1} = c_i + b \wedge x_i \leq x_{i+1}$. This leads to a time bound of $O(nd^4)$ for achieving GAC on the Increasing_Nvalue constraint.

The reformulation based on finite deterministic automaton is detailed in the global constraint catalog[2]. If we use Pesant's algorithm [12], this reformulation leads to a worst-case time complexity of $O(n{\cup_{Di}}^3)$ for achieving GAC, where $\cup_{Di}$ denotes the total number of potential values in the variable domains.

7 Conclusion

Motivated by symmetry breaking, we provide a filtering technique that achieves GAC for a specialized case of the Nvalue constraint where the decision variables are constrained by a chain of non strict inequalities. While finding out whether a Nvalue constraint has a solution or not is NP-hard, our algorithm has a linear time complexity w.r.t. the sum of the domain sizes. We believe that the data structure on matrices of ordered sparse arrays may be useful for decreasing the time worst-case complexity of other filtering algorithms.

Future work may also improve the practical speed of the Increasing_Nvalue constraint by somehow merging consecutive values in the domain of a variable. More important, this work follows the topic of integrating common symmetry breaking constraints directly within core global constraints [9,13].

References

1. Choco: An open source Java CP library, documentation manual (2009), http://choco.emn.fr/
2. Beldiceanu, N., Carlsson, M., Rampon, J.-X.: Global constraint catalog, working version of January 2010. Technical Report T2005-08, Swedish Institute of Computer Science (2005), www.emn.fr/x-info/sdemasse/gccat
3. Beldiceanu, N., Carlsson, M., Thiel, S.: Cost-Filtering Algorithms for the two Sides of the sum of weights of distinct values Constraint. Technical report, Swedish Institute of Computer Science (2002)

4. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: Filtering Algorithms for the *nvalue* Constraint. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 79–93. Springer, Heidelberg (2005)
5. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: SLIDE: A useful special case of the CARDPATH constraint. In: ECAI 2008, Proceedings, pp. 475–479 (2008)
6. Bessière, C., Hebrard, E., Hnich, B., Walsh, T.: The Complexity of Global Constraints. In: 19th National Conference on Artificial Intelligence (AAAI 2004), pp. 112–117. AAAI Press, Menlo Park (2004)
7. Demassey, S., Pesant, G., Rousseau, L.-M.: A cost-regular based hybrid column generation approach. Constraints 11(4), 315–333 (2006)
8. Hermenier, F., Lorca, X., Menaud, J.-M., Muller, G., Lawall, J.: Entropy: a consolidation manager for clusters. In: VEE 2009: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, pp. 41–50 (2009)
9. Katsirelos, G., Narodytska, N., Walsh, T.: Combining Symmetry Breaking and Global Constraints. In: Oddi, A., Fages, F., Rossi, F. (eds.) CSCLP 2008. LNCS, vol. 5655, pp. 84–98. Springer, Heidelberg (2009)
10. Pachet, F., Roy, P.: Automatic Generation of Music Programs. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 331–345. Springer, Heidelberg (1999)
11. Pesant, G.: A filtering algorithm for the stretch constraint. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 183–195. Springer, Heidelberg (2001)
12. Pesant, G.: A Regular Language Membership Constraint for Finite Sequences of Variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004)
13. Ågren, M., Beldiceanu, N., Carlsson, M., Sbihi, M., Truchet, C., Zampelli, S.: Six Ways of Integrating Symmetries within Non-Overlapping Constraints. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 11–25. Springer, Heidelberg (2009)

Improving the Held and Karp Approach with Constraint Programming[⋆]

Pascal Benchimol[1], Jean-Charles Régin[2],
Louis-Martin Rousseau[1], Michel Rueher[2], and Willem-Jan van Hoeve[3]

[1] CIRRELT, École Polytechnique de Montréal
[2] Université de Nice - Sophia Antipolis / CNRS
[3] Tepper School of Business, Carnegie Mellon University

1 Introduction

Held and Karp have proposed, in the early 1970s, a relaxation for the Traveling Salesman Problem (TSP) as well as a branch-and-bound procedure that can solve small to modest-size instances to optimality [4, 5]. It has been shown that the Held-Karp relaxation produces very tight bounds in practice, and this relaxation is therefore applied in TSP solvers such as Concorde [1]. In this short paper we show that the Held-Karp approach can benefit from well-known techniques in Constraint Programming (CP) such as domain filtering and constraint propagation. Namely, we show that filtering algorithms developed for the weighted spanning tree constraint [3, 8] can be adapted to the context of the Held and Karp procedure. In addition to the adaptation of existing algorithms, we introduce a special-purpose filtering algorithm based on the underlying mechanisms used in Prim's algorithm [7]. Finally, we explored two different branching schemes to close the integrality gap. Our initial experimental results indicate that the addition of the CP techniques to the Held-Karp method can be very effective.

The paper is organized as follows: section 2 describes the Held-Karp approach while section 3 gives some insights on the Constraint Programming techniques and branching scheme used. In section 4 we demonstrate, through preliminary experiments, the impact of using CP in combination with Held and Karp based branch-and-bound on small to modest-size instances from the TSPlib.

2 The Held-Karp Approach

Let $G = (V, E)$ be a complete graph with vertex set $\{1, 2, \ldots, n\}$. We let c_{ij} denote the cost of edge $(i, j) \in E$. The cost function extends to any subset of edges by summing their costs. The Traveling Salesman Problem (TSP) asks for a closed tour in G, visiting each vertex exactly once, with minimum cost.

[4, 5] introduced the so-called *1-tree* as a relaxation for the TSP. A 1-tree is defined as a tree on the set of vertices $\{2, \ldots, n\}$, together with two distinct

⋆ This work was partially supported by the European Community's 7th Framework Programme (FP7/2007-2013). It was started when L.-M. Rousseau and W.-J. van Hoeve were visiting the University of Nice-Sophia Antipolis (June/July 2009).

edges incident to vertex 1. The degree of a vertex is the set of edges in the 1-tree incident to that vertex, and we denote it by $\deg(i)$ for $i \in V$. To see that the 1-tree is a relaxation for the TSP, observe that every tour in the graph is a 1-tree, and if a minimum-weight 1-tree is a tour, it is an (optimal) solution to the TSP. Note that the 1-tree is a tour if and onlng y if all the degree of vertices is two.

The iterative approach proposed by [4, 5], uses Lagrangian relaxation to produce a sequence of connected graphs which increasingly resemble tours. We start by computing an initial minimum-weight 1-tree, by computing a minimum-spanning tree on $G \setminus \{1\}$, and adding the two edges with lowest cost incident to vertex 1. If the optimal 1-tree is a tour, we have found an optimal tour. Otherwise, the degree constraint one some of the vertices must be violated, i.e., it is not equal to two. In that case, we proceed by penalizing the degree of such vertices to be different from two by perturbing the edge costs of the graph, as follows. For each vertex $i \in V$, a 'node potential' π_i is introduced, Then, for each edge $(i, j) \in E$, the edge weight $\tilde{c}_{ij}$ is defined as $\tilde{c}_{ij} = c_{ij} + \pi_i + \pi_j$. [4] show that the optimal TSP tour is invariant under these changes, but the optimal 1-tree is not. Once choice for the node potentials is to define $\pi_i = (2 - \deg(i)) \cdot C$, for a fixed constant C. The Held-Karp procedure re-iterates by solving the 1-tree problem and perturbing the edge costs until it reaches a fixed point or meets a stopping criterion. The best lower bound, i.e., the maximum among all choices of the node potentials, is known as the Held-Karp bound and will be denoted by HK.

The overall Held-Karp approach solves the TSP through branch-and-bound, a technique that has been widely used on this problem (see [2] for a survey). A good upper bound, UB, can be computed easily with any of the popular heuristics that have been devised for this problem, e.g., [6].

3 Improving the Approach Using CP

In this section we describe the different refinements introduced to the original Held-Karp approach [4, 5], which consist of two filtering procedures based on the weighted minimum spanning tree (or 1-tree), and one based on the underlying structure of Prim's algorithm.

In the following procedures let T be a minimum 1-tree of G computed by the Held and Karp relaxation described above. For a subset of edges $S \subseteq E$, we let $w(S)$ denote $\sum_{e \in S} c_e$ and $T(e)$ be the minimum 1-tree where e is forced into T.

We note that the filtering in subsection 3.1 has been applied to the weighted minimum spanning tree constraint in [3, 8], and the filtering in subsection 3.2 has been applied to the weighted minimum spanning tree constraint in [3].

3.1 Removing Edges Based on Marginal Costs

The *marginal cost* of an edge e in T is defined as $c'_e = w(T(e)) - w(T)$, that is, the marginal increase of the weight of the minimum 1-tree if e is forced in the 1-tree.

The following algorithm can compute, in $O(mn)$, the marginal costs for edges $e \notin T$. Each non-tree edge $e = (i, j)$ links two nodes i, j, and defines a unique i-j path, say P^e, in T. The replacement cost of (i, j) is defined by $c_e - max(c_a | a \in P^e)$, that is the cost of (i, j) minus the cost of largest edge on the path from i to j in the 1-tree T. Finding P^e can be achieved through DFS in $O(n)$ for all the $O(m)$ edges not in T. If $\text{HK} + c'_e > \text{UB}$, then e can be safely removed from E.

3.2 Forcing Edges Based on Replacement Costs

Conversely, it is possible to compute the *replacement cost* of an edge $e \in T$ as the increase the Held-Karp bound would incur if e would be removed from E, which we define by $c^r_e = w(T \setminus e) - w(T)$.

This computation can be performed for all edges $e \in T$, with the following algorithm: a) set all $c^r_e = \infty \ \forall e \in T$ b) for all $e = (i, j) \notin T$ identify the i-j path P^e in T which joins the end-points of e. Update all edges $a \in P^e$ such that $c^r_a = min(c^r_a, c_e - c_a)$. This computation can be performed in $O(mn)$, or, at no extra cost if performed together with the computation of marginal costs. If $\text{HK} + c^r_e - c_e > \text{UB}$, then e is a mandatory edge in T.

We note that such filtering has been applied to the weighted minimum spanning tree constraint by [3, 8].

3.3 Forcing Edges Based during MST Computation

Recall that Prim's algorithm computes the minimum spanning tree in G (which is easily transformed into a 1-tree) in the following manner. Starting from any node i, it first partitions the graph into disjoints subsets $S = \{i\}$ and $\bar{S} = V \setminus i$ and creates an empty tree T. Then it iteratively adds to T the minimum edge $(i, j) \in (S, \bar{S})$, defined as the set of edges where $i \in S$ and $j \in \bar{S}$, and moves j from $\bar{S}$ to S.

Since we are using MST computations as part of a Held-Karp relaxation to the TSP, we know that there should be at least 2 edges in each possible $(S, \bar{S})$ of V (this property defines one of well known subtour elimination constraints of the TSP). Therefore, whenever we encounter a set $(S, \bar{S})$ that contains only two edges during the computation of the MST with Prim's algorithm, we can force these edges to be mandatory in T.

3.4 Tuning the Propagation Level

The proposed filtering procedures are quite expensive computationally, therefore it is interesting to investigate the amount of propagation that we wish to impose during the search. A first implementation consists in calling each filtering algorithm (as defined in sections 3.1, 3.2 and 3.3) only once before choosing a new branching variable. A second approach would be to repeat these rounds of propagation until none of these procedures is able to delete nor force any edge, that is reaching a fixed point. Finally, if reaching a fixed point allows to reduce the overall search effort, a more efficient propagation mechanism could be developed in order to speed up its computation.

3.5 Choosing the Branching Criterion

Once the initial Held-Karp bound has been computed and the filtering has been performed it is necessary to apply a branching procedure in order to identify the optimal TSP solution. We have investigated two orthogonal branching schemes, both based on the 1-tree associated to the best Held-Karp bound, say T. These strategies consist in selecting, at each branch-and-bound node, one edge e and splitting the search in two subproblems, one where e is forced in the solution and one where it is forbidden. In the strategy *out* we pick $e \in T$ and first branch on the subproblem where it is forbidden while in the strategy *in* we choose $e \notin T$ and first try to force it in the solution.

Since there are $O(n)$ edges in T and $O(n^2)$ edges not in T, the first strategy will tend to create search trees which are narrower but also deeper than the second one. However, since the quality of the HK improves rapidly as we go down the search tree, it is generally possible to cut uninteresting branches before we get to deep. Preliminary experiments, not reported here, have confirmed that strategy *out* is generally more effective than strategy *in*.

Table 1. Results on TSPlib instances

	original HK		1-round		fixpoint	
	time	BnB	time	BnB	time	BnB
burma14	0.1	28	0	0	0	0
ulysses16	0.16	32	0	0	0	0
gr17	0.14	34	0	0	0.01	0
gr21	0.16	42	0	0	0.01	0
ulysses22	0.19	0	0	0	0.01	0
gr24	0.23	44	0.01	0	0.03	0
fri26	0.36	48	0.01	2	0.01	2
bayg29	0.35	54	0.04	6	0.07	6
bays29	0.33	88	0.05	10	0.1	10
dantzig42	0.65	92	0.09	4	0.17	4
swiss42	0.79	112	0.09	8	0.09	8
att48	1.7	140	0.21	18	0.23	15
gr48	94	13554	5.18	2481	7.38	3661
hk48	1.37	94	0.17	4	0.16	4
eil51	15.9	2440	0.39	131	0.84	426
berlin52	0.63	80	0.02	0	0.02	0
brazil58	13	878	1.09	319	1.02	296
st70	236	13418	1.21	183	1.1	152
eil76	15	596	1.03	125	0.88	99
rat99	134	2510	5.44	592	4.88	502
kroD100	16500	206416	11	7236	50.83	4842
rd100	67	782	0.76	0	0.73	0
eil101	187	3692	8.17	1039	9.59	1236
lin105	31	204	1.81	4	1.85	4
pr107	41	442	4.65	45	4.49	48

4 Experimental Results

To evaluate the benefits of using CP within the Held-Karp branch-and-bound algorithm, we ran experiments on several instances of the TSPlib. We report both the number of branching nodes and CPU time required solve each instance, with different propagation levels: no propagation ('original HK'), calling each filtering algorithm once ('1-round'), and propagation until we reach a fixed point ('fixpoint'). To eliminate the impact of the upper bound can have on search tree, we ran these experiments using the optimal value of each instance as its UB.

Table 1 clearly shows the impact of CP filtering techniques on the original Held-Karp algorithm. In fact the reduction of the graph not only considerably reduces the search effort (BnB nodes) but also sufficiently accelerates the computation of 1-trees inside the Held-Karp relaxation to completely absorb the extra computations required by the filtering mechanisms. This can be seen as the proportional reduction in CPU times largely exceeds the reduction in search nodes.

Finally, we cannot conclude that the extra effort required to reach the fixed point is worthwhile, as it is sometimes better and sometimes worse than a single round of filtering. Results on these preliminary tests seem to show that more than one round of computation is most often useless, as the first round of filtering was sufficient to reach the fixed point in about 99.5% of the search nodes. More tests are thus required before investigating more sophisticated propagation mechanisms.

References

[1] Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W.J.: The Traveling Salesman Problem: A Computational Study. Princeton University Press, Princeton (2006)

[2] Balas, E., Toth, P.: Branch and Bound Methods. In: Lawler, E.L., Lenstra, J.K., Rinnooy Kan, A.H.G., Shmoys, D.B. (eds.) The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization, ch. 10. Wiley, Chichester (1985)

[3] Dooms, G., Katriel, I.: The "not-too-heavy spanning tree" constraint. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 59–70. Springer, Heidelberg (2007)

[4] Held, M., Karp, R.M.: The Traveling-Salesman Problem and Minimum Spanning Trees. Operations Research 18, 1138–1162 (1970)

[5] Held, M., Karp, R.M.: The Traveling-Salesman Problem and Minimum Spanning Trees: Part II. Mathematical Programming 1, 6–25 (1971)

[6] Helsgaun, K.: An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. European Journal of Operational Research 126(1), 106–130 (2000)

[7] Prim, R.C.: Shortest connection networks and some generalizations. Bell System Tech. J. 36, 1389–1401 (1957)

[8] Régin, J.-C.: Simpler and Incremental Consistency Checking and Arc Consistency Filtering Algorithms for the Weighted Spanning Tree Constraint. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 233–247. Springer, Heidelberg (2008)

Characterization and Automation of Matching-Based Neighborhoods

Thierry Benoist

Bouygues e-lab, 40 rue de Washington − 75008 Paris, France
tbenoist@bouygues.com

Abstract. This paper shows that that some matching based neighborhood can be automatically designed by searching for stable sets in a graph. This move generation algorithm is illustrated and investigated within the *LocalSolver* framework.

1 Introduction

Autonomous search is a challenging trend in optimization solvers. It consists in partially or totally automating the solving process. The ultimate goal is to reach a *model&run* paradigm where the user merely models the problem to be solved and relies on the solver to find solutions. Automation can apply to various components of an optimization solver, as detailed in [8]. In this note we focus on the automatic design of Very Large Scale Neighborhoods (VLSN) [1] for local search solvers. Neighborhood search is one of the most effective approaches to solve combinatorial optimization problems. It basically consists in moving from one solution to another by applying local changes, tending to improve the objective function. In this context, VLSN are sometimes useful to improve the convergence on difficult instances. In particular, neighborhoods of exponential size explored in polynomial time are often appreciated by researchers [4, 5, 9, 10]. For instance in the car sequencing problem, Estellon et al. noticed in [7] that selecting a set of K positions sufficiently distant one from another allowed optimally repositioning these K cars through the resolution of a transportation problem.

We will show that some of these VLSN encountered in different contexts share common bases and thus can be implemented in a unified way in an autonomous solver. More precisely we prove in section 2 that some matching-based neighborhood can be automatically designed by searching for stable sets in a graph; then we propose in section 3 a convenient way to implicitly pre-compute billions of stable sets. Throughout this paper we will use the Eternity II edge matching puzzle (*http://us.eternityii.com*) for illustrating the introduced ideas and finally for experimenting the algorithms, within the LocalSolver[1] framework. The goal of this

[1] The Author is grateful to the LocalSolver team: Bertrand Estellon, Frédéric Gardi and Karim Nouioua. *LocalSolver* is a free local search solver for combinatorial optimization problems, based on a simple 0-1 formalism. The interested reader is referred to e-lab.bouygues.com for a description of its functionalities and algorithms.

puzzle is to place 256 square tiles on a 16x16 board so as to maximize the number of matching edges (namely adjacent tiles have matching colors along their common edge).

2 Large Neighborhoods and Stable Sets

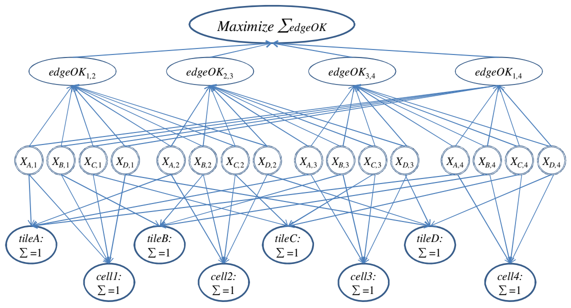
We consider an optimization problem where all decision variables are Booleans. On these binary variables, expressions are stated with various arithmetic and logic operators (*min,* $\leq$*, sum, and,* etc.). Some boolean expressions are tagged as constraints and one numeric expression is the objective function. The syntax graph associated to this classical functional formalism is a *Directed Acyclic Graph* (DAG) whose roots are the decision variables and whose leaves are constraints and objectives, while intermediate expressions are nodes. This DAG is also named the *invariant network* in iOpt [12]. Within this formalism, a transformation (or move) consists in flipping a certain number of binary decision variables. The impact of this transformation is automatically computed by propagating changes up in the DAG, eventually updating the satisfaction of constraints and the value of the objective function. Weighted and unweighted sums will be named *linear nodes*. An edge-matching puzzle can be modelled with "tile to cell" binary variables X_{ijtr} (equal to 1 if tile *t* is assigned to cell *i,j* with rotation *r*), subject to two families of linear equalities "one tile per cell" and "one cell per tile" representing the underlying assignment structure. Matching edges can be detected with simple Boolean expressions for each edge and each color, and the sum of these Booleans is the function to be maximized.

In such a DAG, let C be the set of all non-weighted sums of decision variables, on which an equality constraint is set, and V the set of variables involved in C. Now assume that C can be partitioned into C_1 and C_2 such that each variable of V appears exactly once in a sum of C_1 and once in a sum of C_2. The existence of such a bipartite structure can be detected with a 2-coloring algorithm on the graph (C,V). In our puzzle-matching example, the detected C_1 and C_2 are naturally the "one tile per cell" and "one cell per tile" families. For any valid solution *s* that is to say a vector of values for all decision variables, $\Omega(s)$ is the objective function associated to *s*. A function *u* transforming solution *s* is characterized by the set $\hat{u} \subset V$ of decision variables whose values are flipped by *u* and its impact is $\Delta_u(s) = \Omega(u(s)) - \Omega(s)$. The set of all such functions is *F*. When two functions *u* and *v* share no variable ($\hat{u} \cap \hat{v} = \emptyset$) then $u \oplus v$ is defined by $\widehat{u \oplus v} = \hat{u} \cup \hat{v}$.

Property. *For any sum c∈C, we denote by T(c) the set of nodes of the DAG reachable from a variable of c without crossing a node of C.* $\forall$ *c,d∈C₁, such that T(c)∩T(d) contains only linear nodes,* $\forall$ *s a valid solution,* $\forall$ *u,v∈F such that u (resp. v) only involves values of c (resp. d) and u(s) (resp. v(s)) satisfied all constraints of T(c) (resp. T(d)). Then, if* $u \oplus v$ *preserve the sums in C, the impact of both transformations is valid and additive:* $\Delta_{u\oplus v}(s) = \Delta_u(s) + \Delta_v(s)$. *We will say that c and d are non-adjacent.*

Proof. First $u \oplus v$ is well defined since *c* and *d* share no variable. Then, since $u \oplus v$ preserves the sums in *C*, constraints on these nodes remain satisfied. Other constraints are in $T(c) \cup T(d)$, whose intersection contains no constraints (only linear nodes), that is to say that constraints in $T(c)$ (resp. $T(d)$) are only impacted by variables of c (resp. d). Our hypothesis ensures their satisfaction in this case, hence $u \oplus v(s)$ is valid. Finally, since $T(c)$ and $T(d)$ share linear expressions only, we have $\Delta_{u \oplus v}(s) = \Delta_u(s) + \Delta_v(s)$.

In terms of the edge-matching puzzle, two "one tile per cell" constraints have the above property if and only if they refer to non-adjacent cells. Indeed in this case, no edge-matching detection expression involves variables of both constraints. For instance, in the DAG below (a simple 2x2 board), the sums "cell1"and "cell3" are non-ajacent. Swapping the tiles in these cells preserves the "assignment sums" (C) and thus can be evaluated as the sum of the two changes composing the swap. If all constraints in a subset S_1 of C_1 are pairwise non-adjacent, we say that S_1 is a stable set with respect to this definition of adjacency.



Besides, if all constraints in S_1 are equalities to 1, then this additive property allows defining the following large neighborhood for S_1.We define a bipartite graph based on the two sets C_1 and C_2 such that for each variable v of V involved in $c_1 \in C_1$ and $c_2 \in C_2$, we define an edge from c_1 to c_2 if $v = 1$ and an edge from c_2 to c_1 if $v = 0$. For each constraint c in C_1, we assign weights to incoming edges (those associated to variables equal to 0) as follows. For each variable v of c equal to 0, let g be the function setting to 0 the variable currently instantiated to 1 in c and setting to 1 variable v. Then the weight of the edge representing variable v is $\Delta_g(s) = \Omega(g(s)) - \Omega(s)$, with s the current solution. Outgoing edges receive weight 0. To any cycle in this bipartite graph we associate a move flipping the values of all variables corresponding to edges of the cycle. Such a move preserves the sums in C because each node involved in the cycle has exactly one incoming and one outgoing edge. Besides the cost of this move is the sum of the weights of all involved edges, because:

- for each constraint c in C_1 the variable instantiated to 1 is set to 0 and another one takes value 1, which is the definition that we took for costs on edges
- since S_1 is a stable set, all these costs can be added to get the global cost of the move, thanks to the property established above.

We conclude that if there is a negative cycle in the above graph, then the objective function can be improved by changing the values of variables of this cycle only (while all other decision variables of the problem keep their values). We use the Bellman-Ford-Moore algorithm to search for such a negative cycle in this graph. Worst case complexity is $O(nm)$ with n the number of nodes and m the number of edges. At the end of each of the n iterations, we perform a search for cycles in the parent graph in *O(m)* (see [6] for other cycle detection strategies). For a given set S_1, the complexity is bounded by $O(|S_1|^3)$. For edge-matching puzzles, this neighborhood is the same as the one explored by Schaus & Deville in [11], namely the optimal reassignment of a set of non adjacent cells.

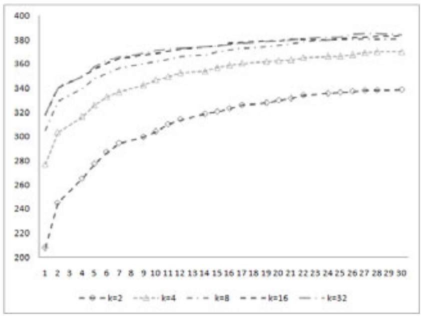
A variant of this algorithm consists in setting small negative costs to edges associated to variables equal to 1, so as to favor the detection of different assignment with the same cost if any (diversification move). In practice we first look for an improving transformation and then for a diversification transformation in the same graph.

3 Stable Sets Generators

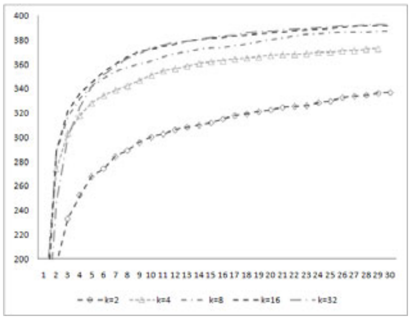
The above neighborhood is based on a stable set S_1 in C_1 but the worst complexity of computing a stable set of size smaller than K with a greedy algorithm (our goal is not to find a maximum stable set) is in the worst case $O(Kd)$ where d is the maximum degree of the graph. Indeed, each time a node is added to the stable set, its neighbor nodes must be removed from the graph. If absence on upper bound on the degree this worst case complexity is $O(K|C_1|)$. Besides such a greedy algorithm may built very small stable set in some cases, for instance if the first selected node is connected to all others. For these reasons it is useful to precompute structures allowing extracting stable sets of size K in $O(K)$. The structure that we define for this purpose is a collection of K disjoint subsets of C_1 such that any pair of nodes appearing in two different sets of this collection are non adjacent. We call such a collection a stable set generator. Building such generators can be achieved with a simple algorithm starting with an empty collection and based on two procedures: *Grow* adds to the growing collection a new non-adjacent singleton and *Merge* adds an adjacent node to one of the sets of the collection. In order to maximize the number of different stable sets that can be generated from this collection, we need to maximize the products of its sizes. Hence our heuristic consists in applying the *merge* procedure on each new singleton, and in favoring the *grow* procedure otherwise. Once the target size K is reached, the *merge* procedure is applied until no node can be added. Applying this randomized algorithm around 10 times at the beginning of the search procedure, we implicitly generate a huge number of stable sets when such stable sets exist.

4 Experimental Results and Conclusions

These algorithms (two-coloring for bipartite detection, adjacency analysis for stable set generation, and LNS neighborhood) have been implemented in the LocalSolver framework and tested on various problems. For the Eternity II problem taken as example in the descriptions of previous sections, we obtain the following results. It shall be noted that analyzing the DAG and creating the stable sets takes less than one second on this problem with more than 250 000 binary variables.



Schaus&Deville Intel Xeon(TM) 2.80GHz (kindly provided by Pierre Schaus)



Automatic LNS, Intel Xeon(TM) 3 GHz

The above graphs represent local search descents for this edge-matching puzzle with a neighborhood ranging from 2 tiles to 32 tiles. The left chart is extracted from [11] while the right one was obtained with *LocalSolver* setting the maximum size of the stable sets to 2, 4, 8, etc. The similarity of these curves confirms that without prior knowledge on the structure of the problem, we achieve to explore the same neighborhood, with the same efficiency. We tested this generic implementation of matching-based neighborhoods on various problems [3]. For the largest instance of each problem, the table below reports the size of the detected bipartite structure, the number of stable sets implicitly generated, the total time of this analysis and the number of moves per second during local search. In these experiments the size of stable sets was limited to 8.

Problem	*Bipartite size*	*# of (implicit) stable sets*	*Moves per second*
Car sequencing with colors	1319x284	10^{24} in 3.2s	590
Car Sequencing CspLib	500x20	10^{17} in 0.1s	545
Eternity II	256x256	10^{13} in 0.5s	990
Social Golfer	30x10 (13 times)	1 in 1.6s	285

These results show that some very-large scale neighborhood can be automatically generated thanks to an analysis of the model. Similarly to small neighborhoods offered by default in *LocalSolver*, these moves preserve the feasibility of the solution and are similar to what an OR researcher would implement.

References

1. Ahuja, R.K., Ergun, O., Orlin, J.B., Punnen, A.P.: A survey of very large-scale neighborhood search techniques. Discrete Appl. Math. 123(1-3), 75 (2002)
2. Angel, E., Bampis, E., Pascual, F.: An exponential (matching based) neighborhood for the vehicle routing problem. Journal of Combinatorial Optimization 15, 179–190 (2008)
3. Benoist, T.: Autonomous Local Search with Very Large-Scale Neighborhoods (manuscript in preparation)
4. Bozejko, W., Wodecki, M.: A Fast Parallel Dynasearch Algorithm for Some Scheduling Problems. Parallel Computing in Electrical Engineering, 275–280 (2006)

5. Cambazard, H., Horan, J., O'Mahony, E., O'Sullivan, B.: Fast and Scalable Domino Portrait Generation. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 51–65. Springer, Heidelberg (2008)
6. Cherkassky, B.V., Goldberg, A.V.: Negative-Cycle Detection Algorithms. In: Díaz, J. (ed.) ESA 1996. LNCS, vol. 1136, pp. 349–363. Springer, Heidelberg (1996)
7. Estellon, B., Gardi, F., Nouioua, K.: Large neighborhood improvements for solving car sequencing problems. RAIRO Operations Research 40(4), 355–379 (2006)
8. Hamadi, Y., Monfroy, E., Saubion, F.: What is Autonomous Search? Microsoft research report. MSR-TR-2008-80 (2008)
9. Hurink, J.: An exponential neighborhood for a one-machine batching problem. OR Spectrum 21(4), 461–476 (1999)
10. Mouthuy, S., Deville, Y., Van Hentenryck, P.: Toward a Generic Comet Implementation of Very Large-Scale Neighborhoods. In: 22nd National Conference of the Belgian Operations Research Society, Brussels, January 16-18 (2008)
11. Shaus, P., Deville, Y.: Hybridization of CP and VLNS for Eternity II. In: JFPC 2008 Quatrième Journées Francophones de Programmation par Contraintes, Nantes (2008)
12. Voudouris, C., Dorne, R., Lesaint, D., Liret, A.: iOpt: A Software Toolkit for Heuristic Search Methods. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 716–719. Springer, Heidelberg (2001)

Rapid Learning for Binary Programs

Timo Berthold[1,⋆], Thibaut Feydy[2], and Peter J. Stuckey[2]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
berthold@zib.de

[2] National ICT Australia[⋆⋆] and the University of Melbourne, Victoria, Australia
{tfeydy,pjs}@csse.unimelb.edu.au

Abstract. Learning during search allows solvers for discrete optimization problems to remember parts of the search that they have already performed and avoid revisiting redundant parts. Learning approaches pioneered by the SAT and CP communities have been successfully incorporated into the SCIP constraint integer programming platform.

In this paper we show that performing a heuristic constraint programming search during root node processing of a binary program can rapidly learn useful nogoods, bound changes, primal solutions, and branching statistics that improve the remaining IP search.

1 Introduction

Constraint programming (CP) and integer programming (IP) are two complementary ways of tackling discrete optimization problems. Hybrid combinations of the two approaches have been used for more than a decade. Recently both technologies have incorporated new *nogood learning* capabilities that derive additional valid constraints from the analysis of infeasible subproblems extending methods developed by the SAT community.

The idea of *nogood learning*, deriving additional valid *conflict constraints* from the analysis of infeasible subproblems, has had a long history in the CP community (see e.g. [1], chapter 6) although until recently it has had limited applicability. More recently adding carefully engineered nogood learning to SAT solving [2] has lead to a massive increase in the size of problems SAT solvers can deal with. The most successful SAT learning approaches use so called *first unique implication point (1UIP)* learning which in some sense capture the nogood closest to the failure that can infer new information.

Constraint programming systems have adapted the SAT style of nogood learning [3,4], using 1UIP learning and efficient SAT representation for nogoods, leading to massive improvements for certain highly combinatorial problems.

Nogood learning has been largely ignored in the IP community until very recently (although see [5]). Achterberg [6] describes a fast heuristic to derive

[⋆] This research was partially funded by the DFG Research Center Matheon in Berlin

[⋆⋆] NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

small conflict constraints by constructing a dual ray with minimal nonzero elements. He shows that nogood learning for general mixed integer problems can result in an average speedup of 10%. Kılınc Karzan et. al. [7] suggest restarting the IP solver and using a branching rule that selects variables which appear in small conflict constraints for the second run. Achterberg and Berthold [8] propose a hybrid branching scheme for IP that incorporates conflict-based SAT and impact-based CP style search heuristics as dynamic tie-breakers.

2 Rapid Learning

The power of nogood learning arises because often search algorithms implicitly repeat the same search in a slightly different context in another part of the search tree. Nogoods are able to recognize such situations and avoid redundant work. As a consequence, the more search is performed by a solver and the earlier nogoods are detected the greater the chance for nogood learning to be beneficial.

Although the nogood learning methods of SAT, CP, and IP approaches are effectively the same, one should note that because of differences in the amount of work per node each solver undertakes there are different design tradeoffs in each implementation. An IP solver will typically spend much more time processing each node than either a SAT or CP solver. For that reason SAT and CP systems with nogoods use 1UIP learning and frequent restarts to tackle problems while this is not the case for IP. IP systems with nogoods typically only restart at the root, and use learning methods which potentially generate several nogoods for each infeasibility (see [6]).

The idea of Rapid Learning is based on the fact that a CP solver can typically perform a partial search on a few hundred or thousand nodes in a fraction of the time that an IP solver needs for processing the root node of the search tree. Rapid Learning applies a fast CP branch-and-bound search for a few hundred or thousand nodes, before we start the IP search, but after IP presolving and cutting plane separation.

Each piece of information collected in this rapid CP search can be used to guide the IP search or even deduce further reductions during root node processing. Since the CP solver is solving the same problem as the IP solver

– each generated conflict constraint is valid for the IP search,
– each global bound change can be applied at the IP root node,
– each feasible solution can be added to the IP solver's solution pool,
– the branching statistics can initialize a hybrid IP branching rule [8], and
– if the CP solver completely solves the problem, the IP solver can abort.

All five types of information may potentially help the IP solver. Rapid Learning performs a limited CP search at the root node, after most of the IP presolving is done to collect potential new information for the IP solver.

The basic idea of Rapid Learning is related to the concept of *Large Neighborhood Search* heuristics in IP. But rather than doing a partial search on a sub-problem using the same (IP search) algorithm, we perform a partial search on the same problem using a much faster algorithm. Rapid Learning also differs from typical IP heuristics in the sense that it can improve both primal and dual bounds at the same time.

3 Computational Results

Our computational study is based on the branch-cut-and-price framework SCIP (Solving Constraint Integer Programs). This system incorporates the idea of *Constraint Integer Programming* [9,10] and implements several state-of-the-art techniques for IP solving, combined with solving techniques from CP and SAT, including nogood learning. The Rapid Learning heuristic presented in this article was implemented as a separator plugin.

For our experiments, we used SCIP 1.2.0.5 with Cplex 12.10 as underlying LP solver, running on a Intel® Core$^{\text{TM}}$2 Extreme CPU X9650 with 6 MB cache and 8 GB RAM. We used default settings and a time limit of one hour for the main SCIP instance which performs the IP search.

For solving the CP problem, we used a secondary SCIP instance with "emphasis cpsolver" (which among other things turns off LP solving) and "presolving fast" settings (which turns off probing and pairwise comparison of constraints) and the parameter "conflict/maxvarsfac" set to 0.05 (which only creates nogoods using at most 5% of the variables of the problem). As node limit we used $\max(500, \min(\mathit{niter}, 5000))$, with *niter* being the number of simplex iterations used for solving the root LP in the main instance. We further aborted the CP search as soon as 1000 conflicts were created, or no useful information was gained after 20% of the node limit.

As test set we chose all 41 *Binary programs (BPs)* of the Miplib 3.0 [11], the Miplib2003 [12] and the IP collection of Hans Mittelmann [13] which have less then 10 000 variables and constraints after SCIP presolving. BPs are an important subclass of IPs and finite domain CPs. where all variables take values 0 or 1. Note, that for a BP, all conflict constraints are Boolean clauses, hence linear constraints.

Table 1 compares the performance of SCIP with and without Rapid Learning applied at the root node (columns "SCIP" and "SCIP-RL"). Columns "RL" provide detailed information on the performance of Rapid Learning. "Ngds" and "Bds" present the number of applied nogoods and global bound changes, respectively, whereas "S" indicates, whether a new incumbent solution was found. For instances which could not be solved within the time limit, we present the lower and upper bounds at termination.

Note first that Rapid Learning is indeed rapid, it rarely consumes more than a small fraction of the overall time (except for `mitre`). We observe that for many instances the application of Rapid Learning does not make a difference. However, there are some, especially the `acc` problems, for which the performance improves dramatically. There are also a few instances, such as `qap10`, for which Rapid Learning deteriorates the performance. The solution time decreases by 12% in geometric mean, the number of branch-and-bound nodes by 13%. For

Table 1. Impact of Rapid Learning on the performance of SCIP

	SCIP		SCIP-RL		Rapid Learning				
Name	Nodes	Time	Nodes	Time	Nodes	Time	Ngds	Bds	S
10teams	197	7.2	197	7.3	716	0.1	0	0	
acc-0	1	0.9	1	0.9	0	0.0	0	0	
acc-1	112	32.6	113	34.4	3600	0.4	1332	0	
acc-2	54	58.8	1	4.4	2045	0.4	427	0	✓
acc-3	462	392.5	64	76.0	2238	0.7	765	0	
acc-4	399	420.2	364	115.4	2284	0.7	722	0	
acc-5	1477	354.1	353	126.6	2054	0.5	756	0	
acc-6	251	71.0	899	138.2	2206	0.5	591	0	
air04	159	45.4	159	45.6	1000	0.2	0	0	
air05	191	22.6	191	22.8	369	0.1	0	0	
cap6000	2755	2.6	2755	2.7	100	0.0	0	17	
disctom	1	2.2	1	2.2	0	0.0	0	0	
eilD76	3	17.2	3	17.2	100	0.0	0	0	
enigma	733	0.5	1422	0.5	500	0.0	9	0	
fiber	51	1.1	53	1.1	100	0.0	0	0	
harp2	352292	209.2	306066	191.3	1135	0.4	7	0	✓
l152lav	56	2.1	56	2.2	423	0.1	0	0	
lseu	366	0.5	450	0.5	500	0.0	146	0	✓
markshare4_0	1823558	111.7	2140552	234.4	500	0.0	305	0	✓
misc03	176	0.8	284	0.8	500	0.0	138	0	
misc07	31972	21.4	34416	22.4	100	0.0	0	0	
mitre	6	7.5	6	10.0	4177	2.5	284	1610	
mod008	366	0.8	366	0.8	100	0.0	0	0	✓
mod010	5	0.8	5	1.0	854	0.2	357	52	
neos1	1	3.1	1	3.2	727	0.1	325	0	✓
neos21	2020	18.7	1538	17.5	141	0.0	0	0	✓
nug08	1	56.2	1	10.2	1011	0.2	460	1392	
p0033	3	0.5	3	0.5	500	0.0	287	4	✓
p0201	76	0.7	76	0.7	100	0.0	0	0	
p0282	24	0.5	24	0.5	100	0.0	0	0	✓
p0548	53	0.5	38	0.5	100	0.0	0	10	
p2756	213	1.7	111	1.6	100	0.0	0	80	
prod1	23015	17.1	25725	20.0	500	0.1	0	0	✓
prod2	68682	80.3	68635	79.2	500	0.1	17	0	✓
qap10	5	146.8	12	542.0	2107	0.5	1666	0	
stein27	4041	0.8	4035	1.1	500	0.0	328	0	✓
stein45	50597	18.0	51247	18.1	500	0.0	0	0	✓
markshare1	[0.0,7.0]		[0.0,5.0]		500	0.0	199	0	✓
markshare2	[0.0,14.0]		[0.0,11.0]		500	0.0	174	0	✓
protfold	[-36.9135,-21.0]		[-37.0898,-22.0]		3078	1.6	510	0	
seymour	[414.318,425.0]		[414.313,426.0]		653	0.0	0	0	✓
geom. mean	212	8.3	185	7.3					
arithm. mean	63 902	57.5	71 357	47.4					

the four unsolved instances, we see that Rapid Learning leads to a better primal bound in three cases. The dual bound is worse for the instance `protfold`. For the instances `acc-2` and `nug08`, Rapid Learning completely solved the problem.

Additional experiments indicate that the biggest impact of Rapid Learning comes from nogoods and learning new bounds, but all the other sources of information are also beneficial to the IP search on average.

4 Conclusion and Outlook

Rapid Learning takes advantage of fast CP search to perform a rapid heuristic learning of nogoods, global bound changes, branching statistics and primal solutions before the IP search begins. Our computational results demonstrate that this information can improve the performance of a state-of-the-art non-commercial IP solver on BPs substantially.

We plan to investigate Rapid Learning for general IP problems, where we need to use bound disjunction constraints [6] to represent nogoods. We also plan to investigate the application of rapid learning at other nodes than the root, and combinations of CP and IP search that continually communicate nogoods, using a hybrid of SCIP and a native CP system.

References

1. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
2. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of DAC 2001, pp. 530–535 (2001)
3. Katsirelos, G., Bacchus, F.: Generalised nogoods in CSPs. In: Proceedings of AAAI 2005, pp. 390–396 (2005)
4. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
5. Davey, B., Boland, N., Stuckey, P.: Efficient intelligent backtracking using linear programming. INFORMS Journal of Computing 14(4), 373–386 (2002)
6. Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optimization 4(1), 4–20 (2007); Special issue: Mixed Integer Programming
7. Kılınç Karzan, F., Nemhauser, G.L., Savelsbergh, M.W.P.: Information-based branching schemes for binary linear mixed-integer programs. Math. Progr. C 1(4), 249–293 (2009)
8. Achterberg, T., Berthold, T.: Hybrid branching. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 309–311. Springer, Heidelberg (2009)
9. Achterberg, T.: Constraint Integer Programming. PhD thesis, TU Berlin (2007)
10. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: A new approach to integrate CP and MIP. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 6–20. Springer, Heidelberg (2008)
11. Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.: An updated mixed integer programming library: MIPLIB 3.0. Optima (58), 12–15 (1998)
12. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Operations Research Letters 34(4), 1–12 (2006)
13. Mittelmann, H.: Decision tree for optimization software: Benchmarks for optimization software (2010), `http://plato.asu.edu/bench.html`

Hybrid Methods for the Multileaf Collimator Sequencing Problem[⋆]

Hadrien Cambazard, Eoin O'Mahony, and Barry O'Sullivan

Cork Constraint Computation Centre,
Department of Computer Science, University College Cork, Ireland
{h.cambazard,e.omahony,b.osullivan}@4c.ucc.ie

Abstract. The multileaf collimator sequencing problem is an important component of the effective delivery of intensity modulated radiotherapy used in the treatment of cancer. The problem can be formulated as finding a decomposition of an integer matrix into a weighted sequence of binary matrices whose rows satisfy a consecutive ones property. In this paper we extend the state-of-the-art optimisation methods for this problem, which are based on constraint programming and decomposition. Specifically, we propose two alternative hybrid methods: one based on Lagrangian relaxation and the other on column generation. Empirical evaluation on both random and clinical problem instances shows that these approaches can out-perform the state-of-the-art by an order of magnitude in terms of time. Larger problem instances than those within the capability of other approaches can also be solved with the methods proposed.

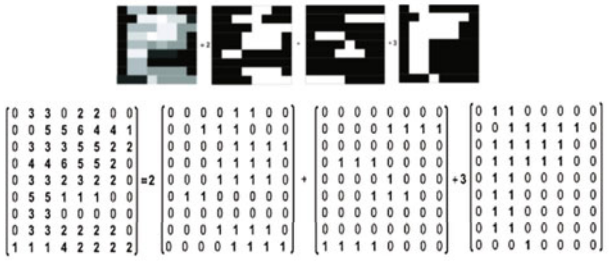
1 Introduction

Radiation therapy represents one of the main treatments against cancer, with an estimated 60% of cancer patients requiring radiation therapy as a component of their treatment. The aim of radiation therapy is to deliver a precisely measured dose of radiation to a well-defined tumour volume whilst sparing the surrounding normal tissue, achieving an optimum therapeutic ratio. At the core of advanced radiotherapy treatments are hard combinatorial optimisation problems. In this paper we focus on the multileaf collimator sequencing in intensity-modulated radiotherapy (IMRT).

What is Intensity-Modulated Radiotherapy? IMRT is an advanced mode of high-precision radiotherapy that utilises computer controlled x-ray accelerators to deliver precise radiation doses to a malignant tumour. The treatment plan is carefully developed based on 3D computed tomography images of the patient, in conjunction with computerised dose calculations to determine the dose intensity pattern that will best conform to the tumour shape. There are three optimisation problems relevant to this treatment. Firstly, the *geometry problem* considers the best positions for the beam head from which to irradiate. Secondly, the *intensity problem* is concerned with computing the exact levels of radiation to use in each area of the tumour. Thirdly, the *realisation problem*, tackled in this paper, deals with the delivery of the intensities computed in

[⋆] This work was supported by Science Foundation Ireland under Grant Number 05/IN/I886.

(a) A multileaf collimator.



$$\begin{bmatrix} 0&3&3&0&2&2&0&0\\ 0&0&5&5&6&4&4&1\\ 0&3&3&3&5&5&2&2\\ 0&4&4&6&5&5&2&0\\ 0&3&3&2&3&2&2&0\\ 0&5&5&1&1&1&0&0\\ 0&3&3&0&0&0&0&0\\ 0&3&3&2&2&2&2&0\\ 1&1&1&4&2&2&2&2 \end{bmatrix} = 2 \begin{bmatrix} 0&0&0&0&1&1&0&0\\ 0&0&1&1&1&0&0&0\\ 0&0&0&0&1&1&1&1\\ 0&0&0&1&1&1&1&0\\ 0&0&0&1&1&1&1&0\\ 0&1&1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&1&1&1&1&0\\ 0&0&0&0&1&1&1&1 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&0&0&0&0\\ 0&0&0&0&1&1&1&1\\ 0&0&0&0&0&0&0&0\\ 0&1&1&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&1&1&1&0&0\\ 0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0\\ 1&1&1&1&0&0&0&0 \end{bmatrix} + 3 \begin{bmatrix} 0&1&1&0&0&0&0&0\\ 0&0&1&1&1&1&1&0\\ 0&1&1&1&1&1&0&0\\ 0&1&1&1&1&1&0&0\\ 0&1&1&0&0&0&0&0\\ 0&1&1&0&0&0&0&0\\ 0&1&1&0&0&0&0&0\\ 0&1&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0 \end{bmatrix}$$

(b) A multileaf collimator sequencing problem.

Fig. 1. A simplified view of the optimisation problem associated with sequencing multileaf collimators in IMRT, Figure 1(b) has been adapted from [3]

the intensity problem. Combinatorial optimisation methods in cancer treatment planning have been reported as early as the 1960s [5] and a recent interesting survey on the topic can be found in [14]. There is a large literature on the optimisation of IMRT, which has tended to focus on the realisation problem [18]. Most researchers consider the sequencing of multileaf collimators (Figure 1(a)). The typical formulation of this problem considers the dosage plan from a particular position as an integer matrix, in which each integer corresponds to the amount of radiation that must be delivered to a particular region of the tumour. The requisite dosage is built up by focusing the radiation beam using a multileaf collimator, which comprises a double set of metal leaves that close from the outside inwards. Therefore, the collimator constrains the possible set of shapes that can be treated at a given time. To achieve a desired dosage, a sequence of settings of the collimator must be used. One such sequence is presented in Figure 1(b). The desired dosage is presented on the left, and it is delivered through a sequence of three settings of the multileaf collimator, which are represented by three matrices. Each matrix is exposed for a specific amount of time, corresponding to the weight associated with the matrix, thus delivering the requisite dosage.

Contribution of this Paper. In our earlier work in this area we presented a novel approach to multileaf collimator sequencing using an approach based on shortest paths [10]. It was shown that such a model significantly out-performed the state-of-the-art and brought clinical-sized instances of the problem within the reach of constraint programming (CP). We now show that the shortest path idea can be exploited to give greater scalability by coupling the CP model with Lagrangian relaxation and column generation techniques. Our shortest-path approach to this problem uniquely provides a basis for benefitting from these techniques. The results presented define the current state-of-the-art for this challenging problem from the domain of cancer treatment planning.

The CP model presented in [10], is briefly introduced in Section 2. We show how to strengthen the CP model with a Lagrangian relaxation in Section 3. An alternative formulation in which the paths are represented explicitly, along with a column generation (CG) model, is presented in Section 4. Section 5 demonstrates that these approaches significantly out-perform the state-of-the-art for this problem.

2 Formulation of the Multileaf Collimator Sequencing Problem

Let I represent the dosage intensity matrix to be delivered. I is an $m \times n$ (rows $\times$ columns) matrix of non-negative integers. We assume that the maximum dosage that is delivered to any region of the tumour is M units of radiation. Therefore, we set $I_{ij} \leq M, 1 \leq i \leq m, 1 \leq j \leq n$. To ensure that each step in the treatment sequence corresponds to a valid setting of the multileaf collimator, we represent each step using a 0/1 matrix over which a row-wise *consecutive ones* property (C1) must hold. Informally, the property requires that if any ones appear in a row, they appear together in a single block. A C1 matrix is a binary matrix in which every row satisfies the consecutive ones property. Formally, X is an $m \times n$ C1 matrix if and only if for any line i, $1 \leq a < b < c \leq n$, $X_{ia} = 1 \wedge X_{ic} = 1 \rightarrow X_{ib} = 1$. A solution to the problem is a sequence of C1 matrices, Ω, in which each X_k is associated with a positive integer b_k such that: $I = \sum_{k \in \Omega}(b_k \cdot X_k)$. Let B and K be the sum of coefficients b_k and the number of matrices X_k used in the decomposition of I, respectively. Then $B = \sum_{k \in \Omega} b_k$ and $K = |\Omega|$. B is referred to as the total *beam-on time* of the plan and K is its *cardinality*; see Figure 1(b) for an example with $K = 3$ and $B = 6$. The overall objective is to minimise the time needed for the complete treatment and the parameters B and K both affect that. Typical problems are to minimise B or K independently (known as the decomposition time and decomposition cardinality problem, respectively) or a linear combination of both: $w_1 K + w_2 B$. We will tackle this general formulation where w_1 accounts for the time needed by the operator to change the settings of the machine and w_2 accounts for the time to deliver one unit of radiation.

The problem of minimising B alone has been widely studied, starting with Bortdeld et al. [7] and Ahuja et al. [2] until a method in linear time was found by Baatar et al. and Engel [4, 15]. Minimising K alone was shown to be strongly NP-Hard [4] even for a single row or column [11] and received a lot of attention [6, 20]. Many heuristics were designed as the problem proved to be very difficult [1, 4]. The problem of minimising K while constraining B (lexicographic objective function) to its optimal value B^* was tackled by Engel and Kalinowski [15, 20]. Exact algorithms were proposed based on dynamic programming, Kalinowski [19], mixed integer linear programming, Langer [21, 26] and Constraint Programming Baatar et al., Ernst et al. and Cambazard et al. [3, 9, 10, 16]. Exact algorithms dealing with a more general objective function as the one used in this paper are designed by Wake et al, Caner Taskin et al [25, 26].

2.1 The Single Row Problem as a Shortest Path

In this section we study a restriction of the minimum cardinality problem DC to a single row. This will help to design efficient inference mechanisms for the general multi-row case. We show a simple construction representing the row problem as a shortest path.

C1 DECOMPOSITION CARDINALITY PROBLEM (DC)
Instance: A row matrix of n integers, $I = \langle I_1, \ldots, I_n \rangle$, a positive integer K.
Question: Find a decomposition of I into at most K C1 row matrices.

In any solution of the DC problem, there must be a subset of the weights of the decomposition that sum to every element I_j of the row. In other words, the decomposition must contain an integer partition of every intensity. We will represent integer partitions with the following notation: $P(a)$ is the set of partitions of integer a, $p \in P(a)$ is a particular partition of a, and $|p|$ the number of integer summands in p. We denote by $occ(v, p)$ the number of occurrences of value v in p. For example, $P(5) = \{\langle 5\rangle, \langle 4, 1\rangle, \langle 3, 2\rangle, \langle 3, 1, 1\rangle, \langle 2, 2, 1\rangle, \langle 2, 1, 1, 1\rangle, \langle 1, 1, 1, 1, 1\rangle\}$, and if $p = \langle 3, 1, 1\rangle$ then $|p| = 3$ and $occ(1, p) = 2$. Observe that the DC problem can be formulated as a shortest path problem in a weighted directed acyclic graph, G, which we refer to as a *partition graph*. A partition graph G of a row matrix $I = \langle I_1, \dots, I_n\rangle$ is a layered graph with $n+2$ layers, the nodes of each layer j corresponding to the set of integer partitions of the row matrix element I_j. The size of this graph is therefore exponential in the maximum intensity. Source and sink nodes, located on layers 0 and $n + 1$ respectively, are associated with the empty partition $\emptyset$. Two adjacent layers form a complete bipartite graph and the cost added to an edge, $p_u \to p_v$, between two partitions, p_u and p_v of adjacent layers, represents the number of additional weights that need to be added to the decomposition to satisfy the C1 property when decomposing the two consecutive elements with the corresponding partitions. The cost of each edge $p_u \to p_v$ in the partition graph is: $c(p_u, p_v) = \sum_{b=1}^{M} c(b, p_u, p_v)$ where $c(b, p_u, p_v) = max(occ(b, p_v) - occ(b, p_u), 0)$. Figure 2 shows the partition graph $I = [3, 2, 3, 1]$.

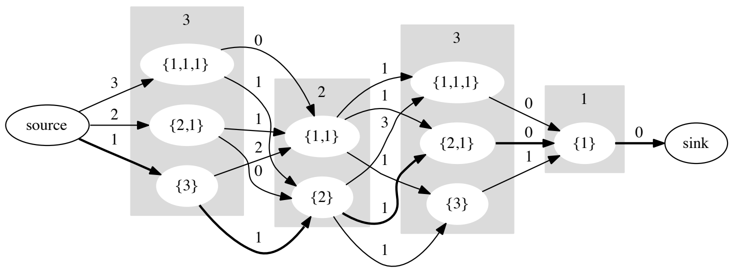


Fig. 2. A partition graph showing transition weights for the single row $I = [3, 2, 3, 1]$

By following the path $\{\{2, 1\}, \{1, 1\}, \{2, 1\}, \{1\}\}$, we build a decomposition:

$[3, 2, 3, 1] = 2[1, ?, ?, ?] + 1[1, ?, ?, ?]$ (choice of $\{2, 1\}$);
$[3, 2, 3, 1] = 2[1, 0, 0, 0] + 1[1, 1, ?, ?] + 1[0, 1, ?, ?]$ (choice of $\{1, 1\}$);
$[3, 2, 3, 1] = 2[1, 0, 0, 0] + 1[1, 1, 0, 0] + 1[0, 1, 1, ?] + 2[0, 0, 1, 0]$ (choice of $\{2, 1\}$);
$[3, 2, 3, 1] = 2[1, 0, 0, 0] + 1[1, 1, 0, 0] + 1[0, 1, 1, 1] + 2[0, 0, 1, 0]$ (choice of $\{1\}$).

The length of the path represents the cardinality of the decomposition and a shortest path therefore provides a decomposition with minimum cardinality. The key idea is that as one moves along a path in this graph, the partition chosen to decompose the element at layer j contains the only weights that can be reused to decompose the element at layer $j + 1$ because of the C1 property. Consider the previous example and the solution given. A coefficient 2 is used by the first partition but not by the second and thus becomes forbidden to decompose any other intensity values. The previous partition alone tells us the available coefficients to decompose the present intensity value. This is why the *cardinality* cost can be defined between consecutive partitions and the whole problem mapped to a shortest path. We could also restrict the cost to a given weight b to obtain the cardinality of this particular coefficient. We will use this idea in the CP model.

2.2 Shortest Path Constraint Programming Model

We present a CP model for the general multi-row case that takes advantage of the property identified for a single row. We index, in lexicographic order, the integer partitions of each element I_{ij} of the intensity matrix, and use an integer variable P_{ij} to denote the index of the partition used to decompose element I_{ij}. For example, if $I_{ij} = 5$ the domain of P_{ij} is $\{1, ..., 7\}$ corresponding to $\{\langle 5\rangle, \langle 4, 1\rangle, \langle 3, 2\rangle, \langle 3, 1, 1\rangle$ $,\langle 2, 2, 1\rangle, \langle 2, 1, 1, 1\rangle, \langle 1, 1, 1, 1, 1\rangle\}$. Thus, $P_{ij} = 4$ means that the coefficients $3, 1$ and 1 are used to sum to 5 in the decomposition. We also have a variable N_b giving the number of occurrences of weight b in the decomposition.

Our CP model uses the constraint SHORTESTPATH$(G, \{P_1, \dots, P_n\}, U)$ [10]. Once instantiated $\{P_1, \dots, P_n\}$ defines a path in the original partition graph. This constraint states that U must be greater than or equal to the length of this path using the cost information G. We refer to it as SHORTESTPATH because it does not enforce U to be equal to the length of the path but rather greater than or equal to it, and the support for the lower bound on U is a shortest path. A layer j of the graph corresponds to variable P_j and the nodes of each layer to the domain values of P_j. Our CP model posts the SHORTESTPATH constraint over three different cost definitions $G_1(i)$, $G_2(i, b)$, $G_3(i)$ (the partition graphs of a line i are topologically identical). Denoting p_u the partition corresponding to value u of P_{ij} and p_v the partition corresponding to value v of $P_{i,j+1}$, the transition costs are as follows: $c_1(p_u, p_v) = \sum_{b=1}^{M} c_2(b, p_u, p_v)$, $c_2(b, p_u, p_v) = max(occ(b, p_v) - occ(b, p_u), 0)$ and $c_3(p_u, p_v) = \sum_{b=1}^{M} b \times c_2(b, p_u, p_v)$. Therefore, our CP model is as follows:

$$
\begin{array}{lll}
minimise \; w_1 K + w_2 B \text{ with} & & K \in \{0, \dots, ub\}, B \in \{B^*, ..., ub\} \\
 & \forall b \le M & N_b \in \{0, \dots, ub\} \\
 & \forall i \le m, j \le n, & P_{ij} \in \{1, \dots, |P(I_{ij})|\} \\
 & & \\
CP_1: & & \sum_{b=1}^{M} b \times N_b = B \\
CP_2: & & \sum_{b=1}^{M} N_b = K \\
CP_3: & \forall i \le m, & \text{SHORTESTPATH}(G_1(i), \{P_{i1}, \dots, P_{in}\}, K) \\
CP_4: & \forall i \le m, b \le M & \text{SHORTESTPATH}(G_2(i, b), \{P_{i1}, \dots, P_{in}\}, N_b) \\
CP_5: & \forall i \le m, & \text{SHORTESTPATH}(G_3(i), \{P_{i1}, \dots, P_{in}\}, B) \\
CP_6: & \forall i \le m, \forall j < m \text{ s.t } I_{ij} = I_{i,j+1} & P_{ij} = P_{i,j+1}
\end{array}
$$

The C1 property of the decomposition is enforced by constraints CP_4. The number of weights of each kind, b, needed so that a C1 decomposition exists for each line i is maintained as a shortest path in $G_2(i, b)$. CP_3 acts as a redundant constraint and provides a lower bound on the cardinality needed for the decomposition of each line i. CP_5 is another useful redundant shortest path constraint that maintains the minimum value of B associated with each line, which can provide valuable pruning by strengthening CP_1. Finally CP_6 breaks some symmetries. We refer the reader to [10] for more details in particular related to the SHORTESTPATH constraint.

3 A Hybrid Model Based on Lagrangian Relaxation

Once the partition variables of a given line i are instantiated, they define a path in the original partition graph of the line. Constraints CP_3, CP_4, CP_5 constrain the length of this path, each with a different transition cost structure for the edges. The $M + 2$ path problems stated by constraints CP_3, CP_4, CP_5 on a given line define together a resource-constrained path problem. In this section we design a propagator to consider these paths simultaneously in order to achieve a higher degree of consistency for a single line. The underlying optimisation problem is the Resource Constrained Shortest Path Problem (RCSPP). The problem is to find a shortest path between a given source and sink so that the quantity of resources accumulated on each arc for each resource do not exceed some limits. Two approaches are often used to solve this problem: dynamic programming and Lagrangian relaxation. We base our propagator on the RCSPP and the multicost-regular constraint [23, 24].

We present one possible mapping of the problem stated by constraints CP_3, CP_4, CP_5 for line i to a RCSPP. We state it as a binary linear formulation where x^j_{uv} is a 0/1 variable denoting whether the edge between partition p_u and p_v of P_{ij} and $P_{i,j+1}$ is used. Layer 0 denotes the layer of the source and $n+1$ the one of the sink ($P_{i0} = P_{i,n+1} = \emptyset$). The problem formulation is as follows:

$$\begin{array}{lll}
z = min & \sum_{j\le n}\sum_{u,v\in P_{ij}\times P_{i,j+1}} c_3(p_u,p_v)\times x^j_{uv} & \\
\forall\, 1\le b\le M & \sum_{j\le n}\sum_{u,v\in P_{ij}\times P_{i,j+1}} c_2(b,p_u,p_v)\times x^j_{uv} & \le \overline{N_b} \\
 & \sum_{j\le n}\sum_{u,v\in P_{ij}\times P_{i,j+1}} c_1(p_u,p_v)\times x^j_{uv} & \le \overline{K} \\
\forall\, 1\le j\le n, u\in P_{ij} & \sum_{v\in P_{i,j-1}} x^{j-1}_{vu} - \sum_{v\in P_{i,j+1}} x^j_{uv} & = 0 \\
 & \sum_{v\in P_{i,1}} x^0_{1v} & = 1 \\
 & \sum_{u\in P_{i,n}} x^n_{u1} & = 1 \\
 & x^j_{uv}\in\{0,1\} &
\end{array} \tag{1}$$

If the optimal value of the RCSPP, z^*, is less than or equal to $\overline{B}$, then there is a solution to constraints CP_3, CP_4, and CP_5, otherwise there is an inconsistency. The first two constraints in the formulation are resource constraints and the last three are the flow conservation constraints enforcing that the x variables define a path.

Lagrangian relaxation is a technique that moves the "complicating constraints" into the objective function with a multiplier, $\lambda \ge 0$, to penalise their violation. For a given value of λ, the resulting problem is the Lagrangian subproblem and, in the context of minimisation, provides a lower bound on the objective of the original problem. The typical approach is to relax the resource constraints, so the Lagrangian function is:

$$\begin{aligned} f(x,\lambda) = \textstyle\sum_j\sum_{u,v} c_3(p_u,p_v)\times x^j_{uv} \;+\; \lambda_0(\sum_j\sum_{u,v} c_1(p_u,p_v)\times x^j_{uv} - \overline{K}) \\ +\; \textstyle\sum_{1\le b\le M}\lambda_b(\sum_j\sum_{u,v} c_2(b,p_u,p_v)\times x^j_{uv} - \overline{N_b}) \end{aligned} \tag{2}$$

The Lagrangian subproblem in this setting is, therefore, a shortest path problem $w(\lambda) = min_x f(x,\lambda)$ and the Lagrangian dual is to find the set of multipliers that provide the best possible lower bound by maximising $w(\lambda)$ over λ. A central result in Lagrangian relaxation is that $w(\lambda)$ is a piecewise linear concave function, and various algorithms can be used to optimise it efficiently.

Solving the Lagrangian Dual. We followed the approach from [23] and used a subgradient method [8]. The algorithm iteratively solves $w(\lambda)$ for different values of λ, initialised to 0 at the first iteration. The values of λ are updated by following the direction of a supergradient of w at the current value λ for a given step length μ. The step lengths have to be chosen to guarantee convergence (see [8]). We refer the reader to [23] for more details. At each iteration t, we solved the shortest path problem with the penalised costs on the edges:

$$c(p_u, p_v) = c_3(p_u, p_v) + \lambda_0^t c_1(p_u, p_v) + \sum_{1 \le b \le M} \lambda_b^t c_2(b, p_u, p_v).$$

This is performed by a traversal of the partition graph; as a byproduct we obtain the values of all shortest paths SO_a from the source to any node a. We can update the lower bound on B using:

$$SO_{sink} - \lambda_0^t \overline{K} - \sum_{1 \le b \le M} \lambda_b^t \overline{N_b}.$$

Then we perform a reversed traversal from the sink to the source to get the values of the shortest path SD_a from all nodes a to the sink. At the end of the iteration we mark all the nodes (partitions) that are infeasible in the current Lagrangian subproblem, i.e.:

$$SO_a + SD_a > \overline{B} + \lambda_0^t \overline{K} + \sum_{1 \le b \le M} \lambda_b^t \overline{N_b}.$$

At the end of the process, all nodes marked during the iterations are pruned from the domains. This is Lagrangian relaxation-based filtering [24]: if a value is proven inconsistent in at least one Lagrangian subproblem, then it is inconsistent in the original problem. The Lagrangian relaxation is incorporated into the constraint model as a global constraint for each line. The independent path constraints are kept and propagated first, whereas the propagation of the resource constrained path constraint is delayed since it is an expensive constraint to propagate.

4 A Column Generation Approach

Numerous linear models have been designed for this problem, see e.g. [14], but the shortest path approach [10] opens the door for a totally new formulation of the problem to be considered. In [10] we designed a linear model representing every integer partition. We now consider an alternative formulation that, rather than representing the partition graph, explicitly encodes the set of possible paths in the partition graph of each line. The resulting formulation is very large, but such models are typical in many settings, e.g. vehicle routing problems. The optimisation of these models can be performed using *column generation* [12]. The key idea is that the Simplex algorithm does not need to have access to all variables (columns) to find a pivot point towards an improving solution. The Simplex algorithm proceeds by iterating from one basic solution to another while improving the value of the objective function. At each iteration, the algorithm looks for a non-basic variable to enter the basis. This is the *pricing problem.* Typically, for a linear minimisation problem written

$$min \sum_i c_i x_i \mid \forall j \sum_i a_{ij} x_i \geq b_j, x_i \geq 0,$$

the pricing problem is to find the i (a variable or column) that minimises $c_i - \sum_j \pi_j a_{ij}$ where π_j is the dual variable associated with constraint j. The explicit enumeration of all i is impossible when the number of variables is exponential. Therefore, the column generation works with a restricted set of variables, which define the *restricted master problem* (RMP) and evaluates reduced costs by implicit enumeration e.g., by solving a combinatorial problem. We now apply these concepts to our shortest path model.

4.1 Column Generation for the Shortest Path Model

We denote by pt_i^k the k^{th} path in the partition graph of line i. A path is a sequence of partitions $\langle p_0, \ldots, p_{n+1} \rangle$ characterised by three costs: the cardinality cost $c_{i1}^k = \sum_{j=0}^{j=n} c_1(p_j, p_{j+1})$, the beam-on time cost $c_{i3}^k = \sum_{j=0}^{j=n} c_3(p_j, p_{j+1})$ and the beam-on time cost restricted to a given coefficient b, $c_{ib2}^k = \sum_{j=0}^{j=n} c_2(b, p_j, p_{j+1})$. The restricted master problem where a subset Ω of the columns are present is denoted RMP(Ω), and can be formulated as follows:

$$\begin{array}{lll}
RMP(\Omega): & minimise & w_1 K + w_2 B \\
C_0 & & \sum_{b \leq M} N_b = K \\
C_1 & & \sum_{b \leq M} b \times N_b = B \\
C_2 & \forall i, & \sum_{k \in \Omega_i} pt_i^k = 1 \\
C_3 & \forall i, & \sum_{k \in \Omega_i} c_{i1}^k \times pt_i^k \leq K \\
C_4 & \forall i, \forall b & \sum_{k \in \Omega_i} c_{ib2}^k \times pt_i^k \leq N_b \\
C_5 & \forall i, & \sum_{k \in \Omega_i} c_{i3}^k \times pt_i^k \leq B \\
 & & K \geq 0, B \geq 0, \ \forall b \, N_b \geq 0 \\
 & \forall i, k \in \Omega_i & pt_i^k \in \{0, 1\}
\end{array}$$

This master problem optimises over a set of paths Ω_i per line i. The task of generating improving columns or paths is delegated to the sub-problem which is partitioned into m problems. The reduced cost of a path in a given line does not affect the computation of the reduced cost on another line. This is a typical structure for Danzig-Wolfe decomposition, and the constraints of the RMP involving the N_b variables are the coupling, or complicating, constraints. An improving column for line i is a path of negative reduced cost where the reduced cost is defined by $c_i - \sum_j \pi_j a_{ij}$. This translates as follows in our context. The M different costs on each edge are modified by a multiplier corresponding to the dual variables of constraints C_3 – C_5. We denote by δ_i, π_{i1}, π_{ib2}, and π_{i3} the dual variables associated with constraints C_2 to C_5, respectively. The subproblem of line i, $PP(i)$, is a shortest path problem where the cost of an edge $c(p_u, p_v)$ is: $c(p_u, p_v) = -\pi_{i1} \times c_1(p_u, p_v) - \sum_b \pi_{ib2} \times c_2(b, p_u, p_v) - \pi_{i3} \times c_3(p_u, p_v)$.

The column generation procedure is summarised in Algorithm 1. Notice that the bound provided by column generation is no better than the one given by the compact linear model because the pricing problem has the *integrality property*. The utility of this

Algorithm 1. ColumnGeneration

```
   Data: Intensity Matrix – A matrix of positive integers
   Result: A lower bound on the optimal decomposition.
1  $\Omega = \emptyset$, $DB = -\infty$, $UB = +\infty$, $\epsilon = 10^{-6}$;
2  for $i \leq m$ do
3      add the path made of $\{1,\ldots,1\}$ partitions for each integer of line $i$ to $\Omega$;
4      set $\pi_{i1} = \pi_{i3} = \pi_{ib2} = -1$ for all $b$, solve $PP(i)$ and add the shortest path to $\Omega$
5  repeat
6      add the paths in $\Omega$ to the restricted master problem, RMP;
7      solve RPM, set $UB$ to the corresponding optimal value and record the dual values
       $(\delta_i, \pi_{i1}, \pi_{i3}, \pi_{ib2})$;
8      $\Omega = \emptyset$;
9      for $i \leq m$ do
10         solve the pricing problem $PP(i)$ and record its optimal value $\gamma_i$;
11         if $(\gamma_i - \delta_i) < -\epsilon$ then
12             add the optimal path to $\Omega$
13     $DB = max(DB, \Sigma_{i \leq m} \gamma_i))$;
   until $\lceil DB - \epsilon \rceil = \lceil UB \rceil$ or $\Omega = \emptyset$ ;
14 return $\lceil UB - \epsilon \rceil$
```

formulation is to give better scaling in terms of memory as we can achieve a tradeoff in the subproblem. We briefly explain the main phases of the algorithm.

Main Process. The algorithm must start with an initial set of columns that contain a feasible solution to obtain valid dual values. Lines 1 to 4 define the initialisation step where two paths, the *unit* path and the shortest path, are computed per line. Lines 6 to 14 specify the main column generation process. First, the new columns are added to the RMP, which is a continuous linear problem, and solved to optimality. UB denotes the upper bound provided by the optimal value of the RMP at each iteration. The pricing problem is then solved for each line using the dual values that are recorded (Line 7). Line 11 checks if a path of negative reduced cost has been found; $\gamma_i - \delta_i$ is the reduced cost of the path solution of $PP(i)$. Then, a lower bound on the original problem, the dual bound DB, is computed. The algorithm stops as soon as no path of negative reduced cost can be found ($\Omega = \emptyset$), or the lower and upper bounds have met ($\lceil DB - \epsilon \rceil = \lceil UB \rceil$).

Dual Lower Bound. The dual solution of the RMP, completed by the best reduced cost, forms a feasible solution of the dual of the original problem and, therefore, provides a lower bound. This dual bound DB is computed on Line 13 and we have:

$$DB = \sum_{i \leq m} \delta_i + \sum_{i \leq m} (\gamma_i - \delta_i) = \sum_{i \leq m} \gamma_i,$$

i.e., the sum of the dual objective function and the best reduced cost (see [12]). The dual bound provides a lower bound on the original problem and can be used for termination. Typically, we can stop as soon as the optimal value is known to be in the interval $]a, a+1]$, in which case one can immediately return $a+1$ as the integer lower bound on the original problem (Condition $\lceil DB - \epsilon \rceil = \lceil UB \rceil$). This last condition is useful to avoid a convergence problem and saves many calls to the subproblems. The use of an ϵ is to avoid rounding issues arising from the use of continuous values.

Solving the Pricing Problem. The pricing problem involves solving a shortest path in a graph whose size is exponential in the maximum element, M. Storing the partition graph explicitly requires $O(n \times P^2)$ space, where P is the (exponentially large) number of partitions of M. Memory remains an issue for the column generation if we solve the pricing problems by explicitly representing the complete graph. To save memory as M increases, the column generation procedure can avoid representing the whole graph by only storing the nodes, thus consuming only $O(nP)$ space. In this case the costs on each edge must be computed on demand as they cannot be stored. In practice, the previous compromise with $O(nP)$ space consumption is perfectly acceptable as instances become very hard before the space again becomes an issue. In our implementation we use a combined approach whereby we store the edges when the two consecutive layers are small enough and only recompute the cost on the fly if it is not recorded.

Speeding up the Column Generation Procedure. The column generation process is known to suffer from convergence problems [12]. In our case, an increase in the value of M implies more time-consuming pricing problems, and the bottleneck lies entirely in this task in practice. We obtained some improvement with a simple stabilisation technique [13, 22] to reduce degeneracy. We added surplus variables, y, to each constraint (except for the convexity constraints) so that constraints C_3 to C_5 read as:

$$\sum_{k\in\Omega_i} c_{i1}^k \times pt_i^k - y_{3i} \le K; \qquad \sum_{k\in\Omega_i} c_{ib2}^k \times pt_i^k - y_{4ib} \le N_b; \qquad \sum_{k\in\Omega_i} c_{i3}^k \times pt_i^k - y_{5i} \le B.$$

We also added slack variables, z, to constraints C_0 and C_1 which now read $\sum_{b\le M} N_b - y_0 + z_0 = K$ and $\sum_{b\le M} b \times N_b - y_1 + z_1 = B$. The slack and surplus variables are constrained in *a box* : $y \le \psi$, $z \le \psi$ and they are penalised in the objective function by a coefficient ρ. The objective function then reads as:

$$w_1 K + w_2 B + \sum_a \rho y_a + \rho z_0 + \rho z_1.$$

This tries to avoid the dual solutions jumping from one extreme to another by restraining the dual variables in a box as long as no relevant dual information is known. ρ and ψ are updated during the process and must end with $\rho = \infty$ or $\psi = 0$ to ensure the sound termination of the algorithm. We simply fix the value of ψ to a small constant (10% of the upper bound given by the heuristic [15]) and we update ρ when the column generation algorithm stalls, using a predefined sequence of values: $[0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1, \infty]$.

4.2 Branch and Price

We went a step further and designed a Branch and Price algorithm by coupling the CP algorithm with the Column Generation approach. The column generation procedure provides a valuable lower bound at the root node which can be often optimal in practice. To benefit from this bound during the search, we will now briefly describe a branch and price algorithm where column generation is called at each node of the CP search tree.

Branching on N_b raises an issue from the column generation perspective: the subproblem becomes a shortest path with resource constraints, one resource per $b \leq M$ limited by the current upper bound on the N_b variables of the CP model. This also means that finding a feasible set of columns to initialise the master problem becomes difficult.

Interaction with CP. Solving the shortest path problem with multi-resource constraints is far too costly. Recall that the original CP model is relaxing the multi-resource path into a set of independent paths. The propagation obtained from this relaxation removes partitions in the partition graph. We can therefore take advantage of this information to prune the graph used by the subproblem of the column generation and solve a shortest path in a restricted graph. We therefore solve a relaxation of the real subproblem that we obtained from the CP propagation. The current bounds on the domains of the N_b variables are also enforced in the master problem RMP. Propagation allows us to strengthen both the master and the subproblems of the column generation.

Initialisation. The initialisation issue can be easily solved by adding slack variables for constraints C_0, C_1, C_3, C_4, and C_5 of the RMP and adding them to the objective function with a sufficiently large coefficient to ensure they will be set to 0 in an optimal solution. Then one simply needs to independently find a path in the current filtered partition graph of each line to obtain a feasible solution.

Column Management. From one node of the search tree to another, we simply keep the columns that are still feasible based on the domains of the N_b and P_{ij} variables and remove all the others. In addition to these removals, if the number of columns increases beyond a threshold (set to 10000 columns in practice), we delete half of the pool starting with the oldest columns to prevent the linear solver from stalling due to the accumulation of too many variables.

Reduced cost propagation. The CG provides a lower bound on the objective function but also the set of optimal reduced costs for the N_b variables. Propagation based on these reduced costs can be performed in the CP model following [17]. At a given node, once the RMP has been solved optimally, we denote by ub and lb the currents bounds on the objective variable. $ub + 1$ corresponds to the value of the best solution found so far and lb is the optimal value of the RMP at the corresponding node. We denote by rc_b, the reduced cost of variable N_b at the optimal solution of the RMP. rc_b represents the increase in the objective function for an increase of one unit of N_b. The upper bound on each N_b in the CP model can be adjusted to $lb(N_b) + \lfloor \frac{ub-lb}{rc_b} \rfloor$.

5 Experimental Results

We evaluated our methods using both randomly generated and clinical problem instances.[1] We used the *randomly generated instances* of [3,9], which comprise 17 categories of 20 instances ranging in size from 12×12 to 40×40 with an M between 10 and 15, which we denote as m-n-M in our results tables. We added 9 additional categories with matrix sizes reaching 80×80 and a maximum intensity value, M, of 25, giving 520 instances in total. The suite of 25 *clinical instances* we used are those from [25].

[1] All the benchmarks are available from `http://www.4c.ucc.ie/datasets/imrt`

Table 1. Comparing quality and time of LP/CG/CG-STAB on the Lex objective function

Inst	Gap (%)	LP	CG				Stabilised CG			
		Time	Time	NbPath	NbIter	Gain	Time	NbPath	NbIter	Gain
mean	0.64	109.08	40.28	849.10	147.54	10.83	**20.42**	579.53	62.47	14.97
median	0.30	14.73	1.44	660.18	123.30	10.58	**1.13**	434.35	54.48	14.32
min	0.00	0.81	**0.12**	262.75	65.95	6.45	**0.12**	198.95	38.85	6.72
max	5.00	1196.51	762.31	1958.10	297.60	21.96	**368.90**	1404.80	104.60	34.46

Table 2. Comparing the effect of the Lagrangian filtering on the Shortest Path Model CPSP

Inst	CPSP					CP + Lagrangian relaxation				
		Time (seconds)			Nodes		Time (seconds)			Nodes
	NS	med	avg	max	avg	NS	med	avg	max	avg
12-12-20	20	**35.30**	**64.38**	**395.18**	816	20	405.22	796.33	4,532.17	**481**
12-12-25	**18**	1,460.39	2,242.19	6,705.01	8,966	0	-	-	-	-
15-15-15	20	**14.49**	**28.39**	**94.74**	938	20	80.76	120.57	399.37	**389**
18-18-15	20	**19.79**	**65.97**	**586.65**	1,366	20	80.36	180.69	807.63	**413**
20-20-15	20	**66.13**	**192.72**	**725.90**	4,436	20	353.09	559.73	2,328.94	**762**
20-20-20	**18**	1,379.72	1,876.12	6,186.78	7,628	6	1,190.96	1,605.77	5,041.88	572
30-30-15	**14**	115.83	698.37	2,638.54	691,318	12	308.04	839.37	3,942.70	937
40-40-10	20	**6.89**	495.90	3,848.14	130,309	20	19.21	**410.94**	**2,706.02**	**1,517**
40-40-15	**10**	512.88	1,555.49	5,687.44	488,133	8	1,003.10	1,645.86	5,029.01	1,189
50-50-10	15	82.04	888.52	5,275.96	4,022,156	**16**	85.36	784.76	5,216.68	10,534
60-60-10	11	1,100.92	1,967.51	6,079.23	8,020,209	**15**	426.73	1,378.31	5,084.95	34,552
70-70-10	7	2,374.97	2,503.82	3,980.76	11,102,664	**9**	2,534.44	2,894.94	5,970.91	131,494
80-80-10	2	464.57	464.57	737.78	14,274,026	**5**	1,877.76	2,193.92	4,147.88	118,408

The experiments ran as a single thread on a Dual Quad Core Xeon CPU, 2.66GHz with 12MB of L2 cache per processor and 16GB of RAM overall, running Linux 2.6.25 x64. A time limit of two hours and a memory limit of 3GB was used for each run.

Experiment 1: Evaluation of the LP Model. Firstly, we examine the quality and speed of the linear models (solved with CPLEX 10.0.0). We use a lexicographic objective function to perform this comparison, i.e. seek a minimum cardinality decomposition for the given minimum beam on-time. In the result tables LP refers to the continuous relaxation of the linear model representing every partition [10], CG to the model based on paths and CG-STAB to its stabilised version. Table 1 reports the average gap (in percentage terms) to the optimal value, the average times for the three algorithms as well as the number of iterations and paths for CG and CG-STAB. The improvement in time over LP is also given (column Gain). The mean, median, min and max across all categories are finally reported as well. The linear relaxation leads to excellent lower bounds but LP becomes quite slow as M grows and could not solve the instances with $M = 25$ due to memory errors. CG improves the resolution time significantly and offers better scalability in terms of memory. Its stabilised version clearly performs fewer iterations and proves to be approximately twice as fast on average.

Experiment 2: Evaluation of the Lagrangian Model. We consider the Lagrangian relaxation and its effect on the CP model.[2] We use a lexicographic objective function. Table 2 reports for the hardest categories the number of instances solved (column NS) within the time limit, along with the median, average and maximum time as well as the

[2] All CP models were implemented in Choco 2.1 – `http://choco.emn.fr`

Table 3. Comparing the CP and Branch and Price

Inst	CPSP			Branch and Price (light)					Branch and Price				
		Time (seconds)			Time (seconds)			Nodes		Time (seconds)			Nodes
	NS	med	avg	NS	med	avg	max	avg	NS	med	avg	max	avg
12-12-20	20	35.30	64.38	20	**33.32**	**41.01**	**88.46**	90	20	67.99	75.68	176.40	**83**
12-12-25	18	1,460.39	2,242.19	20	**1,353.34**	**1,684.32**	**4,826.52**	157	20	2,767.06	2,748.22	5,112.66	**141**
15-15-15	20	14.49	28.39	20	**7.86**	**8.95**	**24.96**	142	20	20.44	21.86	38.18	**127**
18-18-15	20	19.79	65.97	20	**10.34**	**10.23**	**16.12**	202	20	34.03	33.64	45.31	**167**
20-20-15	20	66.13	192.72	20	**14.76**	**15.61**	**27.25**	283	20	47.82	52.74	115.09	**218**
20-20-20	18	1,379.72	1,876.12	20	**235.42**	**230.21**	**387.14**	325	20	823.16	855.54	1,433.07	**221**
30-30-15	14	115.83	698.37	20	**38.13**	**42.85**	**108.88**	2,420	20	322.95	335.43	683.56	**492**
40-40-10	20	6.89	495.90	20	**5.10**	**6.04**	**18.44**	5,932	20	97.56	95.28	128.47	**753**
40-40-15	10	512.88	1,555.49	20	**85.49**	**97.53**	**224.71**	23,755	20	1,101.48	1,172.91	2,354.53	**818**
50-50-10	15	82.04	888.52	20	**14.80**	**27.12**	**178.79**	48,216	20	280.11	265.71	393.71	**1,194**
60-60-10	11	1,100.92	1,967.51	20	**67.06**	**252.60**	**3,337.60**	638,157	20	471.39	492.44	705.08	**1,724**
70-70-10	7	2,374.97	2,503.82	17	686.33	1,443.46	7,118.74	5,778,692	**20**	1,153.71	1,147.24	2,243.58	**2,408**
80-80-10	2	464.57	464.57	8	812.35	1,983.79	6,671.28	11,546,885	**20**	1,854.04	2,069.52	3,830.13	**3,059**

Table 4. Comparisortng the shortest path CP model, the Branch and Price algorithm against [25]

Inst				Caner et al.	CPSP			Branch and Price light		
	m	n	M	Time	Time	Nodes	Obj	Time	Nodes	Obj
c1b1	15	14	20	1.10	8.85	1,144	111	5.26	144	111
c1b2	11	15	20	0.80	0.38	222	104	1.36	77	104
c1b3	15	15	20	11.40	5.90	534	108	3.06	70	108
c1b4	15	15	20	37.00	7.87	389	110	7.10	77	110
c1b5	11	15	20	4.30	0.23	46	104	1.11	37	104
c2b1	18	20	20	26.50	29.68	3,304	132	11.08	665	132
c2b2	17	19	20	20.10	75.30	3,822	132	9.31	255	132
c2b3	18	18	20	14.70	1.86	116	140	6.69	101	140
c2b4	18	18	20	87.30	559.16	42,177	149	64.23	373	149
c2b5	17	18	20	395.60	16.74	911	132	23.39	402	132
c3b1	22	17	20	310.00	21.00	888	132	25.16	322	132
c3b2	15	19	20	4,759.80	48.30	1,527	144	25.82	178	144
c3b3	20	17	20	10,373.90	570.25	12,353	140	617.23	1,228	140
c3b4	19	17	20	524.90	2.18	136	127	13.18	96	127
c3b5	15	19	20	3.30	1.05	0	125	3.24	0	125
c4b1	19	22	20	34.90	0.47	367	152	1.43	356	152
c4b2	13	24	20	20,901.00	42.87	1,183	181	50.83	391	181
c4b3	18	23	20	44.70	17.35	4,059	139	4.99	131	139
c4b4	17	23	20	164.30	13.57	1,069	142	13.85	285	142
c4b5	12	24	20	14,511.40	2,003.76	75,284	192	533.36	1,455	192
c5b1	15	16	20	0.50	0.10	83	96	0.33	60	96
c5b2	13	17	20	14.30	13.33	4,420	125	18.82	248	125
c5b3	14	16	20	3.10	0.56	106	104	2.43	52	104
c5b4	14	16	20	2.20	168.18	19,747	124	37.95	636	124
c5b5	12	17	20	51.90	1.77	547	130	2.27	49	130
Mean				2,091.96	144.43	6,977.36	131.00	**59.34**	307.52	131.00
Median				34.90	13.33	911.00	132.00	**9.31**	178.00	132.00
Min				0.50	**0.10**	0.00	96.00	0.33	0.00	96.00
Max				20,901.00	2,003.76	75,284.00	192.00	**617.23**	1,455.00	192.00

average number of nodes. The Lagrangian relaxation reduces the search space by an order-of-magnitude but turns out to be very slow when M grows.

Experiment 3: Evaluation of the Branch and Price Model. We evaluate the Branch and Price algorithm against the previous CP models with the lexicographic objective function and also using the more general objective function to perform a direct comparison with [25] on clinical instances. Following [25] we set $w_1 = 7$ and $w_2 = 1$. The upper bound of the algorithm is initialised using the heuristic given in [15] whose running time is always well below a second. Table 3 compares the shortest path CP model (CPSP) with two versions of the Branch and Price using the lex objective function. The first version referred to as *Branch and Price (light)* only solves the CG during the first branching phase on the N_b variables whereas the other version solves the CG at each node of the search tree, including when the branching is made on the partition variables. The Branch and Price significantly improves the CP model and is able to optimally solve the integrality of the benchmark whereas the CPSP solves 455 out of the 520 instances. The light version is often much faster but does not scale to the last two larger sets of instances (70×70 and 80×80 matrices). Both branch and price algorithms outperform CPSP on hard instances by orders of magnitude in search space reduction.

Finally, we evaluate the CPSP and the light Branch and Price on 25 clinical instances with the general objective function. Table 4 reports the resolution time, the number of nodes explored (Nodes) and the value of the objective function (Obj). The times reported in [25] are quoted in the table and were obtained on a Pentium 4, 3 Ghz.[3]. The CP model alone already brings significant improvements over the algorithm of [25]. The Branch and Price algorithm shows even more robustness by decreasing the average, median and maximum resolution times.

6 Conclusion

We have provided new approaches to solving the Multileaf Collimator Sequencing Problem. Although the complexity of the resulting algorithms depends on the number of integer partitions of the maximum intensity, which is exponential, it can be used to design very efficient approaches in practice as shown on both random and clinical instances. The hybrid methods proposed in this paper offer performance significantly beyond the current state-of-the-art and rely on a rich exchange of information between OR and CP approaches.

References

1. Agazaryan, N., Solberg, T.D.: Segmental and dynamic intensity-modulated radiotherapy delivery techniques for micro-multileaf collimator. Medical Physics 30(7), 1758–1767 (2003)
2. Ahuja, R.K., Hamacher, H.W.: A network flow algorithm to minimize beamon time for unconstrained multileaf collimator problems in cancer radiation therapy. Netw. 45(1), 36–41 (2005)
3. Baatar, D., Boland, N., Brand, S., Stuckey, P.J.: Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 1–15. Springer, Heidelberg (2007)
4. Baatar, D., Hamacher, H.W., Ehrgott, M., Woeginger, G.J.: Decomposition of integer matrices and multileaf collimator sequencing. Discrete Applied Mathematics 152(1-3), 6–34 (2005)

[3] Two optimal values reported in [25] (for `c3b5` and `c4b5`) are incorrect. The corresponding solutions are pruned by their algorithms during search although it accepts them as valid solutions if enforced as hard constraints.

5. Bahr, G.K., Kereiakes, J.G., Horwitz, H., Finney, R., Galvin, J., Goode, K.: The method of linear programming applied to radiation therapy planning. Radiology 91, 686–693 (1968)
6. Boland, N., Hamacher, H.W., Lenzen, F.: Minimizing beam-on time in cancer radiation treatment using multileaf collimators. Networks 43(4), 226–240 (2004)
7. Bortfeld, T.R., Kahler, D.L., Waldron, T.J., Boyer, A.L.: X-ray field compensation with multileaf collimators. International Journal of Radiation Oncology Biology Physics 28(3), 723–730 (1994)
8. Boyd, S., Xiao, L., Mutapic, A.: Subgradient methods. In: Notes for EE392o, Standford University (2003)
9. Brand, S.: The sum-of-increments constraints in the consecutive-ones matrix decomposition problem. In: SAC 2009: 24th Annual ACM Symposium on Applied Computing (2009)
10. Cambazard, H., O'Mahony, E., O'Sullivan, B.: A shortest path-based approach to the multileaf collimator sequencing problem. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 41–55. Springer, Heidelberg (2009)
11. Collins, M.J., Kempe, D., Saia, J., Young, M.: Nonnegative integral subset representations of integer sets. Inf. Process. Lett. 101(3), 129–133 (2007)
12. Desaulniers, G., Desrosiers, J., Solomon, M.M.: Column Generation. Springer, Heidelberg (2005)
13. du Merle, O., Villeneuve, D., Desrosiers, J., Hansen, P.: Stabilized column generation. Discrete Math. 194(1-3), 229–237 (1999)
14. Ehrgott, M., Güler, Ç., Hamacher, H.W., Shao, L.: Mathematical optimization in intensity modulated radiation therapy. 4OR 6(3), 199–262 (2008)
15. Engel, K.: A new algorithm for optimal multileaf collimator field segmentation. Discrete Applied Mathematics 152(1-3), 35–51 (2005)
16. Ernst, A.T., Mak, V.H., Mason, L.A.: An exact method for the minimum cardinality problem in the planning of imrt. INFORMS Journal of Computing (2009) (to appear)
17. Focacci, F., Lodi, A., Milano, M.: Cost-based domain filtering. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 189–203. Springer, Heidelberg (1999)
18. Hamacher, H.W., Ehrgott, M.: Special section: Using discrete mathematics to model multileaf collimators in radiation therapy. Discrete Applied Mathematics 152(1-3), 4–5 (2005)
19. Kalinowski, T.: The complexity of minimizing the number of shape matrices subject to minimal beam-on time in multileaf collimator field decomposition with bounded fluence. Discrete Applied Mathematics (in press)
20. Kalinowski, T.: A duality based algorithm for multileaf collimator field segmentation with interleaf collision constraint. Discrete Applied Mathematics 152(1-3), 52–88 (2005)
21. Langer, M., Thai, V., Papiez, L.: Improved leaf sequencing reduces segments or monitor units needed to deliver imrt using multileaf collimators. Medical Physics 28(12), 2450–2458 (2001)
22. Lübbecke, M.E., Desrosiers, J.: Selected topics in column generation. Oper. Res. 53(6), 1007–1023 (2005)
23. Menana, J., Demassey, S.: Sequencing and counting with the multicost-regular constraint. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 178–192. Springer, Heidelberg (2009)
24. Sellmann, M.: Theoretical foundations of CP-based lagrangian relaxation. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 634–647. Springer, Heidelberg (2004)
25. Caner Taskin, Z., Cole Smith, J., Edwin Romeijn, H., Dempsey, J.F.: Collimator leaf sequencing in imrt treatment planning. Operations Research 119 (2009) (submitted)
26. Wake, G.M.G.H., Boland, N., Jennings, L.S.: Mixed integer programming approaches to exact minimization of total treatment time in cancer radiotherapy using multileaf collimators. Comput. Oper. Res. 36(3), 795–810 (2009)

Automatically Exploiting Subproblem Equivalence in Constraint Programming

Geoffrey Chu[1], Maria Garcia de la Banda[2], and Peter J. Stuckey[1]

[1] National ICT Australia, Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au

[2] Faculty of Information Technology,
Monash University, Australia
mbanda@infotech.monash.edu.au

Abstract. Many search problems contain large amounts of redundancy in the search. In this paper we examine how to automatically exploit remaining subproblem equivalence, which arises when two different search paths lead to identical remaining subproblems, that is the problem left on the remaining unfixed variables. Subproblem equivalence is exploited by caching descriptions, or keys, that define the subproblems visited, and failing the search when the key for the current subproblem already exists in the cache. In this paper we show how to automatically and efficiently define keys for arbitrary constraint problems. We show how a constraint programming solver with this capability can solve search problems where subproblem equivalence arises orders of magnitude faster. The system is fully automatic, i.e., the subproblem equivalences are detected and exploited without any effort from the problem modeller.

1 Introduction

When solving a search problem, it is common for the search to do redundant work, due to different search paths leading to subproblems that are somehow "equivalent". There are a number of different methods to avoid this redundancy, such as caching solutions (e.g. [19]), symmetry breaking (e.g. [8]), and nogood learning (e.g. [14]). This paper focuses on caching, which works by storing information in a cache regarding every new subproblem explored during the search. Whenever a new subproblem is about to be explored, the search checks whether there is an already explored subproblem in the cache whose information (such as solutions or a bound on the objective function) can be used for the current subproblem. If so, it does not explore the subproblem and, instead, uses the stored information. Otherwise, it continues exploring the subproblem. For caching to be efficient, the lookup operation must be efficient. A popular way is to store the information using a *key* in such a way that problems that can reuse each other's information are mapped to the same (or similar) key.

This paper explores how to use caching *automatically* to avoid redundancy in constraint programming (CP) search. Caching has been previously used in CP search, but either relies on the careful manual construction of the key for each model and search strategy (e.g [19]), or exploits redundancy when the remaining subproblem can be decomposed into independent components (e.g. [10,12]). Instead, we describe an approach that can automatically detect and exploit caching opportunities in arbitrary optimization problems, and does not rely on decomposition. The principal insight of our work is to define a key that can be efficiently computed during the search and can uniquely identify a relatively general notion of reusability (called U-dominance). The key calculation only requires each primitive constraint to be extended to *backproject* itself on the fixed variables involved. We experimentally demonstrate the effectiveness of our approach, which has been implemented in a competitive CP solver, Chuffed. We also provide interesting insight into the relationships between U-dominance and dynamic programming, symmetry breaking and nogood learning.

2 Background

Let $\equiv$ denote syntactic identity and $vars(O)$ denote the set of variables of object O. A *constraint problem* P is a tuple (C, D), where D is a set of *domain constraints* of the form $x \in s_x$ (we will use $x = d$ as shorthand for $x \in \{d\}$), indicating that variable x can only take values in the fixed set s_x, and C is a set of constraints such that $vars(C) \subseteq vars(D)$. We will assume that for every two $x \in s_x, y \in s_y$ in $D : x \not\equiv y$. We will define D_V, the restriction of D to variables V, as $\{(x \in s_x) \in D | x \in V\}$. Each set D and C is logically interpreted as the conjunction of its elements.

A *literal* of $P \equiv (C, D)$ is of the form $x \mapsto d$, where $\exists(x \in s_x) \in D$ s.t. $d \in s_x$. A *valuation θ of P over set of variables $V \subseteq vars(D)$* is a set of literals of P with exactly one literal per variable in V. It is a mapping of variables to values. The *projection* of valuation θ over a set of variables $U \subseteq vars(\theta)$ is the valuation $\theta_U = \{x \mapsto \theta(x) | x \in U\}$. We denote by $fixed(D)$ the set of fixed variables in D, $\{x | (x = d) \in D\}$, and by $fx(D)$ the associated valuation $\{x \mapsto d | (x = d) \in D\}$. Define $fixed(P) = fixed(D)$ and $fx(P) = fx(D)$ when $P \equiv (C, D)$.

A constraint $c \in C$ can be considered a set of valuations $solns(c)$ over the variables $vars(c)$. Valuation θ *satisfies* constraint c iff $vars(c) \subseteq vars(\theta)$ and $\theta_{vars(c)} \in c$. A *solution* of P is a valuation over $vars(P)$ that satisfies every constraint in C. We let $solns(P)$ be the set of all its solutions. Problem P is *satisfiable* if it has at least one solution and *unsatisfiable* otherwise.

Finally, we use $\exists_V.F$ to denote $\exists v_1.\exists v_2 \cdots \exists v_n.F$ where F is a formula and V is the set of variables $\{v_1, v_2, \ldots, v_n\}$. Similarly, we use $\overline{\exists}_V.F$ to denote the formula $\exists_{vars(F)-V}.F$. We let $\Leftrightarrow$ denote logical equivalence and $\Rightarrow$ logical entailment of formulae.

Given a constraint problem $P \equiv (C, D)$, constraint programming solves P by a search process that first uses a constraint solver to determine whether P can immediately be classified as satisfiable or unsatisfiable. We assume a propagation solver, denoted by solv, which when applied to P returns a new set D' of domain constraints such that $D' \Rightarrow D$ and $C \wedge D \Leftrightarrow C \wedge D'$. The solver detects unsatisfiability if any $x \in \emptyset$ appears in D'. We assume that if the solver returns a domain D' where all variables are fixed ($fixed(D) = vars(D)$), then the solver has detected satisfiability of the problem and $fx(D)$ is a solution. If the solver cannot immediately determine whether P is satisfiable or unsatisfiable, the search splits P into n subproblems (obtained by adding one of $c_1, \ldots, c_n$ constraints to P, where $C \wedge D \models (c_1 \vee c_2 \vee \ldots \vee c_n)$) and iteratively searches for solutions to them.

The idea is for the search to drive towards subproblems that can be immediately detected by solv as being satisfiable or unsatisfiable. This solving process implicitly defines a *search tree* rooted by the original problem P where each node represents a new (though perhaps logically equivalent) subproblem P', which will be used as the node's label. For the purposes of this paper we restrict ourselves to the case where each c_i added by the search takes the form $x \in s$. This allows us to obtain the i-th subproblem from $P \equiv (C, D)$ and $c_i \equiv x \in s$ as simply $P_i \equiv (C, join(x, s, D))$, where $join(x, s, D)$ modifies the domain of x to be a subset of s: $join(x, s, D) = (D - \{x \in s_x\}) \cup \{x \in s \cap s_x\}$. While this is not a strong restriction, it does rule out some kinds of constraint programming search.

3 Problem Dominance and Equivalence

Consider two constraint problems $P \equiv (C, D)$ and $P' \equiv (C', D')$ and a set of variables U. Intuitively, we say that P U-dominates P' if variables not in U are fixed, and when P and P' are projected over U, the latter entails the former.

Definition 1. *(C, D) U-dominates (C', D') iff*

- *$(vars(D) - U) \subseteq fixed(D)$ and $(vars(D') - U) \subseteq fixed(D')$, and*
- *$\bar{\exists}_U.(C' \wedge D') \Rightarrow \bar{\exists}_U.(C \wedge D)$.*

Example 1. Consider $P_0 \equiv (C, D)$ where $C \equiv \{x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20\}$, $D \equiv \{x_1 \in \{1..3\}, x_2 \in \{1..4\}, x_3 \in \{2..4\}, x_4 \in \{3..5\}, x_5 \in \{3..5\}\}$, and let $U \equiv \{x_3, x_4, x_5\}$. The subproblem $P \equiv (C, \{x_1 = 3, x_2 = 1\} \cup D_U)$ U-dominates $P' \equiv (C, \{x_1 = 1, x_2 = 3\} \cup D_U)$. □

If one problem P U-dominates another P' we can use the solutions of P to generate the solutions of P', as formalised by the following proposition.

Proposition 1. *If P U-dominates P' then $\theta \in solns(P)$ if $(\theta_U \cup fx(P')_{vars(P')-U}) \in solns(P')$.*

The situation is even simpler if the U-dominance relationship is symmetric.

Definition 2. *P and P' are U-equivalent iff P U-dominates P' and vice versa.*

Example 2. Consider problem (C, D) where $C \equiv \{alldiff([x_1, x_2, x_3, x_4, x_5])\}$ and $D \equiv \{x_1 \in \{1..3\}, x_2 \in \{1..4\}, x_3 \in \{2..4\}, x_4 \in \{3..5\}, x_5 \in \{3..5\}\}$, and let $U \equiv \{x_3, x_4, x_5\}$. The subproblems $P \equiv (C, \{x_1 = 1, x_2 = 2\} \cup D_U)$ and $P' \equiv (C, \{x_1 = 2, x_2 = 1\} \cup D_U)$ are U-equivalent. □

```
cache_search($C, D$)
    $D' :=$ solv($C, D$)
    if ($D' \Leftrightarrow false$) return $false$
    if ($\exists U.\exists P \in Cache$ where $P$ $U$-dominates $(C, D)$) return $false$
    if $fixed(D') \equiv vars(D)$
        [SAT] return $D$
    foreach $(x \in s) \in$ split($C, D$)
        $S :=$ cache_search($C$, join($x, s, D$))
        if ($S \not\equiv false$) return $S$
    $Cache := Cache \cup \{(C, D')\}$
    return $false$
```

Fig. 1. Computing the first solution under subproblem equivalence

Proposition 2. *If P and P' are U-equivalent then $\theta \in solns(P)$ iff $(\theta_U \cup fx(P')_{vars(P')-U}) \in solns(P')$.*

3.1 Searching with Caching

Detecting subproblem domination allows us to avoid exploring the dominated subproblem and reuse the solutions of the dominating subproblem (Proposition 1). This is particularly easy when we are only interested in the first solution, since we know the dominated subproblem must have no solutions. The algorithm for first solution satisfaction search using domination is shown in Figure 1. At each node, it propagates using solv. If it detects unsatisfiability it immediately fails. Otherwise, it checks whether the current subproblem is dominated by something already visited (and, thus, in *Cache*), and if so it fails, It then checks whether we have reached a solution and if so returns it. Otherwise it splits the current subproblem into a logically equivalent set of subproblems and examines each of them separately. When the entire subtree has been exhaustively searched, the subproblem is added to the cache.

The above algorithm can be straightforwardly extended to a branch and bound optimization search. This is because any subproblem cached has failed under a weaker set of constraints, and will thus also fail with a strictly stronger set of constraints. As a result, to extend the algorithm in Figure 1 to, for example, minimize the objective function $\sum_{i=1}^{n} a_i x_i$, we can simply replace the line labelled [SAT] by the following lines:[1]

```
globally store $fx(D)$ as best solution
globally add $\sum_{i=1}^{n} a_i x_i \leq fx(D)(\sum_{i=1}^{n} a_i x_i) - 1$
return $false$
```

Note that in this algorithm, the search always fails with the optimal solution being the last one stored.

[1] We assume there is an upper bound u on the objective function so that we can have a pseudo-constraint $\sum_{i=1}^{n} a_i x_i \leq u$ in the problem from the beginning, and replace it with the new one whenever a new solution is found.

4 Keys for Caching

The principal difficulty in implementing cache_search of Figure 1 is implementing the lookup and test for U-dominance. We need an efficient *key* to represent remaining subproblems that allows U-dominance to be detected efficiently (preferably in $O(1)$ time). Naively, one may think D would be a good key for subproblem $P \equiv (C, D)$. While using D as key is correct, it is also useless since D is different for each subproblem (i.e., node) in the search tree. We need to find a more general key; one that can represent equivalent subproblems with different domain constraints.

4.1 Projection Keys

We can automatically construct such a key by using constraint projection. Roughly speaking, subproblem U-equivalence arises whenever the value of some of the fixed variables in $C \wedge D$ and $C \wedge D'$ is different, but the global effect of the fixed variables on the unfixed variables of C is the same. Therefore, if we can construct a key that characterises exactly this effect, the key should be identical for all U-equivalent subproblems.

To do this, we need to characterize the projected subproblem of each $P \equiv (C, D)$ in terms of its projected variables and constraints. Let $F = \mathit{fixed}(D)$ and $U = \mathit{vars}(C) - F$. The projected subproblem can be characterized as:

$$\begin{aligned}
& \bar{\exists}_U.(C \wedge D) \\
\Leftrightarrow\ & \bar{\exists}_U.(C \wedge D_F \wedge D_U) \\
\Leftrightarrow\ & \bar{\exists}_U.(C \wedge D_F) \wedge D_U \\
\Leftrightarrow\ & \bar{\exists}_U.(\wedge_{c \in C}(c \wedge D_F)) \wedge D_U \\
\Leftrightarrow\ & \wedge_{c \in C}(\bar{\exists}_U.(c \wedge D_F)) \wedge D_U
\end{aligned}$$

The last step holds because all variables being projected out in every $c \wedge D_F$ were already fixed. Importantly, this allows each constraint $c \in C$ to be treated independently.

We can automatically convert this information into a key by *back projecting* the projected constraints of this problem to determine conditions on the fixed variables F. We define the back projection of constraint $c \in C$ for D_F as a constraint $BP(c, D_F)$ over variables $F \cap \mathit{vars}(c)$ such that $\bar{\exists}_U.(c \wedge BP(c, D_F)) \Leftrightarrow \bar{\exists}_U.(c \wedge D_F)$. Clearly, while $D_{F \cap \mathit{vars}(c)}$ is always a correct back projection, our aim is to define the most general possible back projection that ensures the equivalence. Note that if c has no variables in common with F, then $BP(c, D_F) \equiv \mathit{true}$. Note also that when c is implied by D_F, that is $\bar{\exists}_U.(c \wedge D_F) \Leftrightarrow \mathit{true}$, then c can be eliminated. We thus define $BP(c, D_F) \equiv \mathit{red}(c)$, where $\mathit{red}(c)$ is simply a name representing the disjunction of all constraints that force c to be redundant (we will see later how to remove these artificial constraints). The *problem key* for $P \equiv (C, D)$ is then defined as $\mathit{key}(C, D) \equiv \wedge_{c \in C} BP(c, D_F) \wedge D_U$.

Example 3. Consider the problem $C \equiv \{\mathit{alldiff}([x_1, x_2, x_3, x_4, x_5, x_6]), x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20\}$ and domain $D \equiv \{x_1 = 3, x_2 = 4, x_3 = 5, x_4 \in \{0, 1, 2\}, x_5 \in$

$\{0,1,2\}, x_6 \in \{1,2,6\}\}$. Then $F = \{x_1, x_2, x_3\}$ and $U = \{x_4, x_5, x_6\}$. The projected subproblem is characterized by $\mathit{alldiff}([x_4, x_5, x_6]) \wedge x_4 + 2x_5 \leq 4 \wedge D_U$. A correct back projection for *alldiff* onto $\{x_1, x_2, x_3\}$ is $\{x_1, x_2, x_3\} = \{3, 4, 5\}$. A correct back projection of the linear inequality is $x_1 + 2x_2 + x_3 = 16$. Thus, $\mathit{key}(C, D) \equiv \{x_1, x_2, x_3\} = \{3, 4, 5\} \wedge x_1 + 2x_2 + x_3 = 16 \wedge x_4 \in \{0, 1, 2\} \wedge x_5 \in \{1, 2\} \wedge x_6 \in \{1, 2, 6\}$. □

We now illustrate how to use the keys for checking dominance.

Theorem 1. *Let $P \equiv (C, D)$ and $P' \equiv (C, D')$ be subproblems arising during the search. Let $F = \mathit{fixed}(D)$ and $U = \mathit{vars}(C) - F$. If $\mathit{fixed}(D') = F$, $D'_U \Rightarrow D_U$ and $\forall c \in C.(\bar{\exists}_U.c \wedge BP(c, D'_F)) \Rightarrow (\bar{\exists}_U.c \wedge BP(c, D_F))$ then P U-dominates P'.*

Proof. The first condition of U-dominance holds since $\mathit{vars}(D') - U = F$. We show the second condition holds:

$$\begin{array}{rl}
 & \bar{\exists}_U.(C \wedge D') \\
 & \Leftrightarrow \bar{\exists}_U.(C \wedge D'_F \wedge D'_U) \\
(\star) & \Leftrightarrow \wedge_{c \in C}(\bar{\exists}_U.c \wedge D'_F) \wedge D'_U \\
 & \Leftrightarrow \wedge_{c \in C}(\bar{\exists}_U.c \wedge BP(c, D'_F)) \wedge D'_U \\
 & \Rightarrow \wedge_{c \in C}(\bar{\exists}_U.c \wedge BP(c, D_F)) \wedge D_U \\
 & \Leftrightarrow \wedge_{c \in C}(\bar{\exists}_U.c \wedge D_F) \wedge D'_U \\
(\star) & \Leftrightarrow \bar{\exists}_U.(C \wedge D_F \wedge D'_U) \\
 & \Rightarrow \bar{\exists}_U.(C \wedge D)
\end{array}$$

The second and sixth (marked) equivalences hold because, again, all variables being projected out in each $c \wedge D'_F$ and $c \wedge D_F$ were already fixed. □

Corollary 1. *Suppose $P \equiv (C, D)$ and $P' \equiv (C, D')$ are subproblems arising in the search tree for C. Let $F = \mathit{fixed}(D)$ and $U = \mathit{vars}(C) - F$. If $\mathit{key}(C, D) \equiv \mathit{key}(C, D')$ then $\mathit{fixed}(D') = F$ and P and P' are U-equivalent.*

Proof. Let $F' = \mathit{fixed}(D')$, $U' = \mathit{vars}(C) - F'$. Since $\mathit{key}(C, D) \equiv \mathit{key}(C, D')$ we have that $D_U \Leftrightarrow D'_{U'}$ and hence $F = F'$ and $U = U'$. Also clearly $\forall c \in C.BP(c, D_F) \equiv BP(c, D'_F)$ Hence, P U-dominates P' and vice versa. □

While determining a back projection is a form of *constraint abduction* which can be a very complex task, we only need to find simple kinds of abducibles for individual constraints and fixed variables. Hence, we can define for each constraint a method to determine a back projection. Figure 2 shows back projections for some example constraints and variable fixings.

Note that a domain consistent binary constraint c always has either no unfixed variables (and, hence, its back projection is *true*), or all its information is captured by domain constraints (and, hence, it is redundant and its back projection is $\mathit{red}(c)$).

constraint c	D_F	$\bar{\exists}_U.c$	$BP(c, D_F)$
$\mathit{alldiff}([x_1, \ldots, x_n])$	$\wedge_{i=1}^m x_i = d_i$	$\mathit{alldiff}([d_1, \ldots, d_m, x_{m+1}, \ldots x_n])$	$\{x_1, \ldots, x_m\} = \{d_1, \ldots, d_m\}$
$\sum_{i=1}^n a_i x_i = a_0$	$\wedge_{i=1}^m x_i = d_i$	$\sum_{i=m+1}^n a_i x_i = a_0 - \sum_{i=1}^m a_i d_i$	$\sum_{i=1}^m a_i x_i = \sum_{i=1}^m a_i d_i$
$\sum_{i=1}^n a_i x_i \leq a_0$	$\wedge_{i=1}^m x_i = d_i$	$\sum_{i=m+1}^n a_i x_i \leq a_0 - \sum_{i=1}^m a_i d_i$	$\sum_{i=1}^m a_i x_i = \sum_{i=1}^m a_i d_i$
$x_0 = \min_{i=1}^n x_i$	$\wedge_{i=1}^m x_i = d_i$ $x_0 = d_0$	$x_o = \min(\min_{i=1}^m d_i, \min_{i=m+1}^n x_i)$ $\wedge_{i=1}^n x_i \geq d_0 \wedge \vee_{i=1}^n x_i = d_0$	$\min_{i=1}^m x_i = \min_{i=1}^m d_i$ $x_0 = d_0$
$\vee_{i=1}^n x_i$	$x_1 = \mathit{true}$ $\wedge_{i=1}^m x_i = \mathit{false}$	true $\vee_{i=m+1}^n x_i$	$\mathit{red}(\vee_{i=1}^n x_i)$ $\wedge_{i=1}^m x_i = \mathit{false}$

Fig. 2. Example constraints with their fixed variables, projections and resulting back projection

4.2 Using Projection Keys

By Corollary 1, if we store every explored subproblem P in the cache using $\mathit{key}(P)$, and we encounter a subproblem P' such that $\mathit{key}(P')$ appears in the cache, then P' is equivalent to a previous explored subproblem and does not need to be explored.

Example 4. Consider the problem $P \equiv (C, D)$ of Example 3 and the new subproblem $P' \equiv (C, D')$ where $D' \equiv \{x_1 = 5, x_2 = 4, x_3 = 3, x_4 \in \{0, 1, 2\}, x_5 \in \{0, 1\}, x_6 \in \{1, 2, 6\}\}$. The characterisation of the projected subproblem for P' is identical to that obtained in Example 3 and, hence, $\mathit{key}(P) \equiv \mathit{key}(P')$ indicating P and P' are U-equivalent. □

If we are using projection keys for detecting subproblem equivalence, we are free to represent the keys in any manner that illustrates identity. This gives use the freedom to generate space efficient representations, and choose representations for $BP(c, D_F)$ on a per constraint basis.

Example 5. Consider the problem $P \equiv (C, D)$ of Example 3. We can store its projection key $\{x_1, x_2, x_3\} = \{3, 4, 5\} \wedge x_1 + 2x_2 + x_3 = 16 \wedge D_U$ as follows: We store the fixed variables $\{x_1, x_2, x_3\}$ for the subproblem since these must be identical for the equivalence check in any case. We store $\{3, 4, 5\}$ for the *alldiff* constraint, and the fixed value 16 for the linear constraint, which give us enough information given the fixed variables to define the key. The remaining part of the key are domains. Thus, the projection key can be stored as $(\{x_1, x_2, x_3\}, \{3, 4, 5\}, 16, \{0, 1, 2\}, \{0, 1, 2\}, \{1, 2, 6\})$ □

Theorem 1 shows how we can make use of projection keys to determine subproblem dominance. If we store $\mathit{key}(P)$ in the cache we can determine if new subproblem P' is dominated by a previous subproblem by finding a key where the fixed variables are the same, each projection of a primitive constraint for P' is at least as strong as the projection defined by $\mathit{key}(P)$, and the domains of the unfixed variables in P' are at least as strong as the unfixed variables in $\mathit{key}(P)$.

Example 6. Consider $P \equiv (C, D)$ of Example 3 and the new subproblem $P' \equiv (C, D')$ where $D' \equiv \{x_1 = 4, x_2 = 5, x_3 = 3, x_4 \in \{0, 1, 2\}, x_5 \in \{0, 1\}, x_6 \in \{1, 2, 6\}\}$. We have that $\mathit{fixed}(D') = \mathit{fixed}(D) = \{x_1, x_2, x_3\}$ and the back projections of the *alldiff* are identical. Also, the projection of the linear inequality is

$x_4 + 2x_5 \leq 3$. This is stronger than the projection in *key*(P) which is computable as $\exists x_1.\exists x_2.\exists x_3.x_1+2x_2+x_3 = 16 \wedge x_1+2x_2+x_3+x_4+2x_5 \leq 20 \Leftrightarrow x_4+2x_5 \leq 4$. Similarly, $D'_U \Rightarrow D_U$. Hence, P $\{x_4, x_5, x_6\}$-dominates P'. □

To use projection keys for dominance detection we need to check $D'_U \Rightarrow D_U$ and $(\bar{\exists}_U.c \wedge BP') \Rightarrow (\bar{\exists}_U.c \wedge BP)$. Note that if $BP \equiv red(c)$, then the entailment automatically holds and we do not need to store these artificial projection keys. Note also that we can make the choice of how to check for entailment differently for each constraint. We will often resort to identity checks as a weak form of entailment checking, since we can then use hashing to implement entailment.

Example 7. Consider $P \equiv (C, D)$ of Example 3. Entailment for *alldiff* is simply identity on the set of values, while for the linear constraint we just compare fixed values, since $(\exists x_1.\exists x_2.\exists x_3.\ x_1+2x_2+x_3 = k \wedge x_1+2x_2+x_3+x_4+2x_5 \leq 20) \Leftrightarrow x_4+2x_5 \leq 20-k \Rightarrow (\exists x_1.\exists x_2.\exists x_3.\ x_1+2x_2+x_3 = k' \wedge x_1+2x_2+x_3+x_4+2x_5 \leq 20) \Leftrightarrow x_4 + 2x_5 \leq 20 - k'$ whenever $k \geq k'$. For the problem P' of Example 6 we determine the key $(\{x_1, x_2, x_3\}, \{3, 4, 5\}, 17, \{0, 1, 2\}, \{0, 1\}, \{1, 2, 6\})$. We can hash on the first two arguments of the tuple to retrieve the key for P, and then compare 17 versus 16 and check that each of the three last arguments is a superset of that appearing in *key*(P'). Hence, we determine the dominance holds. □

Note that, for efficiency, our implementation checks $D'_U \Rightarrow D_U$ by using identity $(D'_U \equiv D_U)$ so the domains can be part of the hash value. This means that the problem P' of Example 6 will not be detected as dominated in our implementation, since the domain of x_5 is different.

4.3 Caching Optimal Subproblem Values

The presentation so far has concentrated on satisfaction problems; let us examine what happens with optimization problems. Typically, when solving optimization problems with caching one wants to store optimal partial objective values with already explored subproblems. We shall see how our approach effectively manages this automatically using dominance detection with a minor change.

Suppose k is the current best solution found. Then, the problem constraints must include $\sum_{i=1}^{n} a_i x_i \leq k - 1$ where $\sum_{i=1}^{n} a_i x_i$ is the objective function. Suppose we reach a subproblem $P \equiv (C, D)$ where $D_{fixed(D)} \equiv \{x_1 = d_1, \ldots, x_m = d_m\}$ are the fixed variables. The remaining part of the objective function constraint is $\sum_{i=m+1}^{n} a_i x_i \leq k-1-p$ where $p = \sum_{i=1}^{m} a_i d_i$, and the back projection is $\sum_{i=1}^{m} a_i x_i = p$. The projection key contains the representation p for this back projection. If this subproblem fails we have proven that, with D, there is no solution with a value $< k$, nor with $\sum_{i=m+1}^{n} a_i x_i \leq k - 1 - p$.

If we later reach a subproblem $P' \equiv (C, D')$ where $D' \Rightarrow x_1 = d'_1 \wedge \cdots \wedge x_m = d'_m$ are the fixed variables, then dominance requires $p' = \sum_{i=1}^{m} a_i d'_i$ to satisfy $p' \geq p$. If this does not hold it may be that a solution for the projected problem with $\sum_{i=m+1}^{n} a_i x_i \geq k - p$ can lead to a global solution $< k$. Hence, we do have to revisit this subproblem.

Suppose that by the time we reach P', a better solution $k' < k$ has been discovered. Effectively the constraint $\sum_{i=1}^{n} a_i x_i \leq k - 1$ has been replaced by $\sum_{i=1}^{n} a_i x_i \leq k' - 1$. Now we are only interested in finding a solution where $\sum_{i=m+1}^{n} a_i x_i \leq k' - 1 - p'$. To see if this is dominated by a previous subproblem, the stored value p is not enough. We also need the optimal value k when the key was stored. There is a simple fix: rather than storing p in the key for P we store $q = k - 1 - p$. We can detect dominance if $q \leq k' - 1 - p'$ and this value q is usable for all future dominance tests. Note that q *implicitly* represents the partial objective bound on the subproblem P.

5 Related Work

Problem specific approaches to dominance detection/subproblem equivalance are widespread in combinatorial optimization (see e.g. [6,19]) There is also a significant body of work on caching that rely on problem decomposition by fixing variables (e.g [10,12]). This work effectively looks for equivalent projected problems, but since they do not take into account the semantics of the constraints, they effectively use $D_{F \cap vars(c)}$ for every constraint c as the projection key, which finds strictly fewer equivalent subproblems than back-projection. The success of these approaches in finding equivalent subproblems relies on decomposing the projected subproblem into disjoint parts. We could extend our approach to also split the projected problem into connected components but this typically does not occur in the problems of interest to us. Interestingly, [10] uses symmetry detection to make subproblem equivalence detection stronger, but the method used does not appear to scale.

5.1 Dynamic Programming

Dynamic programming (DP) [2] is a powerful approach for solving optimization problems whose optimal solutions are derivable from the optimal solutions of its subproblems. It relies on formulating an optimization as recursive equations relating the answers to optimization problems of the same form. When applicable, it is often near unbeatable by other optimization approaches.

Constraint programming (CP) with caching is similar to DP, but provides several additional capabilities. For example, arbitrary side constraints not easily expressible as recursions in DP can easily be expressed in CP, and dominance can be expressed and exploited much more naturally in CP.

Consider the 0-1 Knapsack problem, a well known NP-hard problem that is easy to formulate using recursive equations suitable for DP. We show how our automatic caching provides a different but similar solution, and how caching can change the asymptotic complexity of the CP solution. The problem is to maximise $\sum_{i=1}^{n} p_i x_i$ subject to the constraints $\sum_{i=1}^{n} w_i x_i \leq W \wedge \forall_{i=1}^{n} x_i \in \{0, 1\}$, where w_i is the nonnegative weight of object i and p_i is the nonnegative profit. A normal CP solver will solve this problem in $O(2^n)$ steps.

The DP formulation defines $knp(j, w)$ as the maximum profit achievable using the first j items with a knapsack of size w. The recursive equation is

$$knp(j, w) = \begin{cases} 0 & j = 0 \lor w \leq 0 \\ \max(knp(j-1, w), knp(j-1, w - w_j) + p_j) & \text{otherwise} \end{cases}$$

The DP solution is $O(nW)$ since values for $knp(j, w)$ are cached and only computed once. Consider a CP solver using a fixed search order $x_1, \ldots, x_n$. A subproblem fixing $x_1, \ldots, x_m$ to $d_1, \ldots, d_m$ respectively generates key value $\sum_{i=1}^{m} w_i d_i$ for the constraint $\sum_{i=1}^{n} w_i x_i \leq W$ and key value $k + 1 - \sum_{i=1}^{m} p_i d_i$ for the optimization constraint $\sum_{i=1}^{n} p_i x_i \geq k + 1$ where k is the best solution found so far. The remaining variable domains are all unchanged so they do not need to be explicitly stored (indeed domains of Boolean or 0-1 variables never need to be stored as they are either fixed or unchanged). The projection key is simply the set of fixed variables $\{x_1, \ldots, x_m\}$ and the two constants. The complexity is hence $O(nWu)$ where u is the initial upper bound on profit.

The solutions are in fact quite different: the DP approach stores the optimal profit for each set of unfixed variables and remaining weight limit, while the CP approach stores the fixed variables and uses weight plus the remaining profit required. The CP approach in fact implements a form of DP with bounding [16]. In particular, the CP approach can detect subproblem dominance, a problem with used weight w' and remaining profit required p' is dominated by a problem with used weight $w \leq w'$ and remaining profit $p \leq p'$. The DP solution must examine both subproblems since the remaining weights are different.

In practice the number of remaining profits arising for the same set of fixed variables and used weight is $O(1)$ and hence the practical number of subproblems visited by the CP approach is $O(nW)$.

Note that while adding a side constraint like $x_3 \geq x_8$ destroys the DP approach (or at least forces it to be carefully reformulated), the CP approach with automatic caching works seamlessly.

5.2 Symmetry Breaking

Symmetry breaking aims at speeding up execution by not exploring search nodes known to be symmetric to nodes already explored. Once the search is finished, all solutions can be obtained by applying each symmetry to each solution. In particular, Symmetry Breaking by Dominance Detection (SBDD) [4] works by performing a "dominance check" at each search node and, if the node is found to be dominated, not exploring the node.

SBDD is related but different to automatic caching. In SBDD $P \equiv (C, D)$ ϕ-dominates $P' \equiv (C, D')$ under symmetry ϕ iff $\phi(D') \Rightarrow D$ since, if this happens, the node associated to symmetric problem $(C, \phi(D'))$ must be a descendant of the node associated to P and, thus, already explored. Note that, in detecting dominance, SBDD places conditions on the domains of *all* variables in D, while automatic caching only does so on the constraints of the problem *once projected on the unfixed variables*. Thus, P' can be ϕ-dominated by P (in the SBDD sense) but not be U-dominated, and vice versa.

Our approach is also related to conditional symmetry breaking [7], which identifies conditions that, when satisfied in a subproblem, ensure new symmetries occur within that subproblem (and not in the original problem). As before, the two approaches capture overlapping but distinct sets of redundancies.

5.3 Nogood Learning

Nogood learning approaches in constraint programming attempt to learn from failures and record these as new constraints in the program. The most successful of these methods use clauses on atomic constraints $v = d$ and $v \leq d$ to record the reasons for failures and use SAT techniques to efficiently manage the nogoods. Automatic caching is also a form of nogood learning, since it effectively records keys that lead to failure.

Any nogood learning technique representing nogoods as clauses has the advantage over caching that it can use the nogoods derived to propagate rather than to simply fail. Restart learning [11] simply records failed subtrees using the set of decisions made to arrive there. This does not allow subproblem equivalence to be detected assuming a fixed search strategy. The usefulness arises because it is coupled with restarting and dynamic search, so it helps avoid repeated search. In that sense it has a very different aim to automatic caching. Nogood learning techniques such as lazy clause generation [14] learn clauses that are derived only from the constraints that are actually involved in conflicts, which is much more accurate than using all non-redundant constraints as in projection keys.

On the other hand nogood learning can come at a substantial price: reason generation and conflict analysis can be costly. Every clause learnt in a nogood approach adds extra constraints and, hence, slows down the propagation of the solver. In contrast, projection keys are $O(1)$ to lookup regardless of their number (at least for the parts that are in the hash).

Because nogood learning use clauses on atomic constraints to define nogoods they may be less expressive than projection keys. Consider the subproblem $x_1 + 2x_2 + x_3 + x_4 + 2x_5 \leq 20 \wedge C$, with $D \equiv \{x_1 = 1 \wedge x_2 = 2 \wedge x_3 = 3\}$. If this subproblem fails, the projection key stores that $x_4+2x_5 \leq 12 \wedge$other keys leads to failure. A nogood system will express this as $x_1 = 1 \wedge x_2 = 2 \wedge x_3 = 3 \wedge$other keys leads to failure, since there are no literals to representing partial sums. This weakness is illustrated by the experimental results for 0-1 Knapsack.

6 Experiments

We compare our solver Chuffed, with and without caching, against Gecode 3.2.2 [17] – widely recognized as one of the fastest constraint programming systems (to illustrate we are not optimizing a slow system) – against the G12 FD solver [15] and against the G12 lazy clause generation solver [5] (to compare against nogood learning). We use the `MurmurHash 2.0` hash function. We use models written in the modelling language MiniZinc [13]. This facilitates a fair comparison between the solvers, as all solvers use the same model and search strategy. Note that caching does not interfere with the search strategies used here, as all it can do is fail subtrees earlier. Thus, Chuffed with caching (denoted as ChuffedC) always finds the same solution as the non-caching version and the other solvers, and any speedup observed comes from a reduced search.

Considerable engineering effort has gone into making caching as efficient as possible: defining as general as possible back projections for each of the many primitive constraints defined in the solver, exploiting small representations for back projections and domains, and eliminating information that never needs storing, e.g. binary domain consistent constraints and Boolean domains.

The experiments were conducted on Xeon Pro 2.4GHz processors with a 900 second timeout. Table 1 presents the number of variables and constraints as reported by Chuffed, the times for each solver in seconds, and the speedup and node reduction obtained from using automatic caching in Chuffed. We discuss the results for each problem below. All the MiniZinc models and instances are available at `www.cs.mu.oz.au/~pjs/autocache/`

Knapsack. 0-1 knapsack is ideal for caching. The non-caching solvers all timeout as n increases, as their time complexity is $O(2^n)$. This is a worst case for lazy clause generation since the nogoods generated are not reusable. ChuffedC, on the other hand, is easily able to solve much larger instances (see Table 1). The node to nW ratio (not shown) stays fairly constant as n increases (varying between 0.86 and 1.06), showing that it indeed has search (node) complexity $O(nW)$. The time to nW ratio grows as $O(n)$ though, since we are using a general CP solver where the linear constraints take $O(n)$ to propagate at each node, while DP requires constant work per node. Hence, we are not as efficient as pure DP.

MOSP. The minimal open stacks problem (MOSP) aims at finding a schedule for manufacturing all products in a given set that minimizes the maximum number of active customers, i.e., the number of customers still waiting for at least one of their products to be manufactured. This problem was the subject of the 2005 constraint modelling challenge [18]. Of the 13 entrants only 3 made use of the subproblem equivalence illustrating that, in general, it may not be easy to detect. Our MOSP model uses customer search and some complex conditional dominance breaking constraints that make the (non-caching) search much faster. We use random instances from [3]. Automatic caching gives up to two orders of magnitude speedup. The speedup grows exponentially with problem size. Lazy clause is also capable of exploiting this subproblem equivalence, but the overhead is so large that it can actually slow the solver down.

Blackhole. In the Blackhole patience game, the 52 cards are laid out in 17 piles of 3, with the ace of spades starting in a "blackhole". Each turn, a card at the top of one of the piles can be played into the blackhole if it is +/-1 from the card that was played previously. The aim is to play all 52 cards. This was one of two examples used to illustrate CP with caching in [19]. The remaining subproblem only depends on the set of unplayed cards, and the value of the last card played.

Thus, there is subproblem equivalence. We use a model from [7] which includes conditional symmetry breaking constraints. We generated random instances and used only the hard ones for this experiment. The G12 solvers do not use a domain consistent *table* constraint for this problem and are several orders of magnitudes slower. Automatic caching gives a modest speedup of around 2-3. The speedup is relatively low on this problem because the conditional symmetry breaking constraints have already removed many equivalent subproblems, and the caching is only exploiting the ones which are left. Note that the manual caching reported in [19] achieves speedups in the same range (on hard instances).

BACP. In the Balanced Academic Curriculum Problem (BACP), we form a curriculum by assigning a set of courses to a set of periods, with certain restrictions on how many courses and how much "course load" can be assigned to each period. We also have prerequisite constraints between courses. The BACP can be viewed as a bin packing problem with a lot of additional side constraints. The remaining subproblem only depends on the set of unassigned courses, and not on how the earlier courses were assigned. We use the model of [9], but with some additional redundant constraints that make it very powerful. The 3 instances *curriculum_*8/10/12 given in CSPLIB can be solved to optimality in just a few milliseconds. We generate random instances with 50 courses, 10 periods, and course credit ranging between 1 and 10. Almost all are solvable in milliseconds so we pick out only the non-trivial ones for the experiment. We also include the 3 standard instances from CSPLIB. Both automatic caching and lazy clause generation are capable of exploiting the subproblem equivalence, giving orders of magnitude speedup. In this case, lazy clause generation is more efficient.

Radiation Therapy. In the Radiation Therapy problem [1], the aim is to decompose an integral intensity matrix describing the radiation dose to be delivered to each area, into a set of patterns to be delivered by a radiation source, while minimising the amount of time the source has to be switched on, as well as the number of patterns used (setup time of machine). The subproblem equivalence arises because there are equivalent methods to obtain the same cell coverages, e.g. radiating one cell with two intensity 1 patterns is the same as radiating it with one intensity 2 pattern, etc. We use random instances generated as in [1]. Both automatic caching and lazy clause generation produce orders of magnitude speedup, though lazy clause generation times are often slightly better.

Memory Consumption. The memory consumption of our caching scheme is linear in the number of nodes searched. The size of each key is dependent on the structure of the problem and can range from a few hundred bytes to tens of thousands of bytes. On a modern computer, this means we can usually search several hundreds of thousands of nodes before running out of memory. There are simple schemes to reduce the memory usage, which we plan to investigate in the future. For example, much like in SAT learning, we can keep an "activity" score for each entry to keep track of how often they are used. Inactive entries can then periodically be pruned to free up memory.

Table 1. Experimental Results

Instance	vars	cons.	ChuffedC	Chuffed	Gecode	G12_fd	G12_lazyfd	Speedup	Node red.
knapsack-20	21	2	0.01	0.01	0.01	0.01	0.10	1.00	2.9
knapsack-30	31	2	0.02	0.83	0.76	1.168	534.5	41.5	67
knapsack-40	41	2	0.03	38.21	34.54	58.25	>900	1274	1986
knapsack-50	51	2	0.07	>900	>900	>900	>900	>12860	>20419
knapsack-60	61	2	0.10	>900	>900	>900	>900	>9000	>14366
knapsack-100	101	2	0.40	>900	>900	>900	>900	>2250	> 2940
knapsack-200	201	2	2.36	>900	>900	>900	>900	>381	> 430
knapsack-300	301	2	6.59	>900	>900	>900	>900	>137	>140
knapsack-400	401	2	13.96	>900	>900	>900	>900	>65	>65
knapsack-500	501	2	25.65	>900	>900	>900	>900	>35	> 34
mosp-30-30-4-1	1021	1861	1.21	4.80	24.1	50.29	29.70	4.0	4.91
mosp-30-30-2-1	1021	1861	6.24	>900	>900	>900	201.8	>144	>187
mosp-40-40-10-1	1761	3281	0.68	0.66	5.85	15.07	29.80	1.0	1.1
mosp-40-40-8-1	1761	3281	1.03	1.15	9.92	27.00	56.96	1.1	1.3
mosp-40-40-6-1	1761	3281	3.79	11.30	75.36	183.9	165.2	3.0	3.5
mosp-40-40-4-1	1761	3281	19.07	531.68	>900	>900	840.4	28	37
mosp-40-40-2-1	1761	3281	60.18	>900	>900	>900	>900	>15	> 18
mosp-50-50-10-1	2701	5101	2.83	3.17	40.70	92.74	134.1	1.1	1.2
mosp-50-50-8-1	2701	5101	6.00	9.12	113.0	292.0	295.9	1.5	1.8
mosp-50-50-6-1	2701	5101	39.65	404.16	>900	>900	>900	10.2	13.1
blackhole-1	104	407	18.35	39.77	103.6	>900	>900	2.17	2.90
blackhole-2	104	411	14.60	21.52	60.06	>900	>900	1.47	1.94
blackhole-3	104	434	18.31	26.14	31.43	>900	>900	1.43	1.81
blackhole-4	104	393	15.77	30.84	69.13	>900	>900	1.96	2.55
blackhole-5	104	429	24.88	58.77	159.5	>900	>900	2.36	3.45
blackhole-6	104	448	11.31	33.27	85.65	>900	>900	2.94	5.11
blackhole-7	104	407	28.02	47.31	127.6	>900	>900	1.69	2.49
blackhole-8	104	380	24.09	43.60	89.02	>900	>900	1.81	2.45
blackhole-9	104	404	38.74	93.92	215.1	>900	>900	2.42	3.52
blackhole-10	104	364	67.85	159.4	418.0	>900	>900	2.35	3.16
curriculum_8	838	1942	0.01	0.01	0.01	0.02	0.08	1.00	1.00
curriculum_10	942	2214	0.01	0.01	0.01	0.03	0.09	1.00	1.00
curriculum_12	1733	4121	0.01	0.01	0.01	0.10	0.23	1.00	1.00
bacp-medium-1	1121	2654	11.47	34.90	29.31	62.4	6.90	3.04	3.03
bacp-medium-2	1122	2650	9.81	>900	>900	>900	0.22	>92	>115
bacp-medium-3	1121	2648	2.42	380.7	461.62	838.6	0.23	157	190
bacp-medium-4	1119	2644	0.61	4.59	5.74	9.92	1.10	7.52	10.1
bacp-medium-5	1119	2641	2.40	56.46	54.03	126.9	0.76	23.5	26.5
bacp-hard-1	1121	2655	54.66	>900	>900	>900	0.16	>16	>16
bacp-hard-2	1118	2651	181.9	>900	>900	>900	0.22	>5	>7
radiation-6-9-1	877	942	12.67	>900	>900	>900	2.89	>71	>146
radiation-6-9-2	877	942	27.48	>900	>900	>900	5.48	>32	>86
radiation-7-8-1	1076	1168	0.84	>900	>900	>900	1.40	>1071	>5478
radiation-7-8-2	1076	1168	0.65	89.18	191.4	173.6	0.93	137	633
radiation-7-9-1	1210	1301	2.39	143.0	315.6	241.9	2.70	59	266
radiation-7-9-2	1210	1301	7.26	57.44	144.4	101.9	8.83	8	34
radiation-8-9-1	1597	1718	27.09	>900	>900	>900	6.21	>33	>114
radiation-8-9-2	1597	1718	12.21	>900	>900	>900	6.53	>74	>267
radiation-8-10-1	1774	1894	22.40	12.17	15.45	12.90	33.2	0.54	1.10
radiation-8-10-2	1774	1894	59.66	>900	>900	>900	12.05	>15	>78

7 Conclusion

We have described how to automatically exploit subproblem equivalence in a general constraint programming system by automatic caching. Our automatic caching can produce orders of magnitude speedup over our base solver CHUFFED, which (without caching) is competitive with current state of the art constraint programming systems like Gecode. With caching, it can be much faster on problems that have subproblem equivalences.

The automatic caching technique is quite robust. It can find and exploit subproblem equivalence even in models that are not "pure", e.g. MOSP with dominance and conditional symmetry breaking constraints, Blackhole with conditional symmetry breaking constraints, and BACP which can be seen as bin packing with lots of side constraints and some redundant constraints. The speedups from caching tends to grow exponentially with problem size/difficulty, as subproblem equivalences also grow exponentially.

Our automatic caching appears to be competitive with lazy clause generation in exploiting subproblem equivalence, and is superior on some problems, in particular those with large linear constraints.

The overhead for caching is quite variable (it can be read from the tables as the ratio of node reduction to speedup). For large problems with little variable fixing it can be substantial (up to 5 times for radiation), but for problems that fix variables quickly it can be very low. Automatic caching of course relies on subproblem equivalence occurring to be of benefit. Note that for dynamic searches this is much less likely to occur. Since it is trivial to invoke, it seems always worthwhile to try automatic caching for a particular model, and determine empirically if it is beneficial.

Acknowledgments. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

References

1. Baatar, D., Boland, N., Brand, S., Stuckey, P.J.: Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 1–15. Springer, Heidelberg (2007)
2. Bellman, R.: Dynamic Programming. Princeton University Press, Princeton (1957)
3. Chu, G., Stuckey, P.J.: Minimizing the maximum number of open stacks by customer search. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 242–257. Springer, Heidelberg (2009)
4. Fahle, T., Schamberger, S., Sellmann, M.: Symmetry breaking. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 93–107. Springer, Heidelberg (2001)
5. Feydy, T., Stuckey, P.J.: Lazy clause generation reengineered. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 352–366. Springer, Heidelberg (2009)
6. Fukunaga, A., Korf, R.: Bin completion algorithms for multicontainer packing, knapsack, and covering problems. J. Artif. Intell. Res. (JAIR) 28, 393–429 (2007)

7. Gent, I., Kelsey, T., Linton, S., McDonald, I., Miguel, I., Smith, B.: Conditional symmetry breaking. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 256–270. Springer, Heidelberg (2005)
8. Gent, I., Petrie, K., Puget, J.-F.: Symmetry in Constraint Programming. In: Handbook of Constraint Programming, pp. 329–376. Elsevier, Amsterdam (2006)
9. Hnich, B., Kiziltan, Z., Walsh, T.: Modelling a balanced academic curriculum problem. In: Proceedings of CPAIOR 2002, pp. 121–131 (2002)
10. Kitching, M., Bacchus, F.: Symmetric component caching. In: Proceedings of IJCAI 2007, pp. 118–124 (2007)
11. Lynce, I., Baptista, L., Marques-Silva, J.: Complete search restart strategies for satisfiability. In: IJCAI Workshop on Stochastic Search Algorithms, pp. 1–5 (2001)
12. Marinescu, R., Dechter, R.: And/or branch-and-bound for graphical models. In: Proceedings of IJCAI 2005, pp. 224–229 (2005)
13. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
14. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
15. Stuckey, P.J., Garcia de la Banda, M., Maher, M.J., Marriott, K., Slaney, J.K., Somogyi, Z., Wallace, M., Walsh, T.: The G12 project: Mapping solver independent models to efficient solutions. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 13–16. Springer, Heidelberg (2005)
16. Puchinger, J., Stuckey, P.J.: Automating branch-and-bound for dynamic programs. In: Proceedings of PEPM 2008, pp. 81–89 (2008)
17. Schulte, C., Lagerkvist, M., Tack, G.: Gecode, `http://www.gecode.org/`
18. Smith, B., Gent, I.: Constraint modelling challenge report 2005 (2005), `http://www.cs.st-andrews.ac.uk/~ipg/challenge/ModelChallenge05.pdf`
19. Smith, B.M.: Caching search states in permutation problems. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 637–651. Springer, Heidelberg (2005)

Single-Facility Scheduling over Long Time Horizons by Logic-Based Benders Decomposition

Elvin Coban and John N. Hooker

Tepper School of Business, Carnegie Mellon University
ecoban@andrew.cmu.edu, john@hooker.tepper.cmu.edu

Abstract. Logic-based Benders decomposition can combine mixed integer programming and constraint programming to solve planning and scheduling problems much faster than either method alone. We find that a similar technique can be beneficial for solving pure scheduling problems as the problem size scales up. We solve single-facility non-preemptive scheduling problems with time windows and long time horizons that are divided into segments separated by shutdown times (such as weekends). The objective is to find feasible solutions, minimize makespan, or minimize total tardiness.

1 Introduction

Logic-based Benders decomposition has been successfully used to solve planning and scheduling problems that naturally decompose into an assignment and a scheduling portion. The Benders master problem assigns jobs to facilities using mixed integer programming (MILP), and the subproblems use constraint programming (CP) to schedule jobs on each facility.

In this paper, we use a similar technique to solve pure scheduling problems with long time horizons. Rather than assign jobs to facilities, the master problem assigns jobs to segments of the time horizon. The subproblems schedule jobs within each time segment.

In particular, we solve single-facility scheduling problems with time windows in which the objective is to find a feasible solution, minimize makespan, or minimize total tardiness. We assume that each job must be completed within one time segment. The boundaries between segments might therefore be regarded as weekends or shutdown times during which jobs cannot be processed. In future research we will address instances in which jobs can overlap two or more segments.

Logic-based Benders decomposition was introduced in [2,8]. Its application to assignment and scheduling via CP/MILP was proposed in [3] and implemented in [9]. This and subsequent work shows that the Benders approach can be orders of magnitude faster than stand-alone MILP or CP methods on problems of this kind [1,7,4,5,6,10,11]. For the pure scheduling problems considered here, we find that the advantage of Benders over both CP and MILP increases rapidly as the problem scales up.

2 The Problem

Each job j has release time, deadline (or due date) d_j, and processing time p_j. The time horizon consists of intervals $[z_i, z_{i+1}]$ for $i = 1, \ldots, m$. The problem is to assign each job j a start time s_j so that time windows are observed ($r_j \leq s_j \leq d_j - p_j$), jobs run consecutively ($s_j + p_j \leq s_k$ or $s_k + p_k \leq s_j$ for all $k \neq j$), and each job is completed within one segment ($z_i \leq s_j \leq z_{i+1} - p_j$ for some i). We minimize makespan by minimizing $\max_j\{s_j + p_j\}$. To minimize tardiness, we drop the constraint $s_j \leq d_j - p_j$ and minimize $\sum_j \max\{0, s_j + p_j - d_j\}$.

3 Feasibility

When the goal is to find a feasible schedule, the master problem seeks a feasible assignment of jobs to segments, subject to the Benders cuts generated so far. Because we solve the master problem with MILP, we introduce 0-1 variables y_{ij} with $y_{ij} = 1$ when job j is assigned to segment i. The master problem becomes

$$\begin{aligned} & \sum_i y_{ij} = 1, \quad \text{all } j \\ & \text{Benders cuts, relaxation} \\ & y_{ij} \in \{0,1\}, \quad \text{all } i, j \end{aligned} \tag{1}$$

The master problem also contains a relaxation of the subproblem, similar to those described in [4,5,6], that helps reduce the number of iterations.

Given a solution $\bar{y}_{ij}$ of the master problem, let $J_i = \{j \mid \bar{y}_{ij} = 1\}$ be the set of jobs assigned to segment i. The subproblem decomposes into a CP scheduling problem for each segment i:

$$\begin{aligned} & \left.\begin{aligned} r_j \leq s_j \leq d_j - p_j \\ z_i \leq s_j \leq z_{i+1} - p_j \end{aligned}\right\}, \text{all } j \in J_i \\ & \texttt{disjunctive}(\{s_j \mid j \in J_i\}) \end{aligned} \tag{2}$$

where the `disjunctive` global constraint ensures that the jobs assigned to segment i do not overlap.

Each infeasible subproblem generates a Benders cut as described below, and the cuts are added to the master problem. The master problem and corresponding subproblems are repeatedly solved until every segment has a feasible schedule, or until the master problem is infeasible, in which case the original problem is infeasible.

Strengthened nogood cuts. The simplest Benders cut is a nogood cut that excludes assignments that cause infeasibility in the subproblem. If there is no feasible schedule for segment i, we generate the cut

$$\sum_{j \in J_i} y_{ij} \leq |J_i| - 1, \quad \text{all } i \tag{3}$$

The cut can be strengthened by removing jobs one by one from J_i until a feasible schedule exists for segment i. This requires re-solving the ith subproblem repeatedly, but the effort generally pays off because the subproblems are much easier to solve than the master problem. We now generate a cut (3) with the reduced J_i.

The cut may be stronger if jobs less likely to cause infeasibility are removed from J_i first. Let the *effective time window* $[\tilde{r}_{ij}, \tilde{d}_{ij}]$ of job j on segment i be its time window adjusted to reflect the segment boundaries. Thus

$$\tilde{r}_{ij} = \max\left\{\min\{r_j, z_{i+1}\}, z_i\right\}, \quad \tilde{d}_{ij} = \min\left\{\max\{d_j, z_i\}, z_{i+1}\right\}$$

Let the *slack* of job j on segment i be $\tilde{d}_{ij} - \tilde{r}_{ij} - p_j$. We can now remove the jobs in order of decreasing slack.

4 Minimizing Makespan

Here the master problem minimizes μ subject to (1) and $\mu \geq 0$. The subproblems minimize μ subject to (2) and $\mu \geq s_j + p_j$ for all $j \in J_i$.

Strengthened nogood cuts. When one or more subproblems are infeasible, we use strengthened nogood cuts (3). Otherwise, for each segment i we use the nogood cut

$$\mu \geq \mu_i^* \left(1 - \sum_{j \in J_i} (1 - y_{ij})\right)$$

where μ_i^* is the minimum makespan for subproblem i. These cuts are strengthened by removing jobs from J_i until the minimum makespan on segment i drops below μ_i^*.

We also strengthen the cuts as follows. Let $\mu_i(J)$ be the minimum makespan that results when in jobs in J are assigned to segment i, so that in particular $\mu_i(J_i) = \mu_i^*$. Let Z_i be the set of jobs that can be removed, one at a time, without affecting makespan, so that $Z_i = \{j \in J_i \mid M_i(J_i \setminus \{j\}) = M_i^*\}$. Then for each i we have the cut

$$\mu \geq \mu_i(J_i \setminus Z_i) \left(1 - \sum_{j \in J_i \setminus Z_i} (1 - y_{ij})\right)$$

This cut is redundant and should be deleted when $\mu_i(J_i \setminus Z_i) = \mu_i^*$.

Analytic Benders Cuts. We can develop additional Benders as follows. Let $J_i' = \{j \in J_i \mid r_j \leq z_i\}$ be the set of jobs in J_i with release times before segment i, and let $J_i'' = J_i \setminus J_i'$. Let $\hat{\mu}_i$ be the minimum makespan of the problem that remains after removing the jobs in $S \subset J_i'$ from segment i. It can be shown as in [6] that

$$\mu_i^* - \hat{\mu}_i \leq p_S + \max_{j \in J_i'}\{\tilde{d}_j\} - \min_{j \in J_i'}\{\tilde{d}_j\} \tag{4}$$

where $p_S = \sum_{j \in S} p_j$. Thus if jobs in J_i' are removed from segment i, we have from (4) a lower bound on the resulting optimal makespan $\hat{\mu}_i$. If jobs in J_i'' are removed, there is nothing we can say. So we have the following Benders cut for each i:

$$\mu \geq \mu_i^* - \left(\sum_{j \in J_i'} p_j(1 - y_{ij}) + \max_{j \in J_i'}\{d_j\} - \min_{j \in J_i'}\{d_j\}\right) - \sum_{j \in J_i''} \mu_i^*(1 - y_{ij}) \tag{5}$$

when one or more jobs are removed from segment i, $\mu \geq 0$ when all jobs are removed, and $\mu \geq \mu_i^*$ otherwise. This can be linearized:

$$\mu \geq \mu_i^* - \sum_{j \in j_i'} p_j(1-y_{ij}) - w_i - \sum_{j \in J_i''} \mu_i^*(1-y_{ij}) - \mu_i^* q_i, \quad q_i \leq 1 - y_{ij},\ j \in J_i$$

$$w_i \leq \left(\max_{j \in J_i'}\{d_j\} - \min_{j \in J_i'}\{d_j\} \right) \sum_{j \in J_i'} (1-y_{ij}), \quad w_i \leq \max_{j \in J_i'}\{d_j\} - \min_{j \in J_i'}\{d_j\}$$

5 Minimizing Tardiness

Here the master problem minimizes τ subject to (1), and each subproblem minimizes $\sum_{j \in J_i} \tau_j$ subject to $\tau_j \geq s_j + p_j - d_j$ and $\tau_j \geq 0$.

Benders cuts. We use strengthened nogood cuts and relaxations similar to those used for minimizing makespan. We also develop the analytic Benders cuts

$$\tau \geq \sum_i \hat{\tau}_i$$

$$\hat{\tau}_i \geq \begin{cases} \tau_i^* - \sum_{j \in J_i} \left(r_i^{\max} + \sum_{\ell \in J_i} p_\ell - d_j \right)^+ (1-y_{ij}), & \text{if } r_i^{\max} + \sum_{\ell \in J_i} p_\ell \leq z_{i+1} \\ \tau_i^* \left(1 - \sum_{j \in J_i} (1-y_{ij}) \right), & \text{otherwise} \end{cases}$$

where the bound on $\hat{\tau}_i$ is included for all i for which $\tau_i^* > 0$. Here τ_i^* is the minimum tardiness in subproblem i, $r_i^{\max} = \max\{\max\{r_j \mid j \in J_i\}, z_i\}$, and $\alpha^+ = \max\{0, \alpha\}$.

6 Problem Generation and Computational Results

Random instances are generated as follows. For each job j, r_j, $d_j - r_j$, and p_j are uniformly distributed on the intervals $[0,\ \alpha R]$, $[\gamma_1 \alpha R,\ \gamma_2 \alpha R]$, and $[0,\ \beta(d_j - r_j)]$, respectively. We set $R = 40\ m$ for tardiness problems, and otherwise $R = 100\ m$, where m is the number of segments. For the feasibility problem we adjusted β to provide a mix of feasible and infeasible instances. For the remaining problems, we adjusted β to the largest value for which most of the instances are feasible.

We formulated and solved the instances with IBM's OPL Studio 6.1, which invokes the ILOG CP Optimizer for CP models and CPLEX for MILP models. The MILP models are discrete-time formulations we have found to be most effective for this type of problem. We used OPL's script language to implement the Benders method.

Table 1 shows the advantage of logic-based Benders as the problem scales up. Benders failed to solve only four instances, due to inability to solve the CP subproblems.

Table 1. Computation times in seconds (computation terminated after 600 seconds). The number of segments is 10% the number of jobs. Tight time windows have $(\gamma_1, \gamma_2, \alpha) = (1/2, 1, 1/2)$ and wide time windows have $(\gamma_1, \gamma_2, \alpha) = (1/4, 1, 1/2)$. For feasibility instances, $\beta = 0.028$ for tight windows and 0.035 for wide windows. For makespan instances, $\beta = 0.025$ for 130 or fewer jobs and 0.032 otherwise. For tardiness instances, $\beta = 0.05$.

	Tight time windows									Wide time windows								
	Feasibility			Makespan			Tardiness			Feasibility			Makespan			Tardiness		
Jobs	CP	MILP	Bndrs	CP	MILP	Bndrs	CP	MILP	Bndrs	CP	MILP	Bndrs	CP	MILP	Bndrs	CP	MILP	Bndrs
50	0.91	8.0	1.5	0.09	9.0	4.0	0.05	1.3	1.1	0.03	7.7	2.5	0.13	13	3.5	0.13	1.3	1.1
60	1.1	12	2.8	0.09	18	5.5	0.14	1.8	1.5	0.05	12	1.6	0.94	29	5.7	0.11	2.3	1.4
70	0.56	17	3.3	0.11	51	6.7	1.3	3.9	2.1	0.13	17	2.3	0.11	39	6.2	0.16	3.0	1.9
80	600	21	2.8	600	188	7.6	0.86	6.0	4.5	600	24	5.0	600	131	7.3	1.9	6.4	5.0
90	600	29	7.5	600	466	10	21	11	4.6	600	32	9.7	600	600	8.5	5.9	9.5	11
100	600	36	12	600	600	16	600	11	2.0	600	44	9.7	600	600	19	600	24	22
110	600	44	20	600	600	17	600	600	600	600	49	17	600	600	24	600	600	600
120	600	62	18	600	600	21	600	15	3.3	600	80	15	600	600	23	600	12	3.1
130	600	68	20	600	600	29	600	17	3.9	600	81	43	600	600	31	600	18	3.9
140	600	88	21	600	600	30	600	600	600	600	175	27	600	*	35	600	600	14
150	600	128	27	600	600	79				600	386	8.5	600	600	43			
160	600	408	82	600	600	34				600	174	5.2	600	600	53			
170	600	192	5.9	600	600	37				600	172	5.9	600	600	600			
180	600	600	6.6	600	*	8.0				600	251	6.5	600	600	56			
190	600	600	7.2	600	*	8.5				600	600	7.3	600	*	78			
200	600	600	8.0	600	*	85				600	600	8.2	600	*	434			

* MILP solver ran out of memory.

References

1. Harjunkoski, I., Grossmann, I.E.: Decomposition techniques for multistage scheduling problems using mixed-integer and constraint programming methods. Computers and Chemical Engineering 26, 1533–1552 (2002)
2. Hooker, J.N.: Logic-based benders decomposition. Technical report, CMU (1995)
3. Hooker, J.N.: Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction. Wiley, New York (2000)
4. Hooker, J.N.: A hybrid method for planning and scheduling. Constraints 10, 385–401 (2005)
5. Hooker, J.N.: An integrated method for planning and scheduling to minimize tardiness. Constraints 11, 139–157 (2006)
6. Hooker, J.N.: Planning and scheduling by logic-based benders decomposition. Operations Research 55, 588–602 (2007)
7. Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. Mathematical Programming 96, 33–60 (2003)
8. Hooker, J.N., Yan, H.: Logic circuit verification by Benders decomposition. In: Principles and Practice of Constraint Programming: The Newport Papers, pp. 267–288. MIT Press, Cambridge (1995)
9. Jain, V., Grossmann, I.E.: Algorithms for hybrid MILP/CP models for a class of optimization problems. INFORMS Journal on Computing 13(4), 258–276 (2001)
10. Maravelias, C.T., Grossmann, I.E.: A hybrid MILP/CP decomposition approach for the continuous time scheduling of multipurpose batch plants. Computers and Chemical Engineering 28, 1921–1949 (2004)
11. Timpe, C.: Solving planning and scheduling problems with combined integer and constraint programming. OR Spectrum 24, 431–448 (2002)

Integrated Maintenance Scheduling for Semiconductor Manufacturing

Andrew Davenport

IBM T. J. Watson Research Center, Yorktown Heights, NY, 10598, USA
davenport@us.ibm.com

1 Introduction

Tools in a manufacturing plant require regular maintenance over their lifespan (e.g. cleaning, calibration, safety checks) in order to keep them running smoothly. In capital intensive industries, such as semi-conductor manufacturing, the scheduling of maintenance operations on the tools used in production is a critical function. Maintenance operations can be expensive to perform, so we should only perform them when necessary. However if maintenance is delayed too long, tools may run sub-optimally or break down (thus requiring even more expensive unplanned, corrective maintenance). Furthermore a tool that is undergoing maintenance may be partly or wholly unavailable for (revenue generating) production operations.

We have developed a system to generate maintenance schedules for the IBM East Fishkill, New York 300mm semiconductor manufacturing plant. In the sections which follow, we give a description of the maintenance scheduling problem in semi-conductor manufacturing and discuss some of the challenges in solving it. We present a goal programming approach that incorporates both constraint programming and mixed-integer programming solution technologies. A system we have developed based on this approach is now in use within IBM.

2 Problem Description

In the semi-conductor manufacturing there are number of different types of maintenance:

1. Preventative maintenance: periodic maintenance recommended by manufacturer of tool, to be carried out at regular time intervals (e.g. every six months).
2. Trigger maintenance: required after a tool reaches a certain state. For example, a wafer count trigger is reached after a certain number of wafers have been processed on a tool.
3. Unplanned, corrective maintenance: in response to unforeseen tool breakdowns or sub-optimal functioning.

The purpose of maintenance scheduling is to generate a detailed schedule for preventative and trigger maintenance operations. For each maintenance operation, we are given a release date, a due date, a processing time, a tool or tool part (on which the maintenance is to be performed) and a demand for some constant number of technicians throughout the entire processing time to perform the maintenance.

The tools in a semi-conductor fab can be partitioned into a number of *toolsets*. A toolset is a set of tools of a certain type (e.g. lithography) manufactured by a certain vendor. Maintenance operations need to be performed by maintenance technicians who are certified to work on tools belonging to a particular toolset. In practice, we have observed that maintenance technicians are mostly certified to work on tools belonging only to a single toolset. For each toolset, we are given a timetable specifying the number of maintenance technicians available during each time period (shift). The availability of technicians is typically the bottleneck in maintenance scheduling.

2.1 Objectives

In practice feasible schedules satisfying release dates, due dates and capacity constraints on the availability of technicians are usually easy to generate. Resource contention is not the main challenge in generating good maintenance schedules. Rather, the challenge is in handling multiple, non-convex objectives. We describe these objectives in the following sections.

Resource leveling. The first objective that we consider relates to the utilization of the available technicians for a toolset over time. When the use of technicians is spread out over time, it is more likely that there will be some idle technicians available to carry out any unforeseen, unplanned maintenance. In time periods when many technicians are busy, there are fewer idle technicians, so any neccessary unplanned maintenance may disrupt planned maintenance or require contracting outside technicians. Unplanned maintenance can be very expensive and disruptive to production operations, so in general it is preferred that we “levelize” the use of maintenance technicians over time so that some technicians are always available to handle unplanned maintenance should it become necessary (as illustrated in Figure 1(right)). We formulate this requirement as an objective that we *minimize the number of technicians* that we utilize throughout the entire schedule in order to perform all of the pending maintenance operations (alternatively we minimize the *maximum* number of technicians that are active (performing maintenance) in any one time period)).

Minimizing disruption. The second objective relates to minimizing the disruption to production that occurs as a result of taking tools out of service in order to perform maintenance on them. Typically, tools in a semiconductor manufacturing fab process lots consisting of a number of silicon wafers. At any one time when a tool is busy processing a lot, there may be a number of wafers waiting in a queue to be processed by the tool. This number of wafers is referred

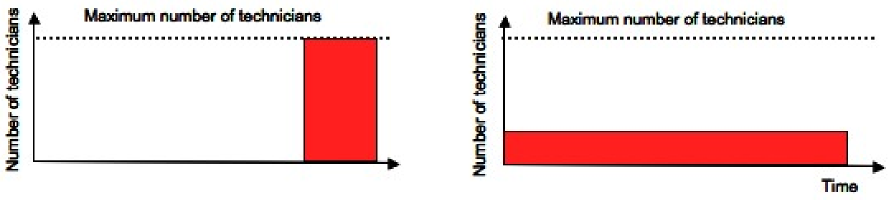


Fig. 1. Bad (left) and good (right) maintenance technician utilization

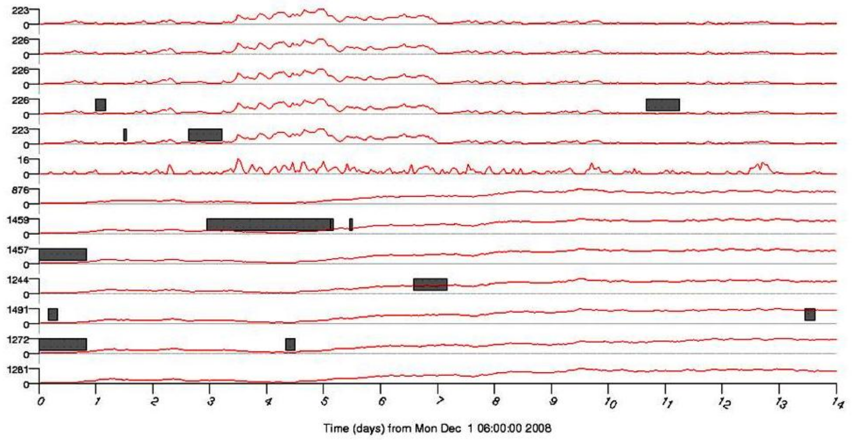


Fig. 2. Illustration of a maintenance schedule for a single toolset. Each row presents a Gantt chart representation of a schedule for a single tool. The plotted line represents the projected work in progress (y-axis) for the tool over each time period (x-axis) in the schedule.

to as the Work In Process (WIP), which is specified for a tool and a time period. When maintenance is performed on a tool, all production is stopped on that tool. As well as delaying wafers that need to be processed on the tool, this can lead to starvation of downstream processes for the wafers, resulting in tool under-utilization. Ideally, we would like to minimize such disruption by performing maintenance operations on tools during time periods when there is as little WIP as possible.

The difficulty we face with using this objective is that typically we do not know what the WIP levels will be in the fab for each tool over the scheduling horizon. Operations in semiconductor fab are usually very dynamic, as wafers can make multiple passes through various processes based on the results of tests of the effectiveness of each production step. Uncertainty also arises due to unplanned tool breakdowns. Detailed production scheduling is usually done using dispatch rules applied whenever a tool becomes available for processing. As such, there is no longer term production schedule we can refer to in order to determine what the WIP levels will be for each tool that we could use in a formulation of the maintenance scheduling problem. Instead we determine the Expected Work In Process will be for each tool and time period, based on a simulation of the flow of wafers through the fab. For this we use the IBM WIP Simulator, developed specifically for semi-conductor manufacturing (described in [1]). From the output of WIP simulator, we obtain the expected WIP level for each tool during each one hour time bucket.

Minimizing earliness and tardiness. The third objective relates to minimizing the long term costs of performing periodic maintenance. There is some flexibility in determining when a preventative maintenance operation j can be performed in the schedule. However the interval of time that can elapse between the completion time e_i of one operation i and the start time s_j of the following operation j (where each operation performs the same periodic maintenance) on a tool should not exceed a given duration $D_{i,j}$. This gives rise to earliness and tardiness costs. Tardiness costs result from scheduling an operation j to start at some time s_j such that $s_j > e_i + D_{i,j}$. Earliness costs arise from scheduling an operation j to start at some time s_j such that $s_j < e_i + D_{i,j}$. Over the long term we want to avoid scheduling periodic maintenance before it's due date, since otherwise this would lead to higher costs as a result of performing more preventative maintenance than is strictly necessary. The due date d_j of an operation j is calculated such $d_j = e_i + D_{i,j}$, i.e. scheduling an operation at it's due date incurs no earliness or tardiness cost. We are given an earliness penalty α_j and a tardiness penalty β_j for each operation j. Given an operation j with a completion time of C, the earliness / tardiness cost $et(j, C)$ of this operation is expressed as $et(j, C) = \max(\alpha_j(d_j - C), \beta_j(C - d_j))$.

3 Solution Approach

The solution approach we developed is motivated by the following observations, based on typical problem data: (a) generating a feasible maintenance schedule is very easy: the main resource bottleneck is the availability of maintenance technicians, and (b) generating an optimal maintenance schedule is difficult, given the multiple, irregular objectives. Since we wish to find an optimal solution, we focus on using exact methods. Constraint programming [2] and mixed integer programming with time-indexed formulations are often the solution techniques of choice for modelling scheduling problems with complex objectives and side constraints. These techniques have different strengths and weaknesses. Constraint programming can model scheduling problems compactly using an event-based formulation, and can be very successful at finding good solutions to problems which are highly resource-constrained. Constraint programming may not be the best choice for solving scheduling problems with irregular objectives. Mixed-integer programming, using time-indexed formulations, can represent scheduling problems with irregular objectives. However the formulations can be very large, since the number of decision variables is dependent on the length of the time horizon. As such, this approach is often limited to small problems.

We rank the objectives, considering them in the following order of decreasing importance: (a) resource levelling, (b) disruption (minimize the overlap of maintenance operations with work in progress) and (c) earliness-tardiness costs. We use a solution approach inspired by lexicographic goal programming. In lexicographic goal programming, we first solve the problem with respect to the most important objective only (we ignore all other objectives). Let f_1 denote the objective value for the first objective in the solution to this problem. We now add a new *constraint* to the problem model, stating that the value for the first objective must be equal to f_1. We then solve the problem for the second objective only, but with the first objective now represented as a constraint in the model. Subsequently, we add a second constraint to the model based on the objective value found for the second objective. We continue this process until we have solved the problem for all objectives.

The objective for resource levelling can be solved very efficiently using constraint programming (in a few seconds). We determine the minimum number of technicians required for a schedule by solving a series of feasibility problems, where for each problem we set a different constraint on the maximum number of technicians available in each time period. We use binary search on the value we set for this constraint to determine the smallest number of technicians for which we can find a schedule. We solve for the objectives concerning disruption and earliness-tardiness using mixed-integer programming. We use a time-indexed formulation with some additional cuts. In practice, the time needed to solve the mixed integer programming formulation to optimality can be quite long, since the number of decision variables can be large (a toolset may have 100 operations to be scheduled over a 2 week horizon). We discretize time into 15 minute time buckets, giving us solve times on the order of 5-20 minutes (using CPLEX 11). While this is acceptable for start of the day scheduling, it does not allow us to use this approach within an interactive system for mixed-initiative scheduling.

4 Summary

We have presented a maintenance scheduling problem for a semi-conductor manufacturing facility. We have developed a goal programming approach combining both constraint programming and mixed-integer programming, which exploits the strengths of both solution techniques. The scheduling system we have developed based on this solution approach has been deployed within IBM.

References

1. Bagchi, S., Chen-Ritzo, C., Shikalgar, S., Toner, M.: A full-factory simulator as a daily decision-support tool for 300mm wafer fabrication productivity. In: Mason, S., Hill, R., Moench, L., Rose, O. (eds.) Proceedings of the 2008 Winter Simulation Conference (2008)
2. Baptiste, P., Pape, C.L., Nuijten, W.: Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems. International Series in Operations Research and Management Science. Springer, Heidelberg (2001)

A Constraint Programming Approach for the Service Consolidation Problem

Kanika Dhyani[1], Stefano Gualandi[2], and Paolo Cremonesi[2]

[1] Neptuny s.r.l., via Durando 10, Milan, Italy
`kanika.dhyani@neptuny.com`
[2] Politecnico di Milano, Dipartimento di Elettronica e Informazione
{`gualandi,cremonesi`}`@elet.polimi.it`

Abstract. In this paper, we present a constraint programming approach for the service consolidation problem that is being currently tackled by Neptuny, Milan. The problem is defined as: Given a data-center, a set of servers with a priori fixed costs, a set of services or applications with hourly resource utilizations, find an allocation of applications to servers while minimizing the data-center costs and satisfying constraints on the resource utilizations for each hour of the day profile and on rule-based constraints defined between services and servers and amongst different services. The service consolidation problem can be modelled as an Integer Linear Programming problem with 0–1 variables, however it is extremely difficult to handle large sized instances and the rule-based constraints. So a constraint programming approach using the Comet programming language is developed to assess the impact of the rule-based constraints in reducing the problem search space and to improve the solution quality and scalability. Computational results for realistic consolidation scenarios are presented, showing that the proposed approach is indeed promising.

1 Introduction

As the complexity of IT infrastructures increases due to mergers or acquisitions, new challenging problems arise in the design and management of the resulting computer systems. Large-scale data-centers are often costly, non-flexible, yielding under-utilized servers and energy wasting. To reduce conflicts among the offered services, many enterprise data-centers host most of their services on dedicated servers without taking into account the possibility of deploying multiple services into a single server. Therefore, many servers are not used at their maximum capabilities and, in turn, expensive hardware investments are often required. Nowadays companies search for IT solutions able to significantly drop data-centers costs, e.g., energy consumption, space costs, and obtain a flexible system satisfying customer demands.

In this framework, the *consolidation* of data-center resources is a current solution adopted by many industries. The objective of a consolidation problem is to reduce the complexity of a data-center while guaranteeing some performance and availability requirements. This is usually achieved by searching for the best mapping between software applications and servers which minimizes data-center costs, hence the name service consolidation problem (SCP).

The SCP is an allocation problem which is NP-hard (see [6]) and which has been extensively tackled using branch-and-bound, simulated annealing, graph theory techniques, clustering, neural networks etc. Previous work on the SCP is based on a dynamic approach taking into account the workloads seasonal behaviour to estimate the server demands in [8]. A similar approach for virtualized systems in [3] is solved with a multidimensional bin-packing approximate algorithm. Nonlinear optimization problems are presented in [1] and optimization models based on queueing networks theory solved with a linear relaxation based heuristic is given in [2].

However, these works incorporate only constraints based on the computational capacity of the target systems, but in reality consolidation scenarios require additional constraints derived from compatibility, availability, performance or support needs. Compatibility constraints require virtual machines to run only on specific type of targets hardware systems (e.g., Intel virtual machines can run only on Intel-based physical systems) and availability constraints require two (or more) virtual machines to run on two distinct physical systems and support needs may require to have a set of virtual machines running on the same pool of physical systems (e.g., in order to simplify support activities). We call these constraints as rule-based constraints. These rules, along with a large number of binary variables in the problem make the problem computationally difficult to handle. Further, quite often this problem turns out to be infeasible either due to the lack of resources on servers to consolidate all the applications or due to the rule-based constraints.

To the best of our knowledge, there are no constraint programming (CP) models for the SCP. A recent work developed for Eventually-Serializable Data Services in [7] defines a CP model, with some rules similar to our rule-based constraints however without any capacity constraints.

2 Problem Formulation

The data-center consists of a set of m servers S with costs c, each characterized by a set R of resources – usually the available cpu, memory, disk and network bandwidth – denoted by u_{lj}, $l \in R, j \in S$, a set C of n candidates which can be applications or services with known requirements of each resource in $l \in R$, for each hour $t \in [1, 24]$ in the day, denoted by r_{lit}, $i \in C$. Additional, rule-based constraints are also known between candidates and/or between candidates and servers. The aim of the service consolidation problem is to find an allocation of candidates on the servers which minimizes the total cost of the data-center (i.e., the cost of all the servers needed for consolidation) while respecting resource constraints of each candidate placed on a server for each hour of the day and the rule-based compatibility constraints.

Let y_j be a 0–1 variable, with $y_j = 1$ if server j is active; else it is 0. Let x_{ij} be an assignment variable, with $x_{ij} = 1$, if candidate i is assigned to server j;

else it is 0. The cost is taken to be $c_j = CPU_j + MEM_j + 1$, where CPU_j and MEM_j are the available cpu and memory of server j. The additional one term in the cost function forces the selection of a costlier server (i.e, a more powerful one) instead of favouring too many cheap (i.e., low power) servers. The Integer Linear Programming (ILP) model is given as:

$$\begin{aligned}
\min \quad & \sum_{j \in S} c_j y_j & & (1)\\
\text{s.t.} \quad & \sum_{i \in C} r_{lit} x_{ij} \leq u_{rj} y_j, & \forall l \in R, j \in S, t \in T, \quad & (2)\\
& \sum_{j \in S} x_{ij} = 1, & \forall i \in C, \quad & (3)\\
& x_{ij} \leq y_j, & \forall i \in C, j \in S, \quad & (4)\\
& + \text{ rule-based constraints} & & \\
& y_j \in \{0,1\}, & \forall j \in S, & \\
& x_{ij} \in \{0,1\}, & \forall i \in C, j \in S. &
\end{aligned}$$

Constraints (2) ensures satisfaction of the resource constraints for each candidate for each hour of the day profile to the server that it is assigned to. Constraints (3) force each application to be assigned to just one server. The activation constraint (4) forces that a candidate can only be assigned to a server that is activated.

To define the rule-based constraints, let $C_1, C_2 \subset C$ and $S_1 \subset S$ be the sets over which these constraints are defined, then we have the following:

– **Candidate-candidate REQUIRE (CCR) rule** states that candidates in C_1 should be placed on the same server with candidates in C_2:

$$\forall j \in S, i_1 \in C_1, i_2 \in C_2, \quad x_{i_1 j} = x_{i_2 j}. \tag{5}$$

– **Candidates-candidate EXCLUSION (CCE) rule** states that the candidates in C_1 should not be placed on the same server with candidates in C_2:

$$\forall j \in S, i_1 \in C_1, i_2 \in C_2, \quad (x_{i_1 j} = 1) \Rightarrow x_{i_2 j} = 0. \tag{6}$$

– **Candidate-candidate REQUIRE AT LEAST ONE (CCRAO) rule** states that candidates in C_1 requires at least one candidate in C_2 to be placed on the same server as it:

$$\forall j \in S, i_1 \in C_1, \quad x_{i_1 j} \Rightarrow \sum_{i_2 \in C_2} x_{i_2 j} \geq 1. \tag{7}$$

– **Candidate-target REQUIRE rule (CTR)** states that candidates in C_1 should be placed on sever S_1:

$$\forall i_1 \in C_1, \quad \sum_{j \in S_1} x_{i_1 j} = 1. \tag{8}$$

– **Candidate-target EXCLUSION (CTE) rule** states that candidates in C_1 should not be placed on sever S_1:

$$\forall j \in S_1, r_1 \in C_1, \quad x_{i_1 i} = 0. \tag{9}$$

The CTR-rule provides a strict lower bound on the number of servers needed for consolidation. Numerically, it is seen that running the above model with an ILP solver that the problem becomes tougher with increase in the size of the system and in particular depends on the number of servers.

For the lack of space, we just sketch the CP model used in our application. For each candidate there is an integer variable X_i with domain equal to the set of available servers S. For each server j there is a 0–1 variable Y_j indicating whether the corresponding server is used or not. By using the same parameters as in the ILP model, the linear knapsack constraints (2) are translated into a set of multi-knapsack constraints (using the Comet syntax):

```
forall(t in T, l in R)
  cp.post(multiknapsack(X, all(i in C) r[i,l,t], all(j in S) u[j,l]));
```

The rule-based constraints (5)–(9) are easily translated into logical and reified constraints on the integer X_i variables.

3 Computational Experiments

We report experimental results for the ILP and CP approaches that are both encoded using Comet [4]. As ILP solver we have used the version of SCIP (ver. 1.1) delivered with the Comet (ver. 2.0).

We have considered different scenarios, and we report the results on the most challenging instances, which have $|S|$=20 and $|S|$=30 servers, a number of candidates ranging from $|C|$=100 to $|C|$=250, and four resources. Each instance is

Table 1. Challenging instances: averaged objective values after 60 sec. and 1000 sec. (averaged over 5 instances for each row). The symbol '-' means that no solution was found.

		SCIP		CP (Comet)	
$\|S\|$	$\|C\|$	60 sec.	1000 sec.	60 sec.	1000 sec.
20	100	-	40	100	80
	150	-	40	100	100
	200	-	-	290	280
	250	-	-	320	300
30	100	-	-	110	80
	150	-	-	245	200
	200	-	-	310	300
	250	-	-	525	500

involved in at least two rule-based constraints, making the whole problem demanding for the ILP solver. Table 1 reports the cost and the computation time with both the ILP and CP solver averaged over 5 different instances of the same dimension (for a total of 40 instances). For both methods, we set two time limits, the first at 60 sec. and the second to 1000 sec. Note that the ILP solver is never able to produce an admissible solution within 60 sec, while CP does. Things are only slightly different for the results after 1000 sec. In many cases the ILP solver does not even find a feasible solution, but when it does, the solutions are of very good quality.

4 Conclusion

We presented the Service Consolidation Problem solved at Neptuny, using both an ILP and a CP approach. The ILP approach is very effective when the number of rule-based constraints is limited. However, when the rule-based constraints are many, as it is the case in our real-life applications, the ILP approach fails in finding a feasible solution in a short time (where short is defined in terms of usability within an application). The CP approach is very effective in finding feasible solutions, and even for the largest instances always finds a solution within 60 seconds. In addition, for other tests not reported here, it is also very fast in detecting infeasibility. As future work, there are two open issues: the first issue concerns the use of explanation techniques (like QuickXplain [5]) to be able to obtain a set of conflicting constraints to show to the final user, and the second issue is to improve the efficiency of the solver in order to tackle even larger instances.

References

1. Almeida, J., Almeida, V., Ardagna, D., Francalanci, C., Trubian, M.: Resource management in the autonomic service-oriented architecture. In: Proc. of International Conference on Autonomic Computing, pp. 84–92 (2006)
2. Anselmi, J., Amaldi, E., Cremonesi, P.: Service consolidation with end-to-end response time constraints. In: Proc. of Euromicro Conference Software Engineering and Advanced Applications, pp. 345–352. IEEE Computer Society, Los Alamitos (2008)
3. Bichler, M., Setzer, T., Speitkamp, B.: Capacity planning for virtualized servers. In: Proc. of the Workshop on Information Technologies and Systems (2006)
4. DYNADEC. Comet release 2.0 (2009), http://www.dynadec.com
5. Junker, U.: Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. In: Proc. of Workshop on Modelling and Solving Problems with Constraints, Seattle, WA, USA (2001)
6. Lawler, E.: Recent results in the theory of machine scheduling. In: Bachem, A., Grotschel, M., Korte, B. (eds.) Mathematical Programming: The State of the Art. Springer, Heidelberg (1983)
7. Michel, L., Shvartsman, A., Sonderegger, E., Van Hentenryck, P.: Optimal Deployment of Eventually-Serializable Data Services. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 188–202. Springer, Heidelberg (2008)
8. Rolia, J., Andrzejak, A., Arlitt, M.F.: Automating enterprise application placement in resource utilities. In: Brunner, M., Keller, A. (eds.) DSOM 2003. LNCS, vol. 2867, pp. 118–129. Springer, Heidelberg (2003)

Solving Connected Subgraph Problems in Wildlife Conservation

Bistra Dilkina and Carla P. Gomes

Department of Computer Science, Cornell University, Ithaca, NY 14853, U.S.A.
{bistra,gomes}@cs.cornell.edu

Abstract. We investigate mathematical formulations and solution techniques for a variant of the Connected Subgraph Problem. Given a connected graph with costs and profits associated with the nodes, the goal is to find a connected subgraph that contains a subset of distinguished vertices. In this work we focus on the budget-constrained version, where we maximize the total profit of the nodes in the subgraph subject to a budget constraint on the total cost. We propose several mixed-integer formulations for enforcing the subgraph connectivity requirement, which plays a key role in the combinatorial structure of the problem. We show that a new formulation based on subtour elimination constraints is more effective at capturing the combinatorial structure of the problem, providing significant advantages over the previously considered encoding which was based on a single commodity flow. We test our formulations on synthetic instances as well as on real-world instances of an important problem in environmental conservation concerning the design of wildlife corridors. Our encoding results in a much tighter LP relaxation, and more importantly, it results in finding better integer feasible solutions as well as much better upper bounds on the objective (often proving optimality or within less than 1% of optimality), both when considering the synthetic instances as well as the real-world wildlife corridor instances.

1 Introduction

A large class of decision and optimization problems can be captured as finding a *connected subgraph* of a larger graph satisfying certain cost and revenue requirements. In different realizations of the Connection Subgraph Problem costs and profits are associated with either edges, nodes or both. Examples of this family of problems are the Minimum Steiner Tree, Maximum-Weighted Connected Subgraph and Point-to-Point Connection Problem. Such problems arise in a large number of applications – e.g. network design, system biology, social networks and facility location planning.

Here, we are concerned with a variant of the Connected Subgraph Problem where we are given a graph with costs and profits associated with nodes and one or more designated nodes called terminals and we seek to find a connected subgraph that includes the terminals with maximal profit and total cost within a specified budget which we refer to as the *Budget-Constrained Steiner Connected*

Subgraph Problem with Node Profits and Node Costs. This problem is known to be NP-hard even for the case of no terminals [2]. Removing the connectivity constraint, we have a 0-1 knapsack problem. On the other hand, the connectivity constraint relates it to other important classes of well-studied problems such as the Traveling Salesman Problem and the Steiner Tree problem. The connectivity constraint plays a key role in the combinatorics of this problem and we propose new mathematical formulations to better capture the structure of the problem w.r.t. the connectivity constraint.

Our work is motivated by an important instance of this problem that arises in Conservation Planning. The general problem consists of selecting a set of land parcels for conservation to ensure species viability. This problem is also known in the literature in its different variants as site selection, reserve network design, and corridor design. Biologists have highlighted the importance of addressing the negative ecological impacts of habitat fragmentation when selecting parcels for conservation. To this effect, ways to increase the spatial coherence among the set of parcels selected for conservation have been investigated (see [14] for a review). We look at the problem of designing so-called wildlife corridors to connect areas of biological significance (e.g. established reserves). Wildlife corridors are an important conservation method in that they increase the genetic diversity and allow for greater mobility (and hence better response to predation and stochastic events such as fire, as well as long term climate change). Specifically, in the wildlife corridor design problem, we are given a set of land parcels, a set of reserves (land parcels that correspond to biologically significant areas), and the cost (e.g. land value) and utility (e.g. habitat suitability) of each parcel. The goal is to select a subset of the parcels that forms a connected network including all reserves. This problem is clearly an instance of the Connected Subgraph Problem with node profits and node costs, where the nodes correspond to parcels, the terminal nodes correspond to the reserves and the edges correspond to adjacency of parcels. Conservation and land use planners generally operate with a limited budget while striving to secure the land that results in the corridor with best habitat suitability. This results in the budget-constrained version of the connected subgraph problem.

The connected subgraph problem in the context of designing wildlife corridors was recently studied in [2, 7]. Conrad et al. [2] designate one of the terminals as a root node and encode the connectivity constraints as a single commodity flow from the root to the selected nodes in the subgraph. This encoding is small and easy to enforce. They present computational results which show an easy-hard-easy runtime pattern with respect to the allowed budget on a benchmark of synthetic instances [7]. Further, when solving large scale real world instances of this optimization problem, the authors report extremely large running time. Here, we try to improve the state-of-the-art for this problem by proposing alternative formulations. We show that the easy-hard-easy pattern in runtime solution for finding optimal solutions observed for synthetic instances aligns with a similar pattern in the relative integrality gap of the LP relaxation of the model. This observation suggests that formulations that have tighter LP relaxations might also lead to faster solution times for finding optimal solutions. To this effect, we propose two additional formulations.

One possible alternative which we explore in this paper is to establish the connectivity of each selected node to the root node by a separate commodity flow. This results in a multi-commodity flow encoding of the connectivity constraints. Although the multi-commodity flow encoding is larger than the single commodity encoding (yet still polynomial size), it can result in a stronger LP relaxation of the problem.

A completely different avenue is to adapt ideas from the vast literature on the Steiner Tree Problem. Encodings of the connectivity requirement with respect to edge decisions successfully applied to the Steiner Tree problem involve exponential number of constraints. The Steiner Tree variants involve costs and/or profits on edges and hence such models explicitly model binary decisions of including or excluding edges from the selected subgraph. In particular, for the Steiner Tree Problem with Node Revenues and Budgets, Costa et al. [4] suggest using the directed Dantzig-Fulkerson-Johnson formulation [5] with subtour elimination constraints enforcing the tree structure of the selected subgraph. For variants of the Connection Subgraph Problem that involve edge costs or edge profits one needs to model explicitly decisions about inclusion of edges in the selected subgraph. Given a graph $G = (V, E)$, in the problem variant we study we only need to make explicit decisions of which nodes to include (i.e., $V' \subseteq V$) and connectivity needs to be satisfied on the induced subgraph $G(V')$ that only contains edges of G whose endpoints belong to V'. Nevertheless, we adapt the directed Dantzig-Fulkerson-Johnson formulation to our problem, therefore considering the graph edges as decision variables instead of the nodes, which in general results in dramatically increasing the search space size from $2^{|V|}$ to $2^{|E|}$. Although at first glance this change seems counterproductive, the added strength that results from explicitly enforcing the connectivity of each selected node to a predefined terminal, in fact, results in a tighter formulation. This formulation involves an exponential number of connectivity constraints that cannot be represented explicitly for real life sized instances. To address this, we present a Bender's decomposition approach that iteratively adds connectivity constraints to a relaxed master problem [1, 12].

We provide computational results on the three different encodings of the connectivity constraints: 1) the single-commodity flow (SCF) encoding [2]; 2) a multi-commodity flow (MCF) encoding; 3) a modified directed Dantzig-Fulkerson-Johnson (DFJ) formulation using node costs. On a benchmark of synthetic instances consisting of grid graphs with random costs and revenues, we show that indeed the multi-commodity encoding provides better LP relaxation bounds than the single commodity flow, and that the directed Dantzig-Fulkerson-Johnson formulation provides the best bounds. Most importantly, the advantage of the bounds provided by the directed Dantzig-Fulkerson-Johnson formulation over the single-commodity flow encoding are greatest exactly in the hard region. The tighter bounds turn out to have a critical effect on the solution times for finding optimal integer feasible solutions. Despite the large size of the DFJ encoding, it works remarkably well for finding integer feasible solutions. The easy-hard-easy pattern with respect to the budget exhibited strongly by the SCF encoding is much less pronounced when using the DFJ encoding – this encoding is considerably more robust to the budget level. We show that the DFJ encoding finds optimal solutions two orders of magnitude faster than the SCF encoding in the interval of budget values that are hardest. This result is particularly relevant when solving real-world instances because the hard region usually falls over a budget interval close to the minimum cost solution to find a connected subgraph – i.e. it helps find solutions for tight budgets.

We test our formulations on real problem instances concerning the design of a Grizzly Bear Wildlife Corridor connecting three existing reserves [2]. We show that, for critically constrained budgets, the DFJ encoding proposed here can find optimal or close to optimal solutions, dramatically speeding up runtime. For the same problem instances and budget levels, the single flow encoding can only find considerably worse feasible solutions and has much worse objective upper bounds. For example, for a budget level which is 10% above the minimum cost required to connect all reserves, the DFJ encoding finds an optimal soltuion and proves optimality in 25 mins, while the SCF encoding after 10 hours has found an inferior solution and has proven an optimality gap of 31%. Similar behavior is observed for a budget of 20% above the minimum cost. Working budgets close to the minimum cost solution is a very likely scenario in a resource-constrained setting such as conservation planning. Hence, with the little money available, it is important to find the best possible solutions. The new DFJ encoding proposed here allows us to find optimal solutions to large scale wildlife corridor problems in exactly the budget levels that are most relevant in practice and that are out of reach in terms of computational time for the previously proposed formulations.

The DFJ encoding is better at capturing the combinatorial structure of the connectivity constraints which is reflected in the tightness of the LP relaxation as well as in the fact that it finds integer feasible solutions much faster and with very strong guarantees in terms of optimality (often proving optimality or within less than 1% of optimality), both when considering the synthetic instances as well as the real-world wildlife corridor instances.

2 Related Work

One of the most studied variant of the Connected Subgraph Problem is perhaps the Steiner Tree which involves a graph $G = (V, E)$, a set of terminal vertices $T \subset V$, and costs associated with edges. In the Minimum Steiner Tree Problem the goal is to select a subgraph $G' = (V' \subseteq V, E' \subseteq E)$ of the smallest cost possible that is a tree and contains all terminals ($T \subseteq V'$). Although including a budget constraint has important practical motivation, budget-constrained variants of the Steiner tree problem are not as nearly widely studied as the minimum Steiner tree or the prize-collecting variant. The variant that is more relevant here is the Budget Prize Collecting Tree Problem where in addition to costs associated with edges, there are also revenues associated with nodes. The goal is to select a Steiner tree with total edge cost satisfying a budget constraint while maximizing the total node revenue of the selected tree. Levin [10] gives a $(4+\epsilon)$-approximation scheme for this problem. Costa et al. [3, 4] study mathematical formulations and solution techniques for this problem in the presence of additional so-called hop constraints. They use a directed rooted tree encoding with an exponential number of connectivity constraints and a Branch-and-Cut solution technique. One can easily see that the Budget Prize Collecting Tree Problem is a special case of the Budget-Constrained Steiner Connected Subgraph with Node Profits and Node Costs by replacing each edge with an artificial node with the corresponding cost and adding edges to the endpoints of the original edge. We adapt some of the vast amount of work on tight formulations for the variants of the Steiner Tree problem with edge costs to the more general node-weighted problem.

Restricted variants of Budget-Constrained Steiner Connected Subgraph Problem with Node Profits and Node Costs have been addressed previously in the literature. Lee and Dooly [9] study the Maximum-weight Connected Subgraph Problem where profits and unit costs are associated with nodes and the goal is to find a connected subgraph of maximal weight and at most a specified R number of nodes. In the constrained variant they consider a designated root node that needs to be included in the selected subgraph.

Moss and Rabani [13] also study the connected subgraph problem with node costs and node profits and refer to this problem as the *Constrained Node Weighted Steiner Tree Problem*. They also only consider the special case where there is either no terminals or only one terminal - a specified root node. For all three optimization variants - the budget, quota and prize-collecting, Moss and Rabani [13] provide an approximation guarantee of $O(\log n)$, where n is the number of nodes in the graph. However, for the budget variant, the result is a bi-criteria approximation, i.e. the cost of the selected nodes can exceed the budget by some fraction. Finding an approximation algorithm for the budget-constrained variant is still an open question, as well as dealing with multiple terminals. Demaine et al. [6] have recently shown that one can improve the $O(\log n)$ approximation guarantee to a constant factor guarantee when restricting the class of graphs to planar but only in the case of the minimum cost Steiner Tree Problem with costs on nodes (but no profits). It is an open research question whether for planar graphs one can design a better approximation scheme for the budget-constrained variant. This is of particular interest because the Wildlife Corridor Design problem corresponds to finding a connected subgraph in a planar graph.

3 Mathematical Formulations

The Connected Subgraph Problem with Node Profits and Node Costs is specified by a connected graph $G = (V, E)$ along with a set of terminal nodes $T \in V$, a cost function on nodes $c : V \rightarrow \mathcal{R}$, and a profit function on nodes $u : V \rightarrow \mathcal{R}$. The goal is to select a subset of the nodes $V' \subseteq V$ such that all terminal nodes T are included $(T \subseteq V')$ and the induced subgraph $G(V')$ is connected. In the Budget-Constrained variant, given a budget C we seek to find a connected subgraph such that the total cost of the nodes in V' do not exceed the budget C, while maximizing the total profit of the selected nodes.

In the following formulations, for each vertex $i \in V$, we introduce a binary variable x_i, representing whether or not i is included in the connected subgraph. Then, the objective function, the budget constraint and the terminal inclusion constraint are stated as:

$$\text{maximize} \sum_{i \in V} u_i x_i, \tag{1}$$

$$\text{s.t.} \sum_{i \in V} c_i x_i \leq C \tag{2}$$

$$x_t = 1, \qquad \forall t \in T \tag{3}$$

$$x_i \in \{0,1\}, \qquad \forall i \in V \tag{4}$$

In the following subsections we outline three different ways of enforcing the connectivity constraints — the selected vertices should induce a connected subgraph of the original graph G.

3.1 Connectivity as Single Commodity Flow

Conrad et al. [2], Gomes et al. [7] use a single-commodity network flow encoding where each undirected edge $\{i,j\} \in E$ is replaced by two directed edges (i,j) and (j,i). Let us call the set of directed edges A. They introduce a source vertex 0, with maximum total outgoing flow $n = |V|$. One arbitrary terminal vertex is chosen as root $r \in T$, and a directed edge $(0,r)$ is defined to insert the flow into the network. Each selected node acts as a "sink" by consuming one unit of flow, and a node can be selected only if it has positive incoming flow. Connectivity of the selected nodes is ensured by enforcing flow conservation constraints at all nodes.

More formally, for each (directed) edge $(i,j) \in A$, there is a non-negative variable y_{ij} to indicate the amount of flow from i to j and the following constraints are enforced:

$$x_0 + y_{0r} = n \tag{5}$$

$$y_{ij} \leq n x_j, \qquad \forall (i,j) \in A \tag{6}$$

$$\sum_{i:(i,j) \in A} y_{ij} = x_j + \sum_{i:(j,i) \in A} y_{ji}, \qquad \forall j \in V \tag{7}$$

$$\sum_{j \in V} x_j = y_{0r} \tag{8}$$

$$y_{ij} \geq 0, \qquad \forall (i,j) \in A \cup (0,r) \tag{9}$$

$$x_0 \geq 0, \qquad \forall (i,j) \in A \tag{10}$$

For the source of the flow, they introduce a variable $x_0 \in [0,n]$, representing the eventual residual flow. Constraint (5) states that the residual flow plus the flow injected into the network corresponds to the total system flow. Each of the vertices with a positive incoming flow retains one unit of flow, i.e., $(y_{ij} > 0) \Rightarrow (x_j = 1), \forall (i,j) \in A$ enforced by Constraint (6). The flow conservation is modeled in Constraint (7). Finally, Constraint (8) enforces that the flow absorbed by the network corresponds to the flow injected into the system. This encoding requires $2|E| + 1$ additional continuous variables and ensures that all selected nodes form a connected component.

3.2 Connectivity as Multi-commodity Flow

In the first encoding we enforce the connectivity of all selected nodes though a single commodity flow. In this model, the key difference is that we enforce the connectivity of the selected set of nodes by associating a separate commodity with each node. There will be one unit of flow from the root to each selected node of its own "commodity" type. We arbitrarily select one of the terminals as a *root* node denoted $r \in T$. Each other node i is a potential sink of one unit of commodity flow of type i that will have to be routed from the root node to i. Let use denote the set of neighbors of a node i as $\delta(i) = \{j|(i,j) \in A\}$.

Similarly to the original model, we still have binary decision variable for each vertex and the objective, the budget constraint and the terminal inclusion constraint are defined as before.

For each (directed) edge $(i,j) \in A$ and each node k different from the *root node* r, we introduce a variable $y_{kij} \geq 0$ which when it is nonzero indicates that the edge carries flow of type k. If a node k is selected then in becomes an active sink for flow of type k.

$$\sum_{j:r\in\delta(j)} y_{kjr} = 0 \qquad \forall k \in V - r \tag{11}$$

$$\sum_{j\in\delta(k)} y_{kjk} = x_k \qquad \forall k \in V - r \tag{12}$$

$$\sum_{j\in\delta(k)} y_{kkj} = 0 \qquad \forall k \in V - r \tag{13}$$

$$\sum_{j\in\delta(i)} y_{kij} = \sum_{i\in\delta(j)} y_{kji} \qquad \forall k, \forall i \in V - r, i \neq k \tag{14}$$

$$y_{kij} \leq x_i \qquad \forall k, \forall i \in V - r, \forall j \in \delta(i) \tag{15}$$

$$y_{kij} \leq x_j \qquad \forall k, \forall i \in V - r, \forall j \in \delta(i) \tag{16}$$

$$y_{kij} \geq 0 \qquad \forall k \in V - r, \forall (i,j) \in A \tag{17}$$

For all nodes k, if node k is selected, then k is a sink for flow of commodity k (Constraint (12, 13)). Constraint (14) imposes conservation of flow for each commodity type k at each node different from the sink k and the source r, and

Constraint (11) imposes that the root does not have any incoming flow. Finally, the capacity of each edge is zero if either end node is not selected, and 1 otherwise (Constraint (15, 16)).

This encoding requires $(|V|-1)2|E|$ additional continuous variables – considerably more than the SCF encoding. However, we will see that enforcing the connectivity of each node to the root separately results in tighter LP relaxation.

3.3 Connectivity as Directed Steiner Tree

As suggested by the multi-commodity flow encoding, to enforce connectivity one may enforce that there exists a path from each selected node to the root node. In this third encoding, we in fact explicitly model the selection of edges as binary variables and insist that we select a set of nodes and edges such that there is a single path from each selected node to the root (using the selected nodes). In other words, we impose stronger constraints than necessary while preserving all feasible solutions in terms of subset of nodes that induce a connected subgraph. In effect, we enforce the connectivity constraints by adding constraints that ensure that we select edges that form a (Steiner) tree. Several studies on Steiner Tree problem variants have shown that often directed edge models are better than undirected ones in solving Steiner Tree problems (e.g. [11, 4]). Following these results, we adapt the directed Dantzig-Fulkerson-Johnson formulation of connectivity. We have a binary variable for each directed edge in A (Constraint (24)). We can avoid explicitly including binary variables x for each node, as these decisions can be inferred from the values of the edge binary variables. The set of selected nodes consists of the nodes that have exactly one incoming edge. Although the vertex variables are not explicitly represented, it will still be useful to refer to them. To this effect, given a solution vector over the edge variables $\mathbf{y}$, let us define an *associated vertex solution vector* $\mathbf{x}$ as $x_k = \sum_{k\in\delta(i)} y_{ik}$. Constraints (18) and (19) express the objective and the budget constraint in terms of edge variables. Constraint (20) enforces that each terminal node should have one incoming edge (i.e. it should be selected). To enforce the directed tree property, each non-root node is allowed to have at most one incoming edge (Constraint (21)). Connectivity is enforced through generalized subtour elimination constraints defined over edge variables (Constraints (23)). We also include Constraint (22) which strengthens the formulation by enforcing that each edge is used in at most one direction.

$$\max \sum_{i\in V}\left(u_i \sum_{j\in\delta(i)} y_{ji}\right) \tag{18}$$

$$\text{s.t.} \sum_{i\in V}\left(c_i \sum_{j\in\delta(i)} y_{ji}\right) \leq C \tag{19}$$

$$\sum_{j \in \delta(i)} y_{ji} = 1 \qquad \forall i \in T \tag{20}$$

$$\sum_{j \in \delta(i)} y_{ji} \leq 1 \qquad \forall i \in V - T \tag{21}$$

$$y_{ij} + y_{ji} \leq 1 \qquad \forall i \in V - T, \forall j \in \delta(i) - r \tag{22}$$

$$\sum_{(i,j) \in A | j \in S, i \in V \setminus S} y_{ij} \geq \sum_{j \in \delta(k)} y_{jk}, \qquad \forall S \subset V - r, \forall k \in S \; [cuts] \tag{23}$$

$$y_{ij} \in \{0,1\} \qquad \forall (i,j) \in A \tag{24}$$

Given the exponential number of connectivity Constraints (23), in the following section we describe a solution approach in which we relax these constraints in the context of cutting plane procedure and only add them as cuts when they become violated.

4 Solution Approaches

Conrad et al. [2], Gomes et al. [7] outline a preprocessing technique for the Budget-constrained Connected Subgraph problem which effectively reduces the problem size for tight budgets. The procedure computes all-pairs shortest paths in the graph and uses these distances to compute for each node the minimal Steiner Tree cost that covers all three terminals as well as the node under consideration. If this minimum cost exceeds the allowed budget, the node does not belong to any feasible solution and hence its variable is assigned to 0.

Gomes et al. [7] also outline a greedy method for finding feasible solutions to the Budget-constrained Connected Subgraph problem by first computing the minimum cost Steiner tree covering all the terminal nodes and then greedily adding additional nodes until the allowed budget is exhausted. They show that providing this greedy solution to their encoding of the Connected Subgraph Problem (the single commodity flow encoding) significantly improves performance.

We use both of these techniques. We apply the preprocessing step to all problem instances. In addition, we provide the greedy solution as a starting point to the SCF encoding.

Our approach to solving the DFJ encoding is based on a cutting plane or Bender's decomposition approach. We solve a relaxed "master" problem which omits the exponential number of connectivity constraints. In a first pass of this procedure all edge variables are relaxed from binary variables to continuous variables $\in [0,1]$. In this first phase, we solve a sequence of progressively tighter LP master problems and in effect this corresponds to a cutting plane approach. Once we find a (fractional) optimal solution to the LP master problem that does not violate any connectivity constraints, we have obtained the optimal solution to the LP relaxation of the DFJ formulation. If that solution is integral, then we have an optimal solution to the original problem. If the LP solution is not integral, we enforce the integrality constraints for all edge variables. We continue the same iteration steps where now the master problem includes the cuts learned during solving the LP relaxation as well as the integrality constraints. In the second phase, we need to solve a sequence of MIP master problems which is in effect a Bender's decomposition approach. At each iteration, the optimal solution to the MIP master might not be connected and more connectivity cuts would need to be added. Once we find an optimal MIP master solution, we have found an optimal integer solution to the original problem. The detailed algorithm is outlined below:

Master Algorithm:
0. (Initialize) Define the initial relaxation P_0 of the problem by Constraints (18, 19,20, 21, 22) as well as the integrality Constraint (24) relaxed to only enforce the bounds. Set iteration count $t = 0$.
1. (Master optimization) Solve P_t and obtain an optimal (edge) solution $\mathbf{y}_t$. Let the associated vertex solution be $\mathbf{x}_t$. If the associated vertex solution $\mathbf{x}_t$ is integral, go to Step 3, otherwise go to Step 4.
2. (Additional Check) Check the connectivity of the induced graph $G(\mathbf{x}_t)$. If it is connected, then $\mathbf{x}_t$ is optimal, and the algorithm returns solution $\mathbf{x}_t$. Otherwise, continue to Step 4.
3. (Master separation) Check if $\mathbf{y}_t$ satisfies all the connectivity constraints (23). If it does, go to Step 4. If a violated constraint is found, then add the corresponding cut to the master problem and let P_{t+1} be the problem obtained. Set $t = t + 1$ and return to Step 1.
4. (Optimality check) If the associated vertex solution $\mathbf{x}_t$ is integral, then $\mathbf{x}_t$ is optimal, and the algorithm returns solution $\mathbf{x}_t$. Otherwise, add the integrality constraints (24) back in to the problem, and let P_{t+1} be the problem obtained. Set $t = t + 1$ and go to Step 1.

Checking the exponential number of connectivity constraints (23) given an edge solution $\mathbf{y}_t$ in Step 3 is done through a polynomial time separation procedure. The separation procedure checks the connectivity of each selected vertex to the root and terminates as soon as it finds a disconnected node and infers a cut to be added. It first checks the connectivity of the terminals to the root and then other selected vertices. We solve a max-flow problem in the directed graph G'=(V,A) between the root and each node $k \in V - r$ selected in the proposed solution, i.e. in the associated vertex solution $x_t(k) > 1 - \epsilon$. The capacities of the edges are the current values of the edge variables $\mathbf{y}_t$ in the master solution. If the maximum flow is less than the sum of the incoming arcs from k, we have found a violated constraint. The dual variables of the max-flow subproblem indicate the partition of nodes $\{S, V\backslash S\}$ that define the minimum cut (let $r \in V\backslash S$).

Now, we can add the cut enforcing that at least one edge across the partition needs to be selected if parcel k is selected:

$$\sum_{(i,j)\in A | i \in V\backslash S, j \in S} y_{ij} \geq x_k \tag{25}$$

Step 2 of the algorithm is a special step that applies to the Connected Subgraph Problems with node costs and node profits. Given a solution $\mathbf{y}_{ij}$ of the DFJ formulation and the associated vertex vector $\mathbf{x}_t$ we can infer a set of selected nodes $V' = \{k \in V | x_k(t) = 1\}$. The original problem only requires that for the selected subset of vertices V' the induced graph $G(V')$ is connected, while the DFJ formulation poses a much stronger requirement to select a subset of edges forming a tree. Hence, it can be the case that that V' induces a connected subgraph in G, but the selected edges $E' = \{(i,j) \in A | y_{ij} = 1\}$ do not form a single connected component. To illustrate this, imagine that the selected edges E' form two vertex-disjoint cycles C_1 and C_2 and such that $u \in C_1$ and $v \in C_2$ and $u, v \in E$. The edge set E' clearly does form a connected subgraph, however the subgraph induced by the selected vertices is connected because of the edge u, v. Without Step 2, our separation procedure in Step 3 will infer a new cut and will wrongly conclude that the selected master solution is not a feasible solution. To avoid such cases, we introduce Step 2 to check the weaker connectivity in terms of the induced subgraph. If this connectivity check fails, then we use the max-flow separation procedure in Step 3 to infer a new connectivity cut to add to the master.

The solution procedure described above solves a series of tighter relaxation of the original problem and therefore the first solution that is feasible w.r.t. all the constraints in the original problem is in fact the optimal solution. One problem with this approach is that we need to wait until the very end to get one integer feasible solution which is also the optimal one. Ideally, one would like to have integer feasible solutions as soon as possible. We achieve this in the context of this solution technique by noticing that while solving the MIP master to optimality we discover a sequence of integer solutions. Some of these integer solutions might satisfy all connectivity constraints (i.e. they are feasible solutions to the original problem), but are discarded by the master as sub-optimal – there might be disconnected solutions to the master of better quality. To detect the discovery of feasible solutions to the original problem while solving the master problem, we introduce a connectivity check at each MIP master incumbent solution (not described in our algorithm outline above). If a MIP master incumbent is connected and is better than any other connected integer solution discovered so far, we record this solution as an incumbent to our original problem.

5 Experimental Results

We evaluate the strength of the LP relaxation of the three alternative encodings on a synthetically generated benchmark of instances [2]. We generate 100 instances of a 10 by 10 grid parcels (100 nodes) and 3 reserves (terminals) with uniformly sampled costs and utilities. We report median running times across the 100 instances where the budget is varied as percentage slack over the minimum cost solution for the particular instance. For example, given a minimum cost solution for connecting the terminal nodes of value c_{min}, 10% slack corresponds to budget $B = 110\% * c_{min}$. All computational experiments were performed using IBM ILOG CPLEX 11[8].

Figure 1 compares the relative gap between the optimal objective of the LP relaxation z^*_{LP} and the optimal objective of the problem z^*_{IP} at different budget levels given by $(z^*_{LP} - z^*_{IP})/z^*_{IP}$. One can see that the DFJ encoding indeed provides a relaxation which is much tighter than the relaxation of the single flow formulation of the problem. In particular, the smaller the budget is (up to some point), the bigger advantage the exponential formulation has. This added strength however is paid in computational time. The LP relaxation of the SCF model is solved really fast compared to the DFJ encoding. On the other had, the multi-commodity encoding does not dominate on either measure – it provides tighter bound on the optimal but not as tight as the DFJ formulation but at the same time takes a considerable computational time. In the rest of the experimental analysis, we concentrate on the single commodity flow and the DFJ encoding. The DFJ-style encodings in the context of Steiner tree problems are known to produce tight LP relaxations. Our results confirm this trend in

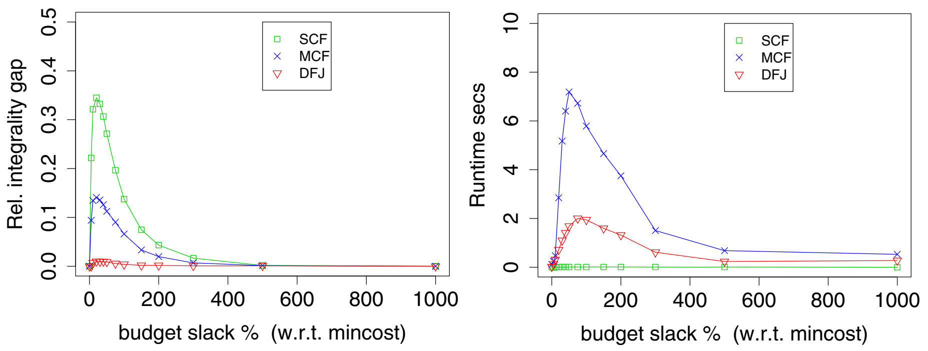


Fig. 1. Optimality gap and run times of LP relaxations of the three encodings on 10x10 lattices with 3 reserves, median over 100 runs

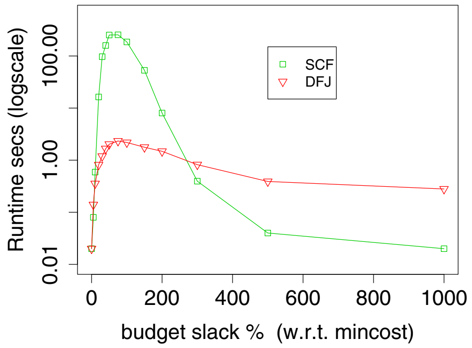


Fig. 2. Compare run times of DFJ and SCF for finding optimal integer solutions on 10x10 lattices with 3 reserves, median over 100 runs

the variant we are studying here. More importantly, one would like to use the strength of this encoding to find the optimal integer feasible solution.

Figure 2 compares the running time of the SCF encoding and the DFJ encoding. An easy-hard-easy pattern of the running time with respect to the budget was already observed by Gomes et al. [7]. Here, we clearly see that the DFJ encoding is in fact most beneficial in exactly the hard budget region. For large budgets, the DFJ encoding in fact has worst running time than the single commodity flow. However, more importantly it improves the running time in the hard region by 2 orders of magnitude.

We are interested in the running time performance of the SCF and the DFJ encoding when looking for integer feasible solutions. We evaluate the performance on a real-world Wildlife Corridor design problem attempting to connect three existing reserves. We tackle this problem at two different spatial scales. The coarser scale considers parcels grid cells of size 40 by 40 km and has 242 parcels (nodes). The finer spatial scale consider parcels of size 10 by 10 km and has 3299 parcels (nodes).

Figure 3 clearly demonstrates the advantage of the DFJ encoding on the 40km problem instance both in terms of the LP relaxation bound (left) and in terms of finding integer optimal solutions (right). The single flow encoding is fast for very tight and very large budgets, but for a critically constrained region the running time is much higher. The DFJ encoding on the other hand shows robust running times which do not vary much with the budget level.

We compare the running time to find integer solutions for the much larger instance at spatial resolution of 10 km. We set the budget at different (tight) levels as percent slack above the minimum cost required to connect the reserves. Table 1 presents solution quality, running times and optimality gap results for three different levels. For comparison, we also include the quality of the solution obtained by the greedy algorithm from [7] (which is usually much worse than the optimal). The results in Table 1 show that the DFJ encoding is much faster at finding optimal or near optimal solutions to the problem than the SCF encoding. Given a 8 hour cutoff time, for all three budget levels DFJ finds equal or better feasible solutions than SCF and also provides very tight optimality guarantee ($< 1\%$ in all cases). On the other hand, SCF in all three cases can only guarantee that the best solution it has founf is within at best 28% of optimality.

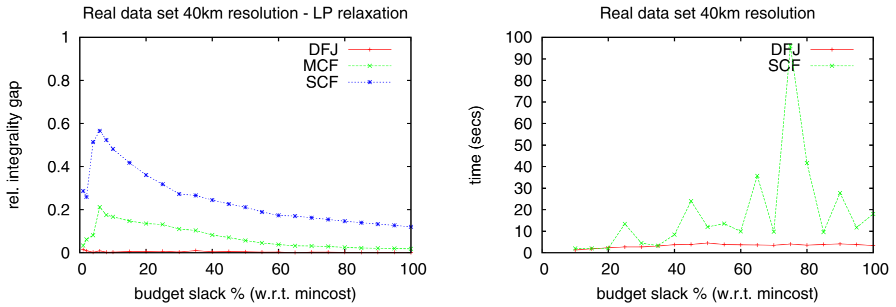


Fig. 3. Results on a Wildlife Corridor Problem at 40 km resolution

Table 1. The performance of the SCF and DFJ encoding on a large real world instance with an 8 hour cutoff time

budget slack	encoding	time	objective	opt. gap
10%	greedy	$<$ 2 mins	10691163	NA
109475	SCF	8 hrs	10877799	31.15%
	DFJ	25 mins	12107793	0.01%
20%	greedy	$<$ 2 mins	12497251	NA
119427	SCF	8 hrs	12911652	30.35%
	DFJ	2 hrs 25 mins	13640629	0.01%
30%	greedy	$<$ 2 mins	13581815	NA
129379	SCF	8 hrs	13776496	28.64%
	DFJ	7 hrs 35 mins	14703920	0.62%

6 Conclusion

The budget-constrained Connection Subgraph Problem is computationally challenging problem with a lot of real world applications. Capturing well the combinatorial structure of the connectivity constraint is critical to effectively solving large scale instances. In this work, we proposed a novel solution approach to this problem that uses an adapted directed Dantzig-Fulkerson-Johnson formulation with subtour elimination constraints in the context of a cut-generation approach. This results in significant speed up in run times when the budget level falls in the interval that results in most computationally challenging instances. We evaluate performance on a relatively large instance of the Wildlife Corridor Design Problem and find optimal solutions for different budget levels. This work is a good example of identifying and extending relevant Computer Science results for problems arising in the area of Computation Sustainability.

Acknowledgments

This research was supported by IISI, Cornell University (AFOSR grant FA9550-04-1-0151), NSF Expeditions in Computing award for Computational Sustainability (Grant 0832782) and NSF IIS award (Grant 0514429).

References

[1] Benders, J.: Partitioning procedures for solving mixed-variables programming problems. Numerische Mathematik 4, 238–252 (1962)

[2] Conrad, J., Gomes, C.P., van Hoeve, W.-J., Sabharwal, A., Suter, J.: Connections in networks: Hardness of feasibility versus optimality. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 16–28. Springer, Heidelberg (2007)

[3] Costa, A.M., Cordeau, J.-F., Laporte, G.: Steiner tree problems with profits. INFOR: Information Systems and Operational Research 4(2), 99–115 (2006)

[4] Costa, A.M., Cordeau, J.-F., Laporte, G.: Models and branch-and-cut algorithms for the steiner tree problem with revenues, budget and hop constraints. Networks 53(2), 141–159 (2009)
[5] Dantzig, G., Fulkerson, R., Johnson, S.: Solution of a Large-Scale Traveling-Salesman Problem. Operations Research 2(4), 393–410 (1954)
[6] Demaine, E.D., Hajiaghayi, M.T., Klein, P.: Node-weighted steiner tree and group steiner tree in planar graphs. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5555. Springer, Heidelberg (2009)
[7] Gomes, C.P., van Hoeve, W.-J., Sabharwal, A.: Connections in networks: A hybrid approach. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 303–307. Springer, Heidelberg (2008)
[8] ILOG, SA, CPLEX 11.0 Reference Manual (2007)
[9] Lee, H.F., Dooly, D.R.: Decomposition algorithms for the maximum-weight connected graph problem. Naval Research Logistics 45(8), 817–837 (1998)
[10] Levin, A.: A better approximation algorithm for the budget prize collecting tree problem. Operations Research Letters 32(4), 316–319 (2004)
[11] Ljubic, I., Weiskircher, R., Pferschy, U., Klau, G.W., Mutzel, P., Fischetti, M.: Solving the prize-collecting steiner tree problem to optimality. In: ALENEX/ANALCO, pp. 68–76 (2005)
[12] McDaniel, D., Devine, M.: A Modified Benders' Partitioning Algorithm for Mixed Integer Programming. Management Science 24(3), 312–319 (1977)
[13] Moss, A., Rabani, Y.: Approximation algorithms for constrained node weighted steiner tree problems. SIAM J. Comput. 37(2), 460–481 (2007)
[14] Williams, J.C., Snyder, S.A.: Restoring habitat corridors in fragmented landscapes using optimization and percolation models. Environmental Modeling and Assessment 10(3), 239–250 (2005)

Consistency Check for the Bin Packing Constraint Revisited

Julien Dupuis[1], Pierre Schaus[2], and Yves Deville[1]

[1] Department of Computer Science and Engineering, UCLouvain, Belgium
{julien.dupuis,yves.deville}@uclouvain.be
[2] Dynadec Europe, Belgium
pschaus@dynadec.com

1 Introduction

The bin packing problem (BP) consists in finding the minimum number of bins necessary to pack a set of items so that the total size of the items in each bin does not exceed the bin capacity C. The bin capacity is common for all the bins.

This problem can be solved in Constraint Programming (CP) by introducing one placement variable x_i for each item and one load variable l_j for each bin.

The $\texttt{Pack}([x_1, \ldots, x_n], [w_1, \ldots, w_n], [l_1, \ldots, l_m])$ constraint introduced by Shaw [1] links the placement variables $x_1, \ldots, x_n$ of n items having weights $w_1, \ldots, w_n$ with the load variables of m bins $l_1, \ldots, l_m$ with domains $\{0, \ldots, C\}$. More precisely the constraint ensures that $\forall j \in \{1, \ldots, m\} : l_j = \sum_{i=1}^{n}(x_i = j) \cdot w_i$ where $x_i = j$ is reified to 1 if the equality holds and to 0 otherwise. The Pack constraint was successfully used in several applications.

In addition to the decomposition constraints $\forall j \in \{1, \ldots, m\} : l_j = \sum_{i=1}^{n}(x_i = j) \cdot w_i$ and the redundant constraint $\sum_{i=1}^{n} w_i = \sum_{j=1}^{n} l_j$, Shaw introduced:

1. a filtering algorithm based on a knapsack reasoning inside each bin, and
2. a failure detection algorithm based on a reduction of the partial solution to a bin packing problem.

This work focuses on improvements of the failure detection algorithm.

2 Reductions to Bin Packing Problems

Shaw describes in [1] a fast failure detection procedure for the Pack constraint using a bin packing lower bound (BPLB). The idea is to reduce the current partial solution (i.e. where some items are already assigned to a bin) of the Pack constraint to a bin packing problem. Then a failure is detected if the BPLB is larger than the number of available bins m.

We propose two new reductions of a partial solution to a bin packing problem. The first one can in some cases dominate Shaw's reduction and the second one theoretically dominates the other two.

Paul Shaw's reduction: R0. Shaw's reduction consists in creating a bin packing problem with the following characteristics. The bin capacity is the largest upper bound of the load variables, *i.e.* $c = \max_{j\in\{1,\ldots,m\}}(l_j^{\max})$. All items that are not packed in the constraint are part of the items of the reduced problem. Furthermore, for each bin, a virtual item is added to the reduced problem to reflect (1) the upper bound dissimilarities of the load variables and (2) the already packed items. More precisely, the size of the virtual item for a bin j is $(c - l_j^{\max} + \sum_{\{i|x_i=j\}} w_i)$, that is the bin capacity c reduced by the actual capacity of the bin in the constraint plus the total size of the already packed items in this bin. An example is shown in Figure 1(b).

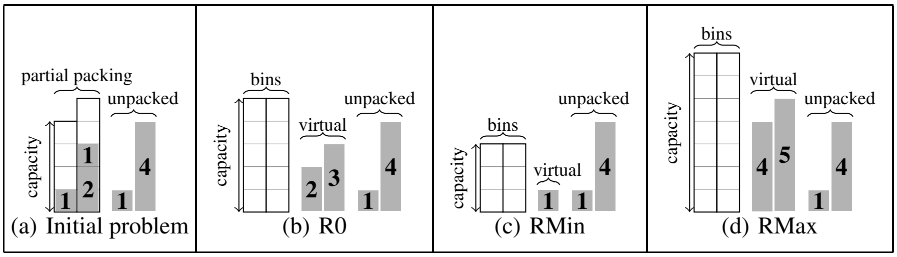


Fig. 1. Example of the three reductions for the bin packing problem

RMin. We introduce RMin that is obtained from R0 by reducing the capacity of the bins and the size of all the virtual items by the size of the smallest virtual item. The virtual items have a size of $(c - l_j^{\max} + \sum_{\{i|x_i=j\}} w_i - min_k(c - l_k^{\max} + \sum_{\{i|x_i=k\}} w_i))$. This reduction is illustrated in Figure 1(c).

RMax. We propose RMax that consists in increasing the capacity and the size of the virtual items by a common quantity, so that, when distributing the items with a bin packing algorithm, it is guaranteed that each virtual item will occupy a different bin. In order to achieve this, each virtual item's size must be larger than half the bin capacity.

In R0, let p be the size of the smallest virtual item, and c the capacity of the bins. The size of the virtual items and the capacity must be increased by $(c - 2p + 1)$. The smallest virtual item will have a size of $s = (c - p - 1)$ and the capacity of the bins will be $(2c - 2p + 1) = 2s - 1$. As one can observe, the smallest virtual item is larger than the half of the capacity. If $c = 2p - 1$, this reduction is equivalent to Shaw's reduction. Note that if $c < 2p - 1$, the capacity and the virtual items will be reduced.

The virtual items have a size of $(2c - 2p + 1 - l_j^{\max} + \sum_{\{i|x_i=j\}} w_i)$. This reduction is illustrated in Figure 1(d).

Generic reduction: Rδ. All these reductions are particular cases of a generic reduction (Rδ) which, based on R0, consists in adding a positive or negative delta (δ) to the capacity and to all the virtual items' sizes.

For R0, $\delta = 0$. For RMin, δ is the minimum possible value that keeps all sizes positive. A smaller δ would create an inconsistency, as the smallest virtual item would have a negative size. δ_{RMin} is always negative or equal to zero. For RMax, δ is the smallest value guaranteeing that virtual items cannot pile up. Note that in some cases, δ_{RMin} or δ_{RMax} can be zero. Also note that δ_{R0} can be larger than the others.

3 Theoretical Comparison of the Three Reductions

Definition 1 (Dominate). *Let A and B be two reductions of the `Pack` constraint to bin packing. We say that A* dominates *B if, for any instance of the `Pack` constraint, the number of bins required in A is larger than the number of bins required in B.*

Theorem 1. *Rδ is a relaxation of the problem of testing the consistency of the `Pack` constraint.*

Proof. If a partial solution of the `Pack` constraint can be extended to a solution with every item placed, then Rδ also has a solution: if each virtual item is placed in its initial bin, then the free space of each bin is equal to its free space in the partial solution, and so all the unplaced items can be placed in the same bin as in the extended solution from the partial assignment.

Theorem 2. *R0 does not dominate RMin and RMin does not dominate R0.*

Proof. Consider the partial packing $\{4, 2\}$ of two bins of capacity 6, and the unpacked items $\{3, 3\}$. R0 only needs two bins, where RMin needs three bins.

Now consider the partial packing $\{2, 3, 1\}$ of three bins of capacity 4, and the unpacked items $\{3, 3\}$. In this case, R0 needs four bins, where RMin only needs three bins.

Theorem 3. *RMax is equivalent to testing the consistency of the `Pack` constraint*

Proof. By Theorem 1, RMax is a relaxation of the partial solution of the BP problem. There remains to show that if there is a solution for RMax, then the partial solution can be extended to a complete solution of the `Pack` constraint. Let's call v the bin from which the virtual item v is from. It is guaranteed by the size of the virtual items that they will each be placed in a different bin b_v. The remaining space in each bin b_v corresponds to the free space in bin v in the original problem. An extended solution of the `Pack` constraint is obtained by packing in v all items packed in b_v.

Corollary 1. *RMax dominates R0 and RMin.*

From a theroretical standpoint, the RMax reduction is always better or equivalent to R0, RMin, and any other instance of Rδ. In practice, though, this is not always the case, as it is shown in the next section.

4 Experimental Comparison

The failure test of Shaw [1] uses the bin packing lower bound $\mathcal{L}_2$ of Martello and Toth [2] that can be computed in linear time. Recently the lower bound $\mathcal{L}_3$ of Labbé [3] has been proved [4] to be always larger than or equal to $\mathcal{L}_2$ and to benefit from a better worst case asymptotic performance ratio ($3/4$ for $\mathcal{L}_3$ [4] and $2/3$ for $\mathcal{L}_2$ [2]), while still having a linear computation time. Experiments show us that $\mathcal{L}_3$ can help detect about 20% more failures than $\mathcal{L}_2$. Throughout the next experiments, we are using $\mathcal{L}_3$.

Although in theory, RMax always outperforms R0 and RMin, the practical results are less systematic. This is because $\mathcal{L}_3$ (as well as $\mathcal{L}_2$) is not monotonic, which means that a BP instance requiring a larger number of bins than a second instance can have a lower bound smaller than the second one. In fact, $\mathcal{L}_3$ is more adapted to instances where most item sizes are larger than the third of the capacity. RMax increases the capacity, making unpacked items proportionally smaller. For each of R0, RMin and RMax, there are instances where they contribute to detecting a failure, while the other two do not.

Table 1 presents the performance of the failure detection using each one of the reductions. It shows the ratio of failures found using each reduction over the total number of failures found by at least one filter. Additional reductions have been experimented, with δ being respectively 25%, 50% and 75% on the way between δ_{RMin} and δ_{RMax}. These results were obtained by generating more than 1,000 random instances and computing $\mathcal{L}_3$ on each of their reductions. Here is how the instances were produced:

Inst1. Number of bins, number of items and capacity C each randomly chosen between 30 and 50. Bins already filled up to $1..C$. Random item sizes in $\{1, \ldots, C\}$.
Inst2. 50 bins. Capacity = 100. Number of items is 100 or 200. Size with normal distribution ($\mu = 5000/n$, $\sigma \in \{3n, 2n, n, n/2, n/3\}$ where n is the number of items). Among these, percentage of items already placed $\in \{10\%, 20\%, 30\%, 40\%, 50\%\}$.
Inst3. Idem as 2, but the number of placed items is 90% or 95%.

Table 1. Comparison of the number of failures found with different reductions

Instances	Number of failures detected (%)					
	RMin	R25	R50	R75	RMax	R0
Inst1	74.16	78.87	86.40	89.53	**99.58**	74.79
Inst2	**99.93**	86.75	87.03	87.8	87.15	**99.93**
Inst3	80.64	86.55	93.37	97.75	**99.39**	98.52

This reveals that some types of instances are more adapted to R0 or RMin, while some are more adapted to RMax. The intermediate reductions R25, R50 and R75 were never better in average than RMin and RMax. Thus, they were not considered in the following experiments.

Comparison on benchmark instances. For the analysis to be more relevant, we compared the behavior of the three proposed reductions on real instances. CP algorithms were run over Scholl's SALBP-1 benchmark [5] and on Scholl's bin packing instances [6] (first data set with n=50 and n=100), and at every change in the domains of the variables, the current partial solution was extracted. We randomly selected 30,000 extracted instances from each. In the second case, only instances for which at least one reduction could detect a failure were selected. The three reductions using $\mathcal{L}_3$ were applied on these selected instances. Figure 2 gives a schema of the results.

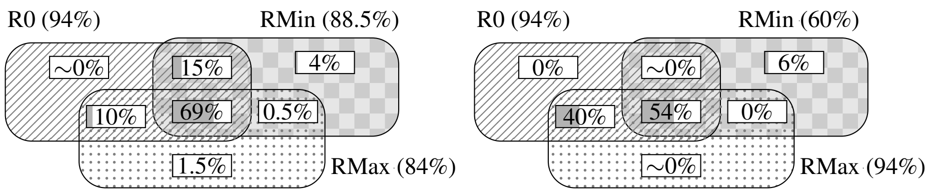


Fig. 2. Proportions of failure detections using each reduction on SALBP-1 instances (left) and BP instances (right)

These results show that R0 detects a larger number of failures. But (almost) all of its failures are also detected by one of the others. Hence, combining RMin and RMax is better than using R0 alone. It is also useless to combine R0 with RMin and RMax.

Impact on a CP search. We compared the effect of applying the failure detection strategy in a CP search on Scholl's bin packing instances N1 and N2 (360 instances), using R0, RMin, RMax and then RMin and RMax combined, with a time limit of five minutes for each instance. For the instances for which all reductions leaded to the same solution, the mean computation time of the searches was computed. All these results are presented in Table 2. One can observe that RMin and Rmax combined find more optimal solutions (though there is no significative difference with R0), and lead faster to the solution than the others (33% speedup compared to R0).

Table 2. Comparison of the reductions on solving the BPLB problem

	No pruning	R0	RMin	RMax	RMin & RMax
Number of optimal solutions	281	317	315	309	**319**
Mean time (s)	5.39	1.88	1.60	3.50	**1.25**

5 Conclusion

We presented two new reductions of a partial solution of the `Pack` constraint to a bin packing problem. Through a CP search, these reductions are submitted to a bin packing lower bound algorithm in order to detect failures of the `Pack` constraint as suggested by Shaw in [1].

We proved that our second reduction (RMax) theoretically provides a better failure detection than the others, assuming a perfect lower-bound algorithm. We conclude that the best strategy is to consider both RMin and RMax filters in a CP search.

Acknowledgments. The first author is supported by the Belgian FNRS (National Fund for Scientific Research). This research is also partially supported by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy) and the FRFC project 2.4504.10 of the Belgian FNRS.

References

1. Shaw, P.: A constraint for bin packing. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 648–662. Springer, Heidelberg (2004)
2. Martello, S., Toth, P.: Lower bounds and reduction procedures for the bin packing problem. Discrete Appl. Math. 28(1), 59–70 (1990)
3. Labbé, M., Laporte, G., Mercure, H.: Capacitated vehicle routing on trees. Operations Research 39(4), 616–622 (1991)
4. Bourjolly, J.M., Rebetez, V.: An analysis of lower bound procedures for the bin packing problem. Comput. Oper. Res. 32(3), 395–405 (2005)
5. Scholl, A.: Data of assembly line balancing problems. Technische Universität Darmstadt (93)
6. Scholl, A., Klein, R., Jürgens, C.: Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. Computers & Operations Research 24(7), 627–645 (1997)

A Relax-and-Cut Framework for Gomory's Mixed-Integer Cuts

Matteo Fischetti[1] and Domenico Salvagnin[2]

[1] DEI, University of Padova, Italy
matteo.fischetti@unipd.it
[2] DMPA, University of Padova, Italy
salvagni@math.unipd.it

Abstract. Gomory's Mixed-Integer Cuts (GMICs) are widely used in modern branch-and-cut codes for the solution of Mixed-Integer Programs. Typically, GMICs are iteratively generated from the optimal basis of the current Linear Programming (LP) relaxation, and immediately added to the LP before the next round of cuts is generated. Unfortunately, this approach prone to instability.

In this paper we analyze a different scheme for the generation of rank-1 GMIC read from a basis of the original LP—the one before the addition of any cut. We adopt a relax-and-cut approach where the generated GMIC are not added to the current LP, but immediately relaxed in a Lagrangian fashion.

Various elaborations of the basic idea are presented, that lead to very fast—yet accurate—variants of the basic scheme. Very encouraging computational results are presented, with a comparison with alternative techniques from the literature also aimed at improving the GMIC quality, including those proposed very recently by Balas and Bonami and by Dash and Goycoolea.

Keywords: Mixed-integer programming, Gomory's cuts, Lagrangian relaxation, Relax and Cut.

1 Introduction

Gomory's Mixed-Integer Cuts (GMICs) are of fundamental importance for branch-and-cut Mixed-Integer Program (MIP) solvers, that however are quite conservative in their use because of known issues due to the iterative accumulation of GMICs in the optimal Linear Programming (LP) basis, which leads to numerical instability due a typically exponential growth of the determinant of the LP basis.

Recent work on the subject suggests however that stability issues are largely due to the overall framework where GMICs are used, rather than to the GMICs themselves. Indeed, the two main cutting plane modules (the LP solver and the cut generator) form a closed-loop system that is intrinsically prone to instability—unless a "decoupling filter" is introduced in the loop. Breaking the feedback is therefore a must if one wants to really exploit the power of GMICs.

In this paper we propose a new mechanism to break the entanglement between LP bases and GMICs cuts. More specifically, in our framework the generated GMICs are not added to the current LP, but immediately relaxed in a Lagrangian fashion—following an approach known as *relax-and-cut*. In this way, GMICs are always generated from a (Lagrangian near-optimal) basis of the original LP, hence their quality is not likely to deteriorate in the long run as we do not allow GMIC to accumulate in the LP basis.

The paper is organized as follows. Section 2 briefly reviews some relevant literature. In Section 3 we introduce our notation and describe the relax-and-cut framework. Various elaborations of the basic idea are presented in Section 4, that lead to faster yet accurate variants of the basic relax-and-cut scheme. Very encouraging computational results are presented in Section 5, with a comparison with alternative techniques from the literature also aimed at improving the GMIC quality, namely those proposed very recently by Balas and Bonami [1] and by Dash and Goycoolea [2]. Some conclusions and possible directions of work are finally drawn in Section 6.

We assume the reader has some familiarity with MIP cuts; see, e.g., Cornuéjols [3] for a recent survey on the subject.

2 Literature

GMICs for general MIPs have been introduced by Ralph Gomory about 50 years ago in his seminal paper [4]. However, these cuts were not used in practice until the work of Balas, Ceria, Cornuéjols and Natraj [5], who found for the first time an effective way to exploit them in a branch-and-cut context [6]. In particular, the authors stressed the importance of generating GMICs in *rounds*, i.e., from all the tableau rows with fractional right hand side.

The explanation of GMIC instability in terms of closed-loop systems was pointed out by Zanette, Fischetti and Balas [7], who presented computational experiments showing that reading the LP optimal solution to cut *and* the Gomory cuts from the same LP basis almost invariably creates a dangerous feedback in the long run.

The same explanation applies to other cutting plane procedures that derive cuts directly from tableau information of the enlarged LP that includes previously-generated cuts (e.g., those related to Gomory's corner polyhedron, including cyclic-group cuts, intersection cuts, multi-row cuts, etc.). This is not necessarily the case when using methods based on an external cut generation LP (e.g., disjunctive methods using disjunctions not read from the optimal tableau), or when the derived cuts are post-processed so as to reduce their correlation with the optimal tableau (e.g., through lexicographic search [7] or by cut strengthening methods [2,8]).

A different framework for Gomory's cuts was recently proposed by Fischetti and Lodi [9]. The aim of that paper was actually to compute the best possible bound obtainable with rank-1 fractional Gomory's cuts. The fact of restricting to rank-1 cuts forced the authors to get rid of the classical separation scheme, and to model the separation problem through an auxiliary MIP to be solved by an external module. The surprising outcome was a numerically stable cutting plane method where rank-1 fractional Gomory's cuts alone produced very tight bounds—though the separation overhead was too large to be practical in most cases. Note that, in that scheme, the separation procedure did not have access to the optimal LP basis, but only received on input the point to be separated—hence loosening the optimization and separation entanglement. As a consequence, even if the point x^* to be separated possibly did not change significantly at some iterations, it was unlikely that the separated cuts were as heavily correlated as in the classical scheme—in this context, the well-known erratic behavior of MIP solvers that often return quite different solutions for almost identical input, turned out to be beneficial in that it acted as a diversification in the cut selection policy. These results were later confirmed for GMICs by Balas and Saxena [10] and by Dash, Günlük, and Lodi [11], who adopted the same scheme but generalized the MIP separation module so as to deal with GMIC separation.

The above discussion suggests that an improved performance can be attained if one does not insist on reading GMICs from the optimal basis of the current LP, that includes previously generated GMICs. Progresses in this direction have been obtained recently by using one of the following two approaches. Let x^* be an optimal vertex of the *large LP* (the one defined by the original constraints plus the GMICs generated in the previous iterations), and let B^* be an associated optimal basis.

(i) Balas and Perregaard [8] perform a sequence of pivots on the tableau of the large LP leading to a (possibly non-optimal or even infeasible) basis of the same large LP that produces a deeper cut w.r.t. the given x^*.
(ii) Dash and Goycoolea [2] heuristically look for a basis B of the *original* LP that is "close enough to B^*", in the hope of cutting the given x^* with rank-1 GMICs associated with B; this is done, e.g., by removing from A all the columns that are nonbasic with respect to x^*, thus restricting B to be a submatrix of B^*.

The approach of relaxing cuts right after their separation is known in the literature as the *Relax-and-Cut* strategy. It was introduced independently by Lucena [12], and by Escudero, Guignard and Malik [13]—who actually proposed the relax-and-cut name; see Lucena [14] for a survey of the technique and of its applications. Very recently, Lucena [15] applied a relax-and-cut approach to the solution of hard single 0-1 knapsack problems, where fractional Gomory's cuts were used, for the first time, in a Lagrangian framework.

3 A Relax-and-Cut Framework for GMICs

Consider a generic MIP of the form

$$\min\{cx : Ax = b,\ x \geq 0,\ x_j \text{ integer } \forall j \in J\}$$

where $A \in \mathbb{Q}^{m \times n}$, $b \in \mathbb{Q}^m$, $c \in \mathbb{Q}^n$, and $J \subseteq \{1, \cdots, n\}$ is the index set of the integer variables. As customary, let $P := \{x \in \mathbb{R}^n_+ : Ax = b\}$ denote the LP relaxation polyhedron, that we assume to be bounded.

Given a large (possibly exponential) family of linear cuts

$$\alpha^i x \geq \alpha^i_0, \quad i = 1, \ldots, M \tag{1}$$

we aim at computing—possibly in an approximate way—the value

$$z_1 := \begin{cases} \min cx \\ x \in P \\ \alpha^i x \geq \alpha^i_0, \quad i = 1, \ldots, M \end{cases} \tag{2}$$

In our basic application, family (1) consists of the GMICs associated with all possible primal-feasible bases of system $Ax = b$, i.e., z_1 is a (typically very tight) lower bound on the first GMIC closure addressed by Balas and Saxena [10] and by Dash, Günlük, and Lodi [11]. However, as discussed in the computational section, family (1) is in principle allowed to contain any kind of linear inequalities, including problem-specific cuts and/or GMICs of any rank, or even invalid linear conditions related, e.g., to branching conditions.

A standard solution approach for (2) consists in dualizing cuts (1) in a Lagrangian fashion, thus obtaining the Lagrangian dual problem

$$\max_{u \geq 0} \left\{ L(u) := \min\{cx + \sum_{i=1}^{M} u_i(\alpha^i_0 - \alpha^i x) : \ x \in P\} \right\} \tag{3}$$

whose optimal value is known to coincide with z_1.

The solution of (3) can be attempted through very simple iterative procedures known as subgradient methods, or through more sophisticated and accurate schemes such as the bundle method; see e.g. [16]. All the above solution schemes generate a sequence of dual points $u^k \geq 0$ meant to converge to an optimal dual solution u^*. For each u^k in the sequence, an optimal solution $x^k \in P$ of the inner-minimization in (3) is computed, along with the associated Lagrangian value

$$L(u^k) = cx^k + \sum_{i=1}^{M} u^k_i(\alpha^i_0 - \alpha^i x^k)$$

and subgradient $s^k \in \mathbb{R}^M$, whose components are just the cut violations

$$s^k_i := \alpha^i_0 - \alpha^i x^k, \quad i = 1, \ldots, M$$

($s^k_i > 0$ for violated cuts, and $s^k_i \leq 0$ for slack/tight cuts). In particular, the ability of computing the subgradient is essential for the convergence of overall scheme—this is not a trivial task when the cut family is described only implicitly.

In our context, family (1) is by far too large to be computed explicitly, so we store only some of its members, using a data structure called *cut pool*. Cut duplication in the pool is heuristically avoided by normalizing the cut right-hand-side and by using a hash function.

The cut pool is initially empty, or it can contain some heuristic collection of warm-start cuts. The pool is then iteratively extended by means of rank-1 GMICs that are heuristically generated, on the fly, during the process of solving the Lagrangian dual problem. More specifically, if the Lagrangian subproblem at a certain u^k is solved by the simplex method and an optimal vertex x^k of P with fractional components is found, we can just read a round of rank-1 GMICs from the optimal LP basis and feed the cut pool. Note that these cuts are always associated with a primal-feasible basis of the original system P, so they are globally valid for our MIP problem even if the cut pool contains invalid cuts (e.g., branching conditions or temporary diversification cuts). Also note that, although violated by x^k, some of these cuts can actually belong already to the current pool—an indication that their Lagrangian multiplier should be increased in the next iterations.

A characteristic of relax-and-cut methods is that, differently from traditional cutting plane schemes, there is no natural "fractional point to cut", and the discovery of new cuts to be added to the pool is beneficial mainly because new components of the "true" subgradient s^k are computed, thus improving the chances of convergence to the "true" optimal dual value z_1 of the overall Lagrangian scheme.

4 Implementations

We next describe three very basic heuristics for the Lagrangian dual problem (3), that are intended to evaluate the potentials of using GMICs in a relax-and-cut framework. The investigation of more sophisticated schemes such as the bundle method is left to future investigation.

4.1 Subgradient Optimization

The basic algorithm underlying our heuristics is the subgradient method. The subgradient method is an adaptation of the gradient method to the case of maximization of nondifferentiable concave functions, such as $L(u)$. It starts with a tentative point $u_0 \geq 0$ and then iteratively constructs a sequence of points u^k according to the following rule:

$$u^{k+1} = (u^k + \lambda_k s^k)_+$$

where s^k is a subgradient of $L(\cdot)$ in u^k, $\lambda_k > 0$ is an appropriate parameter called *step length*, and $(\cdot)_+$ denotes the projection onto the nonnegative orthant.

The asymptotic convergence of the method is guaranteed by the properties of the subgradient and by the choice of appropriate step sizes. A step-size rule often used in practice, usually known as *relaxation step length* or *Polyak's step length*, computes

$$\lambda_k = \frac{\mu_k(UB - L(u^k))}{||s^k||^2}$$

where μ_k is a parameter satisfying $0 < \mu_k \leq 2$, and UB is the unknown optimal dual value z_1, typically replaced by an upper bound on z_1. In our code, this upper bound is computed as the objective value of the best integer solution found by a MIP solver at the end of root node, or an appropriate multiple of the LP relaxation if none is found. As to μ_k, it is adjusted dynamically in order to try to speed up convergence, using quite an elaborated update strategy inspired by the computational studies reported in [17,18]. In particular, at the beginning of each Lagrangian iteration, we compute a "reference" interval $\Delta = UB - bestLB$, that we use to guide our strategy. If, in the last $p = 100$ iterations, $bestLB$ has improved by less than 0.01Δ, then we update μ_k as follows:

$$\mu_k = \begin{cases} 10\mu_k & \text{if } bestLB - avgLB < 0.001\Delta \\ 2\mu_k & \text{if } 0.001\Delta \leq bestLB - avgLB < 0.01\Delta \\ \mu_k/2 & \text{otherwise} \end{cases}$$

where $avgLB$ is the average value of $L(u)$ in the last p iterations. Finally, if $L(u) < bestLB - \Delta$ for 10 consecutive iterations we halve μ_k and backtrack to the best u^k so far.

In the following, we will denote by `subg` our implementation of a pure subgradient method for solving (3), with a limit of 10,000 iterations. The starting step size parameter is aggressively set to $\mu_0 = 10$. This is justified by the fact that in our scenario the convergence of the method is not guaranteed (and is also unlikely in practice), because we are dealing explicitly only with a small subset of cuts. In particular, we always deal with truncated subgradients and, even more importantly, we have no way of generating violated GMICs apart from reading them from the LP tableau. According to our computational experience, in this scenario a small initial value for μ is quite unappropriate because it causes the method to saturate far from an optimal Lagrangian dual solution u^*, with no possibility for recovery.

Finally, to avoid overloading the cut pool, we read a round of GMICs at every K-th subgradient iteration, where $K = 10$ in our implementation. In addition, the length of the Lagrangian vector u^k is not increased every time new cuts are added to the pool, but only every 50 subgradient iterations, so as to let the subgradient method stabilize somehow before adding new Lagragian components. In this view, our implementation is between the so-called *delayed* and *non-delayed* relax-and-cut methods [14].

4.2 Hybrid LP and Subgradient Optimization

The basic subgradient method presented in the previous subsection has several drawbacks when applied to (3). In particular, finding the right combination of parameters to obtain a satisfactory convergence is definitely tricky. In our setting, we found beneficial to recompute, from time to time, the optimal vector u of for *all* the cuts in the current pool, which amounts to solving a large LP by means of a standard dynamic pricing of the cuts in the pool, akin to the one proposed in [19]. Note that this policy is usually not attractive in a classical setting where the number of dualized constraints is fixed—solving the large LP would be just as hard as solving (3). This is however not the case in our context, because the pool is extended dynamically and stores a (large but) manageable subset of cuts.

In what follows, we will denote by `hybr` our implementation of a hybrid subgradient method for solving (3), where we periodically compute the optimal multipliers of the pool cuts by solving the large LP (note however that we do not read GMICs from the optimal basis of the large LP). In our code this is done every $1,000$ subgradient iterations. All other parameters are the same as in `subg`.

4.3 Fast Hybrid Framework

Although the hybrid version `hybr` is definitely an improvement over `subg`, both methods are still quite demanding as far as running time is concerned. The reason is twofold. First, we may spend a lot of time generating GMICs from useless bases (contrarily to popular belief, reading cuts from the tableau comes *not* for free, although it is very cheap compared to other separation methods). Second, the LPs change significantly from one iteration to the next one, because of the zig-zagging nature of the dual multipliers induced by the standard subgradient algorithm, hence the usual warm-start of the simplex algorithm is less effective—note that this drawback may be reduced by using more stabilized algorithms like the bundle method.

We developed some variants of `hybr` tweaked for speed, trying to sacrifice the quality of the computed bound on z_1 as little as possible. Speed is obtained by drastically reducing the number of subgradient iterations and by using a very small step length parameter ($\mu_k = 0.01$ in our code). The small step size yields more parametrized LPs where warm-start is more effective, and the reduced number of iterations speeds up the whole approach. In a sense, we are no longer relying on the step size for the convergence of the method—which is taken care of by the large LPs used to get the optimal multipliers—and we use the subgradient method just to sample near-optimal Lagrangian bases of the original system generating rank-1 GMICs (this will be called the *sampling phase* in the sequel). It is worth noting that the small step length parameter and the reduced number of iterations essentially turn off the step-length update strategy that we have described in Section 4.1.

We implemented two variants of the above method, namely `fast` and `faster`. In both variants we solve the large LP to compute the Lagrangian optimal multipliers only 10 times, and we generate GMICs at every subgradient iteration. The difference is in the number of subgradient iterations in the sampling phase between two consecutive large-LP resolutions, which is 100 for `fast`, and just 50 for `faster`.

It is worth observing that the methods above can be interpreted *à la Dantzig-Wolfe* as a way to decompose the optimal solution x^* of the large LP into a suitable convex combination $\sum_j \lambda_j x^j$ of vertices x^j of P, and to separate these x^j in the attempt of finding valid cuts violated by x^*. This links those variants to the work of Ralphs, Kopman, Pulleyblank, and Trotter [20], where a similar idea was applied to separate capacity cuts for the Capacitated Vehicle Routing Problem—the fractional CVRP vertex being expressed as the convex combination of m-TSP integer solutions, each of which is easily evaluated to find violated capacity cuts.

5 Computational Results

We tested our variants of the relax-and-cut framework for GMICs on the problem instances in MIPLIB 3.0 [21] and MIPLIB 2003 [22]. Following [2], we omitted all instances where there is no improvement after one round of GMICs read from the optimal tableau, or where no integer solution is known. Moreover, we excluded instances `mod011` and `rentacar`, because of the presence of ranged constraints in the formulation, that are not handled by our current GMIC code. In the end, we were left with 52 instances from MIPLIB 3.0, and 20 instances from MIPLIB 2003. For the sake of space, we will only report aggregated statistics; detailed tables are available, on request, from the authors.

We implemented our code in C++, using IBM ILOG Cplex 11.2 as black box LP solver (its primal heuristics were also used to compute the subgradient upper bound UB). All tests have been performed on a PC with an Intel Q6600 CPU running at 2.40GHz, with 4GB of RAM (only one CPU was used by each process). As far as the GMIC generation is concerned, for a given LP basis we try to generate a GMIC from every row where the corresponding basic variable has a fractionality of at least 0.001. The cut is however discarded if its final dynamism, i.e., the ratio between the greatest and smallest absolute value of the cut coefficients, is greater than 10^{10}.

5.1 Approximating the First GMI Closure

In our first set of experiments we compared the ability (and speed) of the proposed methods in approximating the first GMI closure for the problems in our testbed. The first GMI closure has received quite a lot of attention in the last years, and it was computationally proved that it can provide a tight approximation of the convex hull of the feasible solutions. In addition, rank-1 GMICs are read from the original tableau, hence they are generally considered safe from the numerical point of view. Note that our method can only generate cuts from *primal-feasible* bases, hence it can produce a weaker bound than that associated with the first GMI closure [23].

In Table 1 we report the average gap closed by all methods that generate rank-1 GMICs only, as well as the corresponding computing times (geometric means). We recall that for a given istance, the gap closed is defined as $100 \cdot (z - z_0)/(opt - z_0)$, where z_0 is the value of the initial LP relaxation, z is the value of the final LP relaxation, and opt is the best known solution. For comparison, we report also the average gap closed by one round of GMIC read from the the first optimal tableau (`1gmi`), as well as the average gap closed with the default method proposed by Dash and Goycoolea (`dgDef`), as reported in [2].

Table 1. Average gap closed and computing times for rank-1 methods

	MIPLIB 3.0		MIPLIB 2003	
method	cl.gap	time (s)	cl.gap	time (s)
`1gmi`	26.9%	0.02	18.3%	0.54
`faster`	57.9%	1.34	43.3%	33.33
`fast`	59.6%	2.25	45.5%	58.40
`hybr`	60.8%	15.12	48.6%	315.21
`subg`	56.0%	25.16	43.5%	291.21
`dgDef`	61.6%	20.05	39.7%	853.85

All computing times are given in CPU seconds on our Intel machine running at 2.4 GHz, except for `dgDef` where we just report the computing times given in [2], without any speed conversion—the entry for MIPLIB 3.0 refers to a 1.4 GHz PowerPC machine (about 2 times slower than our PC), while the entry for MIPLIB 2003 refers to a 4.2 GHz PowerPC machine (about twice as fast as our PC).

According to the table, the relax-and-cut methods performed surprisingly well, in particular for the hard MIPLIB 2003 instances where all of them outperformed `dgDef` in terms of both quality and speed.

As far as the bound quality is concerned, the best method appears to be `hybr`, mainly because of its improved convergence with respect to `subg`, and of the much larger number of subgradient iterations (and hence of LP bases) generated with respect to the two fast versions.

The two fast versions also performed very well, in particular `faster` that proved to be really fast (more than 10 times faster than `dgDef`) and quite accurate. It is worth observing that about 75% of the computing time for `fast` and `faster` was spent in the sampling phase: 40% for LP reoptimizations, and 35% for actually reading the GMICs from the tableau and projecting slack variables away. Quite surprisingly, the solution of the large LPs through a dynamic pricing of the pool cuts required just 15% of the total computing time.

5.2 A Look to Higher Rank GMICs

In this subsection we investigate the possibility of generating GMICs of rank greater than 1. Unfortunately there is no fast way to compute the exact rank of a cut, hence we use an easy upper bound where the rows of the original system $Ax = b$ are defined to be of rank 0, and the rank of a GMIC is computed as the maximum rank of the involved rows, plus one. Having computed the above upper bound for each GMIC, we avoid storing in the pool any GMICs whose upper bound exceeds an input rank limit k (k =2 or 5, in our tests).

Our relax-and-cut framework can be extended in many different ways to generate higher-rank GMICs. In particular, given a maximum allowed rank k, it is possible to:

a) Generate k rounds of GMICs in a standard way, use them to intialize the cut pool, and then apply our method to add rank-1 GMICs on top of them. This very simple strategy turned out not to work very well in practice, closing significantly less gap than the rank-1 version.
b) Apply one of the relax-and-cut variants of the previous subsection until a termination condition is reached. At this point add to the original formulation (some of) the GMICs that are tight at the large-LP optimal solution, and repeat k times. This approach works quite well as far the final bound is concerned, but it is computationally expensive because we soon have to work with bigger (and denser) tableaux.
c) Stick to rank-1 GMICs in the sampling phase, never enlarging the original system. However, each time a large LP is solved to recompute the dual multipliers (this can happen at most k times), add to the pool (but not to the original formulation) all the GMICs read from the large-LP optimal basis.
d) As before, stick to rank-1 GMICs in the sampling phase. If however no cut separating the previous large-LP solution x^* is found in the sampling phase, then add to the pool all GMICs read from the large LP optimal basis, and continue. This way, the generation of higher-rank cuts acts as a diversification step, used to escape a local deadlock, after which standard rank-1 separation is resumed.

According to our preliminary computational experience, the last two schemes give the best compromise between bound quality and speed. In particular, c) takes almost the same computing time as its rank-1 counterpart in Table 1, and produces slightly improved bounds. Option d) is slower than c) but closes significantly more gap, hence it seems more attractive for a comparison with rank-1 cuts.

Table 2. Average gap closed and computing times for higher rank methods

		MIPLIB 3.0		MIPLIB 2003	
method	rank	cl.gap	time (s)	cl.gap	time (s)
`gmi`	1	26.9%	0.02	18.3%	0.54
`faster`	1	57.9%	1.34	43.3%	33.33
`fast`	1	59.6%	2.25	45.5%	58.40
`gmi`	2	36.0%	0.03	24.0%	0.88
`faster`	2	62.1%	2.75	47.2%	58.37
`fast`	2	64.1%	5.12	48.5%	106.76
`gmi`	5	47.8%	0.07	30.3%	2.17
`faster`	5	65.6%	5.47	49.9%	126.65
`fast`	5	67.2%	10.09	51.1%	238.33
`L&P`	10	57.0%	3.50	30.7%	95.23

In Table 2 we report the average gap closed by our fast versions when higher-rank GMICs are generated according to scheme d) above. Computing times (geometric means) are also reported. Rank-1 rows are taken from the previous table.

In the table, row `gmi` refers to 1, 2 or 5 rounds of GMICs. For the sake of comparison, we also report the average gap closed by 10 rounds of Lift&Project cuts (`L&P`), as described in [1]. To obtain the Lift&Project bounds and running times we ran the latest version of separator `CglLandP` [24] contained in the COIN-OR [25] package Cgl 0.55, using Clp 1.11 as black box LP solver (the separator did not work with Cplex because of the lack of some pivoting procedures). This separation procedure was run with default settings, apart from the minimum fractionality of the basic variables used to generate cuts, which was set to 0.001 as in the other separators. All computing times are given in seconds on our Intel machine running at 2.4 GHz.

Our fast procedures proved quite effective also in this setting, providing significantly better bounds than `L&P` in a comparable or shorter amount of time, even when restricting to rank-1 GMICs. As expected, increasing the cut rank improves the quality of the bound by a significant amount, though it is not clear whether this improvement is worth the time overhead—also taking into account that GMICs of higher rank tend to be numerically less reliable. Similarly, it is not clear whether the bound improvement achieved by `fast` w.r.t. `faster` is worth the increased computing time.

6 Conclusions and Future Work

We have considered Gomory Mixed-Integer Cuts (GMICs) read from an optimal LP basis, as it is done customary in branch-and-cut methods, but in a new shell aimed at overcoming the notoriously bad behavior of these cuts in the long run. The new shell uses a relax-and-cut approach where the generated GMICs are not added to the current LP, but are stored in a cut pool and immediately relaxed in a Lagrangian fashion.

We have presented some variants of our basic method and we have computationally compared them with other methods from the literature. The results have shown that even simple implementations of the new idea are quite effective, and outperform their competitors in terms of both bound quality and speed. We are confident however that there is still room for improvement of our basic methods.

Future work should investigate the following research topics:

- The use of a more sophisticated Lagrangian dual optimizer to replace the simple subgradient procedure we implemented.
- Our method is meant to add rank-1 GMICs on top of a collection of other cuts collected in a cut pool. In our current experiments the cut pool only contains GMICs collected in the previous iterations. However, it seems reasonable to allow the pool to contain other classes of (more combinatorial) cuts, e.g., all those generated at the root node by a modern MIP solver. In this setting,

the preprocessed model and the generated cuts (stored in the cut pool) can be provided as input to our relax-and-cut scheme, in the attempt of reducing even further the integrality gap at the root node.
- During Lagrangian optimization, a large number of (possibly slightly fractional or even integer) vertices of P are generated, that could be used heuristically (e.g., through rounding) to provide good primal MIP solutions.

Finally, in the process of developing our method we realized that cutting plane schemes miss an overall "meta-scheme" to control cut generation and to escape "local optima" by means of diversification phases—very well in the spirit of Tabu or Variable Neighborhood Search meta-schemes for primal heuristics. The development of sound meta-schemes on top of a basic separation tool is therefore an interesting topic for future investigations—our relax-and-cut framework for GMICs can be viewed as a first step in this direction.

References

1. Balas, E., Bonami, P.: Generating lift-and-project cuts from the LP simplex tableau: open source implementation and testing of new variants. Mathematical Programming Computation 1(2-3), 165–199 (2009)
2. Dash, S., Goycoolea, M.: A heuristic to generate rank-1 GMI cuts. Technical report, IBM (2009)
3. Cornuéjols, G.: Valid inequalities for mixed integer linear programs. Mathematical Programming 112(1), 3–44 (2008)
4. Gomory, R.E.: An algorithm for the mixed integer problem. Technical Report RM-2597, The RAND Cooperation (1960)
5. Balas, E., Ceria, S., Cornuéjols, G., Natraj, N.: Gomory cuts revisited. Operations Research Letters 19, 1–9 (1996)
6. Cornuéjols, G.: Revival of the Gomory cuts in the 1990's. Annals of Operations Research 149(1), 63–66 (2006)
7. Zanette, A., Fischetti, M., Balas, E.: Lexicography and degeneracy: can a pure cutting plane algorithm work? Mathematical Programming (2009)
8. Balas, E., Perregaard, M.: A precise correspondence between lift-and-project cuts, simple disjunctive cuts, and mixed integer Gomory cuts for 0-1 programming. Mathematical Programming 94(2-3), 221–245 (2003)
9. Fischetti, M., Lodi, A.: Optimizing over the first Chvàtal closure. Mathematical Programming 110(1), 3–20 (2007)
10. Balas, E., Saxena, A.: Optimizing over the split closure. Mathematical Programming 113(2), 219–240 (2008)
11. Dash, S., Günlük, O., Lodi, A.: MIR closures of polyhedral sets. Mathematical Programming 121(1), 33–60 (2010)
12. Lucena, A.: Steiner problems in graphs: Lagrangian optimization and cutting planes. COAL Bulletin (21), 2–8 (1982)
13. Escudero, L.F., Guignard, M., Malik, K.: A Lagrangian relax-and-cut approach for the sequential ordering problem with precedence relationships. Annals of Operations Research 50(1), 219–237 (1994)
14. Lucena, A.: Non delayed relax-and-cut algorithms. Annals of Operations Research 140(1), 375–410 (2005)

15. Lucena, A.: Lagrangian relax-and-cut algorithms. In: Handbook of Optimization in Telecommunications, pp. 129–145. Springer, Heidelberg (2006)
16. Hiriart-Hurruty, J.B., Lemaréchal, C.: Convex Analysis and Minimization Algorithms. Springer, Heidelberg (1993)
17. Caprara, A., Fischetti, M., Toth, P.: A heuristic method for set covering problem. Operations Research 47(5), 730–743 (1999)
18. Guta, B.: Subgradient Optimization Methods in Integer Programming with an Application to a Radiation Therapy Problem. PhD thesis, University of Kaiserslautern (2003)
19. Andreello, G., Caprara, A., Fischetti, M.: Embedding cuts in a branch and cut framework: a computational study with {0,1/2}-cuts. INFORMS Journal on Computing (19), 229–238 (2007)
20. Ralphs, T.K., Kopman, L., Pulleyblank, W.R., Trotter, L.E.: On the capacitated vehicle routing problem. Mathematical Programming 94(2-3), 343–359 (2003)
21. Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.P.: An updated mixed integer programming library: MIPLIB 3.0. Optima 58, 12–15 (1998), http://www.caam.rice.edu/bixby/miplib/miplib.html
22. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Operations Research Letters 34(4), 1–12 (2006), http://miplib.zib.de
23. Cornuéjols, G., Li, Y.: Elementary closures for integer programs. Operations Research Letters 28(1), 1–8 (2001)
24. CglLandP: website, https://projects.coin-or.org/Cgl/wiki/CglLandP
25. COIN-OR: website, http://www.coin-or.org/

An In-Out Approach to Disjunctive Optimization

Matteo Fischetti[1] and Domenico Salvagnin[2]

[1] DEI, University of Padova, Italy
matteo.fischetti@unipd.it
[2] DMPA, University of Padova, Italy
salvagni@math.unipd.it

Abstract. Cutting plane methods are widely used for solving convex optimization problems and are of fundamental importance, e.g., to provide tight bounds for Mixed-Integer Programs (MIPs). This is obtained by embedding a cut-separation module within a search scheme. The importance of a sound search scheme is well known in the Constraint Programming (CP) community. Unfortunately, the "standard" search scheme typically used for MIP problems, known as the Kelley method, is often quite unsatisfactory because of saturation issues.

In this paper we address the so-called *Lift-and-Project closure* for 0-1 MIPs associated with all disjunctive cuts generated from a given set of elementary disjunction. We focus on the search scheme embedding the generated cuts. In particular, we analyze a general meta-scheme for cutting plane algorithms, called in-out search, that was recently proposed by Ben-Ameur and Neto [1]. Computational results on test instances from the literature are presented, showing that using a more clever meta-scheme on top of a black-box cut generator may lead to a significant improvement.

Keywords: Mixed-integer programming, cutting planes, disjunctive optimization.

1 Introduction

Cutting plane methods are widely used for solving convex optimization problems and are of fundamental importance, e.g., to provide tight bounds for Mixed-Integer Programs (MIPs). These methods are made by two equally important components: (i) the separation procedure (oracle) that *produces* the cut(s) used to tighten the current relaxation, and (ii) the overall search framework that actually *uses* the generated cuts and determines the next point to cut.

In the last 50 years, a considerable research effort has been devoted to the study of effective families of MIP cutting planes, as well as to the definition of sound separation procedures and cut selection criteria [2,3]. However, the search component was much less studied, at least in the MIP context where one typically cuts a vertex of the current LP relaxation, and then reoptimizes the new LP to get a new vertex to cut—a notable exception is the recent paper [4] dealing with Benders' decomposition. The resulting approach—known as "the Kelley method" [5]—can however be rather inefficient, the main so if the separation procedure is not able to produce strong (e.g., facet defining or, at least, supporting) cuts. As a matter of fact, alternative search schemes are available that work with non-extreme (internal) points [6,7], including the famous ellipsoid [8,9] and analytic center [10,11,12] methods; we refer the reader to [13] for an introduction. The convergence behavior of these search methods is less dependant on the quality of the generated cuts, which is a big advantage when working with general MIPs where separation procedures tend to saturate and to produce shallow cuts. A drawback is that, at each iteration, one needs to recompute a certain "core" point, a task that can be significantly more time consuming than a simple LP reoptimization. An interesting hybrid search method, called *in-out search*, was recently proposed by Ben-Ameur and Neto [1].

In this paper we address disjunctive optimization [14] in the MIP context. It essentially consists of a cutting plane method where cuts are separated by exploiting a given set of valid disjunctions. In particular, we consider 0-1 MIPs and the associated *Lift-and-Project closure*, defined by all the disjunctive cuts that can be derived from the "elementary" set of disjunctions of the type $x_j \leq 0$ or $x_j \geq 1$ for each integer-constrained variable x_j. This topic is currently the subject of intensive investigation by the Mathematical Programming community. Our current research topic is in fact to move the research focus from the widely investigated separation module to the search scheme where the generated cuts are actually embedded. A first step in this direction is reported in the present paper, where we investigate the use of disjunctive cuts within an in-out search shell. Computational results show that the resulting scheme outperforms the standard one, in that it produces tighter bounds within shorter computing times and need much fewer cuts—though they use exactly the same separation module.

2 In-Out Search

Let us consider a generic MIP of the form

$$\min\{c^T x : Ax = b,\ l \leq x \leq u,\ x_j \in \mathbb{Z}\ \forall j \in I\}$$

and let $P := \{x \in \mathbb{R}^n : Ax = b,\ l \leq x \leq u\}$ denote the associated LP relaxation polyhedron. In addition, let us assume the oracle structure allows one to define a "cut closure", P_1, obtained by intersecting P with the half-spaces induced by all possible inequalities returned by the oracle. Cutting plane methods are meant to compute $z_1 := \min\{c^T x : x \in P_1\}$, with P_1 described implicitly through the oracle.

In-out search works with two points: an "internal" (possibly non optimal) point $q \in P_1$, and an optimal vertex x^* of P (possibly not in P_1). By construction, the final (unknown) value z_1 belongs to the *uncertainty interval* $[c^T x^*, c^T q]$, i.e., at each iteration both a lower and an upper bound on z_1 are available. If the two points q and x^* coincide, the cutting plane method ends. Otherwise, we apply a bisection step over the line segment $[x^*, q]$, i.e., we invoke the separation procedure in the attempt of cutting the middle point $y := (x^* + q)/2$. (In the original proposal, the separation point is more generally defined as $y := \alpha x^* + (1 - \alpha)q$ for a given $\alpha \in (0, 1]$.) If a violated cut is returned, we add it to the current LP that is reoptimized to update x^*, hopefully reducing the current lower bound $c^T x^*$. Otherwise, we update $q := y$, thus improving the upper bound and actually *halving* the current uncertainty interval.

The basic scheme above can perform poorly in its final iterations. Indeed, it may happen that x^* already belongs to P_1, but the search is not stopped because the internal point q is still far from x^*. We then propose a simple but quite effective modification of the original scheme where we just count the number of consecutive updates to q, say k, and separate directly x^* in case $k > 3$. If the separation is unsuccessful, then we can terminate the search, otherwise we reset counter k and continue with the usual strategy of cutting the middle point y.

As to the initialization of $q \in P_1$, this is a simple task in many practical settings, including the MIP applications where finding a feasible integer solution q is not difficult in practice.

3 Disjunctive Cuts

Consider the generic MIP of the previous section. To simplify notation, we concentrate on 0-1 MIPs where $l_j = 0$ and $u_j = 1$ for all $j \in I$. Our order of business is to optimize over the Lift-and-Project closure, say P_1, obtained from P by adding all linear inequalities valid for $P^j := conv(\{x \in P : x_j \leq 0\} \cup \{x \in P : x_j \geq 1\})$ for $j \in I$. To this end, given a point $x^* \in P$ (not necessarily a vertex), for each $j \in I$ with $0 < x^*_j < 1$ we construct a certain *Cut Generation Linear Program* (CGLP) whose solution allows us to detect a valid inequality for P^j violated by x^* (if any). Various CGLPs have been proposed in the literature; the one chosen for our tests has a size comparable with that of the original LP, whereas other versions require to roughly double this size. Given x^* and a disjunction $x_j \leq 0 \vee x_j \geq 1$ violated by x^*, our CGLP reads:

$$\max x_j - d^* \tag{1}$$

$$Ax = d^* b \tag{2}$$

$$d^* l \leq x \leq d^* l + (x^* - l) \tag{3}$$

$$d^* u - (u - x^*) \leq x \leq d^* u \tag{4}$$

where $d^* = x^*_j > 0$ (the two sets of bound constraints can of course be merged). Given the optimal dual multipliers $(\lambda, -\sigma'', \sigma', -\tau', \tau'')$ associated with the constraints of the CGLP, it is possible to derive a most-violated disjunctive cut $\gamma x \geq \gamma_0$, where $\gamma = \sigma' - \tau' - u_0 e_j$, $\gamma_0 = \sigma' l - \tau' u$, and $u_0 = 1 - \lambda b - (\sigma' - \sigma'') + (\tau' - \tau'')u$.

4 Computational Results (Sketch)

We implemented both the standard (`kelley`) and in-out (`in-out`) separation schemes and we compared them on a collection of 50 0-1 MIP instances from

MIPLIB 3.0 [15] and 2003 [16], and on 15 set covering instances from ORLIB [17]. We used IBM ILOG Cplex 11.2 as black-box LP solver, and to compute a first heuristic solution to initialize the in-out internal point q. Both schemes are given a time limit of 1 hour, and generate only one cut at each iteration–taken from the disjunction associated to the most fractional variable. Cumulative results are reported in Table 1, where `time` denotes the geometric mean of the computing times (CPU seconds on an Intel Q6600 PC running at 2.4 GHz), `itr` denotes the geometric mean of the number of iterations (i.e., cuts), `cl.gap` denotes the average gap closed w.r.t the best known integer solution, and `L&P cl.gap` denotes the average gap closed w.r.t. the best known upper bound on z_1 (this upper bound is obtained as the minimum between the best-known integer solution value and the last upper bound on z_1 computed by the in-out algorithm). The results clearly show the effectiveness of in-out search, in particular for set covering instances.

Table 1. Cumulative results on Lift-and-Project optimization

testbed	method	time (s)	itr	cl.gap	L&P cl.gap
MIPLIB	kelley	38.18	1,501	40.8%	63.7%
	in-out	28.42	592	41.2%	64.1%
set covering	kelley	2,281.60	16,993	35.2%	71.8%
	in-out	757.29	1,575	38.7%	85.8%

References

1. Ben-Ameur, W., Neto, J.: Acceleration of cutting-plane and column generation algorithms: Applications to network design. Networks 49(1), 3–17 (2007)
2. Cornuéjols, G.: Valid inequalities for mixed integer linear programs. Mathematical Programming 112(1), 3–44 (2008)
3. Cornuéjols, G., Lemaréchal, C.: A convex analysis perspective on disjunctive cuts. Mathematical Programming 106(3), 567–586 (2006)
4. Naoum-Sawaya, J., Elhedhli, S.: An interior-point branch-and-cut algorithm for mixed integer programs. Technical report, Department of Management Sciences, University of Waterloo (2009)
5. Kelley, J.E.: The cutting plane method for solving convex programs. Journal of the SIAM 8, 703–712 (1960)
6. Elzinga, J., Moore, T.J.: A central cutting plane algorithm for the convex programming problem. Mathematical Programming 8, 134–145 (1975)
7. Ye, Y.: Interior Point Algorithms: Theory and Analysis. John Wiley, New York (1997)
8. Tarasov, S., Khachiyan, L., Erlikh, I.: The method of inscribed ellipsoids. Soviet Mathematics Doklady 37, 226–230 (1988)
9. Bland, R.G., Goldfarb, D., Todd, M.J.: The ellipsoid method: a survey. Operations Research 29(6), 1039–1091 (1981)
10. Atkinson, D.S., Vaidya, P.M.: A cutting plane algorithm for convex programming that uses analytic centers. Mathematical Programming 69, 1–43 (1995)

11. Nesterov, Y.: Cutting plane algorithms from analytic centers: efficiency estimates. Mathematical Programming 69(1), 149–176 (1995)
12. Goffin, J.L., Vial, J.P.: On the computation of weighted analytic centers and dual ellipsoids with the projective algorithm. Mathematical Programming 60, 81–92 (1993)
13. Boyd, S., Vandenberghe, L.: Localization and cutting-plane methods (2007), http://www.stanford.edu/class/ee364b/notes/localization_methods_notes.pdf
14. Balas, E.: Disjunctive programming. Annals of Discrete Mathematics 5, 3–51 (1979)
15. Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.P.: An updated mixed integer programming library: MIPLIB 3.0. Optima 58, 12–15 (1998), http://www.caam.rice.edu/bixby/miplib/miplib.html
16. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Operations Research Letters 34(4), 1–12 (2006), http://miplib.zib.de
17. Beasley, J.: OR-Library: distributing test problems by electronic mail. Journal of the Operational Research Society 41(11), 1069–1072 (1990), http://people.brunel.ac.uk/~mastjjb/jeb/info.html

A SAT Encoding for Multi-dimensional Packing Problems

Stéphane Grandcolas and Cédric Pinto*

LSIS - UMR CNRS 6168,
Avenue Escadrille Normandie-Niemen,
13397 Marseille Cedex 20, France
{stephane.grandcolas,cedric.pinto}@lsis.org

Abstract. The Orthogonal Packing Problem (OPP) consists in determining if a set of items can be packed into a given container. This decision problem is NP-complete. Fekete et al. modelled the problem in which the overlaps between the objects in each dimension are represented by interval graphs. In this paper we propose a SAT encoding of Fekete et al. characterization. Some results are presented, and the efficiency of this approach is compared with other SAT encodings.

1 Introduction

The multi-dimensional Orthogonal Packing Problem (OPP) consists in determining if a set of items of known sizes can be packed in a given container. Although this problem is NP-complete, efficient algorithms are crucial since they may be used to solve optimization problems like the strip packing problem, the bin-packing problem or the optimization problem with a single container.

S. P. Fekete et al. introduced a new characterization for OPP [1]. For each dimension i, a graph G_i represents the items overlaps in the i^{th} dimension. In these graphs, the vertices represent the items. The authors proved that solving the d-dimensional orthogonal packing problem is equivalent to finding d graphs $G_1, \ldots, G_d$ such that **(P1)** each graph G_i is an interval graph , **(P2)** in each graph G_i, any stable set is i-feasible, that is the sum of the sizes of its vertices is not greater than the size of the container in dimension i, and **(P3)** there is no edge which occurs in each of the d graphs. They propose a complete search procedure [1] which consists in enumerating all possible d interval graphs, choosing for each edge in each graph if it belongs to the graph or not. The condition (P3) is always satisfied, forbidding the choice for any edge which occurs in d-1 graphs in the remaining graph. Each time a graph G_i is an interval graph, the i-feasibility of its stable sets is verified, computing its maximum weight stable set (the weights are the sizes of the items in the dimension i). As soon as the three conditions are satisfied the search stops and the d graphs represent then a class of equivalent solutions to the packing problem. Figure 1 shows an example in two dimensions with two packings among many others corresponding to the same pair of interval graphs.

There are very few SAT approaches for packing. In 2008 T. Soh et al. proposed a SAT encoding for the strip packing problem in two dimensions (SPP) [2]. This problem

* This work is supported by Region Provence-Alpes-Cote-d'Azur and the ICIA Technologies company. We also thank P. Jegou and D. Habet for helpfull discussions.

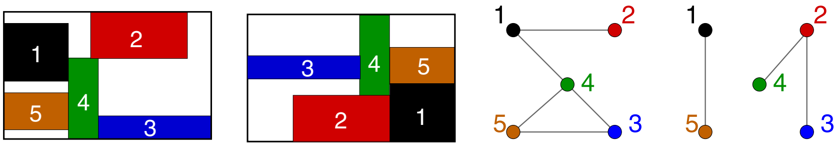


Fig. 1. Two packings corresponding to the same interval graphs in a two-dimensional space

consists in finding the minimal height of a fixed width container containing all the items. For that purpose they perform successive searches with different heights (selected with a dichotomy search strategy). Each time, the decision problem is encoded in a SAT formula which is solved with an external SAT solver (Minisat). In their formulation the variables represent the exact positions of the items in the container. Additional variables represent the relative positions of the items one with the others (on the left, on the right, above, under). T. Soh et al. also introduce constraints to avoid reconsidering symmetric equivalent packings. Finally the new clauses that the SAT solver Minisat generates to represent the conflicts are memorised and re-used in further searches. This is possible since successive searches are incremental SAT problems. T. Soh et al. SAT encoding involves $\mathcal{O}(W \times H \times n + n^2)$ Boolean variables for a problem with n items and a container of width W and height H.

2 A New SAT Encoding

We propose a new SAT encoding based on Fekete et al. characterization for the d-dimensional packing problem. Recall that each graph G_i must be an interval graph, and that if this is the case, then there exists a linear ordering of the maximal cliques of G_i such that each vertex occurs in consecutive cliques. This ordering is called *a consecutive linear ordering* and its size, the number of maximal cliques, is less then or equal to the number of items.

Basically, for each dimension i, Boolean variables indicate the presence of the edges in the graph G_i, that is the overlaps between the objects in dimension i. Furthermore, Boolean variables represent a linear clique decomposition of the graph G_i, ensuring that the graph is an interval graph if this decomposition is a consecutive linear ordering. The cliques are numbered from 1 to n. Then, Boolean variables indicate for each item and for each clique if the item occurs in the clique. Finally additional variables have been introduced to simplify the formulation of the constraints. The variables used in our formulation are defined as follows (note that some of these variables are not necessary in the basic formalisation of the packing problem):

$e^i_{x,y}$: **true** if the edge $\{x, y\}$ is in G_i,
$c^i_{x,a}$: **true** if item x is in clique a,
$p^i_{x,y,a}$: **true** if items x and y both occur in clique a,
u^i_a : **true** if clique a is not empty,

The stable set feasability of the graph G_i is verified with clauses that forbid the unfeasible stable sets. The set of all the unfeasible stable sets in dimension i is denoted S^i. Then the packing problem is encoded by the following formulas:

1. **[All objects are packed]**
   $x \in O, 1 \leq i \leq d,$
   $c^i_{x,1} \vee \ldots \vee c^i_{x,n}$
2. **[Consecutive linear ordering]**
   $x \in O, 1 \leq i \leq d, 1 \leq a < b - 1 < n,$
   $(c^i_{x,a} \wedge c^i_{x,b}) \Rightarrow c^i_{x,a+1}$
3. **[No-overlap Constraint]**
   $x, y \in O,$
   $\neg e^1_{x,y} \vee \ldots \vee \neg e^d_{x,y}$
4. **[Stable set feasibility]**
   $1 \leq i \leq d, N \in S^i,$
   $\bigvee_{x,\, y\, \in N} e^i_{x,y}$
5. **[No empty cliques]**
   $1 \leq i \leq d, 1 \leq a \leq n,$
   $(\neg c^i_{1,a} \wedge \ldots \wedge \neg c^i_{n,a}) \Rightarrow (\neg c^i_{1,a+1} \wedge \ldots \wedge \neg c^i_{n,a+1})$
6. **[Correlations between the variables]**
   $x, y \in O, 1 \leq a \leq n, 1 \leq i \leq d,$
   $p^i_{x,y,a} \Leftrightarrow (c^i_{x,a} \wedge c^i_{y,a})$ and $(p^i_{x,y,1} \vee \ldots \vee p^i_{x,y,k}) \Leftrightarrow e^i_{x,y}$

The formulas (1) force each item to occur in at least one clique, while the formulas (2) force each item to occur in consecutive cliques (Fekete et al. property $P1$: the graphs are interval graphs). The formulas (3) state that no two objects may intersect in all the dimensions (Fekete et al. property $P3$). The stable set feasability is enforced by the formulas (4): for each unfeasible stable set $N \in S^i$ in the dimension i, a clause ensures that at least two items of the stable set intersect each other. In fact only the minimal unfeasible stable sets are considered. For example, if two items x and y are too large to be packed side by side in the ith dimension, then $\{x, y\}$ is a stable set of S^i and the unit clause $e^i_{x,y}$ is generated. Then the SAT solver will immediately assign to the variable $e^i_{x,y}$ the value true and propagate it. The formulas (5) forbid empty cliques. Finally the formulas (6) establish the relations between the Boolean variables.

The following constraints are not necessary but they may help during the search:

7. **[Consective linear ordering (bis)]**
   $x \in O, 1 \leq a \leq n, 1 \leq i \leq d,$
   $(c^i_{x,a} \wedge \neg c^i_{x,a+1}) \Rightarrow (\neg c^i_{x,a+2} \wedge \ldots \wedge \neg c^i_{x,n})$
   $(c^i_{x,a} \wedge \neg c^i_{x,a-1}) \Rightarrow (\neg c^i_{x,a-2} \wedge \ldots \wedge \neg c^i_{x,1})$
8. **[Maximal cliques]**
   $1 \leq a \leq n, 1 \leq i \leq d,$
   $u^i_a \Leftrightarrow (c^i_{1,a} \vee \ldots \vee c^i_{n,a})$
   $(u^i_a \wedge u^i_{a+1}) \Rightarrow ((c^i_{1,a} \wedge \neg c^i_{1,a+1}) \vee \ldots \vee (c^i_{n,a} \wedge \neg c^i_{n,a+1}))$
   $(u^i_a \wedge u^i_{a+1}) \Rightarrow ((\neg c^i_{1,a} \wedge c^i_{1,a+1}) \vee \ldots \vee (\neg c^i_{n,a} \wedge c^i_{n,a+1}))$

9. **[Identical items ordering]**

$$x, y \in O, x \equiv y \text{ and } x \prec y, 1 \leq a < n, a < b \leq n$$
$$(c^{\delta}_{y,a} \wedge c^{\delta}_{x,b}) \Rightarrow c^{\delta}_{x,a}$$

The formulas (7) propagates the consecutive cliques ordering property, the formulas (8) forbid cliques which are not maximal, and the formulas (9) force identical objects to respect a given *a priori* ordering in only one dimension δ, so as to avoid the generation of equivalent permutations of these objects. This SAT encoding involves $\mathcal{O}(n^3)$ and $\mathcal{O}(n^4+2^n)$ clauses. However, since only the minimal unfeasible stable sets are encoded, in the general case there are much less than 2^n clauses of type (4).

3 Experimental Results

3.1 Orthogonal Packing Problem

The problem consists to determine if a given set of items may be packed into a given container. We have compared our approach with that Fekete et al. on a selection of two-dimensional problems, using as reference the results published by Clautiaux et al. [3]. Table 1 shows the characteristics of the instances, the results of Fekete et al. (**FS**), and the results of our approach with two modelisations: the modelisation **M1** corresponds to the formulas from (1) to (6) and (9), while the modelisation **M2** contains, furthermore, the facultative formulas (7) and (8). All of our experimentations were run on Pentium IV 3.2 GHz processors and 1 GB of RAM, using Minisat 2.0.

Table 1. Comparison with Fekete et al

Instance				FS	M1			M2		
Name	Space	Fais.	n	Time (s)	Time (s)	#var.	#claus.	Time (s)	#var.	#claus.
E02F17	02	*F*	17	7	**4.95**	5474	26167	13.9	6660	37243
E02F20	02	*F*	20	-	5.46	8720	55707	**1.69**	10416	73419
E02F22	02	*F*	22	167	**7.62**	11594	105910	21.7	13570	129266
E03N16	03	*N*	16	**2**	39.9	4592	20955	47.3	5644	30259
E03N17	03	*N*	17	**0**	4.44	5474	27401	9.32	6660	38477
E04F17	04	*F*	17	13	**0.64**	5474	26779	1.35	6660	37855
E04F19	04	*F*	19	560	3.17	7562	46257	**1.43**	9040	61525
E04F20	04	*F*	20	22	5.72	8780	59857	**2.22**	10416	77569
E04N18	04	*N*	18	**10**	161	6462	32844	87.7	7790	45904
E05F20	05	*F*	20	491	6.28	8780	59710	**0.96**	10416	77422
Average				> 217	23.9	7291	46159	**18.8**	8727	60894

Our approach outperforms FS on satisfiable instances, and even the instance E02F20 is not solved by Fekete et al. within the timeout (15 minutes). On unsatisfiable instances they have better performances, probably because they compute very relevant bounds (see DFF in [4]) which help them to detect dead ends during the search very early.

3.2 Strip Packing Problem

We have also compared our approach with Soh and al. on two-dimensional strip packing problems of the OR-Library available at `http://www.or.deis.unibo.it/`

Table 2. Results for OR-Library instances

Instance				Soh et	M1				M2			
Name	n	Width	LB	al.	Height	#var.	#claus.	Time	Height	#var.	#claus.	Time
HT01	16	20	20	**20**	**20**	4592	22963	13.3	**20**	5644	32267	19.4
HT02	17	20	20	**20**	**20**	5474	28669	744	**20**	6660	39745	444
HT03	16	20	20	**20**	**20**	4592	24222	18.5	**20**	5644	33526	25.5
HT04	25	40	15	**15**	16	16850	271500	1206	19	19396	305392	521
HT05	25	40	15	**15**	16	16850	337395	438	16	19396	372287	536
HT06	25	40	15	**15**	16	16850	494500	146	16	19396	528392	295
CGCUT01	16	10	23	**23**	**23**	4592	26745	5.89	**23**	5644	36049	9.71
CGCUT02	23	70	63	65	66	13202	115110	1043	70	15360	188222	1802
GCUT01	10	250	1016	**1016**	**1016**	1190	4785	0.11	**1016**	1606	7237	0.04
GCUT02	23	250	1133	1196	1259	8780	105810	37.3	1196	10416	123522	1241
NGCUT01	10	10	23	**23**	**23**	1190	5132	0.23	**23**	1606	7584	0.09
NGCUT02	17	10	30	**30**	**30**	5474	29662	1.6	**30**	6660	40738	2.74
NGCUT03	21	10	28	**28**	**28**	10122	108138	273	**28**	11924	128542	580
NGCUT04	7	10	20	**20**	**20**	434	1661	0.01	**20**	640	2577	0.01
NGCUT05	14	10	36	**36**	**36**	3122	15558	6.01	**36**	3930	21906	4.44
NGCUT06	15	10	31	**31**	**31**	3810	18629	1.92	**31**	4736	26361	2.91
NGCUT07	8	20	20	**20**	**20**	632	2535	0	**20**	900	3855	0
NGCUT08	13	20	33	**33**	**33**	2522	11870	2.74	**33**	3220	17010	9.73
NGCUT09	18	20	49	**50**	50	6462	33765	391	50	7790	46825	53.3
NGCUT10	13	30	80	**80**	**80**	2522	11790	0.75	**80**	3220	16930	0.39
NGCUT11	15	30	50	**52**	**52**	3810	18507	19.7	**52**	4736	26239	25.9
NGCUT12	22	30	79	**87**	**87**	11594	173575	886	**87**	13570	196931	24.5

`research.html`. The problem is to determine the minimal height of a fixed width container which may contain a given set of items. As Soh et al. we perform a sort of dichotomy search starting with a lower bound given by Martello and Vigo [5] and an upper bound which is calculated using a greedy algorithm. In table 2 we have reported the sizes of the encodings (numbers of variables and clauses) and the minimal height which was found within the timeout of 3600 seconds. Optimal heights are in bold (this occurs when the minimal height is equal to the lower bound or when the solver proves that there is no solution with a smaller height). Instances in which the number of items is large have been discarded, since the number of unfeasible stable sets becomes too important and so the number of corresponding clauses. Note that Soh and al. used also the solver Minisat. For 16 instances among 22 our system discovers the optimal height. Furthermore, among these 16 instances, 14 are solved in less than 30 seconds with one of our two modelisations. The ability of Soh and al. solver to reuse the conflict clauses that Minisat generates during the search is a real advantage since many unsuccessfull searches are then avoided.

4 Conclusions and Future Works

We have proposed a SAT encoding which outperforms significantly Fekete et al. method on satisfiable instances. Moreover, we have experimented this encoding on strip-packing problems. In future work we will try to integrate the DFF computation to improve the search on unsolvable problems. We will also try to characterize the situations in which the conflicts clauses which are generated by the SAT solver, may be re-used. This occurs in particular when successive calls to the solver are performed, for example when searching the minimal height in strip-packing problems.

References

1. Fekete, S.P., Schepers, J., van der Veen, J.: An exact algorithm for higher-dimensional orthogonal packing. Operations Research 55(3), 569–587 (2007)
2. Soh, T., Inoue, K., Tamura, N., Banbara, M., Nabeshima, H.: A SAT-based Method for Solving the Two-dimensional Strip Packing Problem. In: Proceedings of the 15th RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion (2008)
3. Clautiaux, F., Carlier, J., Moukrim, A.: A new exact method for the two-dimensional orthogonal packing problem. European Journal of Operational Research 183(3), 1196–1211 (2007)
4. Fekete, S.P., Schepers, J.: A general framework for bounds for higher-dimensional orthogonal packing problems. Mathematical Methods of Operations Research 60(2), 311–329 (2004)
5. Martello, S., Monaci, M., Vigo, D.: An exact approach to the strip-packing problem. Journal on Computing 15(3), 310–319 (2003)

Job Shop Scheduling with Setup Times and Maximal Time-Lags: A Simple Constraint Programming Approach

Diarmuid Grimes[1] and Emmanuel Hebrard[1,2]

[1] Cork Constraint Computation Centre & University College Cork, Ireland
{d.grimes,e.hebrard}@4c.ucc.ie
[2] LAAS-CNRS, Toulouse, France
hebrard@laas.fr

Abstract. In previous work we introduced a simple constraint model that combined generic AI strategies and techniques (weighted degree heuristic, geometric restarts, nogood learning from restarts) with naive propagation for job shop and open shop scheduling problems. Here, we extend our model to handle two variants of the job shop scheduling problem: job shop problems with setup times; and job shop problems with maximal time lags. We also make some important additions to our original model, including a solution guidance component for search.

We show empirically that our new models often outperform the state of the art techniques on a number of known benchmarks for these two variants, finding a number of new best solutions and proving optimality for the first time on some problems. We provide some insight into the performance of our approach through analysis of the constraint weighting procedure.

1 Introduction

Scheduling problems have proven fertile research ground for constraint programming and other combinatorial optimization techniques. There are numerous such problems occurring in industry, and whilst relatively simple in their formulation - they typically involve only *Sequencing* and *Resource* constraints - they remain extremely challenging to solve. After such a long period as an active research topic (more than half a century back to Johnson's seminal work [18]) it is natural to think that methods specifically engineered for each class of problems would dominate approaches with a broader spectrum. However, it was recently shown [27,15,26] that generic SAT or constraint programming models can approach or even outperform state of the art algorithms for open shop scheduling and job shop scheduling. In particular, in a previous work [15] we introduced a constraint model that advantageously trades inference strength for brute-force search speed and adaptive learning-based search heuristics combined with randomized restarts and a form of nogood learning.

Local search algorithms are generally the most efficient approach for solving job shop scheduling problems. The best algorithms are based on tabu search, e.g. i-TSAB [21], or use a CP/local search hybrid [29]. Pure CP approaches can also be efficient, especially when guided by powerful search strategies that can be thought of as metaheuristics [4]. The best CP approach uses inference from the *Edge-finding* algorithm

[8,22] and dedicated variable ordering heuristics such as *Texture* [3]. On the other hand, we take a minimalistic approach to modelling the problem. In particular, whilst most algorithms consider resource constraints as global constraints, devising specific algorithms to filter them, we simply decompose them into primitive disjunctive constraints ensuring that two tasks sharing a resource do not run concurrently. To this naive propagation framework, we combine slightly more sophisticated, although generic heuristics and restart policies. In this work, we have also incorporated the idea of solution guided search [4].

We showed recently that this approach can be very effective with respect to the state of the art. However, it is even more evident on variants of these archetypal problems where dedicated algorithms cannot be applied in a straightforward manner. In the first variant, running a task on a machine requires a setup time, dependent on the task itself, and also on the previous task that ran on the same machine. In the second variant, maximum time lags between the starting times of successive tasks of each job are imposed. In both cases, most approaches decompose the problem into two subproblems, for the former the traveling salesman problem with time windows [1,2] is used, while the latter can be decomposed into sequencing and timetabling subproblems [10]. On the other hand, our approach can be easily adapted to handle these additional constraints. Indeed, it found a number of new best solutions and proved optimality for the first time on some instances from a set of known benchmarks.

It may appear surprising that such a method, not reliant on domain specific knowledge, and whose components are known techniques in discrete optimization, could be so effective. We therefore devised some experiments to better understand how the key component of our approach, the constraint weighting, affects search on these problems. These empirical results reveal that although the use of constraint weighting is generally extremely important to our approach, it is not always so. In particular on *no-wait* job shop scheduling problems (i.e. problems with maximal time-lag of 0 between tasks), where our approach often outperforms the state of the art, the weight even seems to be detrimental to the algorithm.

In Section 2, we describe our approach. In Section 3, after outlining the experimental setup, we provide an experimental comparison of our approach with the state-of-the-art on standard benchmarks for these two problems. Finally we detail the results of our analysis of the impact of weight learning in these instances in Section 4.

2 A Simple Constraint Programming Approach

In this section we describe the common ground of constraint models we used to model the variants of JSP tackled in this paper. We shall consider the minimization of the total makespan (C_{max}) as the objective function in all cases.

2.1 Job Shop Scheduling Problem

An $n \times m$ job shop problem (JSP) involves a set of nm *tasks* $\mathcal{T} = \{t_i \mid 1 \leq i \leq nm\}$, partitioned into n *jobs* $\mathcal{J} = \{J_x \mid 1 \leq x \leq n\}$, that need to be scheduled on m *machines* $\mathcal{M} = \{M_y \mid 1 \leq y \leq m\}$. Each job $J_x \in \mathcal{J}$ is a set of m tasks $J_x =$

$\{t_{(x-1)*m+y} \mid 1 \leq y \leq m\}$. Conversely, each machine $M_y \in \mathcal{M}$ denotes a set of n tasks (to run on this machine) such that: $\mathcal{T} = (\bigcup_{1\leq x\leq n} J_x) = (\bigcup_{1\leq y\leq m} M_y)$.

Each task t_i has an associated duration, or processing time, p_i. A *schedule* is a mapping of tasks to time points consistent with: sequencing constraints which ensure that the tasks of each job run in a predefined order; and *resource* constraints which ensure that no two tasks run simultaneously on any given machine.

In this paper we consider the standard objective function defined as the minimization of the *makespan* C_{max}, that is, the total duration to run all tasks. If we identify each task t_i with its start time in the schedule, the job shop scheduling problem (JSP) can thus be written as follow:

$$(JSP)\ \ minimise\, C_{max}\ \ \text{subject to :}$$

$$C_{max} \geq t_i + p_i \qquad \forall t_i \in \mathcal{T} \tag{2.1}$$

$$t_i + p_i \leq t_{i+1} \qquad \forall J_x \in \mathcal{J},\ \forall t_i, t_{i+1} \in J_x \tag{2.2}$$

$$t_i + p_i \leq t_j\ \vee\ t_j + p_j \leq t_i \qquad \forall M_y \in \mathcal{M},\ t_i \neq t_j \in M_y \tag{2.3}$$

2.2 Constraint Model

The objective to minimise (total makespan) is represented by a variable C_{max} and the start time of each task t_i is represented by a variable $t_i \in [0, \ldots, max(C_{max}) - p_i]$. Next, for every pair of tasks t_i, t_j sharing a machine, we introduce a Boolean variable b_{ij} which represents the relative ordering between t_i and t_j. A value of 0 for b_{ij} means that task t_i precedes task t_j, whilst a value of 1 stands for the opposite ordering. The variables t_i, t_j and b_{ij} are linked by the following constraint:

$$b_{ij} = \begin{cases} 0 \Leftrightarrow t_i + p_i \leq t_j \\ 1 \Leftrightarrow t_j + p_j \leq t_i \end{cases}$$

Bounds consistency (BC) is maintained on these constraints. A *range support* of a constraint $C(x_1, \ldots, x_k)$ is an assignment of $\{x_1, \ldots, x_k\}$ satisfying C, and where the value assigned to each variable x_i is an integer taken in the interval $[min(x_i)..max(x_i)]$. A constraint $C(x_1, \ldots, x_k)$ is *bounds consistent* (BC) iff, for every variable x_i in the scope of C, $min(x_i)$ and $max(x_i)$ have a range support. Here, the scope of the constraint involves three variables, b_{ij}, t_i and t_j, therefore BC can be achieved in constant time for a single constraint, by applying simple rules. For n jobs and m machines, this model involves $nm(n-1)/2$ Boolean variables and as many ternary disjunctive constraints. Using an AC3 type constraint queue, the wort case time complexity for achieving bounds consistency on the whole network is therefore $O(C_{max}{*}nm(n-1)/2)$ since in the worst case bounds can be reduced by one unit at a time. For instance, consider three tasks t_i, t_j and t_k such that $p_i = p_j = p_k = 1$ and assume that $b_{ij} = b_{jk} = 0$ (hence $t_i \leq t_j \leq t_k$). Moreover, suppose that the domain of b_{ik} is reduced to the value 1, so that the cycle is closed. Since the domains are reduced by a constant amount at each propagation, the number of iterations necessary to obtain a failure is in $O(C_{max})$. However, it rarely reaches this bound in practice. Observe, moreover, that artificially increasing the size of the instance by a fixed amount will not affect the propagation loop as long as the durations increase proportionally to the horizon.

2.3 Search Strategy

We use the model described above in two different ways. Initially the lower bound on C_{max} is set to the duration of the longest job/machine, whilst the upper bound ub is initialised by a greedy algorithm in one case (Section 3.1), or by simply summing the durations of every task (Section 3.2). Since this starting upper bound is often very poor, especially in the latter case, we reduce the gap by performing a dichotomic search. We repeatedly solve the decision problem with a makespan fixed to $\frac{ub+lb}{2}$, updating lb and ub accordingly, until they have collapsed. Each dichotomic step has a fixed time cutoff, if the problem is unsolved the lb is updated, although not stored as the best proven lb. Moreover, we observed that in many cases, the initial upper bound is so overestimated that it helps to slightly bias the dichotomic pivot toward lower values until a first solution is found.

If the problem has not been solved to optimality during the dichotomic search, we perform a branch & bound search with the best makespan from the dichotmic search as our upper bound, and the best proven lb as our lower bound. Branch & bound search is performed until either optimality is proven or an overall cutoff is reached.

Branching: Instead of searching by assigning a starting time to a single value on the left branches, and forbidding this value on the right branches, it is common to branch on *precedences*. An unresolved pair of tasks t_i, t_j is selected and the constraint $t_i + p_i \leq t_j$ is posted on the left branch whilst $t_j + p_j \leq t_i$ is posted on the right branch. In our model, branching on the Boolean variables precisely simulates this branching strategy and thus significantly reduces the search space. Indeed, the existence of a partial ordering of the tasks (compatible with start times and durations, and such that its projection on any job or machine is a total order) is equivalent to the existence of a solution. In other words, if we successfully assign all Boolean variables in our model, the existence of a solution is guaranteed. Assigning each task variable to its lowest domain value gives the minimum C_{max} for this solution.

Variable Selection: We use the domain/weighted-degree heuristic [5], which chooses the variable minimising the ratio of current domain size to total weight of its neighboring constraints (initialised to 1). A constraint's weight is incremented by one each time the constraint causes a failure during search. It is important to stress that the behaviour of this heuristic is dependent on the modelling choices. Indeed, two different, yet logically equivalent, sets of constraints may distribute the weights differently. In this model, every constraint involves at most one search variable. Moreover, the relative light weight of the model allows the search engine to explore many more nodes than would a method relying on stronger inference, thus learning weights quicker.

However, at the start of the search, this heuristic is completely uninformed since every Boolean variable has the same domain size and the same degree. We therefore use an augmented version of the heuristic, where, instead of the domain size of b_{ij}, we use the domain size of the two associated task variables t_i, t_j. We denote $dom(t_i) = (max(t_i) - min(t_i) + 1)$ the domain size of task t_i, that is, the residual time windows of its starting time. Moreover, we denote $w(i, j)$ the number of times the search failed while propagating the constraint between t_i, t_j and b_{ij}. We choose the variable minimising the sum of the tasks' domain size divided by the weighted degree:

$$\frac{dom(t_i) + dom(t_j)}{w(i,j)} \tag{2.4}$$

Moreover, one can also use the weighted degree associated with the task variables. Let $\Gamma(t_j)$ denote the set of tasks sharing a resource with t_j. We call $w(t_j) = \sum_{t_i \in \Gamma(t_j)} w(i,j)$ the sum of the weights of every ternary disjunctive constraint involving t_j. Now we can define an alternative variable ordering as follows:

$$\frac{dom(t_i) + dom(t_j)}{w(t_i) + w(t_j)} \tag{2.5}$$

We refer to these heuristics as $tdom/bweight$ and $tdom/tweight$, $tdom$ refers to the sum of the domain sizes of the tasks associated with the Boolean variable, and $bweight$ ($tweight$) refers to the weighted degree of the Boolean (tasks). Ties were broken randomly.

Value Selection: Our value ordering is based on the solution guided approach (SGMPCS) proposed by Beck for JSPs [4]. This approach involves using previous solution(s) as guidance for the current search, intensifying search around a previous solution in a similar manner to i-TSAB [21]. In SGMPCS, a set of elite solutions is initially generated. Then, at the start of each search attempt, a solution is randomly chosen from the set and is used as a value ordering heuristic for search. When an improving solution is found, it replaces the solution in the elite set that was used for guidance. The logic behind this approach is its combination of intensification (through solution guidance) and diversification (through maintaining a set of diverse solutions).

Interestingly Beck found that the intensification aspect was more important than the diversification. Indeed, for the JSPs studied, there was little difference in performance between an elite set of size 1 and larger elite sets (although too large a set did result in a deterioration in performance). We use an elite set of 1 for our approach, i.e. once an initial solution has been found this solution is used, and updated, throughout our search.

Furthermore, up until the first solution is found during dichotomic search, we use a value ordering working on the principle of best *promise* [11]. The value 0 for b_{ij} is visited first iff the domain reduction directly induced by the corresponding precendence ($t_i + p_i \leq t_j$) is less than that of the opposite precedence ($t_j + p_j \leq t_i$).

Restart policy: It has previously been shown that randomization and restarts can greatly improve systematic search performance on combinatorial problems [12]. We use a geometric restarting strategy [28] with random tie-breaking. The geometric strategy is of the form $s, sr, sr^2, sr^3, ...$ where s is the base and r is the multiplicative factor. In our experiments the base was 64 failures and the multiplicative factor was 1.3. We also incorporate the nogood recording from restarts strategy of Lecoutre et al. [19], where nogoods are generated from the final search state when the cutoff has been reached. To that effect, we use a global constraint which essentially simulates the unit propagation procedure of a SAT solver. After every restart, for every minimal subset of decisions leading to a failure, the clause that prevents exploring the same path on subsequent restarts is added to the base. This constraint is not weighted when a conflict occurs.

3 Experimental Evaluation

We compare our model with state-of-the-art solvers (both systematic and non-sysytematic) on 2 variants of the JSP, job shop problems with sequence dependent setup times and job shop problems with time lags. All our experiments were run on an Intel Xeon 2.66GHz machine with 12GB of ram on Fedora 9. Due to the random component of our algorithm, each instance was solved ten times and we report our results in terms of both best and average makespan found per problem. Each algorithm run on a problem had an overall time limit of 3600s.

The number of algorithms we need to compare against makes it extremely difficult to run all experiments on a common setting.[1] We therefore decided to compare with the results taken from their associated papers. Since they were obtained on different machines with overall cutoffs based on different criteria, a direct comparison of cpu time is not possible. However, an improvement on the best known makespan is sufficient to observe that our approach is competitive. Therefore, we focus our analysis of the results on the objective value (although we do include average cpu time over the 10 runs for problems where we proved optimality).

3.1 Job Shop Scheduling Problem with Sequence Dependent Setup-Times

A job shop problem with sequence-dependent setup times, involves, as in a regular JSP, m machines and nm tasks, partitioned into n Jobs of m tasks. As for a JSP, the tasks have to run in a predefined order for every job and two tasks sharing a machine cannot run concurrently, that is, the starting times of these tasks should be separated by at least the duration of the first. However, for each machine and each pair of tasks running on this machine, the machine needs to be setup to accommodate the new task. During this setup the machine must stand idle. The duration of this operation depends on the sequence of tasks, that is, for every pair of tasks (t_i, t_j) running on the same machine we are given the setup time $s(i, j)$ for t_j following t_i and the setup time $s(j, i)$ for t_i following t_j. The setup times respect the triangular inequality, that is $\forall i, j, k \; s(i, j) + s(j, k) \geq s(i, k)$. The objective is to minimise the *makespan*. More formally:

$$(SDST - JSP) \;\; minimise \, C_{max} \;\; \text{subject to :}$$

$$C_{max} \geq t_i + p_i \qquad \forall t_i \in \mathcal{T} \tag{3.1}$$

$$t_i + p_i \leq t_{i+1} \qquad \forall J_x \in \mathcal{J}, \; \forall t_i, t_{i+1} \in J_x \tag{3.2}$$

$$t_i + p_i + s_{i,j,y} \leq t_j \;\vee\; t_j + p_j + s_{j,i,y} \leq t_i \qquad \forall M_y \in \mathcal{M}, \; \forall t_i \neq t_j \in M_y \tag{3.3}$$

State of the art: This problem represents a challenge for CP and systematic approaches in general, since the inference from the Edge-finding algorithm is seriously weakened as it cannot easily take into account the setup times. Therefore there are two main approaches to this problem. The first by Artigues *et al.* [1] (denoted AF08 in Table 1) tries to adapt the reasoning for simple unary resources to unary resources with setup times. The approach relies on solving a TSP with time windows to find the shortest permutation of tasks, and is therefore computationally expensive.

[1] The code may be written for different OS, not publicly available, or not open source.

Table 1. SDST-JSP: Comparison vs state-of-the-art (best & mean C_{max}, 10 runs)

Instance	AF08	BSV08	GVV08		GVV09		*tdom/bweight*		
	Best	Best	Best	Avg	Best	Avg	Best	Avg	Time
t2-ps01	**798**	**798**	**798**	**798**			**798**	798.0	0.1
t2-ps02	**784**	**784**	**784**	**784**			**784**	784.0	0.2
t2-ps03	**749**	**749**	**749**	**749**			**749**	749.0	0.2
t2-ps04	**730**	**730**	**730**	**730**			**730**	730.0	0.1
t2-ps05	**691**	693	**691**	692			**691**	691.0	0.1
t2-ps06	**1009**	1018	1026	1026			**1009**	1009.0	20.3
t2-ps07	**970**	1003	**970**	971			**970**	970.0	46.1
t2-ps08	**963**	975	**963**	966			**963**	963.0	86.1
t2-ps09	1061	**1060**	**1060**	**1060**			**1060***	1060.0	1025.1
t2-ps10	**1018**	**1018**	**1018**	**1018**			**1018**	1018.0	11.0
t2-ps11	1494	1470	**1438**	1439	**1438**	1441	1443	1463.6	-
t2-ps12	1381	1305	**1269**	1291	**1269**	1277	**1269**	1322.2	-
t2-ps13	1457	1439	**1406**	1415	1415	1416	1415	1428.8	-
t2-ps14	1483	1485	**1452**	1489	**1452**	1489	**1452**	1470.5	-
t2-ps15	1661	1527	**1485**	1502	**1485**	1496	1486	1495.8	-
t2-pss06		1126					**1114***	1114.0	600.9
t2-pss07		1075					**1070***	1070.0	274.1
t2-pss08		1087					**1072***	1073.0	-
t2-pss09		1181					**1161***	1161.0	-
t2-pss10		1121					**1118***	1118.0	47.2
t2-pss11		1442					**1412***	1425.9	-
t2-pss12		1290			**1258**	1266	1269	1287.6	-
t2-pss13		1398			**1361**	1379	1365	1388.0	-
t2-pss14		1453					**1452***	1453.0	-
t2-pss15		1435					**1417***	1427.4	-

The second type of approach relies on metaheuristics. Balas *et al.* [2] proposed combining a shifting bottleneck algorithm with guided local search (denoted BSV08 in Table 1[2]), where the problem is also decomposed into a TSP with time windows. Hybrid genetic algorithms have also been proposed by González *et al.* for this problem, firstly a hybrid GA with local search [13] and more recently GA combined with tabu search [14] (denoted GVV08 and GVV09 *resp.* in Table 1). For both GA hybrids, the problem is modeled using the disjunctive graph representation.

Specific Implementation Choices: Our model is basically identical to the generic scheduling model introduced in Section 2. However, the setup time between two tasks is added to the duration within the disjunctive constraints. That is, given two tasks t_i and t_j sharing a machine, let $s_{i,j}$ (resp. $s_{j,i}$) be the setup time for the transition between t_i and t_j (resp. between t_j and t_i), we replace the usual disjunctive constraint with:

$$b_{ij} = \begin{cases} 0 \Leftrightarrow t_i + p_i + s_{i,j} \leq t_j \\ 1 \Leftrightarrow t_j + p_j + s_{j,i} \leq t_i \end{cases}$$

Evaluation: Table 1 summarizes the results of the state-of-the-art and our approach on a set of benchmarks proposed by Brucker and Thiele [7]. The problems are grouped based on the number of jobs and machines (nxm), *01-05 are of size 10x5, *06-10 are of size 15x5, while *11-15 are of size 20x5. Each step of the dichotomic search had a 30 second cutoff, the search heuristic used was *tdom/bweight*. We use the following

[2] Results for t2-pss-*06-11 and 14-15 are from
`http://www.andrew.cmu.edu/user/neils/tsp/outt2.txt`

notation for Table 1 (we shall reuse it for Tables 3 and 4): underlined values denote the fact that optimality was proven, bold face **values** denote the best value achieved by any method and finally, values* marked with a star denote instances where our approach improved on the best known solution or built the first proof of optimality. We also include the average time over the 10 runs when optimality was proven (a dash means optimality wasn't proven before reaching the 1 hour cutoff).

We report the first proof of optimality for four instances (`t2-ps09, t2-pss06, t2-pss07, t2-pss10`) and 8 new upper bounds for t2-pss* instances (however it should be noted that there is no comparison available for GVV09 on these 8 instances). In general, our approach is competitive with the state-of-the-art (GVV09) and outperforms both dedicated systematic and non-systematic solvers.

3.2 Job Shop Scheduling Problem with Time Lags

An $n \times m$ job shop problem with time lags (JTL) involves the same variables and constraints as a JSP of the same order. However, there is an additional upper bound on the time lag between every pair of successive tasks in every job. Let l_i denote the maximum amount of time allowed between the completion of task t_i and the start of task t_j. More formally:

$$(TL - JSP) \;\; minimise\, C_{max} \;\; \text{subject to :}$$
$$C_{max} \geq t_i + p_i \qquad \forall t_i \in \mathcal{T} \tag{3.4}$$
$$t_i + p_i \leq t_{i+1} \qquad \forall J_x \in \mathcal{J},\ \forall t_i, t_{i+1} \in J_x \tag{3.5}$$
$$t_{i+1} - (p_i + l_i) \leq t_i \qquad \forall J_x \in \mathcal{J},\ \forall t_i, t_{i+1} \in J_x \tag{3.6}$$
$$t_i + p_i \leq t_j \;\vee\; t_j + p_j \leq t_i \qquad \forall M_y \in \mathcal{M},\ \forall t_i \neq t_j \in M_y \tag{3.7}$$

This type of constraint arises in many situations. For instance, in the steel industry, the time lag between the heating of a piece of steel and its moulding should be small. Similarly when scheduling chemical reactions, the reactives often cannot be stored for a long period of time between two stages of a process to avoid interactions with external elements. This type of problem has been studied in a number of areas including the steel and chemical industries [24].

State of the art: Caumond *et al.* introduced in 2008 a genetic algorithm able to deal with general time lag constraints [9]. However most of the algorithms introduced in the literature have been designed for a particular case of this problem: the *no-wait* job shop. In this case, the maximum time-lag is null, i.e. each task of a job must start directly after its preceding task has finished.

For the no-wait job shop problem, the best methods are a tabu search method by Schuster (TS [25]), another metaheuristic introduced by Framinian and Schuster (CLM [10]) and a hybrid constructive/tabu search algorithm introduced by Bożejko and Makuchowski in 2009 (HTS [6]). We report the best results of each paper. It should be noted that for HTS, the authors reported two sets of results, the ones we report for the "hard" instances were "without limit of computation time".

Table 2. Results summary for JTL- and NW-JSP

(a) JTL-JSP: C_{max} & Time

Instance Sets	CLT		*tdom/bweight*	
	C_{max}	Time	C_{max}	Time
car[5-8]_0_0,5	**7883.25**	322.19	**7883.25**	2.16
car[5-8]_0_1	**7731.25**	273.75	**7731.25**	4.16
car[5-8]_0_2	**7709.25**	297.06	**7709.25**	6.31
la[06-08]_0_0,5	1173.67	2359.33	**980.00**	2044.77
la[06-08]_0_1	1055.33	1870.92	**905.33**	2052.41
la[06-08]_0_2	1064.33	1853.67	**904.67**	2054.81

(b) NW-JSP: Summary of APRD per problem set

Instance	TS	HTS	CLM	CLT	tdom/ twdeg	tdom
ft	-8.75	**-10.58**			**-10.58**	-9.79
abz	-20.77	-25.58			**-25.89**	-25.1
orb	2.42	0.77	1.44		**0.00**	**0.00**
la01-10	4.43	1.77	3.31	4.53	**0.00**	**0.00**
la11-20	9.52	-5.40	5.14	29.14	-6.32	**-6.36**
la21-30	-33.93	**-39.96**	-34.62		-39.85	-39.04
la31-40	-36.69	**-42.39**	-36.87		-41.65	-40.36
swv01-10	-34.41	**-37.22**	-34.39		-36.88	-35.33
swv11-20	-40.62	**-42.25**			-39.17	-33.87
yn	-34.87	**-41.84**			-38.78	-39.03

Specific Implementation Choices: The constraint to represent time lags between two tasks of a job are simple precedences in our model. For instance, a time lag l_i between t_i and t_{i+1}, will be represented by the following constraint: $t_{i+1} - (p_i + l_i) \leq t_i$.

Although our generic model was relatively efficient on these problems, we made a simple improvement for the no-wait class based on the following observation: if no delay is allowed between any two consecutive tasks of a job, then the start time of every task is functionally dependent on the start time of any other task in the job. The tasks of each job can thus be viewed as one block. In other words we really need only one task in our model to represent all the tasks of a job. We therefore use only n variables standing for the jobs: $\{J_x \mid 1 \leq x \leq n\}$.

Let h_i be the total duration of the tasks coming before task t_i in its job. That is, if job $J = \{t_1, \dots, t_m\}$, we have: $h_i = \sum_{k<i} p_k$. For every pair of tasks $t_i \in J_x, t_j \in J_y$ sharing a machine, we use the same Boolean variables to represent disjuncts as in the original model, however linked by the following constraints:

$$b_{ij} = \begin{cases} 0 \Leftrightarrow J_x + h_i + p_i - h_j \leq J_y \\ 1 \Leftrightarrow J_y + h_j + p_j - h_i \leq J_x \end{cases}$$

Notice that while the variables and constants are different, these are still exactly the same ternary disjuncts used in the original model.

The no-wait job shop scheduling problem can therefore be reformulated as follows, where the variables $J_1, \dots, J_n$ represent the start time of the jobs, $J_{x(i)}$ stands for the job of task t_i, and $f(i,j) = h_i + p_i - h_j$.

$$(NW - JSP) \;\; minimise\, C_{max} \;\text{ subject to :}$$

$$C_{max} \geq J_x + \sum_{t_i \in J_x} p_i \qquad \forall J_x \in \mathcal{J} \tag{3.8}$$

$$J_{x(i)} + f(i,j) \leq J_{x(j)} \vee J_{x(j)} + f(j,i) \leq J_{x(i)} \qquad \forall M_y \in \mathcal{M},\ t_i, t_j \in M_y \tag{3.9}$$

Evaluation: On general JTL problems, it is difficult to find comparable results in the literature. To the best of our knowledge, the only one available is the genetic algorithm by Caumond *et al.* [9] that we shall denote CLT. In Table 2a, we report the results from our model on the instances used in that paper, where instances are grouped based on type (*car* (4 instances) / *la* (3 instances)) and maximum time lag (0.5 / 1 / 2).

For the no-wait job shop problem, we first present our results in terms of each solver's average percentage relative deviation (PRD) from the reference values given in [6] per problem set in Table 2b. The PRD is given by the following formula:

$$PRD = ((C_{Alg} - C_{Ref})/C_{Ref}) * 100 \qquad (3.10)$$

where C_{Alg} is the best makespan found by the algorithm and C_{Ref} is the reference makespan for the instance given in [6]. There are 82 instances overall.

Interestingly, the search heuristic *tdom/tweight* performed much better with our no-wait model than *tdom/bweight*, thus we report the results for this heuristic. This was somewhat surprising because this heuristic is less discriminatory as the task weights for a Boolean are the weights of the two jobs, which will be the same for all Booleans between these two jobs. Further investigation revealed that ignoring the weight yielded better results on a number of problems. Thus we also include the heuristic *tdom*.

Our approach was better than the local search approaches on the smaller problem sets, and remained competitive on the larger problem sets. In Table 3 we provide results for the instances regarded as easy in [6], these had been proven optimal by Mascis [20].

Table 3. NW-JSP: Comparison vs state-of-the-art on *easy* instances (best & mean C_{max}, 10 runs).

Instance	Size	Ref	TS	HTS	CLM	CLT	*tdom/tweight*			*tdom*		
	nxm		Best	Best	Best	Best	Best	Avg	Time	Best	Avg	Time
ft06	6x6	73	**73**	**73**	**73**		**73**	73	0.01	**73**	73	0.02
ft10	10x10	1607	1620	**1607**	1619		**1607**	1607	4.08	**1607**	1607	2.49
abz5		2150	2233	2182			**2150**	2150	9.28	**2150**	2150	8.87
abz6		1718	1758	1760			**1718**	1718	1.25	**1718**	1718	0.71
orb01		1615	1663	1615	1646		**1615**	1615	1.65	**1615**	1615	1.45
orb02		1485	1555	1518	1518		**1485**	1485	1.16	**1485**	1485	1.12
orb03		1599	1603	**1599**	1603		**1599**	1599	4.22	**1599**	1599	3.10
orb04		1653	**1653**	**1653**	**1653**		**1653**	1653	1.56	**1653**	1653	1.11
orb05		1365	1415	1367	1371		**1365**	1365	3.91	**1365**	1365	4.43
orb06		1555	**1555**	1557	**1555**		**1555**	1555	0.31	**1555**	1555	0.26
orb07		689	706	717	706		**689**	689	6.10	**689**	689	3.34
orb08		1319	**1319**	**1319**	**1319**		**1319**	1319	2.22	**1319**	1319	2.12
orb09		1445	1535	1449	1515		**1445**	1445	1.02	**1445**	1445	0.68
orb10		1557	1618	1571	1592		**1557**	1557	4.55	**1557**	1557	4.78
la01	10x5	971	1043	975	1031	975	**971**	971	0.13	**971**	971	0.11
la02		937	990	975	**937**	**937**	**937**	937	0.24	**937**	937	0.19
la03		820	832	**820**	832	**820**	**820**	820	0.14	**820**	820	0.15
la04		887	889	889	889	911	**887**	887	0.28	**887**	887	0.17
la05		777	817	**777**	797	818	**777**	777	0.30	**777**	777	0.22
la06	15x5	1248	1299	**1248**	1256	1305	**1248**	1248	115.19	**1248**	1248	81.70
la07		1172	1227	**1172**	1253	1282	**1172**	1172	66.96	**1172**	1172	57.30
la08		1244	1305	1298	1307	1312	**1244**	1244	50.35	**1244**	1244	38.63
la09		1358	1450	1415	1451	1547	**1358**	1358	181.55	**1358**	1358	102.10
la10		1287	1338	1345	1328	1333	**1287**	1287	54.14	**1287**	1287	30.78
la16	10x10	1575	1637	**1575**	1637	1833	**1575**	1575	2.09	**1575**	1575	1.37
la17		1371	1430	1384	1389	1591	**1371**	1371	2.34	**1371**	1371	1.70
la18		1417	1555	**1417**	1555	1790	**1417**	1417	1.38	**1417**	1417	1.31
la19		1482	1610	1491	1572	1831	**1482**	1482	3.14	**1482**	1482	3.08
la20		1526	1705	**1526**	1580	1828	**1526**	1526	0.70	**1526**	1526	0.66

Table 4. NW-JSP: Improvement on *hard* instances (best & mean C_{max}, 10 runs)

Instance	Size	Ref	TS	HTS	CLM	$tdom/tweight$			$tdom$		
	nxm		Best	Best	Best	Best	Avg	Time	Best	Avg	Time
swv06	20x15	3291	3502	3290	3291	**3278***	3378.0	-	3391	3500.4	-
la11	20x5	2821	1737	1621	1714	**1619***	1646.9	-	1622	1632.2	-
la12		2434	1550	1434	1507	**1414**	1432.7	-	**1414***	1414.0	2892.37
la14		2662	1771	1610	1773	**1578***	1628.5	-	**1578***	1611.1	-
la15		2765	1808	1686	1771	**1679***	1693.2	-	1681	1691.9	-
la21	15x10	2092	2242	**2030**	2149	**2030**	2030.0	-	**2030***	2030.0	579.69
la22		1928	2008	**1852**	1979	**1852**	1854.3	-	**1852**	1852.0	1013.45
la23		2038	2093	**2021**	2038	**2021**	2033.2	-	**2021**	2021.0	1160.13
la24		2061	2061	**1972**	2133	**1972**	1982.7	-	**1972**	1972.0	1128.55
la25	20x10	2034	2072	**1906**	2050	**1906**	1906.0	1336.92	**1906**	1906.0	218.60
la27		2933	2968	2675	2933	**2671***	2750.3	-	2675	2743.0	-
la36	15x15	2810	2993	**2685**	2810	**2685**	2715.5	-	**2685**	2685.0	1530.39
la37		3044	3171	**2831**	3161	2937	2974.0	-	**2831**	2930.4	-
la38		2726	2734	**2525**	2726	**2525**	2556.9	-	**2525**	2525.0	2898.77
la39		2752	2804	2687	2784	**2660***	2686.0	-	**2660***	2662.7	3564.28
la40		2838	2977	2580	2880	**2564***	2660.8	-	**2564***	2591.9	2879.08

We proved optimality on all these instances, in under 10s for most cases. It is of interest to note that $tdom$ was nearly always quicker than $tdom/tweight$ at proving optimality. In Table 4, we report results for the "hard" instances where our approach found an improving solution, and the first proofs of optimality for 10 (la12, la21-25, la36 and la38-40) of the 53 open problems.

4 Weight Learning Analysis

We have previously shown that the *weighted degree* is a key element of our approach [16]. In particular the gap in performance between $tdom/bwdeg$ and $tdom$ was quite large for open shop scheduling problems. Here we try to give a more precise characterization of the importance of learning weights, by gradually reducing the influence of these weights in the variable selection heuristic. We observe that the impact of the weights is very much problem-dependent. It is extremely important for job shop with setup times model and for the standard model for job shop with time lags. However, for the specific model for no-wait job shop problems, it can be detrimental in some cases.

4.1 Evaluation of Weighted Degree

In order to evaluate the effect of weight learning on search, we devised the following variable ordering heuristic, that we denote $tdom/(K + bweight)$, and that selects first the variable b_{ij} minimising the value of:

$$\frac{dom(t_i) + dom(t_j)}{w(i,j) + K} \tag{4.1}$$

Observe that when $K = 0$, this heuristic is equivalent to $tdom/(bweight)$, whereas, when K tends toward infinity, the weights become insignificant in the variable selection. For $K = \infty$ the next variable is selected with respect to $tdom$ only.

We can therefore tune the impact of the weights in the variable choice, by setting the constant K. As K increases, the role of the weights is increasingly restricted to a tie breaker. We selected a subset of instances small enough to be solved by $tdom/(\infty + bweight)$. For the selected subset of small instances, we ran each version of the heuristic ten times with different random seeds. We report the average cpu time across the ten runs in Table 5. When the run went over a one hour time cutoff, we report the deviation to the optimal solution (in percentage) instead.

Table 5. Weight evaluation: cpu-time or deviation to the optimal for increasing values of K

Instance	$tdom/(K + bweight)$						
	$K = 0$	$K = 10$	$K = 100$	$K = 1000$	$K = 10000$	$K = 100000$	$K = \infty$
`t2-ps07`	26.55	**23.33**	26.67	41.60	77.27	403.90	*+12.9%*
`t2-ps08`	41.08	**35.85**	93.60	128.96	194.96	665.28	*+9.8%*
`t2-ps09`	971.83	956.63	**948.28**	957.85	1164.94	1649.19	*+8.8%*
`t2-ps10`	**13.04**	13.95	13.63	19.44	100.25	422.24	*+15.7%*
`la07_0_3`	*+0.0%*	*+0.0%*	*+0.0%*	*+0.0%*	*+0.0%*	*+0.0%*	*+5.8%*
`la08_0_3`	15.63	**12.45**	23.03	30.22	117.50	391.99	3098.87
`la09_0_3`	1.61	**0.51**	1.44	10.16	129.62	169.02	2115.98
`la10_0_3`	3.42	2.25	**0.41**	0.69	1.39	3.44	39.66
`la07_0_0`	1751.16	549.58	392.71	151.70	66.18	**49.67**	57.28
`la08_0_0`	2231.18	575.44	309.04	113.95	42.04	**35.74**	38.63
`la09_0_0`	2402.76	1291.29	691.96	407.68	147.73	**89.28**	102.03
`la10_0_0`	3274.86	833.28	214.51	53.75	26.85	**26.51**	30.82

For job shop with setup times, the best compromise is for $K = 10$. For very large values of K, the domain size of the tasks takes complete precedence on the weights, and the performance degrades. However, as long as the weights are present in the selection process, even simply as tie breaker, the cpu time stays within one order of magnitude from the best value for K. On the other hand, when the weights are completely ignored, the algorithm is not able to solve any of the instances. Indeed the gap to optimality is quite large, around 9% to 15%.

For job shop with time lags, the situation is a little bit different. As in the previous case, the best compromise is for $K = 10$ and the performance degrades slowly when K increases. However, even when the weights are completely ignored, the gap stays within a few orders of magnitude from the best case. Finally, for the no-wait job shop, we observe that the opposite is true. Rather than increasing with K, the cpu time actually *decreases* when K grows.

One important feature of a heuristic is its capacity to focus the search on a small subset of variables that would constitute a backdoor of the problem. It is therefore interesting to find out if there is a correlation between a high level of inequality in the weight distribution and the capacity to find small backdoors. We used the *Gini* coefficient to characterize the weight distribution. The Gini coefficient is a metric of inequality, used for instance to analyse distribution of wealth in social science.

The Gini coefficient is based on the *Lorenz* curve, mapping the cumulative proportion of income y of a fraction x of the poorest population. When the distribution is perfectly fair, the Lorenz curve is $y = x$. The Gini coefficient is the ratio of the area lying between the Lorenz curve and $x = y$, over the total area below $x = y$.

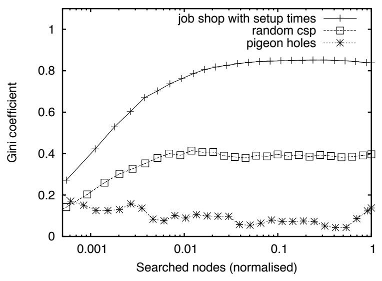
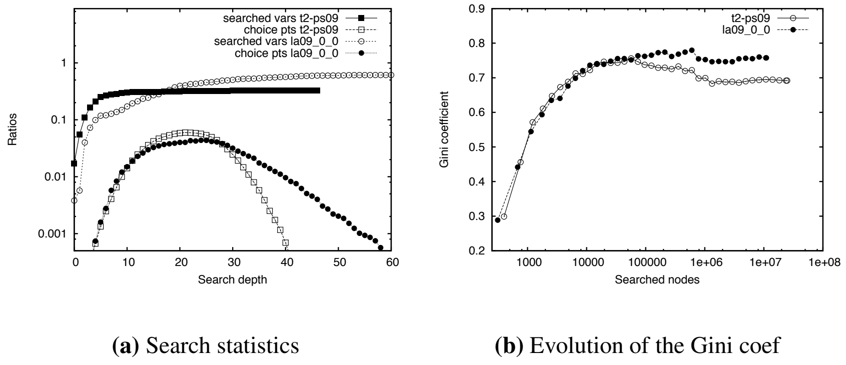


Fig. 1. Weight distribution bias: Gini coefficient over the (normalised) number of searched nodes

We consider only search trees for unsatisfiable instances. In an ideal situation, when the search converges immediately toward a given set of variables from which a short proof of unsatisfiability can be extracted, the Gini coefficient of the weight distribution typically increases rapidly and monotonically. In Figure 1 we plot the Gini coefficient of the proofs for the instance `t2-ps07`; for an instance of random CSP with 100 variables, a domain size of 15, 250 binary constraints of tightness 0.53 uniformly distributed; and a pigeon holes instance. After each geometric restart, the Gini coefficient is computed and plotted against the current number of explored nodes. We observe that the weight distribution is quickly and significantly biased on the job shop instance. On the other hand, there is much less discrimination on the random CSP instance, where constraints are uniformly distributed, and almost no discrimination at all on the pigeon hole problem. We were interested in checking if one could predict, from the fairness of the weight distribution, how beneficial the weighted degree heuristic is for the considered problem. However, when comparing two proofs that required a comparably large amount of search, but for which we showed that, in one case the weights are beneficial, and in the other case detrimental, it is in fact extremely difficult to differentiate the evolution of the coefficient. It took 11 million nodes to prove that $C_{max} = 1357$ is unsatisfiable for `la09_0_0` and 24 million nodes to prove that $C_{max} = 1059$ is unsatisfiable for `t2-ps09`. It is clear from the results in Table 5 however, that the weights helped in the latter case, whereas they did not in the former case. We report two statistics collected during search showing some clear differences: the ratio of (Boolean) variables that are selected at a choice point up to each depth in the search tree, over the total number of (Boolean) variables; the ratio



(a) Search statistics **(b)** Evolution of the Gini coef

Fig. 2. Search tree and weight distribution for `t2-ps09` and `la09_0_0`

of the number of choice points, that is nodes of the search tree, at each depth, over the total number of explored nodes.

Clearly for `t2-ps09`, where the weights are useful, the search is more focused on lower depth, and on a smaller ratio of variables. Indeed, the cumulative ratio of searched variables tops at 0.3 (See Figure 2a). On the other hand, for `la09_0_0`, even very deep in the tree, new choice points are opened (the ratio of choice points is more spread out), and they involve a large proportion of new variables (the cumulative number of searched variables increases almost linearly up to 0.6). The evolution of the Gini coefficient during search is, however, very similar in both cases (See Figure 2b).

One possibility is that the build up of contention is more important for the no wait problems due to the stronger propagation between tasks of the one job. Preliminary results suggest that initially both $tdom$ and $tdom/bweight$ repeatedly select Booleans between the same pair of jobs, once a pair has been selected. The heuristics diverge when search backs up from deep in search, $tdom$ will still often choose Booleans from the same pair of jobs as the variable above the choice point, while the weights learnt deep in search may result in the heuristics that use $bweight$ and $tweight$ choosing variables associated with a different pair of jobs. Obviously, this effect will be stronger for $bweight$ as the weights are associated with individual Booelans.

5 Conclusions

We have shown how our constraint model can be easily extended to handle two variants of the job shop scheduling problem. In both cases we found our approach to be competitive with the state-of-the-art, most notably in proving optimality on some of the open problems of both problem types.

Whereas it appeared to uniformly improve search efficiency for standard job shop and open shop scheduling problems, our analysis of constraint weighting revealed that it can actually be detrimental for some variants of these problems.

References

1. Artigues, C., Feillet, D.: A branch and bound method for the job-shop problem with sequence-dependent setup times. Annals OR 159(1), 135–159 (2008)
2. Balas, E., Simonetti, N., Vazacopoulos, A.: Job shop scheduling with setup times, deadlines and precedence constraints. J. of Scheduling 11(4), 253–262 (2008)
3. Beck, J.C., Davenport, A.J., Sitarski, E.M., Fox, M.S.: Texture-Based Heuristics for Scheduling Revisited. In: AAAI 1997, pp. 241–248 (1997)
4. Christopher Beck, J.: Solution-Guided Multi-Point Constructive Search for Job Shop Scheduling. Journal of Artificial Intelligence Research 29, 49–77 (2007)
5. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting Systematic Search by Weighting Constraints. In: ECAI 2004, pp. 482–486 (2004)
6. Bozejko, W., Makuchowski, M.: A fast hybrid tabu search algorithm for the no-wait job shop problem. Computers & Industrial Engineering 56(4), 1502–1509 (2009)
7. Brucker, P., Thiele, O.: A branch and bound method for the general- shop problem with sequence-dependent setup times. Operation Research Spektrum 18, 145–161 (1996)

8. Carlier, J., Pinson, E.: An Algorithm for Solving the Job-shop Problem. Management Science 35(2), 164–176 (1989)
9. Caumond, A., Lacomme, P., Tchernev, N.: A memetic algorithm for the job-shop with time-lags. Computers & OR 35(7), 2331–2356 (2008)
10. Framinan, J.M., Schuster, C.J.: An enhanced timetabling procedure for the no-wait job shop problem: a complete local search approach. Computers & OR 33, 1200–1213 (2006)
11. Geelen, P.A.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: Proc. Tenth European Conference on Artificial Intelligence, ECAI 1992, pp. 31–35 (1992)
12. Gomes, C.P., Selman, B., Kautz, H.: Boosting combinatorial search through randomization. In: AAAI 1998, pp. 431–437 (1998)
13. González, M.A., Vela, C.R., Varela, R.: A new hybrid genetic algorithm for the job shop scheduling problem with setup times. In: ICAPS, pp. 116–123. AAAI, Menlo Park (2008)
14. González, M.A., Vela, C.R., Varela, R.: Genetic algorithm combined with tabu search for the job shop scheduling problem with setup times. In: Mira, J., Ferrández, J.M., Álvarez, J.R., de la Paz, F., Toledo, F.J. (eds.) IWINAC 2009. LNCS, vol. 5601, pp. 265–274. Springer, Heidelberg (2009)
15. Grimes, D., Hebrard, E., Malapert, A.: Closing the Open Shop: Contradicting Conventional Wisdom. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 400–408. Springer, Heidelberg (2009)
16. Grimes, D., Hebrard, E., Malapert, A.: Closing the Open Shop: Contradicting Conventional Wisdom on Disjunctive Temporal Problems. In: 14th ERCIM International Workshop on Constraint Solving and Constraint Logic Programming, CSCLP 2009 (2009)
17. Hodson, A., Muhlemann, A.P., Price, D.H.R.: A microcomputer based solution to a practical scheduling problem. The Journal of the Operational Research Society 36(10), 903–914 (1985)
18. Johnson, S.M.: Optimal two- and three-stage production schedules with setup times included. Naval Research Logistics Quarterly 1(1), 61–68 (1954)
19. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Nogood Recording from Restarts. In: IJCAI 2007, pp. 131–136 (2007)
20. Mascis, A., Pacciarelli, D.: Job-shop scheduling with blocking and no-wait constraints. European Journal of Operational Research 143(3), 498–517 (2002)
21. Nowicki, E., Smutnicki, C.: An Advanced Tabu Search Algorithm for the Job Shop Problem. Journal of Scheduling 8(2), 145–159 (2005)
22. Nuijten, W.: Time and Resource Constraint Scheduling: A Constraint Satisfaction Approach. PhD thesis, Eindhoven University of Technology (1994)
23. Raaymakers, W.H.M., Hoogeveen, J.A.: Scheduling multipurpose batch process industries with no-wait restrictions by simulated annealing. European Journal of Operational Research 126(1), 131–151 (2000)
24. Rajendran, C.: A no-wait flowshop scheduling heuristic to minimize makespan. The Journal of the Operational Research Society 45(4), 472–478 (1994)
25. Schuster, C.J.: No-wait job shop scheduling: Tabu search and complexity of problems. Math. Meth. Oper. Res. 63, 473–491 (2006)
26. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.: Why cumulative decomposition is not as bad as it sounds. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 746–761. Springer, Heidelberg (2009)
27. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 590–603. Springer, Heidelberg (2006)
28. Walsh, T.: Search in a Small World. In: IJCAI 1999, pp. 1172–1177 (1999)
29. Watson, J.-P., Beck, J.C.: A Hybrid Constraint Programming / Local Search Approach to the Job-Shop Scheduling Problem. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 263–277. Springer, Heidelberg (2008)

On the Design of the Next Generation Access Networks

Stefano Gualandi, Federico Malucelli, and Domenico L. Sozzi

Politecnico di Milano, Dipartimento di Elettronica e Informazione
{gualandi,malucell,sozzi}@elet.polimi.it

Abstract. We present a class of problems that arise in the design of the Next Generation Access Networks. The main features of these networks are: to be based on fiber links of relatively long length with respect to traditional copper based networks, users may be reached directly by fibers, and the presence of few central offices managing a large number of users. We present an Integer Programming model that captures the technological constraints and the deployment costs. The model serves as a basis for a decision support tool in the design of the Next Generation Access Networks. Pure Integer Programming cannot handle real-life problem instances, giving rise to new challenges and opportunities for hybrid Constraint Programming-Mathematical Programming methods. In this paper, we compare a LP-based randomized rounding algorithm with a Constraint-based Local Search formulation. The use of an LP relaxation is twofold: it gives lower bounds to the optimal solution, and it is easily embedded into a randomized rounding algorithm. The Constraint-based Local Search algorithm is then exploited to explore the set of feasible solutions. With these algorithms we are able to solve real-life instances for one of the problems presented in this paper.

1 Introduction

In the last decade, network design has been one of the most important application domains for Integer Programming methods. Typical application areas are transportations and telecommunications, where even a small optimization factor can have an important economical impact.

Even for Constraint Programming, network design has been a source of applications. See e.g., Simonis [1] for a recent overview.

In this paper, we present challenges that arise optimizing the design of the Next Generation Access Networks completely based on fiber cable technology that, in certain cases, may reach single users and for this reason are called Fiber To The Home networks (FTTH).

The new network characteristics and the upcoming deployment motivate the investigations on quantitative optimization models and algorithms for the planning that can help investors to decide which type of fiber network to select and how to operationally implement it, that is where to install *central offices*, that is the centers connected to the backbone network managing customer connections, and possible intermediate cabinets and how to reach users considering network link capacity.

1.1 Technological Aspects

This technology has been proposed many years ago, but due to many factors, including the telecommunication crisis and the absence of a really band eager application, is becoming interesting only now.

We present the main issues arising in the design of FTTH networks considering only the most important technological features affecting the optimization process. For a review on technical aspects refer to [2].

The main features of these networks are: to be based on fiber links of relatively long length with respect to traditional copper based networks, users may be reached directly by fibers, and the presence of few central offices managing a large number of users. The use of fiber optics technology on the one hand involves massive investments in the deployment phase, on the other hand it allows to reduce the yearly maintenance costs and to increase the reliability.

From a mathematical point of view there are two main classes of architectures of FTTH networks: the single star networks and the double star networks.

In single star networks, each user is reached by a fiber starting directly from a central office. The fibers can be up to 20 km long without needing any intermediate device between the central office and the final destination. The number of users that are connected to the same central office depends only on the managing capacity that usually ranges from 1,000 to 100,000. The long haul of cables may allow to cover large areas. In case of densely populated areas, as metropolitan areas, instead of reaching each user with a fiber, an alternative solution introduces so called *splicing cabinets* in each building where the fiber is terminated. From each splicing cabinet copper cable drops are used to reach users. The relatively short length of copper cables allows to provide a broad band even though the architecture is not entirely based on fiber optic (about 1Gb/s for drops of less than 150 m). We will refer to single star networks as *fiber to the cabinet.*

Double star networks exploit the fact that cables from central offices to subsets of nearby users may often follow the same path for a long distance and eventually split in the last portion. Therefore an intermediate *cabinet* is introduced. The cabinets are usually placed at the intersection of streets and can manage up to 30,000 users. In the cabinets multiplexing may take place, allowing to better exploit the cable capacity in the leg from the central office to the cabinet and thus to reduce the number of cables. We can have different architectures depending on the level where multiplexing takes place. It may be at electrical level, requiring a powered cabinet, or at optical level. In the latter case cabinets do not need to be powered. As in the single star case, also in the double star architectures, instead of having a cable from the cabinet to each user, we may introduce a splicing cabinet for each building serving more users with a single fiber cable. These networks can be seen as two level hierarchical networks, where the first layer represents the distribution from the central office to the cabinets and the second layer represents the distribution from the cabinet to the users. We will refer to double star networks as *fiber to the basement* or *fiber to the home.*

Variants of these problems consider additional features as reliability constraints for subsets of customers and mixed cable and radio links for the last mile.

1.2 Problem Statement and Notation

Planning a FTTH network involves several decisions: where to install central offices, where to install cabinets, the assignment of cabinets to central offices and how many fibers are needed for the connection, which multiplexing capacity to install in each cabinet and, in the case of fiber to the home or fiber to the basement networks, the assignment of basements or homes to the cabinets.

More specifically, we have two assignment problems: one that maps cabinets to central offices and the other basements/homes to cabinets. The constraints are the following:

a) each active cabinet must be connected to at least a central office;
b) the maximum number of fibers reaching each central office cannot exceed its managing capacity;
c) each cabinet has a demand that must be satisfied, given by the number of connections that must be established between the cabinet and its central office; this demand, in the case of double star networks, is given by the number of users connected to the cabinet;
d) the connections between a cabinet and its central office (or between basement/home and cabinet) cannot exceed a given distance.

Deployment costs are affected by three factors:

1. the number of activated central offices;
2. the type of multiplexing technology installed in each cabinet;
3. the fiber laying cost, proportional to the distance between the cabinet and the central office it is assigned to, and between the basement or home and the cabinet it is assigned to.

Let us denote by O the set of candidate sites for central offices, by C be the set of candidate sites for cabinets and by S the set of basements/homes to be served. Let s_i^1 and M_i^1 be the cost and the capacity (in terms of number of fibers) of central office i. Let T be the types of technologies that can be installed in cabinets. Multiplexing technology t in a cabinet allows to send m_t channels on a single fiber towards the central office. Let s_{jt}^2 be the installation cost of cabinet j with technology t, and and M_j^2 its maximum capacity in terms of number of fibers coming from the users. With d_{ij} we indicate the known distance (computed on the street graph) between any two sites i and j.

Figure 1 shows a micro example on the Politecnico campus in Milan. The map shows the street graph along with 2 candidate sites for central offices, 3 candidate sites for cabinets, and 9 building basements where the splitters will be installed. The problem is formalized using a tripartite graph defined on three sets of vertices: the set of central offices O, the set of cabinets C, and the set of basements S. Figure 2 shows the tripartite graph corresponding to the micro example of Fig. 1. The subgraph induced by the sets O and C is referred to as the primary network, while the subgraph induced by the sets C and S is the secondary network. There is an edge in both the primary or in the secondary network only if the distance constraints are satisfied, that is, there is the edge

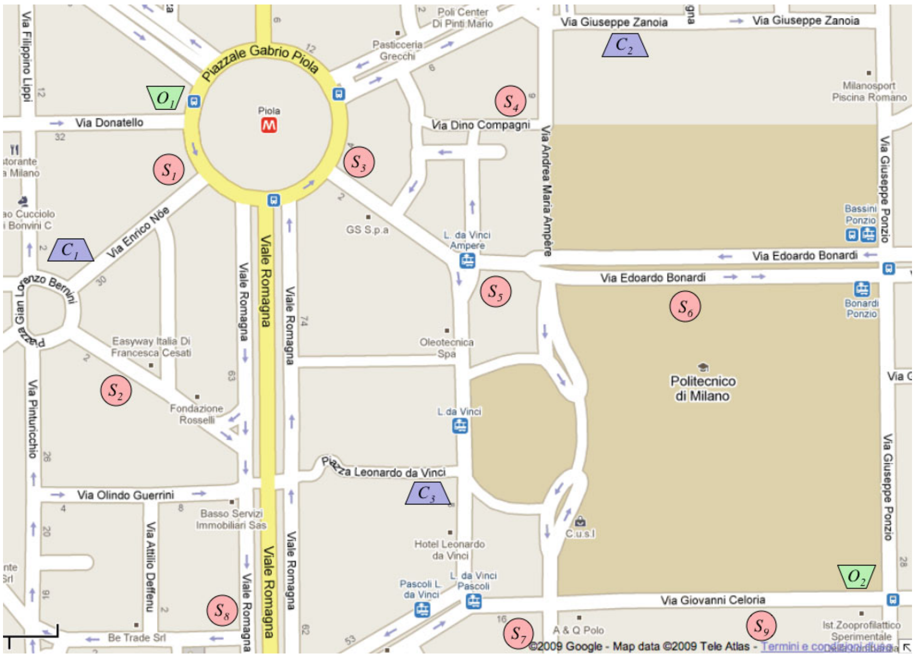


Fig. 1. Micro example on the Politecnico of Milano campus, with two candidate sites for the central offices O_i (downward trapezia), three canidate sites for the cabinets C_j (upward trapezia), and nine basements S_l (circles)

(i, j) with $i \in O$ and $j \in C$ if $d_{ij} \le L^1$, and there is the edge (j, l) with $j \in C$ and $l \in S$ if $d_{jl} \le L^2$.

The two level nature of the problem is quite evident. Further on we will use superscript 1 to denote the level between central offices and cabinets, and superscript 2 to denote that between cabinets and customers.

1.3 Related Work

The problem studied here, to the best of our knowledge, was not considered before in the optimization literature. A related network design problem is presented in [3], where, given the positions of central offices and of users, the problem consists in finding the position of the optical splitters in such a way of minimizing the overall costs. In that model, there is not an actual list of candidate sites, since a rural (or greenfield) scenario is considered, and the coordinates of the position of the splitters are part of the decision variables of the problem. A mixed integer non linear model is presented with the only purpose of formulating the problem and it is not exploited in the heuristic algorithm.

Our problem is related to the Two-level Uncapacitated Facility Location problem (TUFL), well studied in the literature (e.g., see [4] for a polyhedral study

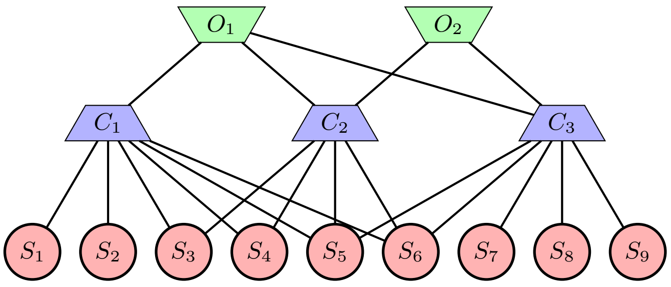


Fig. 2. A tripartite (sub)graph correspoding to the example of Figure 1

and see [5] for recent advances in approximation algorithms). However, there are two important differences with our problem: first, the capacity constraints that are not considered in the literature, and, second, the multiplexing technology constraints that make the problem much more complex. Note that, differently from TUFL, in our case distinct paths from a central office to customers are not profitable, since the multiplexing occurring at the cabinets can merge more links coming from the secondary network into a single link of primary network. Thus the techniques developed for the TUFL can hardly be exploited in our case.

Hybrid constraint and integer programming methods for the networks problems are presented in [1]. Recently [6] tackled a problem of routing and wavelength assignment on optical networks. For each demand the set of frequencies is given, and the problem consist in deciding which demands to select and how to route them. To solve this problem a decomposition approach is implemented using a MIP model to solve the allocation subproblem, i.e., to select and to route a subset of demands, and the wavelength assignment problem is formalized as a graph coloring and solved with constraint programming. In case the CP subproblem becomes infeasible, the MIP allocation problem is somehow relaxed.

Another network design problem is presented in [7], where particular attention is paid to problem of breaking symmetries.

A hybrid local search and constraint propagation method for a network routing problem is presented in [8], where the problem consists in, given a directed capacitated network and a set of traffic demands, minimizing the cost of the lost traffic demands.

2 Fiber to the Basement

Let us introduce a mathematical model for the design of Fiber To The Basement/Home networks, which is the most general problem. The model for the Fiber to the Cabinet problem can be derived as a special case. The problem can be seen as a variant of the capacitated facility location problem, where facilities belong to two levels (i.e. central offices and cabinets). We need to introduce two sets of binary variables: $y^1_i, i \in O$ whose value is 1 if a central office is activated in site i, and $y^2_{jt}, j \in C, t \in T$ if a cabinet with multiplexing technology t is activated in site j. We need another set of binary variables x^2_{jl} whose value is 1 if basement l is assigned to cabinet j. Integer variables x^1_{ij} give the number of fibers connecting central office i with cabinet j. The last two sets of variables are defined for all pairs i, j and j, l such that the distance between the corresponding sites is less than or equal to the maximum allowed distance. In order to consider only pairs of sites within a feasible distance, we introduce a set E including all pairs i, j with $i \in O$ and $j \in C$ such that $d_{ij} \leq L^1$, and a set F of pairs j, l with $j \in C$ and $l \in S$ such that $d_{jl} \leq L^2$.

The Integer Programming model is as follows:

$$\min \quad \sum_{i \in O} s^1_i y^1_i + \sum_{j \in C} \sum_{t \in T} s^2_{jt} y^2_{jt} + \sum_{ij \in E} c^1_{ij} x^1_{ij} + \sum_{jl \in F} c^2_{jl} x^2_{jl} \tag{1}$$

$$\text{s.t.} \quad \sum_{jl \in E} x^2_{jl} = 1, \qquad \forall l \in S, \tag{2}$$

$$\sum_{ij \in E} x^1_{ij} \leq M^1_i y^1_i, \qquad \forall i \in O, \tag{3}$$

$$\sum_{t \in T} y^2_{jt} \leq 1, \qquad \forall j \in C, \tag{4}$$

$$\sum_{jl \in F} x^2_{jl} \leq M^2_j \sum_{t \in T} y^2_{jt}, \qquad \forall j \in C, \tag{5}$$

$$\sum_{t \in T} y^2_{jt} \leq \sum_{ij \in E} x^1_{ij}, \qquad \forall j \in C, \tag{6}$$

$$m_t \sum_{ij \in E} x^1_{ij} \geq \sum_{jl \in F} x^2_{jl} - M^2_j (1 - y^2_{jt}), \qquad \forall j \in C, \forall t \in T, \tag{7}$$

$$y^1_i \in \{0, 1\}, \qquad \forall i \in O, \tag{8}$$

$$y^2_{jt} \in \{0, 1\}, \qquad \forall j \in C, \forall t \in T, \tag{9}$$

$$x^1_{ij} \in \mathbb{Z}_+, \qquad \forall ij \in E, \tag{10}$$

$$x^2_{jl} \in \{0, 1\}, \qquad \forall jl \in F. \tag{11}$$

Constraints (2) state that each user must be connected to a cabinet. Constraints (3) are twofold: they force the activation of central office i (i.e. it sets variable y_i to 1) if at least one cabinet j is assigned to it, and they limit the number of cabinets assigned to i according to the capacity. Constraints (4) determine that either a cabinet is not active (when the left hand side is equal to 0) or at most a multiplexing technology is assigned to it. Constraints (6) state that if a cabinet is activated it must be connected to a central office. While constraints (7) relate the number of incoming fibers in a cabinet from users with the number of outgoing fibers towards the central office. This number must account for the multiplexing factor installed in the cabinet. Note that in the group of constraints referring to a cabinet at most one is significant, while the others are made redundant by big constants.

The objective function (1) accounts for the cost s_i^1 of each activated central office, the cost s_{jt}^2 for installing the technology t in cabinet j and the connection costs for the fibers between central offices and cabinets and between cabinets and the users.

Note that for real life instances the ILP model (1)–(11) has more than one million of variables and constraints, and cannot be solved in a reasonable time with a pure Integer Programming approach.

3 Computational Approaches

We have developed two approaches to solve the FTTH problem. The first approach is an LP-based Randomized Rounding algorithm, the second is a Constraint-based Local Search algorithm. Both approaches are implemented exploiting features of the Comet constraint language [9].

3.1 LP-Based Randomized Rounding

The FTTH problem recalls a capacitated facility location, but it has two levels of facilities: at the first level we have the candidate sites for the central offices, and at the second level there are the candidate sites for the cabinets, in addition at this level multiplexing technologies must be accounted for. LP-based randomized rounding algorithms have proved to be successful for traditional capacitated facility location problems [10]. This motivated our design of an LP-based Randomized Rounding.

Let us call (P) the problem obtained by substituting integrality constraints in (1)–(11) with the following linear constraints:

$$0 \leq y_i^1 \leq 1, \quad 0 \leq y_{ij}^2 \leq 1, \quad x_{ij}^1 \geq 0, \quad 0 \leq x_{jl}^2 \leq 1. \tag{12}$$

Problem (P) can be solved easily with standard linear programming software. An alternative option could utilize the Volume algorithm, but we leave this to future investigations.

Our LP-based Randomized Rounding algorithm is based on the observation that once we have decided which central offices and which cabinets to open, that is, the variables y^1 and y^2 have been fixed to either 1 or 0, the remaining problem is reduced to a generalized minimum cost flow problem on a two level bipartite graph. Even if the generalized minimum cost flow problem is polynomial (e.g., see [11]), we solve it with a linear programming software.

We define three auxiliary subproblems:

1. The Continuous Generalized Minimum Cost Flow Problem (C-GFP) obtained by fixing all of location variables y_i^1 and y_{jt}^2 either to 1 or 0.
2. The Partial Generalized Minimum Cost Flow Problem (P-GFP) obtained by fixing to 1 some selected y_i^1 and y_{jt}^2 variables, and leave open the remaining ones (that is we do not fix to 0 any variable);
3. The Integer Generalized Minimum Cost Flow Problem (I-GFP) obtained by adding the integrality constraint to (C-GFP).

Algorithm 1 sketches the main steps of our LP-based Randomized Rounding algorithm. The randomized algorithm first solves the linear problem (P), then it randomly rounds the variables y^1 and y^2 to either 1 or 0. In a second step, the algorithm solves the (C-GFP) so obtained. In the case (C-GFP) is not feasible, the algorithm solves the corresponding (P-GFP) and tries with a different randomized rounding. Otherwise, if (C-FGP) is feasible, but some variable x^1 or x^2 are not integer, the algorithms solves the corresponding (I-GFP). The algorithm cycles over these steps a given number r of times.

Since the LP relaxation is rather weak and most of y^1 variables are only slightly bigger than zero, rather than performing a standard randomized rounding, where variable y_i^1 is set to 1 with probability $\bar{y}_i^1$, we perform a normalization of the relaxation as follows. For each variable y_i^1 we compute the ratio:

$$B_i = \frac{\sum_{ij \in E} \bar{x}_{ij}^1}{\sum_{i'j \in E} \bar{x}_{i'j}^1} \tag{13}$$

This corresponds to normalize for each central office i the sum of the values assigned in the LP relaxation to the link variables $\bar{x}_{ij}^1$ entering in i by the sum of all the link variables $x_{i'j}^1$. On the contrary, for variables y^2 we perform a standard randomized rounding. The randomized rounding is preceded by a preprocessing phase that fixes to 1 all the facility variables having a value greater than δ.

3.2 Constraint-Based Local Search

We investigated also the use of Constraint-based Local Search (CBLS) that has proved to be effective on other very large optimization problems [12].

CBLS Model. The CBLS approach proposed in this paper is based on a model different from (1)–(11), and it relies on the use of *invariants* (see [13]) to incrementally maintain the necessary information to guide the search procedure. In order to use a different notation from the ILP formulation we will use upper case letters to denote the variables of the CBLS model.

The decision variables are the following:

- For each basement l there is an integer variable X_l^2 with domain equal to the subset of cabinets $C_l \subseteq C$ reachable from j, i.e., $C_l = \{j \mid \exists(l,j) \in F\}$. If $X_l^2 = j$ it means that basement l is linked to cabinet j.
- For each cabinet j there is an integer variable Z_j with domain equal to T. $Z_j = t$ means that in cabinet j the t-th multiplexing technology is installed.
- For each possible link $(i,j) \in E$ there is an integer variable X_{ij}^1 (equivalent to variable x_{ij}^1), that gives the number of fibers installed between central office i and cabinet j.

These are the actual decision variables, since once they have been determined we can derive which are the open central offices and the open cabinets (with the

Algorithm 1. LP-based Randomized Rounding Algorithm.

```
1: for $i$ in $1..r$ do
2:     $LB, \bar{x}, \bar{y} \leftarrow$ solve problem (P)
3:     Fix to 1 the variables $y^1$ and $y^2$ with value greater than $\delta$
4:     for $i \in O$ do
5:         if Uniform01() $< \frac{\sum_{j \in C} \bar{x}^1_{ij}}{\sum_{i' j \in E} \bar{x}^1_{i' j}}$ then
6:             Set $y^1_i = 1$
7:         else
8:             Set $y^1_i = 0$
9:         end if
10:    end for
11:    for $j \in C, t \in T$ do
12:        if Uniform01() $< \bar{y}^2_{jt}$ then
13:            Set $y^2_{jt} = 1$                    ▷ and set to zero all other $t' \neq t$
14:        else
15:            Set $y^2_{jt} = 0$
16:        end if
17:    end for
18:    Solve the (C-GFP) obtained by fixing $y^1$ and $y^2$ in steps 6,8,13, and 15
19:    if (C-GFP) is not feasible then
20:        Solve the (P-GFP) and go to 4        ▷ consider only the fixes in 6 and 13
21:    end if
22:    if $x^1$ and $x^2$ are not all integer then
23:        Solve the so obtained (I-GFP)
24:    end if
25:    if (I-GFP) is feasible then update UB else go to 4
26: end for
```

corresponding technology). In order to keep track of the open facilities, we use the following invariants:

– $Y^1_i \in \{0,1\}$: it is equal to 1 if at least a cabinets j linked to i exists:

$$Y^1_i \Leftrightarrow (\exists j \in C.X^1_{ij} > 0), \qquad \forall j \in J.$$

– $Y^2_j \in \{0,1\}$: it is equal to 1 if at least a basement l linked to j exists:

$$Y^2_j \Leftrightarrow (\exists l \in S.X^2_l = j), \qquad \forall j \in C.$$

Once we have assigned a value to each decision variable, and these values have propagated to the invariants, the objective function is computed as follows:

$$\sum_{i \in O} s^1_i Y^1_i + \sum_{j \in C: Y^2_j = 1} s^2_{jZ_j} + \sum_{ij \in E} c^1_{ij} X^1_{ij} + \sum_{l \in S} c^2_{X^2_l l} \tag{14}$$

Note that in the second and the fourth term, we use variable subscription, as it were an `element` constraint, that is, we have a variable appearing in the subscript of a cost parameters, which it is not possible in Integer Linear Programming.

Search procedure. Basically, the CBLS performs a search in the space of possible assignments to the decision variables X^1, X^2, and Z. The initial solution is obtained by the following greedy procedure: first, we open a cabinet j, second, we assign to it the M_j^1 nearest basements, then we open a second cabinet and we "fill it" with basements, and we repeat these steps until every basement is linked to a cabinet. Then, we decide which multiplexing technology to install on every open cabinet while minimizing the installation costs. Finally, with a similar greedy procedure, we keep in opening a new central office at the time, until every open cabinet is assigned to a number of central offices in such a way that the number of incoming fibers (in a cabinet) is equal to the number of multiplexed outgoing fibers. Ties are always broken randomly.

The local search is based on a simple move: select the basement l connected to a cabinet j, and select a different open cabinet $j' \neq j$ that is not saturated (it has some capacity left) such that *moving* l from j to j' gives the best improvement in the objective function (14). After this move, it may happen that we need to increase by one the multiplexing technology at the cabinet j', and possibly to increase the number of fibers outgoing cabinet j'. We use this move to perform a best improvement local search, until we get stalled in a local minimum. Once we get stuck in a local minimum, we use a different neighborhood by trying to change the multiplexing technology, i.e. variable Z_j, and to modify the variable X_{ij}^1. Again, we perform a best improvement local search by selecting the moves that decrease the objective function.

After that the local search algorithm get stuck in a local minimum a certain number of times, we perform a simple *diversification* by randomly swapping the assignment of basements to cabinets, and of fibers outgoing the cabinets to central offices.

4 Computational Results

The two approaches presented in this paper are evaluated on realistic instances. By realistic we mean that they are randomly generated in such a way to be as close as possible to the real scenario of the metropolitan area of the city of Rome. Using the street graph of Rome, with the link lengths in meters, we have generated 21 different instances using values for the installation and deployment costs and for the central office and cabinet capacities, as provided by our collaborators working at Alcatel-Lucent.

The biggest instance has 35 candidate sites for the central offices (consider that the currently operated traditional network in Milan has 28 central offices, but they would be more than really necessary in a fiber based network), 150 candidate sites for the cabinets, and 10.000 basements. This is equivalent to approximately serve 300.000 final users. The smaller instances are generated in order to compare our heuristics with an exact ILP method.

4.1 Implementation Details

The two approaches have been implemented with the COMET constraint languages (version 2.0), using COIN-CLP as linear solver and SCIP as Integer Linear

Programming solver. The tests were carried over a computer with linux-ubuntu 32 bits, an Intel Q6600 CPU (Quadcore 2.4GHz, 8Mb L2 cache, 1066 MHz FSB) and with 4Gb of ram.

4.2 Comparing the Two Algorithms

The first set of experiments was performed to compare the LP-based Randomized Rounding (LP-RR) algorithm with the Constraint-based Local Search algorithm.

Table 1 shows the results of the comparison for a first set of Rome instances. Each instance is described in terms of number of central office candidate sites ($|O|$) the cabinet candidate sites ($|C|$) and basements ($|S|$). Each line reports the results averaged over 5 runs, and gives the average value (Cost) of the objective function, the average execution time, and the best result found (Best-Cost). The LP-RR algorithm is executed with $r = 20$, that is, it runs 20 rounds for each execution, with threshold $\delta = 0.9$. The CBLS algorithm has a limit on the number of restarts (equal to 20).

Both algorithms provide solutions of very good quality, but the CBLS provides the best results for big instances and it is clearly faster. For the instance with 10.000 basements the CBLS algorithm found better solutions in a time that is two order of magnitude less in comparison with the randomized rounding. Note that, for the first instance, the optimal value is 2383, as verified by the ILP solver SCIP, and both heuristic algorithms have been able to obtain this result.

Table 1. LP-based Randomized Rounding (LP-RR) versus Constraint-based Local Search (CBLS). Cost and Time (in seconds) are averaged over 5 runs for each instance.

				LP-RR			CBLS		
Inst.	$\|O\|$	$\|C\|$	$\|S\|$	Cost	Time	Best-Cost	Cost	Time	Best-Cost
1	3	10	100	2383	31	**2383**	**2383**	0.6	**2383**
2	10	35	400	6979	716	6966	6864	1.2	**6860**
3	15	65	841	13630	1735	13599	13349	44.6	**13306**
4	20	100	1521	25499	2465	25427	24850	316	**24752**
5	25	120	3025	55073	4768	55052	51752	330	**51646**
6	30	140	6084	121794	7705	121974	118224	1105	**118135**
7	35	150	10000	239668	26915	239668	229677	1817	**229244**

The main limitation of the LP-based Randomized Rounding algorithm is that it solves several times a large generalized minimum cost flow problem and (frequently) an integer problem as well. By increasing the threshold in line 3 of Algorithm 1, for instance from $\delta = 0.9$ to $\delta = 0.95$, we get indeed solutions of better quality, but involving an important increment of the computation time. Therefore we have focused on the CBLS algorithm.

Table 2 shows the results for a second set of Rome instances. The CBLS algorithm described in Section 3.2 is run five times on each instance and we have computed the average over each run. Each instance is described in terms of number of central office and cabinet candidate sites and number of basements. For each instance the table reports the Cost and the running Time averaged over 5 runs, the corresponding standard deviations (stdev), the Best-Cost, the optimum solution (IP) found with an ILP solver. The last column gives the percentage gap with the optimal solution. Note that the CBLS is fast, and it also provides solutions with a very small percentage gap. In particular, for the smaller instances it does find the optimum.

Table 2. Solving small Rome instances with the CBLS approach: gaps with respect to the optimal solution computed with SCIP. Cost and Time (in seconds) are averaged over 5 runs for each instance.

Inst.	$\|O\|$	$\|C\|$	$\|S\|$	Cost	(stdev)	Time	(stdev)	Best-Cost	IP	Gap
8	5	10	109	3244	0.04%	1.5	2.7%	**3243**	**3243**	0.0%
9	10	20	204	**13888**	0.00%	2.3	1.4%	**13888**	**13888**	0.0%
10	20	100	1462	419929	0.04%	87.1	0.6%	419823	**417554**	0.5%
11	25	120	3139	1011821	0.02%	567.7	0.7%	1011457	**1009710**	0.2%

Finally, Table 3 reports additional results for other bigger Rome instances, reporting the percentage gap (LP-Gap) computed with the value of the linear relaxation of the problem. The CBLS is pretty stable both in the quality of the solution and in the computation time required. For the bigger instances, those with 10,000 basements, the computation time can be more the one-hour (see instance 18,20, and 21), but still is always better than the LP-RR algorithm. We remark that the percentage gap on the lower bound computed by solving the linear relaxation (P) is in the worse case 2.4%.

Table 3. Solving big Rome instances with the CBLS approach: gaps computed with respect to the linear relaxation (P).

Inst.	$\|O\|$	$\|C\|$	$\|S\|$	Cost	(stdev)	Time	(stdev)	Best-Cost	LP-Gap
12	30	140	5960	4558323	(0.02%)	1350.1	(0.46%)	4557601	1.1%
13	30	140	5981	3954325	(0.01%)	1008.4	(0.05%)	3953619	1.2%
14	30	140	5982	4561215	(0.01%)	1803.6	(0.14%)	4560780	0.9%
15	30	140	5995	4164941	(0.01%)	2168.7	(0.69%)	4164724	1.1%
16	30	140	6014	3462920	(0.01%)	1426.9	(0.35%)	3462857	1.4%
17	35	150	10020	3126763	(0.02%)	2511.8	(0.44%)	3126385	2.4%
18	35	150	10040	5937585	(0.01%)	3484.7	(0.55%)	5936733	1.1%
19	35	150	10072	6663950	(0.01%)	1183.6	(0.54%)	6663481	0.9%
20	35	150	9978	6261704	(0.01%)	4252.8	(0.49%)	6261046	1.0%
21	35	150	9983	5980627	(0.01%)	3846.9	(0.65%)	5979618	1.1%

5 Conclusions

We presented a computational approach to the FTTH problem that arises in the context of designing the Next Generation Access Network. The FTTH has an ILP formulation, but realistic instances leads to very large size problems, and therefore we have only focused on heuristic algorithms. First, we discussed a LP-based Randomized Rounding algorithm that exploits a substructure of the problem, reducing the problem to a generalized minimum cost flow problem. Second, we presented a Constraint-based Local Search algorithm that despite its simplicity is very effective, providing, in short time, solutions with small percentage gap to lower bounds of the problem.

Currently, we are investigating a new type of network topology for the secondary network between the cabinets and the basements. Instead of using direct links between basements and cabinets, we plan to use a tree topology, allowing more basements to share portion of fibers before of reaching a cabinet. This network topology leads to new challenges, because the ILP formulation has an exponential number of constraints. As future work, we plan to extend our Constraint-based Local Search algorithm to tree-based network topologies.

Acknowledgments

The authors are grateful indebted with Carlo Spinelli of Alcatel-Lucent for helpful discussions and for providing details about the technological aspects of the problem.

References

1. Simonis, H.: Constraint applications in networks. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
2. Kramer, G., Pesavento, G.: Ethernet Passive Optical Network (EPON): building a next-generation optical access network. IEEE Communications Magazine 40(2), 66–73 (2002)
3. Li, J., Shen, G.: Cost Minimization Planning for Greenfield Passive Optical Networks. Journal of Optical Communications and Networking 1(1), 17–29 (2009)
4. Aardal, K., Labbe, M., Leung, J., Queyranne, M.: On the two-level uncapacitated facility location problem. INFORMS Journal on Computing 8, 289–301 (1996)
5. Zhang, J.: Approximating the two-level facility location problem via a quasi-greedy approach. Mathematical Programming 108(1), 159–176 (2006)
6. Simonis, H.: A Hybrid Constraint Model for the Routing and Wavelength Assignment Problem. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 104–118. Springer, Heidelberg (2009)
7. Smith, B.: Symmetry and Search in a Network Design Problem. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 336–350. Springer, Heidelberg (2005)
8. Lever, J.: A local search/constraint propagation hybrid for a network routing problem. International Journal of Artificial Intelligence Tools 14(1-2), 43–60 (2005)

9. Van Hentenryck, P., Michel, L.: Control abstractions for local search. Constraints 10(2), 137–157 (2005)
10. Barahona, F., Chudak, F.: Near-optimal solutions to large-scale facility location problems. Discrete Optimization 2(1), 35–50 (2005)
11. Ahuja, R., Magnanti, T., Orlin, J.: Network Flows: Theory, Algorithms, and Applications. Prentice-Hall, Englewood Cliffs (1993)
12. Van Hentenryck, P., Michel, L.: Constraint-based local search. MIT Press, Cambridge (2005)
13. Van Hentenryck, P., Michel, L.: Differentiable invariants. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 604–619. Springer, Heidelberg (2006)

Vehicle Routing for Food Rescue Programs: A Comparison of Different Approaches

Canan Gunes, Willem-Jan van Hoeve, and Sridhar Tayur

Tepper School of Business, Carnegie Mellon University

1 Introduction

The 1-Commodity Pickup and Delivery Vehicle Routing Problem (1-PDVRP) asks to deliver a single commodity from a set of supply nodes to a set of demand nodes, which are unpaired. That is, a demand node can be served by any supply node. In this paper, we further assume that the supply and demand is unsplittable, which implies that we can visit each node only once. The 1-PDVRP arises in several practical contexts, ranging from bike-sharing programs in which bikes at each station need to be redistributed at various points in time, to food rescue programs in which excess food is collected from, e.g., restaurants and schools, and redistributed through agencies to people in need. The latter application is the main motivation of our study.

Pickup and delivery vehicle routing problems have been studied extensively; see, e.g., [1] for a recent survey. However, the 1-commodity pickup and delivery vehicle routing problem (1-PDVRP) has received limited attention. When only one vehicle is considered, the problem can be regarded as a traveling salesman problem, or 1-PDTSP. For the 1-PDTSP, different solution methods have been proposed, including [3, 4]. On the other hand, the only paper that addresses the 1-PDVRP is by [2], to the best of our knowledge. [2] present different approaches, including MIP, CP and Local Search, which are applied to instances involving up to nine locations.

The main goal of this work is to compare off-the-shelf solution methods for the 1-PDVRP, using state-of-the-art solvers. In particular, how many vehicles, and how many locations, can still be handled (optimally) by these methods? The secondary goal of this work is to evaluate the potential (cost) savings in the context of food rescue programs. We note that the approaches we consider (MIP, CP, CBLS) are similar in spirit to those of [2]. Our MIP model is quite different, however. Further, although the CP and CBLS models are based on the same modeling concepts, the underlying solver technology has been greatly improved over the years.

2 Different Approaches to the 1-PDVRP

2.1 Input Data and Parameters

Let the set V denote the set of locations, and let $O \in V$ denote the origin (or depot) from which the vehicles depart and return. With each location i in V we associate a number $q_i \in \mathbb{R}$ representing the quantity to be picked up ($q_i > 0$) or delivered ($q_i < 0$) at i. The distance between two locations i and j in V will be denoted by d_{ij}. Distance can be represented by length or time units.

Let T denote the set of vehicles (or trucks). For simplicity, we assume that all vehicles have an equal 'volume' capacity Q of the same unit as the quantities q to be picked up (e.g., pounds). In addition, all vehicles are assumed to have an equal 'horizon' capacity H of the same unit as the distances d.

2.2 Mixed Integer Programming

Our MIP model is based on column generation. The master problem of our column generation procedure consists of a set of 'columns' S representing feasible routes. The routes are encoded as binary vectors on the index set V of locations; that is, the actual order of the route is implictly encoded. The columns are assumed to be grouped together in a matrix A of size V by S. The length of the routes is represented by a 'cost' vector $c \in \mathbb{R}^{|S|}$. We let $z \in \{0,1\}^{|S|}$ be a vector of binary variables representing the selected routes. The master problem can then be encoded as the following set covering model:

$$\begin{aligned} &\min\, c^{\mathsf{T}} z \\ &\text{s.t. } Az = \mathbf{1} \end{aligned} \tag{1}$$

For our column generation procedure, we will actually solve the continous relaxation of (1), which allows us to use the shadow prices corresponding to the constraints. We let λ_j denote the shadow price of constraint j in (1), where $j \in V$.

The subproblem for generating new feasible routes uses a model that employs a flow-based representation on a layered graph, where each layer consists of nodes representing all locations. The new route comprises M steps, where each step represents the next location to be visited. We can safely assume that M is the minimum of $|V| + 1$ and (an estimate on) the maximum number of locations that 'fit' in the horizon H for each vehicle.

We let x_{ijk} be a binary variable that represents whether we travel from location i to location j in step k. We further let y_j be a binary variable representing whether we visit location j at any time step. The vector of variables y will represent the column to be generated. Further, variable I_k represents the inventory of the vehicle, while variable D_k represents the total distance traveled up to step k, where $k = 0, \dots, M$. We let $D_0 = 0$, while $0 \le I_0 \le Q$. The problem of finding an improving route can then be modeled as presented in Figure 1.

In this model, the first four sets of constraints ensure that we leave from and finish at the origin. The fifth set of constraints enforce that we can enter the origin at any time, but not leave it again. The sixth set of constraints model the flow conservation at each node, while the seventh set of constraints (the first set in the right column) prevent the route from visiting a location more than once. The following four sets of constraints represent the capacity constraints of the vehicle in terms of quantities picked up and delivered, and in terms of

$$\min \sum_{i\in V}\sum_{j\in V}\sum_{k=1}^{M} d_{ij}x_{ijk} - \sum_{j\in V}\lambda_j y_j$$

$$\begin{array}{lll}
\text{s.t.} & \sum_{j\in V} x_{O,j,1} = 1 & \\
& \sum_{j\in V} x_{i,j,1} = 0 & \forall i \in V\setminus\{O\} \\
& \sum_{i\in V} x_{i,O,M} = 1 & \\
& \sum_{i\in V} x_{i,j,M} = 0 & \forall j \in V\setminus\{O\} \\
& x_{O,j,k} = 0 & \forall j \in V\setminus\{O\}, \forall k\in[1..M] \\
& \sum_{i\in V} x_{ijk} = \sum_{l\in V} x_{j,l,k+1} & \forall j\in V, \forall k\in[1..M-1] \\
& \sum_{j\in V\setminus\{O\}}\sum_{k=1}^{M} x_{ijk} \le 1 & \forall j \in V\setminus\{O\} \\
& I_k = I_{k-1} + \sum_{i\in V}\sum_{j\in V} q_i x_{ijk} & \forall k\in[1..M] \\
& 0 \le I_k \le Q & \forall k\in[0..M] \\
& D_k = D_{k-1} + \sum_{i\in V}\sum_{j\in V} d_{ij}x_{ijk} & \forall k\in[1..M] \\
& 0 \le D_k \le H & \forall k\in[0..M] \\
& \sum_{i\in V}\sum_{k=1}^{M} x_{ijk} = y_j & \forall j\in V
\end{array}$$

Fig. 1. MIP model for finding an improving route

distance. The last set of constraints link together the ‘flow’ variables x with the new column represented by the variables y.

As noted above, throughout the iterative process, we apply a continuous relaxation of the master problem (1). When this process terminates (it reaches a fixed point, or it meets a stopping criterion), we run the master problem as an integer program. Therefore, our procedure may not provably find the optimal solution, but it does provide a guaranteed optimality gap.

As a final remark, when only one vehicle is involved, the MIP model amounts to solving only the subproblem, to which the constraints are added that we must visit all locations.

2.3 Constraint Programming

Our CP model is based on a well-known interpretation of the VRP as a multi-machine job scheduling problem with sequence-dependent setup times. In the CP literature, this is usually modeled using alternative resources (the machines) and activities (the jobs). That is, each visit to a location corresponds to an activity, and each vehicle corresponds to two (linked) resources: one ‘unary resource’ modeling the distance constraint, and one ‘reservoir’ modeling the inventory of the vehicle. With each activity we associate variables representing its start time and end time, as well as a fixed duration (this can be 0 if we assume that the (un-)loading time is negligible). Further, each activity either depletes or replenishes the inventory reservoir of a vehicle. The distance between two locations is modeled as the ‘transition time’ between the corresponding activities. We minimize the sum of the completion times of all vehicles.

All these concepts are readily available in most industrial CP solvers. We have implemented the model in ILOG Solver 6.6 (which includes ILOG Scheduler). A snapshot of the ILOG model for a single vehicle is provided in Figure 2. It shows that the concepts presented above can almost literally be encoded as a CP model.

2.4 Constraint-Based Local Search

Our final approach uses Constraint-Based Local Search (CBLS). With CBLS we can express the problem similar to a CP model, which will then be used

```
IloReservoir truckReservoir(ReservoirCapacity, 0);
truckReservoir.setLevelMax(0, TimeHorizon, ReservoirCapacity);

IloUnaryResource truckTime();
IloTransitionTime T(truckTime, Distances);

vector<IloActivity> visit;
visit = vector<IloActivity>(N);

for (int i=0; i<N; i++) {
  visit[i].requires(truckTime);
  if (supply[i] > 0)
    visit[i].produces(truckReservoir, supply[i]);
  else
    visit[i].consumes(truckReservoir, -1*supply[i]);
}
```

```
class RoutingModel {
   ...
   IloDimension2 _time;
   IloDimension2 _distance;
   IloDimension1 _weight;
   ...
}

IloNode node( <read coordinates from file> );

IloVisit visit(node);
visit.getTransitVar(_weight) == Supply);
minTime <= visit.getCumulVar(_time) <= maxTime;
visit.getCumulVar(_weight) >= 0);

IloVehicle vehicle(firstNode, lastNode);
vehicle.setCapacity(_weight, Capacity);
vehicle.setCost(_distance);
```

Fig. 2. Snapshots of the ILOG Scheduler model (left) and ILOG Dispatcher model (right), for a single vehicle

to automatically derive the neighborhoods and penalty function needed to define a local search procedure. Our CBLS is based on the semantics offered by ILOG Dispatcher (included in ILOG Solver 6.6). These semantics are specifically designed to model routing problems.

ILOG Dispatcher uses the concepts *nodes*, *vehicles*, and *visits*. The nodes are defined by the coordinates of the locations, and contain as an attribute the amount to be picked up or delivered. The vehicles contain several attributes, including time, distance, and weight (load). Vehicles also contain, by default, a 'unary resource' constraint with respect to time, and a 'capacity' constraint with respect to the load, similar to the resources in ILOG Scheduler. The attributes of visits include the location, the quantity to be picked up (positive) or delivered (negative), a time window, and possibly other problem-specific constraints.

In a first phase, we create a feasible solution. ILOG Dispatcher uses various heuristics for this, including a nearest-neighbour heuristic that we applied in our experiments. Where applicable, we started from the current schedule that we extracted from the data.

The second phase improves upon the starting solution using various local search methods. We applied successively the methods IloTwoOpt, IloOrOpt, IloRelocate, IloCross and IloExchange. Within each method, we take the first legal cost-decreasing move encountered.

3 Evaluation

Our experimental results are performed on data provided by the Pittsburgh Food Bank. Their food rescue program visits 130 locations per week. The provided data allowed us to extract a fairly accurate estimate on the expected pickup amount for the donor locations. The precise delivery amounts were unknown, and we therefore approximate the demand based on the population served by each location (which is known accurately), scaled by the total supply. We allow the total demand to be slightly smaller than the total supply, to avoid pathological behavior of the algorithm. We note however, that although this additional 'slack' influences the results, the qualitative behavior of the different techniques remains the same. The MIP model is solved using ILOG CPLEX 11.2, while the CP and CBLS model are solved using ILOG Solver 6.6, all on a 2.33GHz Intel Xeon machine.

The first set of instances are for individual vehicles, on routes serving 13 to 18 locations (corresponding to a daily schedule). The second set of instances group together schedules over multiple days, ranging from 30 to 130 locations. The results are presented in Figure 3. We report for each instance the cost savings (in terms of total distance traveled) with respect to the current operational schedule. Here, $|V|$ and $|T|$ denote the number of locations and vehicles, respectively. The optimal solutions found with MIP and CP took several (2–3) minutes to compute, while the solutions found with CBLS took several seconds or less. The time limit was set to 30 minutes.

Our experimental results indicate that on this problem domain, our MIP model is outperformed by our CP model to find an optimal solution (we note that a specialized 1-PDTSP MIP approach such as [4] might perform better than our 'generic' MIP model on the single-vehicle instances). Further, the CP model is able to find optimal solutions for up to 18 locations and one vehicle; for a higher number of locations or vehicles, the CP model is unable to find even a single solution. Lastly, the CBLS approach is able to handle large-scale instances, up to 130 locations and 9 vehicles. The expected savings are substantial, being at least 10% on the largest instance.

$\lvert V\rvert$	$\lvert T\rvert$	MIP	CP	CBLS
13	1	12%	12%	12%
14	1	15%	15%	14%
15	1	-	7%	6%
16	1	-	5%	3%
18	1	-	16%	15%
30	2	-	-	4%
60	4	-	-	8%
130	9	-	-	10%

Fig. 3. Savings obtained with different approaches

References

[1] Berbeglia, G., Cordeau, J.F., Gribkovskaia, I., Laporte, G.: Static pickup and delivery problems: A classification scheme and survey. TOP 15(1), 1–31 (2007)

[2] Dror, M., Fortin, D., Roucairol, C.: Redistribution of self-service electric cars: A case of pickup and delivery. Technical Report W.P. 3543, INRIA-Rocquencourt (1998)

[3] Hernández-Pérez, H., Salazar-González, J.J.: A branch-and-cut algorithm for a traveling salesman problem with pickup and delivery. Discrete Applied Mathematics 145, 126–139 (2004)

[4] Hernández-Pérez, H., Salazar-González, J.J.: The one-commodity pickup-and-delivery traveling salesman problem: Inequalities and algorithms. Networks 50, 258–272 (2007)

Constraint Programming and Combinatorial Optimisation in Numberjack[⋆]

Emmanuel Hebrard[1,2], Eoin O'Mahony[1], and Barry O'Sullivan[1]

[1] Cork Constraint Computation Centre,
Department of Computer Science, University College Cork, Ireland
{e.hebrard,e.omahony,b.osullivan}@4c.ucc.ie
[2] LAAS-CNRS Toulouse, France
hebrard@laas.fr

Abstract. Numberjack is a modelling package written in Python for embedding constraint programming and combinatorial optimisation into larger applications. It has been designed to seamlessly and efficiently support a number of underlying combinatorial solvers. This paper illustrates many of the features of Numberjack through the use of several combinatorial optimisation problems.

1 Introduction

We present Numberjack[1], a Python-based constraint programming system. Numberjack brings the power of combinatorial optimisation to Python programmers by supporting the specification of complex problem models and specifying how these should be solved. Numberjack provides a common API for constraint programming, mixed-integer programming and satisfiability solvers. Currently supported are: the CP solvers Mistral and Gecode; a native Python CP solver; the MIP solver SCIP; and the satisfiability solver MiniSat[2]. Users of Numberjack can write their problems once and then specify which solver should be used. Users can incorporate combinatorial optimisation capabilities into any Python application they build, with all the benefits that it brings.

2 Modelling in Numberjack

Numberjack is provided as a Python module. To use Numberjack one must import all Numberjack's classes, using the command: `from Numberjack import *`. Similarly, one needs to import the modules corresponding to the solvers that will be invoked in the program, for instance: `import Mistral` or `import Gecode`. The Numberjack module essentially provides a class `Model` whereas the solver modules provide a class `Solver`, which are built from a `Model`. The structure of a typical Numberjack program is presented in Figure 1. Notice that it is possible to use several types of solver to solve the same model by explicitly invoking the modules. To solve a model, the various methods implemented in the back-end solvers can be invoked through Python.

[⋆] Supported by Science Foundation Ireland Grant Number 05/IN/I886.
[1] Available under LGPL from http://numberjack.ucc.ie
[2] Mistral: http://4c.ucc.ie/~ehebrard/Software.html; Gecode: http://gecode.org; SCIP: http://scip.zib.de/; MiniSat: http://minisat.se;

```
from Numberjack import * # Import all Numberjack classes
import Gecode            # Import the Gecode solver interface
import Mistral           # Import the Mistral solver interface

model = Model()          # Declare a new model
...                      # Define the constraints and objectives

gsolver = Gecode.Solver(model)   # Declare a Gecode solver
msolver = Mistral.Solver(model)  # Declare a Mistral solver
gsolver.solve()                  # Solve the model with Gecode
msolver.solve()                  # Solve the model with Mistral
```

Fig. 1. The structure of a typical Numberjack program

Almost every statement in Numberjack is an *expression*. Variables are expressions, and constraints are expressions on a set of sub-expressions. Variable objects are created by specifying its domain by passing a lower and an upper bound, or a set of values. One can also use floating point values for the bounds, however the result will depend on the back-end solver. MIP solvers will, by default, treat variables declared with floating point values as continuous and integer otherwise. A model is a set of expressions.

It is possible to define classes of objects to help write concise models. For instance, the objects `VarArray` and `Matrix` are syntactic sugars for one-dimensional and two-dimensional arrays of Numberjack expressions, respectively. The `Matrix` object allows us to reference the rows, the columns, and a flattened version of the matrix using `.row`, `.col` and `.flat`, respectively. The overloaded bracket (`[]`) operator tied to the Python object method `getitem`. The operator takes one argument representing the index of the object to be returned. For `VarArray` and `Matrix` objects this argument can either be a Numberjack expression or an integer. The bracket operator of the `Matrix` object returns the `VarArray` object representing the row at the given index. When the index argument is itself a Numberjack expression, the result is interpreted as an `Element` constraint. Objective functions are also expressions.

2.1 Some Example Models

Costas Array. A Costas array[3] is an arrangement of N points on a $N \times N$ checkerboard, such that each column or row contains only one point, and that all of the $N(N-1)/2$ vectors defined by these points are distinct. We model this problem in Figure 2 as follows: for each row, we introduce a variable whose value represents the column at which a point is placed in this row. To ensure that no two points share the same column, we post an `AllDiff` constraint on the rows (Line 4). To each value $y \in [1..N-2]$, we can map a set of vectors whose vertical displacements are equal, i.e., the vectors defined by the points $(row[i], i)$ and $(row[i+y], i+y)$. To ensure that these vectors are distinct, we use another `AllDiff` constraint.

Golomb Ruler. In the Golomb ruler problem the goal is to minimise the position of the last mark on a ruler such that the distance between each pair of marks is different. The Numberjack model is shown in Figure 3.

[3] http://mathworld.wolfram.com/CostasArray.html

```
N   = 10
row = [Variable(1,N) for i in range(N)]

model =  Model()
model += AllDiff(row)
for y in range(N-2):
   model += AllDiff([row[i] - row[i+y+1] for i in range(N-y-1)])

solver = Mistral.Solver(model)
solver.solve()
```

Fig. 2. A Numberjack model for a 10×10 instance of the Costas array problem

```
M = [Variable(1,rulerSize) for i in range(N)]

model =  Model()
model += AllDiff([M[i]-M[j] for i in range(1,N) for j in range(i)])

model += Minimise(marks[nbMarks-1])  # The objective function
```

Fig. 3. A Numberjack model for the Golomb ruler problem

Magic Square. In this problem one wants every number between 1 and N^2 to be placed in an $N \times N$ matrix such that every row, column and diagonal sum to the same number. A model for that problem making use of the `Matrix` class is presented in Figure 4.

```
N = 10
sum_val = N*(N*N+1)/2
square  = Matrix(N,N,1,N*N)

model = Model(

   # The values in each cell must be distinct
   AllDiff(square.flat),

   # Each row and column must add to sum_val
   [Sum(row) == sum_val for row in square.row],
   [Sum(col) == sum_val for col in square.col],

   # Each diagonal must add to sum_val
   Sum([square[a][a] for a in range(N)]) == sum_val,
   Sum([square[a][N-a-1] for a in range(N)]) == sum_val )
```

Fig. 4. A Numberjack model for the Magic Square problem

Quasigroups. A quasigroup is $m \times m$ multiplication defined by a matrix which from a Latin square, i.e. every element occurs once in every row and column. The result of the product $a * b$ corresponds to the element at row a and column b of the matrix. Figure 5 presents a model for the problem in which for all a, b we have: $((b * a) * b) * b = a$.

2.2 Extending Numberjack

Numberjack provides a facility to add custom constraints. Consider the following optical network monitoring problem taken from [1]. An optical network consists of nodes

```
N = 8
x = Matrix(N,N,N)

model = Model(

   # The rows and columns form a Latin square
   [AllDiff(row) for row in x.row],
   [AllDiff(col) for col in x.col],

   # Enforce the QG5 Property
   [x[ x[  x[b][a] ][ b ] ][b] == a for a in range(N) for b in range(N)] )
```

Fig. 5. A Numberjack model for the Quasigroup Existence problem

and fibre channels. When a node fails in the network, all lightpaths passing through that node are affected. Monitors attached to the nodes present in the affected lightpaths trigger alarms. Hence, a single fault will generate multiple alarms. By placing monitors in the right way, we can minimize the number of alarms generated for a fault while keeping the fault-detection coverage maximum. In the problem we model below, we add the additional constraint that for any node failure that might occur, it triggers a unique set of alarms. This problem requires that each combination of monitor alarms is unique for each node fault. This requires that every pair of vectors of variables differ on at least one element. This can be specified in Numberjack by introducing a `HammingDistance` constraint. The Numberjack model for this problem is presented as Figure 6.

```
class HammingDistance(Expression):

 def __init__(self, row1, row2):
   Expression.__init__(self, "HammingDistance")
     self.set_children(row1+row2)
     self.rows = [row1, row2]

 def decompose(self):
   return [Sum([(var1 != var2) for var1, var2 in zip(self.rows[0],self.rows[1])])]

Nodes = 6        # We consider a graph with 6 nodes
Monitors = 10    # Faults on the nodes trigger 10 monitors

alarm_matrix = [ # Each vector specifies the monitors triggered by each node
   [1, 2, 3,                   10], [                  7          ],
   [               6, 7,         ], [            5, 6, 7,         ],
   [   2, 3, 4,          8,    10], [      3, 4,          8, 9, 10]  ]

monitors_on     = VarArray(Monitors)        # The decision variables
being_monitored = Matrix(Nodes, Monitors)

model = Model()                             # Specify the model...
model.add( Minimise(Sum(monitors_on)) )
model.add( [ monitor == ( Sum(col) >= 1 ) for col, monitor in
                           zip(being_monitored.col, monitors_on) ])
model.add( [ Sum(row) > 0 for row in being_monitored] )
model.add([HammingDistance(x1,x2) > 0 for x1, x2 in pair_of(being_monitored)])
for monitored_row, possible_monitor_row in zip(being_monitored, alarm_matrix):
   model.add([monitored_row[idx - 1] == 0 for idx in
              [x for x in range(Monitors) if x not in possible_monitor_row]])
```

Fig. 6. A Numberjack model for the optical network monitoring problem [1]

3 Experiments

Experiments ran on an Intel Xeon 2.66GHz machine with 12GB of ram on Fedora 9.

Experiment 1: Overhead due to Numberjack. We first assess the overhead of using a solver within Numberjack. We ran three back-end solvers, Mistral, MiniSat and SCIP on three arythmetic puzzles (Magic Square, Costas Array and Golomb Ruler). For each run, we used a profiler to separate the time spent executing Python code from the time spent executing code from the back-end solver. We report the results in Table 1. For every problem we report results averaged across 7 instances[4] of various size and 10 randomized runs each. The time spent executing the Python code is very modest, and of course independent of the hardness of the instance.

Table 1. Solver Time vs Python Time (Arithmetic puzzles)

Instance	Mistral Time (s)		MiniSat Time (s)		SCIP Time (s)	
	Solver	Python	Solver	Python	Solver	Python
`Magic-Square` (3 to 9)	0.0205	0.0101	59.7130	0.0116	35.85	0.0107
`Costas-Array` (6 to 12)	0.0105	0.0098	0.0666	0.0095	78.2492	0.0134
`Golomb-Ruler` (3 to 9)	0.5272	0.0056	56.0008	0.0055	118.1979	0.0076

Experiment: Comparison of Back-end Solvers. It is well known in the fields of Constraint Programming and Mixed Integer Programming that the areas have different strengths and weaknesses. For example, the CP and SAT solvers were much more efficient than the MIP solver for the arythmetic puzzles used in the first set of experiments (See Table 1). However, of course, the situation can be completely reversed on problems more suited to mathematical programming. We ran Numberjack on the Warehouse allocation problem (P34 of the CSPLib). This problem is easily solved using the Mixed Integer Solver SCIP as back end (1.86 seconds and 4.8 nodes in average over the 5 instances on the CSPLib) whilst Mistral ran over a time limit of one hour, staying well over the optimal allocation and exploring several million nodes.

4 Conclusion

Numberjack is a Python-based constraint programming system. It brings the power of combinatorial optimisation to Python programmers by supporting the specification of complex models and specifying how these should be solved. We presented the features of Numberjack through the use of several combinatorial problems.

Reference

1. Nayek, P., Pal, S., Choudhury, B., Mukherjee, A., Saha, D., Nasipuri, M.: Optimal monitor placement scheme for single fault detection in optical network. In: Proceedings of 2005 7th International Conference, vol. 1, pp. 433–436 (2005)

[4] The results of SCIP on the 3 hardest Magic Square instances are not taken into account since the cutoff of 1000 seconds was reached.

Automated Configuration of Mixed Integer Programming Solvers

Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown

University of British Columbia, 2366 Main Mall, Vancouver BC, V6T 1Z4, Canada
{hutter,hoos,kevinlb}@cs.ubc.ca

Abstract. State-of-the-art solvers for mixed integer programming (MIP) problems are highly parameterized, and finding parameter settings that achieve high performance for specific types of MIP instances is challenging. We study the application of an automated algorithm configuration procedure to different MIP solvers, instance types and optimization objectives. We show that this fully-automated process yields substantial improvements to the performance of three MIP solvers: CPLEX, GUROBI, and LPSOLVE. Although our method can be used "out of the box" without any domain knowledge specific to MIP, we show that it outperforms the CPLEX special-purpose automated tuning tool.

1 Introduction

Current state-of-the-art mixed integer programming (MIP) solvers are highly parameterized. Their parameters give users control over a wide range of design choices, including: which preprocessing techniques to apply; what balance to strike between branching and cutting; which types of cuts to apply; and the details of the underlying linear (or quadratic) programming solver. Solver developers typically take great care to identify default parameter settings that are robust and achieve good performance across a variety of problem types. However, the best combinations of parameter settings differ across problem types, which is of course the reason that such design choices are exposed as parameters in the first place. Thus, when a user is interested only in good performance for a given family of problem instances—as is the case in many application situations—it is often possible to substantially outperform the default configuration of the solver.

When the number of parameters is large, finding a solver configuration that leads to good empirical performance is a challenging optimization problem. (For example, this is the case for CPLEX: in version 12, its 221-page parameter reference manual describes 135 parameters that affect the search process.) MIP solvers exist precisely because humans are not good at solving high-dimensional optimization problems. Nevertheless, parameter optimization is usually performed manually. Doing so is tedious and laborious, requires considerable expertise, and often leads to results far from optimal.

There has been recent interest in automating the process of parameter optimization for MIP. The idea is to require the user to only specify a set of problem instances of interest and a performance metric, and then to trade machine time for human time to automatically identify a parameter configuration that achieves good performance. Notably, IBM ILOG CPLEX—the most widely used commercial MIP solver—introduced an automated tuning tool in version 11. In our own recent work, we proposed several methods for the automated configuration of various complex algorithms [20, 19, 18, 15]. While we mostly focused on solvers for propositional satisfiability (based on both local and tree search), we also conducted preliminary experiments that showed the promise of our methods for MIP. Specifically, we studied the automated configuration of CPLEX 10.1.1, considering 5 types of MIP instances [19].

The main contribution of this paper is a thorough study of the applicability of one of our black-box techniques to the MIP domain. We go beyond previous work by configuring three different MIP solvers (GUROBI, LPSOLVE, and the most recent CPLEX version 12.1); by considering a wider range of instance distributions; by considering multiple configuration objectives (notably, performing the first study on automatically minimizing the optimality gap); and by comparing our method to CPLEX's automated tuning tool. We show that our approach consistently sped up all three MIP solvers and also clearly outperformed the CPLEX tuning tool. For example, for a set of real-life instances from computational sustainability, our approach sped up CPLEX by a factor of 52 while the tuning tool returned the CPLEX defaults. For GUROBI, speedups were consistent but small (up to a factor of 2.3), and for LPSOLVE we obtained speedups up to a factor of 153.

The remainder of this paper is organized as follows. In the next section, we describe automated algorithm configuration, including existing tools and applications. Then, we describe the MIP solvers we chose to study (Section 3) and discuss the setup of our experiments (Section 4). Next, we report results for optimizing both the runtime of the MIP solvers (Section 5) and the optimality gap they achieve within a fixed time (Section 6). We then compare our approach to the CPLEX tuning tool (Section 7) and conclude with some general observations and an outlook on future work (Section 8).

2 Automated Algorithm Configuration

Whether manual or automated, effective algorithm configuration is central to the development of state-of-the-art algorithms. This is particularly true when dealing with $\mathcal{NP}$-hard problems, where the runtimes of weak and strong algorithms on the same problem instances regularly differ by orders of magnitude. Existing theoretical techniques are typically not powerful enough to determine whether one parameter configuration will outperform another, and therefore algorithm designers have to rely on empirical approaches.

2.1 The Algorithm Configuration Problem

The algorithm configuration problem we consider in this work involves an algorithm to be configured (a *target algorithm*) with a set of parameters that affect its performance, a set of problem instances of interest (*e.g.*, 100 vehicle routing problems), and a performance metric to be optimized (*e.g.*, average runtime; optimality gap). The target algorithm's parameters can be *numerical* (*e.g.*, level of a real-valued threshold); *ordinal* (*e.g.*, low, medium, high); *categorical* (*e.g.*, choice of heuristic), Boolean (*e.g.*, algorithm component active/inactive); and even *conditional* (*e.g.*, a threshold that affects the algorithm's behaviour only when a particular heuristic is chosen). In some cases,

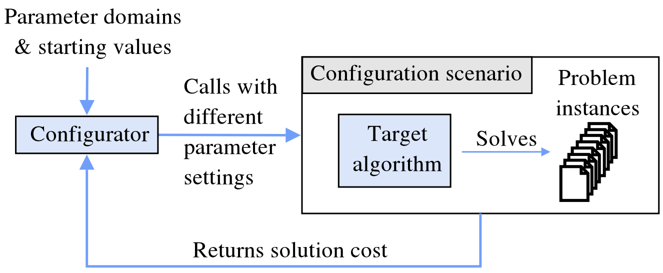


Fig. 1. A configuration procedure (short: configurator) executes the target algorithm with specified parameter settings on one or more problem instances, observes algorithm performance, and uses this information to decide which subsequent target algorithm runs to perform. A configuration scenario includes the target algorithm to be configured and a collection of instances.

a value for one parameter can be incompatible with a value for another parameter; for example, some types of preprocessing are incompatible with the use of certain data structures. Thus, some parts of parameter configuration space are *forbidden*; they can be described succinctly in the form of forbidden partial instantiations of parameters (*i.e.*, constraints).

We refer to instances of this algorithm configuration problem as *configuration scenarios*, and we address these using automatic methods that we call *configuration procedures*; this is illustrated in Figure 1. Observe that we treat algorithm configuration as a black-box optimization problem: a configuration procedure executes the target algorithm on a problem instance and receives feedback about the algorithm's performance without any access to the algorithm's internal state. (Because the CPLEX tuning tool is proprietary, we do not know whether it operates similarly.)

2.2 Configuration Procedures and Existing Applications

A variety of black-box, automated configuration procedures have been proposed in the CP and AI literatures. There are two major families: *model-based* approaches that learn a response surface over the parameter space, and *model-free* approaches that do not. Much existing work is restricted to scenarios having only relatively small numbers of numerical (often continuous) parameters, both in the model-based [7, 13, 17] and model-free [6, 1] literatures. Some relatively recent model-free approaches permit both larger numbers of parameters and categorical domains, in particular Composer [12], F-Race [9, 8], GGA [3], and our own ParamILS [20, 19]. As mentioned above, the automated tuning tool introduced in CPLEX version 11 can also be seen as a special-purpose algorithm configuration procedure; we believe it to be model free.

Blackbox configuration procedures have been applied to optimize a variety of parametric algorithms. Gratch and Chien [12] successfully applied the Composer system to optimize the five parameters of LR-26, an algorithm for scheduling communication between a collection of ground-based antennas and spacecraft in deep space. Adenso-Diaz and Laguna [1] demonstrated that their Calibra system was able to optimize the parameters of six unrelated metaheuristic algorithms, matching or surpassing the performance achieved manually by their developers. F-Race and its extensions have been used to optimize numerous algorithms, including iterated local search for the quadratic assignment problem, ant colony optimization for the travelling salesperson problem, and the best-performing algorithm submitted to the 2003 timetabling competition [8].

Our group successfully used various versions of PARAMILS to configure algorithms for a wide variety of problem domains. So far, the focus of that work has been on the configuration of solvers for the propositional satisfiability problem (SAT); we optimized both tree search [16] and local search solvers [21], in both cases substantially advancing the state of the art for the types of instances studied. We also successfully configured algorithms for the most probable explanation problem in Bayesian networks, global continuous optimization, protein folding, and algorithm configuration itself (for details, see Ref. 15).

2.3 Configuration Procedure Used: FOCUSEDILS

The configuration procedure used in this work is an instantiation of the PARAMILS framework [20, 19]. However, we do not mean to argue for the use of PARAMILS in particular, but rather aim to provide a lower bound on the performance improvements that can be achieved by applying general-purpose automated configuration tools to MIP solvers; future tools may achieve even better performance.

PARAMILS performs an iterated local search (ILS) in parameter configuration space; configurations are evaluated by running the target algorithm with them. The search is initialized at the best out of ten random parameter configurations and the target algorithm's default configuration. Next, PARAMILS performs a first-improvement local search that ends in a local optimum. It then iterates three phases: (1) a random perturbation to escape the local optimum; (2) another local search phase resulting in a new local optimum; and (3) an acceptance criterion that typically accepts the new local optimum if it is better than the previous one. The PARAMILS instantiation we used here is FOCUSEDILS version 2.4, which aggressively rejects poor configurations and focuses its efforts on the evaluation of good configurations. Specifically, it starts with performing only a single target algorithm run for each configuration considered, and performs additional runs for good configurations as the search progresses. This process guarantees that—given enough time and a training set that is perfectly representative of unseen test instances—FOCUSEDILS will identify the best configuration in the given design space [20, 19]. (Further details of PARAMILS and FOCUSEDILS can be found in our previous publications [20, 19].)

In practice, we are typically forced to work with finite sets of benchmark instances, and performance on a small training set is often not very representative for performance on other, unseen instances of similar origin. PARAMILS (and any other configuration tool) can only optimize performance on the training set it is given; it cannot guarantee that this leads to improved performance on a separate set of test instances. In particular, with very small training sets, a so-called *over-tuning* effect can occur: given more time, automated configuration tools find configurations with better training but worse test performance [8, 20].

Since target algorithm runs with some parameter configurations may take a very long (potentially infinite) time, PARAMILS requires the user to specify a so-called *captime*

Table 1. Target algorithms and characteristics of their parameter configuration spaces. For details, see `http://www.cs.ubc.ca/labs/beta/Projects/MIP-Config/`

Algorithm	Parameter type	# parameters of this type	# values considered	Total # configurations
CPLEX MILP (MIQCP)	Boolean	6 (7)	2	$1.90 \cdot 10^{47}$ ($3.40 \cdot 10^{45}$)
	Categorical	45 (43)	3–7	
	Integer	18	5–7	
	Continuous	7	5–8	
GUROBI	Boolean	4	2	$3.84 \cdot 10^{14}$
	Categorical	16	3–5	
	Integer	3	5	
	Continuous	2	5	
LPSOLVE	Boolean	40	2	$1.22 \cdot 10^{15}$
	Categorical	7	3–8	

κ_{max}, the maximal amount of time after which PARAMILS will terminate a run of the target algorithm as unsuccessful. FOCUSEDILS version 2.4 also supports *adaptive capping*, a speedup technique that sets the captimes $\kappa \leq \kappa_{max}$ for individual target algorithm runs, thus permitting substantial savings in computation time.

FOCUSEDILS is a randomized algorithm that tends to be quite sensitive to the ordering of its training benchmark instances. For challenging configuration tasks some of its runs often perform much better than others. For this reason, in previous work we adopted the strategy to perform 10 independent parallel runs of FOCUSEDILS and use the result of the run with best *training* performance [16, 19]. This is sound since no knowledge of the test set is required in order to make the selection; the only drawback is a 10-fold increase in overall computation time. If none of the 10 FOCUSEDILS runs encounters any successful algorithm run, then our procedure returns the algorithm default.

3 MIP Solvers

We now discuss the three MIP solvers we chose to study and their respective parameter configuration spaces. Table 1 gives an overview.

IBM ILOG CPLEX is the most-widely used commercial optimization tool for solving MIPs. As stated on the CPLEX website (`http://www.ilog.com/products/cplex/`), currently over 1 300 corporations and government agencies use CPLEX, along with researchers at over 1 000 universities. CPLEX is massively parameterized and end users often have to experiment with these parameters:

> "Integer programming problems are more sensitive to specific parameter settings, so you may need to experiment with them." (ILOG CPLEX 12.1 user manual, page 235)

Thus, the automated configuration of CPLEX is very promising and has the potential to directly impact a large user base.

We used CPLEX 12.1 (the most recent version) and defined its parameter configuration space as follows. Using the CPLEX 12 "parameters reference manual", we identified 76 parameters that can be modified in order to optimize performance. We were careful to keep all parameters fixed that change the problem formulation (*e.g.*, parameters such as the optimality gap below which a solution is considered optimal). The

76 parameters we selected affect all aspects of CPLEX. They include 12 preprocessing parameters (mostly categorical); 17 MIP strategy parameters (mostly categorical); 11 categorical parameters deciding how aggressively to use which types of cuts; 9 numerical MIP "limits" parameters; 10 simplex parameters (half of them categorical); 6 barrier optimization parameters (mostly categorical); and 11 further parameters. Most parameters have an "automatic" option as one of their values. We allowed this value, but also included other values (all other values for categorical parameters, and a range of values for numerical parameters). Exploiting the fact that 4 parameters were conditional on others taking certain values, these 76 parameters gave rise to $1.90 \cdot 10^{47}$ distinct parameter configurations. For mixed integer quadratically-constrained problems (MIQCP), there were some additional parameters (1 binary and 1 categorical parameter with 3 values). However, 3 categorical parameters with 4, 6, and 7 values were no longer applicable, and for one categorical parameter with 4 values only 2 values remained. This led to a total of $3.40 \cdot 10^{45}$ possible configurations.

GUROBI is a recent commercial MIP solver that is competitive with CPLEX on some types of MIP instances [23]. We used version 2.0.1 and defined its configuration space as follows. Using the online description of GUROBI's parameters,[1] we identified 26 parameters for configuration. These consisted of 12 mostly-categorical parameters that determine how aggressively to use each type of cuts, 7 mostly-categorical simplex parameters, 3 MIP parameters, and 4 other mostly-Boolean parameters. After disallowing some problematic parts of configuration space (see Section 4.2), we considered 25 of these 26 parameters, which led to a configuration space of size $3.84 \cdot 10^{14}$.

LPSOLVE is one of the most prominent open-source MIP solvers. We determined 52 parameters based on the information at `http://lpsolve.sourceforge.net/`. These parameters are rather different from those of GUROBI and CPLEX: 7 parameters are categorical, and the rest are Boolean switches indicating whether various solver modules should be employed. 17 parameters concern presolving; 9 concern pivoting; 14 concern the branch & bound strategy; and 12 concern other functions. After disallowing problematic parts of configuration space (see Section 4.2), we considered 47 of these 52 parameters. Taking into account one conditional parameter, these gave rise to $1.22 \cdot 10^{15}$ distinct parameter configurations.

4 Experimental Setup

We now describe our experimental setup: benchmark sets, how we identified problematic parts in the configuration spaces of GUROBI and LPSOLVE, and our computational environment.

4.1 Benchmark Sets

We collected a wide range of MIP benchmarks from public benchmark libraries and other researchers, and split each of them 50:50 into disjoint training and test sets; we detail these in the following.

[1] `http://www.gurobi.com/html/doc/refman/node378.html#sec:Parameters`

MJA. This set comprises 343 machine-job assignment instances encoded as mixed integer quadratically constrained programming (MIQCP) problems [2]. We obtained it from the Berkeley Computational Optimization Lab (BCOL).[2] On average, these instances contain 2 769 variables and 2 255 constraints (with standard deviations 2 133 and 1 592, respectively).

MIK. This set comprises 120 mixed-integer knapsack instances encoded as mixed integer linear programming (MILP) problems [4]; we also obtained it from BCOL. On average, these instances contain 384 variables and 151 constraints (with standard deviations 309 and 127, respectively).

CLS. This set of 100 MILP-encoded capacitated lot-sizing instances [5] was also obtained from BCOL. Each instance contains 181 variables and 180 constraints.

REGIONS100. This set comprises 2 000 instances of the combinatorial auction winner determination problem, encoded as MILP instances. We generated them using the `regions` generator from the Combinatorial Auction Test Suite [22], with parameters *goods*=100 and *bids*=500. On average, the resulting MILP instances contain 501 variables and 193 inequalities (with standard deviations 1.7 and 2.5, respectively).

REGIONS200. This set contains 2 000 instances similar to those in REGIONS100 but larger; we created it with the same generator using *goods*=200 and *bids*=1 000. On average, the resulting MILP instances contain 1 002 variables and 385 inequalities (with standard deviations 1.7 and 3.4, respectively).

MASS. This set comprises 100 integer programming instances modelling multi-activity shift scheduling [10]. On average, the resulting MILP instances contain 81 994 variables and 24 637 inequalities (with standard deviations 9 725 and 5 391, respectively).

CORLAT. This set comprises 2 000 MILP instances based on real data used for the construction of a wildlife corridor for grizzly bears in the Northern Rockies region (the instances were described by Gomes et al. [11] and made available to us by Bistra Dilkina). All instances had 466 variables; on average they had 486 constraints (with standard deviation 25.2).

4.2 Avoiding Problematic Parts of Parameter Configuration Space

Occasionally, we encountered problems running GUROBI and LPSOLVE with certain combinations of parameters on particular problem instances. These problems included segmentation faults as well as several more subtle failure modes, in which incorrect results could be returned by a solver. (CPLEX did not show these problems on any of the instances studied here.) To deal with them, we took the following measures in our experimental protocol. First, we established reference solutions for all MIP instances using CPLEX 11.2 and GUROBI, both run with their default parameter configurations for up to one CPU hour per instance.[3] (For some instances, neither of the two solvers could find a solution within this time; for those instances, we skipped the correctness check described in the following.)

[2] `http://www.ieor.berkeley.edu/~atamturk/bcol/`, where this set is called `conic.sch`.

[3] These reference solutions were established before we had access to CPLEX 12.1.

In order to identify problematic parts of a given configuration space, we ran 10 PARAMILS runs (with a time limit of 5 hours each) until one of them encountered a target algorithm run that either produced an incorrect result (as compared to our reference solution for the respective MIP instance), or a segmentation fault. We call the parameter configuration $\boldsymbol{\theta}$ of such a run *problematic*. Starting from this problematic configuration $\boldsymbol{\theta}$, we then identified what we call a *minimal problematic configuration* $\boldsymbol{\theta}_{min}$. In particular, we iteratively changed the value of one of $\boldsymbol{\theta}$'s parameters to its respective default value, and repeated the algorithm run with the same instance, captime, and random seed. If the run still had problems with the modified parameter value, we kept the parameter at its default value, and otherwise changed it back to the value it took in $\boldsymbol{\theta}$. Iterating this process converges to a problematic configuration $\boldsymbol{\theta}_{min}$ that is minimal in the following sense: setting any single non-default parameter value of $\boldsymbol{\theta}_{min}$ to its default value resolves the problem in the current target algorithm run.

Using PARAMILS's mechanism of forbidden partial parameter instantiations, we then forbade any parameter configurations that included the partial configuration defined by $\boldsymbol{\theta}_{min}$'s non-default parameter values. (When all non-default values for a parameter became problematic, we did not consider that parameter for configuration, clamping it to its default value.) We repeated this process until no problematic configuration was found in the PARAMILS runs: 4 times for GUROBI and 14 times for LPSOLVE. Thereby, for GUROBI we removed one problematic parameter and disallowed two further partial configurations, reducing the size of the configuration space from $1.32 \cdot 10^{15}$ to $3.84 \cdot 10^{14}$. For LPSOLVE, we removed 5 problematic binary flags and disallowed 8 further partial configurations, reducing the size of the configuration space from $8.83 \cdot 10^{16}$ to $1.22 \cdot 10^{15}$. Details on forbidden parameters and partial configurations, as well as supporting material, can be found at `http://www.cs.ubc.ca/labs/beta/Projects/MIP-Config/`

While that first stage resulted in concise bug reports we sent to GUROBI and LPSOLVE, it is not essential to algorithm configuration. Even after that stage, in the experiments reported here, target algorithm runs occasionally disagreed with the reference solution or produced segmentation faults. We considered the empirical cost of those runs to be ∞, thereby driving the local search process underlying PARAMILS away from problematic parameter configurations. This allowed PARAMILS to gracefully handle target algorithm failures that we had not observed in our preliminary experiments. We could have used the same approach without explicitly identifying and forbidding problematic configurations.

4.3 Computational Environment

We carried out the configuration of LPSOLVE on the 840-node Westgrid Glacier cluster, each with two 3.06 GHz Intel Xeon 32-bit processors and 2–4GB RAM. All other configuration experiments, as well as all evaluation, was performed on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 10.1; runtimes were measured as CPU time on these reference machines.

Table 2. Results for minimizing the runtime required to find an optimal solution and prove its optimality. All results are for test sets disjoint from the training sets used for the automated configuration. We report the percentage of timeouts after 24 CPU hours as well as the mean runtime for those instances that were solved by both approaches. Bold-faced entries indicate better performance of the configurations found by PARAMILS than for the default configuration. (To reduce the computational burden, results for LPSOLVE on REGIONS200 and CORLAT are only based on 100 test instances sampled uniformly at random from the 1000 available ones.)

Algorithm	Scenario	% test instances unsolved in 24h		mean runtime for solved [CPU s]		Speedup
		default	PARAMILS	default	PARAMILS	factor
CPLEX	MJA	0%	0%	3.40	**1.72**	**1.98**×
	MIK	0%	0%	4.87	**1.61**	**3.03**×
	REGIONS100	0%	0%	0.74	**0.35**	**2.13**×
	REGIONS200	0%	0%	59.8	**11.6**	**5.16**×
	CLS	0%	0%	47.7	**12.1**	**3.94**×
	MASS	0%	0%	524.9	**213.7**	**2.46**×
	CORLAT	0%	0%	850.9	**16.3**	**52.3**×
GUROBI	MIK	0%	0%	2.70	**2.26**	**1.20**×
	REGIONS100	0%	0%	2.17	**1.27**	**1.71**×
	REGIONS200	0%	0%	56.6	**40.2**	**1.41**×
	CLS	0%	0%	58.9	**47.2**	**1.25**×
	MASS	0%	0%	493	**281**	**1.75**×
	CORLAT	0.3%	**0.2%**	103.7	**44.5**	**2.33**×
LPSOLVE	MIK	63%	63%	21 670	21 670	1×
	REGIONS100	0%	0%	9.52	**1.71**	**5.56**×
	REGIONS200	12%	**0%**	19 000	**124**	**153**×
	CLS	86%	**42%**	39 300	**1 440**	**27.4**×
	MASS	83%	83%	8 661	8 661	1×
	CORLAT	50%	**8%**	7 916	**229**	**34.6**×

5 Minimization of Runtime Required to Prove Optimality

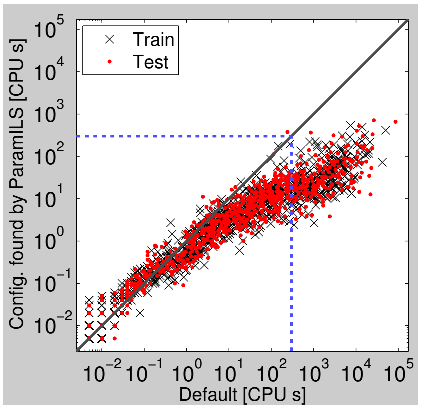
In our first set of experiments, we studied the extent to which automated configuration can improve the time performance of CPLEX, GUROBI, and LPSOLVE for solving the seven types of instances discussed in Section 4.1. This led to $3 \cdot 6 + 1 = 19$ configuration scenarios (the quadratically constrained MJA instances could only be solved with CPLEX).

For each configuration scenario, we allowed a total configuration time budget of 2 CPU days for each of our 10 PARAMILS runs, with a captime of $\kappa_{max} = 300$ seconds for each MIP solver run. In order to penalize timeouts, during configuration we used the penalized average runtime criterion (dubbed "PAR-10" in our previous work [19]), counting each timeout as $10 \cdot \kappa_{max}$. For evaluation, we report timeouts separately.

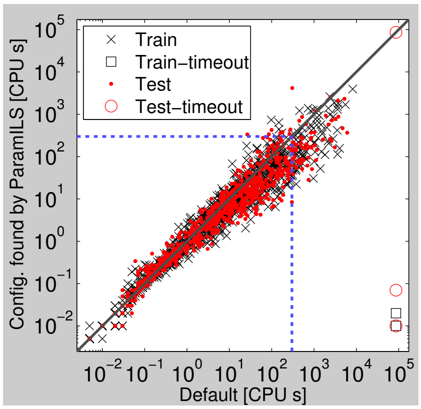
For each configuration scenario, we compared the performance of the parameter configuration identified using PARAMILS against the default configuration, using a test set of instances disjoint from the training set used during configuration. We note that this default configuration is typically determined using substantial time and effort; for example, the CPLEX 12.1 user manual states (on p. 478):

"A great deal of algorithmic development effort has been devoted to establishing default ILOG CPLEX parameter settings that achieve good performance on a wide variety of MIP models."

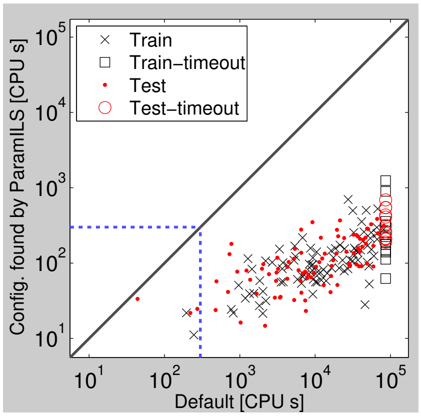
Table 2 describes our configuration results. For each of the benchmark sets, our approach improved CPLEX's performance. Specifically, we achieved speedups ranging from 2-fold to 52-fold. For GUROBI, the speedups were also consistent, but less pronounced (1.2-fold to 2.3-fold). For the open-source solver LPSOLVE, the speedups were most substantial, but there were also 2 cases in which PARAMILS did not improve over LPSOLVE's default, namely the MIK and MASS benchmarks. This occurred because, within the maximum captime of $\kappa_{max} = 300s$ we used during configuration, none of the thousands of LPSOLVE runs performed by PARAMILS solved a single benchmark



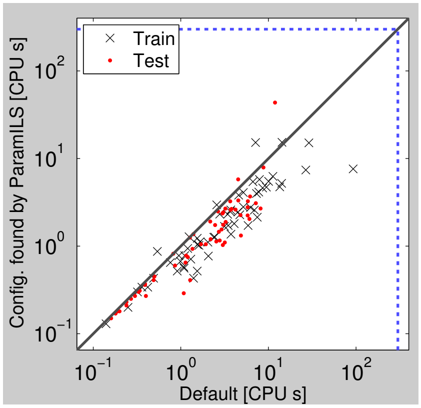
(a) CPLEX, CORLAT. Speedup factors: train 48.4×, test 52.3×.



(b) GUROBI, CORLAT. Speedup factors: train 2.24×, test 2.33×.



(c) LPSOLVE, REGIONS200. Speedup factors: train 162×, test 153×.



(d) GUROBI, MIK. Speedup factors: train 2.17×, test 1.20×.

Fig. 2. Results for configuration of MIP solvers to reduce the time for finding an optimal solution and proving its optimality. The dashed blue line indicates the captime ($\kappa_{max} = 300s$) used during configuration.

instance for either of the two benchmark sets. For the other benchmarks, speedups were very substantial, reaching up to a factor of 153 (on REGIONS200).

Figure 2 shows the speedups for 4 configuration scenarios. Figures 2(a) to (c) show the scenario with the largest speedup for each of the solvers. In all cases, PARAMILS's configurations scaled better to hard instances than the algorithm defaults, which in some cases timed out on the hardest instances. PARAMILS's *worst* performance was for the 2 LPSOLVE scenarios for which it simply returned the default configuration; in Figure 2(d), we show results for the more interesting second-worst case, the configuration of GUROBI on MIK. Observe that here, performance was actually rather good for most instances, and that the poor speedup in test performance was due to a single hard test instance. Better generalization performance would be achieved if more training instances were available.

6 Minimization of Optimality Gap

Sometimes, we are interested in minimizing a criterion other than mean runtime. Algorithm configuration procedures such as PARAMILS can in principle deal with various optimization objectives; in our own previous work, for example, we have optimized median runlength, average speedup over an existing algorithm, and average solution quality [20, 15]. In the MIP domain, constraints on the time available for solving a given MIP instance might preclude running the solver to completion, and in such cases, we may be interested in minimizing the optimality gap (also known as MIP gap) achieved within a fixed amount of time, T.

To investigate the efficacy of our automated configuration approach in this context, we applied it to CPLEX, GUROBI and LPSOLVE on the 5 benchmark distributions with

Table 3. Results for configuration of MIP solvers to reduce the relative optimality gap reached within 10 CPU seconds. We report the percentage of test instances for which no feasible solution was found within 10 seconds and the mean relative gap for the remaining test instances. Bold face indicates the better configuration (recall that our lexicographic objective function cares first about the number of instances with feasible solutions, and then considers the mean gap among feasible instances only to break ties).

Algorithm	Scenario	% test instances for which no feas. sol. was found		mean gap when feasible		Gap reduction factor
		default	PARAMILS	default	PARAMILS	
CPLEX	MIK	0%	0%	0.15%	**0.02%**	**8.65×**
	CLS	0%	0%	0.27%	**0.15%**	**1.77×**
	REGIONS200	0%	0%	1.90%	**1.10%**	**1.73×**
	CORLAT	28%	**1%**	4.43%	**1.22%**	**2.81×**
	MASS	88%	**86%**	1.91%	**1.52%**	**1.26×**
GUROBI	MIK	0%	0%	0.02%	**0.01%**	**2.16×**
	CLS	0%	0%	0.53%	**0.44%**	**1.20×**
	REGIONS200	0%	0%	3.17%	**2.52%**	**1.26×**
	CORLAT	14%	**5%**	3.22%	**2.87%**	**1.12×**
	MASS	68%	68%	76.4%	**52.2%**	**1.46×**
LPSOLVE	MIK	0%	0%	652%	**14.3%**	**45.7×**
	CLS	0%	0%	29.6%	**7.39%**	**4.01×**
	REGIONS200	0%	0%	10.8%	**6.60%**	**1.64×**
	CORLAT	68%	**13%**	4.19%	**3.42%**	**1.20×**
	MASS	100%	100%	-	-	-

the longest average runtimes, with the objective of minimizing the average relative optimality gap achieved within $T = 10$ CPU seconds. To deal with runs that did not find feasible solutions, we used a lexicographic objective function that counts the fraction of instances for which feasible solutions were found and breaks ties based on the mean relative gap for those instances. For each of the 15 configuration scenarios, we performed 10 PARAMILS runs, each with a time budget of 5 CPU hours.

Table 3 shows the results of this experiment. For all but one of the 15 configuration scenarios, the automatically-found parameter configurations performed substantially better than the algorithm defaults. In 4 cases, feasible solutions were found for more instances, and in 14 scenarios the relative gaps were smaller (sometimes substantially so; consider, *e.g.*, the 45-fold reduction for LPSOLVE, and note that the gap is not bounded by 100%). For the one configuration scenario where we did not achieve an improvement, LPSOLVE on MASS, the default configuration of LPSOLVE could not find a feasible solution for *any* of the training instances in the available 10 seconds, and the same turned out to be the case for the thousands of configurations considered by PARAMILS.

7 Comparison to CPLEX Tuning Tool

The CPLEX tuning tool is a built-in CPLEX function available in versions 11 and above.[4] It allows the user to minimize CPLEX's runtime on a given set of instances. As in our approach, the user specifies a per-run captime, the default for which is $\kappa_{max} = 10\,000$ seconds, and an overall time budget. The user can further decide whether to minimize mean or maximal runtime across the set of instances. (We note that the mean is usually dominated by the runtimes of the hardest instances.) By default, the objective for tuning is to minimize mean runtime, and the time budget is set to infinity, allowing the CPLEX tuning tool to perform all the runs it deems necessary.

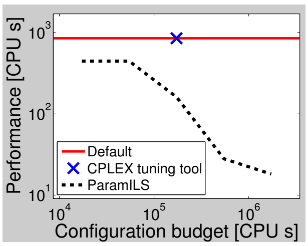
Since CPLEX is proprietary, we do not know the inner workings of the tuning tool; however, we can make some inferences from its outputs. In our experiments, it always started by running the default parameter configuration on each instance in the benchmark set. Then, it tested a set of named parameter configurations, such as 'no_cuts', 'easy', and 'more_gomory_cuts'. Which configurations it tested depended on the benchmark set.

PARAMILS differs from the CPLEX tuning tool in at least three crucial ways. First, it searches in the vast space of all possible configurations, while the CPLEX tuning tool focuses on a small set of handpicked candidates. Second, PARAMILS is a randomized procedure that can be run for any amount of time, and that can find different solutions when multiple copies are run in parallel; it reports better configurations as it finds them. The CPLEX tuning tool is deterministic and runs for a fixed amount of time (dependent on the instance set given) unless the time budget intervenes earlier; it does not return a configuration until it terminates. Third, because PARAMILS does not rely on domain-specific knowledge, it can be applied out of the box to the configuration of other MIP

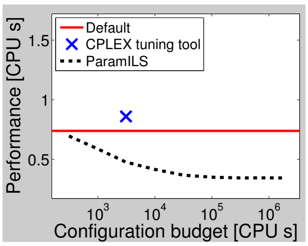
[4] Incidentally, our first work on the configuration of CPLEX predates the CPLEX tuning tool. This work, involving Hutter, Hoos, Leyton-Brown, and Stützle, was presented and published as a technical report at a doctoral symposium in Sept. 2007 [14]. At that time, no other mechanism for automatically configuring CPLEX was available; CPLEX 11 was released Nov. 2007.

Table 4. Comparison of our approach against the CPLEX tuning tool. For each benchmark set, we report the time t required by the CPLEX tuning tool (it ran out of time after 2 CPU days for REGIONS200 and CORLAT, marked by '*') and the CPLEX name of the configuration it judged best. We report the mean runtime of the default configuration; the configuration the tuning tool selected; and the configurations selected using 2 sets of 10 PARAMILS runs, each allowed time $t/10$ and 2 days, respectively. For the PARAMILS runs, in parentheses we report the speedup over the CPLEX tuning tool. Boldface indicates improved performance.

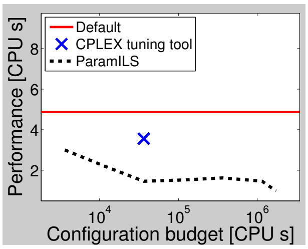
Scenario	CPLEX tuning tool stats		CPLEX mean runtime [CPU s] on test set, with respective configuration			
	Tuning time t	Name of result	Default	CPLEX tuning tool	10× PARAMILS(t/10)	10× PARAMILS(2 days)
CLS	104 673	'defaults'	48.4	48.4	**15.1(3.21×)**	**10.1(4.79×)**
REGIONS100	3 117	'easy'	0.74	0.86	**0.48(1.79×)**	**0.34(2.53×)**
REGIONS200	172 800*	'defaults'	59.8	59.8*	**14.2(4.21×)**	**11.9(5.03×)**
MIK	36 307	'long_test1'	4.87	3.56	**1.46(2.44×)**	**0.98(3.63×)**
MJA	2 266	'easy'	3.40	3.18	**2.71(1.17×)**	**1.64(1.94×)**
MASS	28 844	'branch_dir'	524.9	**425.8**	627.4(0.68×)	478.9(0.89×)
CORLAT	172 800*	'defaults'	850.9	850.9*	**161.1(5.28×)**	**18.2(46.8×)**



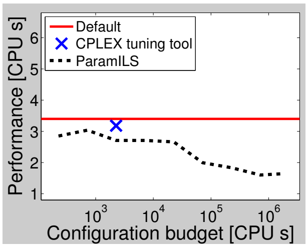
(a) CORLAT



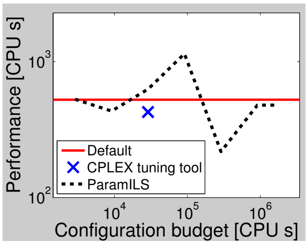
(b) REGIONS100



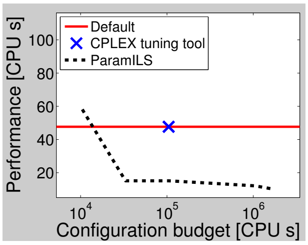
(c) MIK



(d) MJA



(e) MASS



(f) CLS

Fig. 3. Comparison of the default configuration and the configurations returned by the CPLEX tuning tool and by our approach. The x-axis gives the total time budget used for configuration and the y-axis the performance (CPLEX mean CPU time on the test set) achieved within that budget. For PARAMILS, we perform 10 runs in parallel and count the total time budget as the sum of their individual time requirements. The plot for REGIONS200 is qualitatively similar to the one for REGIONS100, except that the gains of PARAMILS are larger.

solvers and, indeed, arbitrary parameterized algorithms. In contrast, the few configurations in the CPLEX tuning tool appear to have been selected based on substantial domain insights, and the fact that different parameter configurations are tried for different types of instances leads us to believe that it relies upon MIP-specific instance characteristics.

While in principle this could be an advantage, in its current form it appears to be rather restrictive.

We compared the performance of the configurations found by the CPLEX tuning tool to that of configurations found by PARAMILS. For this comparison, we used the tuning tool's default settings to optimize mean runtime on the same training sets used for PARAMILS, and tested performance on the same test sets (disjoint from the training sets). We ran both configuration approaches with a time limit of 2 CPU days. In most cases, the CPLEX tuning tool finished before that time limit was reached and—in contrast to PARAMILS—could not use the remaining time in order to further improve performance. As before, we used 10 independent parallel runs of PARAMILS, at each time step reporting the performance of the one with best training performance.

First, we discuss the performance of the CPLEX tuning tool, summarized in Table 4. We note that in two cases (REGIONS200 and CORLAT), it reached the time limit of 2 CPU days and returned the algorithm defaults in both cases. Out of the remaining 5 cases, it returned the default configuration in 1 (CLS), yielded a configuration with slightly *worse* performance than the default in 1 (REGIONS100), and moderately improved performance in the remaining 3 (up to a factor of 1.37). The 3 non-default configurations it returned only differed in the following few parameters from the default: 'easy' (perform only 1 cutting plane pass, apply the periodic heuristic every 50 nodes, and branch based on pseudo-reduced costs); 'long_test1' (use aggressive probing and aggressive settings for 8 types of cuts); and 'branch_dir' (at each node, select the up branch first).

PARAMILS outperformed the tuning tool for 6 of the 7 configuration scenarios, sometimes substantially so. Specifically, PARAMILS found configurations with up to 5.2 times lower mean runtime when its total time budget was set to exactly the amount of time t the CPLEX tuning tool ran before terminating (*i.e.*, $t/10$ for each of the 10 PARAMILS runs; t varied widely across the scenarios, see Table 4). For the one remaining scenario, MASS, the configuration it found was slower by a factor of $1/0.68 = 1.47$ (which we attribute to an over-tuning effect to be discussed shortly). With a fixed time budget of two days for each PARAMILS run, PARAMILS's performance improved for all seven domains, reaching a speedup factor of up to 46.

Figure 3 visualizes the anytime test performance of PARAMILS compared to the default and the configuration found by the CPLEX tuning tool. Typically, PARAMILS found good configurations quickly and improved further when given more time. The main exception was configuration scenario MASS (see Figure 3(e)), the one scenario where PARAMILS performed worse than the CPLEX tuning tool in Table 4. Here, test performance did not improve monotonically: given more time, PARAMILS found configurations with better training performance but worse test performance. This example of the over-tuning phenomenon mentioned in Section 2.3 clearly illustrates the problems arising from benchmark sets that are too small (and too heterogeneous): good results for 50 (rather variable) training instances are simply not enough to confidently draw conclusions about the performance on additional unseen test instances. On all other 6 configuration scenarios, training and test sets were similar enough to yield near-monotonic improvements over time, and large speedups over the CPLEX tuning tool.

8 Conclusions and Future Work

In this study we have demonstrated that by using automated algorithm configuration, substantial performance improvements can be obtained for three widely used MIP solvers on a broad range of benchmark sets, in terms of minimizing runtime for proving optimality (with speedup factors of up to 52), and of minimizing the optimality gap given a fixed runtime (with gap reduction factors of up to 45). This is particularly noteworthy considering the effort that has gone into optimizing the default configurations for commercial MIP solvers, such as CPLEX and GUROBI. Our approach also clearly outperformed the CPLEX tuning tool. The success of our fully automated approach depends on the availability of training benchmark sets that are large enough to allow generalization to unseen test instances. Not surprisingly, when using relatively small benchmark sets, performance improvements on training instances sometimes do not fully translate to test instances; we note that this effect can be avoided by using bigger benchmark sets (in our experience, about 1000 instances are typically sufficient).

In future work, we plan to develop more robust and more efficient configuration procedures. In particular, here (and in past work) we ran our configurator PARAMILS 10 times per configuration scenario and selected the configuration with best performance on the training set in order to handle poorly-performing runs. We hope to develop more robust approaches that do not suffer from large performance differences between independent runs. Another issue is the choice of captimes. Here, we chose rather large captimes during training to avoid the risk of poor scaling to harder instances; the downside is a potential increase in the time budget required for finding good configurations. We therefore plan to investigate strategies for automating the choice of captimes during configuration. We also plan to study *why* certain parameter configurations work better than others. The algorithm configuration approach we have used here, PARAMILS, can identify very good (possibly optimal) configurations, but it does not yield information on the importance of each parameter, interactions between parameters, or the interaction between parameters and characteristics of the problem instances at hand. Partly to address those issues, we are actively developing an alternative algorithm configuration approach that is based on response surface models [17, 18, 15].

Acknowledgements

We thank the authors of the MIP benchmark instances we used for making them available, in particular Louis-Martin Rousseau and Bistra Dilkina, who provided the previously unpublished instance sets MASS and CORLAT, respectively. We also thank IBM and Gurobi Optimization for making a full version of their MIP solvers freely available for academic purposes; and Westgrid for support in using their compute cluster. FH gratefully acknowledges support from a postdoctoral research fellowship by the Canadian Bureau for International Education. HH and KLB gratefully acknowledge support from NSERC through their respective discovery grants, and from the MITACS NCE for seed project funding.

References

[1] Adenso-Diaz, B., Laguna, M.: Fine-tuning of algorithms using fractional experimental design and local search. Operations Research 54(1), 99–114 (2006)
[2] Aktürk, S.M., Atamtürk, A., Gürel, S.: A strong conic quadratic reformulation for machine-job assignment with controllable processing times. Research Report BCOL.07.01, University of California-Berkeley (2007)
[3] Ansotegui, C., Sellmann, M., Tierney, K.: A gender-based genetic algorithm for the automatic configuration of solvers. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 142–157. Springer, Heidelberg (2009)
[4] Atamtürk, A.: On the facets of the mixed–integer knapsack polyhedron. Mathematical Programming 98, 145–175 (2003)
[5] Atamtürk, A., Muñoz, J.C.: A study of the lot-sizing polytope. Mathematical Programming 99, 443–465 (2004)
[6] Audet, C., Orban, D.: Finding optimal algorithmic parameters using the mesh adaptive direct search algorithm. SIAM Journal on Optimization 17(3), 642–664 (2006)
[7] Bartz-Beielstein, T.: Experimental Research in Evolutionary Computation: The New Experimentalism. Natural Computing Series. Springer, Berlin (2006)
[8] Birattari, M.: The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective. PhD thesis, Université Libre de Bruxelles, Brussels, Belgium (2004)
[9] Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K.: A racing algorithm for configuring metaheuristics. In: Proc. of GECCO 2002, pp. 11–18 (2002)
[10] Cote, M., Gendron, B., Rousseau, L.: Grammar-based integer programing models for multi-activity shift scheduling. Technical Report CIRRELT-2010-01, Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport (2010)
[11] Gomes, C.P., van Hoeve, W.-J., Sabharwal, A.: Connections in networks: A hybrid approach. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 303–307. Springer, Heidelberg (2008)
[12] Gratch, J., Chien, S.A.: Adaptive problem-solving for large-scale scheduling problems: A case study. JAIR 4, 365–396 (1996)
[13] Huang, D., Allen, T.T., Notz, W.I., Zeng, N.: Global optimization of stochastic black-box systems via sequential kriging meta-models. Journal of Global Optimization 34(3), 441–466 (2006)
[14] Hutter, F.: On the potential of automatic algorithm configuration. In: SLS-DS2007: Doctoral Symposium on Engineering Stochastic Local Search Algorithms, pp. 36–40. Technical report TR/IRIDIA/2007-014, IRIDIA, Université Libre de Bruxelles, Brussels, Belgium (2007)
[15] Hutter, F.: Automated Configuration of Algorithms for Solving Hard Computational Problems. PhD thesis, University of British Columbia, Department of Computer Science, Vancouver, Canada (2009)
[16] Hutter, F., Babić, D., Hoos, H.H., Hu, A.J.: Boosting Verification by Automatic Tuning of Decision Procedures. In: Proc. of FMCAD 2007, Washington, DC, USA, pp. 27–34. IEEE Computer Society, Los Alamitos (2007a)
[17] Hutter, F., Hoos, H.H., Leyton-Brown, K., Murphy, K.P.: An experimental investigation of model-based parameter optimisation: SPO and beyond. In: Proc. of GECCO 2009, pp. 271–278 (2009a)
[18] Hutter, F., Hoos, H.H., Leyton-Brown, K., Murphy, K.P.: Time-bounded sequential parameter optimization. In: Proc. of LION-4. LNCS. Springer, Heidelberg (to appear, 2010)

[19] Hutter, F., Hoos, H.H., Leyton-Brown, K., Stützle, T.: ParamILS: an automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36, 267–306 (2009b)

[20] Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: Proc. of AAAI 2007, pp. 1152–1157 (2007b)

[21] KhudaBukhsh, A., Xu, L., Hoos, H.H., Leyton-Brown, K.: SATenstein: Automatically building local search SAT solvers from components. In: Proc. of IJCAI 2009, pp. 517–524 (2009)

[22] Leyton-Brown, K., Pearson, M., Shoham, Y.: Towards a universal test suite for combinatorial auction algorithms. In: Proc. of EC 2000, pp. 66–76. ACM, New York (2000)

[23] Mittelmann, H.: Mixed integer linear programming benchmark, serial codes (2010), `http://plato.asu.edu/ftp/milpf.html` (version last visited on January 26, 2010)

Upper Bounds on the Number of Solutions of Binary Integer Programs

Siddhartha Jain, Serdar Kadioglu*, and Meinolf Sellmann*

Brown University, Department of Computer Science,
115 Waterman Street, P.O. Box 1910, Providence, RI 02912
{sj10,serdark,sello}@cs.brown.edu

Abstract. We present a new method to compute upper bounds of the number of solutions of binary integer programming (BIP) problems. Given a BIP, we create a dynamic programming (DP) table for a redundant knapsack constraint which is obtained by surrogate relaxation. We then consider a Lagrangian relaxation of the original problem to obtain an initial weight bound on the knapsack. This bound is then refined through subgradient optimization. The latter provides a variety of Lagrange multipliers which allow us to filter infeasible edges in the DP table. The number of paths in the final table then provides an upper bound on the number of solutions. Numerical results show the effectiveness of our counting framework on automatic recording and market split problems.

Keywords: solution counting, CP-based Lagrangian relaxation, surrogate relaxation, dynamic programming.

1 Introduction

Solution counting has become a new and exciting topic in combinatorial research. Counting solutions of combinatorial problem instances is relevant for example for new branching methods [23,24]. It is also relevant to give user feedback in interactive settings such as configuration systems. Moreover, it plays an ever more important role in post-optimization analysis to give the user of an optimization system an idea how many solutions there are within a certain percentage of the optimal objective value. The famous mathematical programming tool Cplex for example now includes a solution counting method. Finally, from a research perspective the problem is interesting in its own right as it constitutes a natural extension of the mere optimization task.

Solution counting is probably best studied in the satisfaction (SAT) community where a number of approaches have been developed to estimate the number of solutions of under-constrained instances. First attempts to count the number of solutions often simply consisted in extending the run of a solution finding systematic search after a first solution has been found [3]. More sophisticated randomized methods estimate upper and lower bounds with high probability. In [8], e.g., in a trial an increasing number of random XOR constraints are added to the problem. The upper and lower bounds

* This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

on the number of solutions depends on how many XORs can be added before the instance becomes infeasible, whereby the probability that the bound is correct depends on the number of trials where (at least or at most) the same number of XORs can be added before the instance changes its feasibility status.

An interesting trend in constraint programming (CP) is to estimate solution density via solution counting for individual constraints [23,24]. Since the solution density information is used for branching, it is important that these methods run very fast. Consequently, they are constraint-based and often give estimates on the number of solutions rather than hard upper and lower bounds or bounds that hold with high probability.

In mathematical programming, finally, the IBM Cplex IP solution counter [5,10] enumerates all solutions while aiming at finding diverse set of solutions, and the Scip solution counter finds the number of all feasible solutions using a technique to collect several solutions at once [1]. Stopped prematurely at some desired time-limit, these solvers provide lower bounds on the number of solutions.

Considering the literature, we find that a big emphasis has been laid on the computation of lower bounds on the number of solutions of a given problem instance. Apart from the work in [8] and the upper bounding routine for SAT in [11], we are not aware of any other approaches that provide hard or high probability upper bounds on the number of solutions. Especially solution counters that are part of the IBM Cplex and the Scip solver would benefit if an upper bound on the number of solutions could be provided alongside the lower bound in case that counting needs to be stopped prematurely.

With this study we attempt to make a first step to close this gap. In particular, we consider binary integer programs and propose a general method for computing hard upper bounds on the number of feasible solutions. Our approach is based on the exploitation of relaxations, in particular surrogate and Lagrangian relaxations. Experimental results on automatic recording and market split problems provide a first proof of concept.

2 Upper Bounds on the Number of Solutions for Binary Integer Programs

We assume that the problem instance is given in the format

$$\begin{array}{ll} (BIP) & p^T x \geq B \\ & Ax \leq b \\ & x_i \in \{0,1\}. \end{array}$$

Wlog, we assume that the profit coefficients are integers. Although we could multiply the first inequality with minus one, we make it stand out as the original objective of the binary integer program (BIP) that was to be maximized. Usually, in branch and bound approaches, we consider relaxations to compute upper bounds on that objective. For example, we may solve the linear program (LP)

$$\begin{array}{ll} Maximize & L = p^T x \\ & Ax \leq b \\ & 0 \leq x_i \leq 1 \end{array}$$

and check whether $L \geq B$ for an incumbent integer solution with value B to prune the search.

For our task of computing upper bounds on the number of solutions, relaxing the problem is the first thing that comes to mind. However, standard LP relaxations are not likely to be all that helpful for this task. Assume that there are two (potentially fractional) solutions that have an objective value greater or equal B. Then, there exist infinitely many fractional solutions that have the same property.

Consequently, we need to look for a relaxation which preserves the discrete character of the original problem. We propose to use the surrogate relaxation for this purpose. In the surrogate relaxation, we choose multipliers $\lambda_i \geq 0$ for each linear inequality constraint i and then aggregate all constraints into one. We obtain:

$$\begin{array}{ll} Maximize & S = p^T x \\ & \lambda^T A x \leq \lambda^T b \\ & x_i \in \{0, 1\}. \end{array}$$

This problem is well known, it is a knapsack problem (that may have negative weights and/or profits). Let us set $w \leftarrow w_\lambda \leftarrow A^T \lambda$ and $C \leftarrow C_\lambda \leftarrow \lambda^T b$. Then, we insert the profit threshold B back into the formulation. This is sound as S is a relaxation of the original problem. We obtain a knapsack constraint

$$\begin{array}{ll} (KP) & p^T x \geq B \\ & w^T x \leq C \\ & x_i \in \{0, 1\}. \end{array}$$

2.1 Filtering Knapsack Constraints

In [12], knapsack constraints were studied in great detail and exact pseudo-polynomial time filtering algorithms were developed which are based on a dynamic programming formulation of knapsack. First, the knapsack instance is modified so that all profits p_i are non-negative. This can be achieved by potentially replacing a binary variable x_i (in the context of knapsack we often refer to the index i as an *item*) with its 'negation' $x_i' = 1 - x_i$. Then, in a cell $M_{q,k}$ we store the minimum knapsack weight needed to achieve exactly profit q when only items $\{1, \ldots, k\}$ can be included in the knapsack (i.e., when all variables in $\{x_{k+1}, \ldots, n\}$ are set to 0). Then, the following recursion equation holds:

$$M_{q,k} = \min\{M_{q,k-1}, M_{q-p_k,k-1} + w_k\}. \tag{1}$$

To filter the constraint we interpret the DP as a weighted directed acyclic graph (DAG) where the cells are the nodes and nodes that appear in the same recursion are connected (see left graph in Figure 1). In particular, we define $G = (V, E, v)$ by setting

- $V_M := \{M_{q,k} \mid 0 \leq k \leq n\}$.
- $V := V_M \cup \{t\}$.
- $E_0 := \{(M_{q,k-1}, M_{q,k}) \mid k \geq 1,\ M_{q,k} \in V_M\}$.
- $E_1 := \{(M_{q-p_k,k-1}, M_{q,k}) \mid k \geq 1,\ q \geq p_k,\ M_{q,k} \in V_M\}$.
- $E_t := \{(M_{q,n}, t) \mid q \geq B,\ M_{q,n} \in V_M\}$.
- $E := E_0 \cup E_1 \cup E_t$.

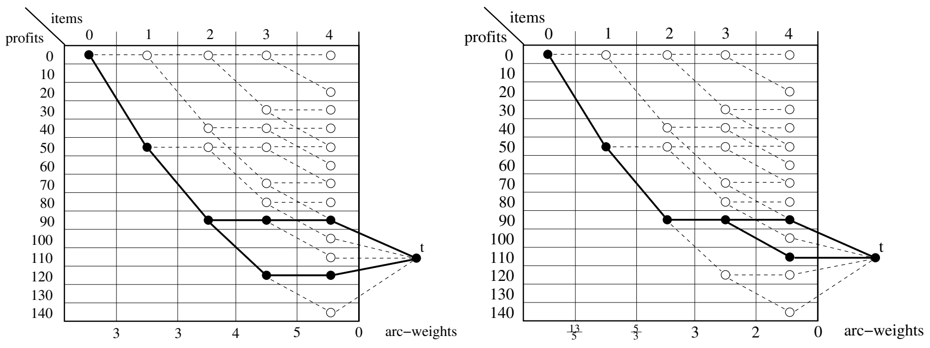


Fig. 1. The figure shows dynamic programming tables for a knapsack constraint with four variables, profits $p^T = (50, 40, 30, 20)$, and profit constraint threshold $B = 82$. In the left figure the weights are $w^T = (3, 3, 4, 5)$, and the knapsack's capacity is $C = 10$. In the right figure the weights are $w^T = (13/5, 5/3, 3, 2)$, and the capacity is $C = 19/3$. The node-labels are defined by their row and column number, the sink node t is marked separately. The value of non-horizontal arcs that cross a vertical line is given under that line, horizontal arcs have weight 0. Hollow nodes and dashed arcs mark those nodes and arcs that are removed by the filtering algorithm, because there exists no path from $M_{0,0}$ to t with weight lower or equal C that visits them.

- $v(e) := 0$ for all $e \in E_0 \cup E_t$.
- $v(M_{q-p_k,k-1}, M_{q,k}) := w_k$ for all $(M_{q-p_k,k-1}, M_{q,k}) \in E_1$.

We consider the DAG G because there is a one-to-one correspondence between paths from $M_{0,0}$ to t and variable instantiations that yield a profit greater than B. Moreover, the length of such a path is exactly the weight of the corresponding instantiation. Therefore, every path from $M_{0,0}$ (the *source*) to t (the *sink*) with length lower or equal C defines a feasible, improving solution (we call such paths *admissible*). Vice versa, every feasible, improving solution also defines an admissible path from source to sink with length lower or equal C.

The filtering algorithm in [12] removes edges from G that cannot be part of any admissible path. This is done using a filtering routine for shorter path constraints on DAGs from [13]: We first compute the shortest path distances from the source to all nodes using the topological ordering of the DAG, thus being able to handle shortest path problems even non-negative edge weights in time linear in the size of the graph. In an analogous second pass which begins at the sink we compute the shortest path distances to the sink. Equipped with both distances for each node, we can compute the shortest path lengths from source to sink through each edge in the graph – and remove all edges and nodes which cannot lie on any admissible path.

2.2 Upper Bounds on the Number of Solutions

In [12], the resulting DP is analyzed using a technique from [20,21] to identify which variables cannot take value 0 or value 1. We do not perform this last step. Instead, we use the resulting DP to count the number of paths from source to sink using the technique in [23,24]. Note that any solution to (BIP) fulfills the redundant constraint (KP) and therefore corresponds to an admissible path in our DP. Moreover, two different solutions also define two different paths in the DP. Therefore, the number of paths in the DP gives an upper bound on the number of solutions in (BIP).

Now, the quality of the upper bound will depend on the choice of the initial vector λ. In ordinary optimization, we aim for a choice of λ for which the surrogate relaxation gives the tightest relaxation value. However, for the purpose of filtering we know that sub-optimal multipliers λ can provide better filtering effectiveness [14]. Consider the following example:

$$\begin{array}{ll}(EX) & 50x_1 + 40x_2 + 30x_3 + 20x_4 \geq 82 \\ & 3x_1 + x_2 + 3x_3 \leq 5 \\ & 2x_2 + x_3 + 5x_4 \leq 5 \\ & x_i \in \{0, 1\}.\end{array}$$

If we use $\lambda = (1,1)^T$, e.g., then we get the knapsack constraint as shown in the left graph of Figure 1 with a relaxation value of 120 (as that is the highest profit visited by the remaining admissible paths). On the other hand, had we chosen $\lambda = (13/15, 6/15)^T$, we would have obtained the knapsack constraint in the right graph of Figure 1 with an improved upper bound of 110.

Comparing the two DPs, we find that the two choices for λ yield incomparable filtering effectiveness. Although the second set of multipliers gives a strictly better upper bound, it cannot remove the edge $(M_{90,3}, M_{110,4})$. On the other hand, the second choice for λ allows us to remove the edges $(M_{90,2}, M_{120,3})$ and $(M_{120,3}, M_{120,4})$. This effect has been studied before in [14]. The explanation for the different filtering behavior is that, in principle, each edge has its own vector λ that maximally challenges admissibility (as measured by the shortest path length through that edge).

In principle, we could employ a probing procedure. For each edge, we remove all edges on the same level, thus enforcing that each path from source to sink must pass through this edge. Then, we start with some selection for λ and compute the shortest path length according to the corresponding weights w_λ as well as the corresponding BIP solution x_λ. If $w_\lambda^T x_\lambda > C_\lambda$, then we can remove the edge. Otherwise, we modify λ to minimize $C_\lambda - w_\lambda^T x_\lambda$ as much as possible. From the theory of Lagrangian relaxation (see for example [2]) we know that finding the optimal choice for λ consists in minimizing a piecewise linear convex function. Consequently, we can use a subgradient search algorithm to find the vector $\lambda \geq 0$ which will minimize $C_\lambda - w_\lambda^T x_\lambda$ as much as possible and thus enable us to decide whether any λ exists that would allow us to remove the edge under consideration.

The problem with this procedure is of course that it takes way too much time to probe each individual edge. Instead, we follow the same method as in CP-based Lagrangian relaxation [15]. That is, we employ a subgradient search to find a vector λ that minimizes $C_\lambda - w_\lambda^T x_\lambda$ in the DP. Then, for each λ that the subgradient search considers, we use our edge-filtering algorithms to remove edges from the graph. That way, we hope to visit a range of different settings for λ that will hopefully remove a large percentage of edges in the DP that can be discarded.

Consider again our example (EX) from before. If we first prune the graph with respect to the weight vector w from the left graph in Figure 1 and then, in the pruned graph, remove edges based on the weight vector w from the right graph in Figure 1, then we end up with only one path which corresponds to the only solution to (EX) which is $x = (1, 1, 0, 0)^T$.

2.3 The Algorithm

The complete procedure is sketched in Algorithm 1. Note how we first increase the number of solutions by considering the cardinality of the set $R \leftarrow \{x \in \{0,1\}^n \mid p^T x \geq B\}$ instead of $P \leftarrow \{x \in \{0,1\}^n \mid p^T x \geq B \;\&\; Ax \leq b\}$. Then, to reduce the number of solutions again, we heuristically remove edges from the DP that has exactly one path for each $x \in R$ by propagating constraints $\lambda^T Ax \leq \lambda^T b$ for various choices of λ in the DP. The resulting number of paths in the DP gives a hard upper bound on the number of solutions to the original BIP.

Algorithm 1. BIP Counting Algorithm

```
1: Negate binary variables with profit $p_i < 0$.
2: Set up the graph $G$ for $\{x \in \{0,1\}^n \mid p^T x \geq B\}$.
3: Initialize $\lambda$.
4: while subgradient method not converged do
5:     Set $w \leftarrow \lambda^T A$, $C \leftarrow \lambda^T b$.
6:     Propagate $w^T x \leq C$ in $G$ removing inadmissible edges.
7:     Compute the solution $x$ that corresponds to the shortest path from source to sink in $(G, w)$.
8:     Update $\lambda$ according to the current gap $C - w^T x$ and the subgradient $Ax - b$.
9: end while
10: Count the number of paths from source to sink in G and return that number.
```

2.4 Strengthening the Bound – Cutting Planes, Tree Search, Compatibility Labels, and Generate and Check

A nice property of our approach is that we can use all the usual methods for strengthening linear continuous relaxations, such as preprocessing and especially adding valid inequalities, so-called cutting planes, to the BIP which tighten the continuous relaxation.

To strengthen the upper bound on the solution count further, we can embed our procedure in a branch-and-bound tree search algorithm which we truncate at some given depth-limit. The sum of all solutions at all leafs of the truncated tree then gives an upper bound on the number of solutions.

For very hard combinatorial counting problems we may consider doing even more. In our outline above, we use the profit constraint to define the graph G. In principle, we could use any vector μ of natural numbers and consider the constraint $(p^T - \mu^T A)x \geq B - \mu^T b$ to set up the DP. This is needed in particular when there is no designated objective function. We notice, however, that we do not need to restrict us to using just one DP. Instead, we can set up multiple DPs for different choices of μ.

The simplest way to strengthen the upper bound on the number of solutions is to take the minimum count over all DPs. However, we can do much better than that. Following an idea presented in [9], we can compute compatibility labels between the different DPs: Let us denote with G_A and G_B the graphs that correspond to two different DPs for our problem. Our filtering algorithm ensures that each edge in the graph is visited by at least one admissible path. The compatibility labels from [9] aim to ensure that an edge in G_A is also supported by a (not necessarily admissible) path in G_B. More precisely, for each edge in G_A we ensure that there is a path from source to sink in G_A that visits the edge and which corresponds to a solution which also defines a path from source to sink in G_B.

Finally, if we have found an upper bound on the solution count that is rather small, we can generate all potential solutions which is very easy given our DAG G. Then, we test each assignment for feasibility and thus provide an exact count.

3 Numerical Results

3.1 Automatic Recording

We first consider the automatic recording problem (ARP) that was introduced in [15].

3.2 Problem Formulation

The technology of digital television offers to hide meta-data in the content stream. For example, an electronic program guide with broadcasting times and program annotation can be transmitted. An intelligent video recorder like the TIVO$^{\mathrm{tm}}$ system [19] can exploit this information and automatically record TV content that matches the profile of the system's user. Given a profit value for each program within a predefined planning horizon, the system has to make the choice which programs shall be recorded, whereby two restrictions have to be met:

– The disk capacity of the recorder must not be exceeded.
– Only one program can be recorded at a time.

While the problem originally emerged from automatic video recording, it has other applications, for example in satellite scheduling. Various algorithms for the ARP have been studied in [15,14,16,17]. The problem can be stated as a binary integer program:

$$\begin{array}{lll} Maximize & p^T x & \\ & w^T x \le K & \\ & x_i + x_j \le 1 & \forall\, 0 \le i \le j \le n, I_i \cap I_j \ne \emptyset \\ & x \in \{0,1\}^n & \end{array} \tag{ARP 1}$$

where p_i and w_i represent the profit and the storage requirement of program i, K is the storage capacity, and $I_i := [startTime(i), endTime(i)]$ corresponds to the broadcasting interval of program i. The objective function maximizes the user satisfaction while the first constraint enforces the storage restrictions. Constraints of the form $x_i + x_j \le 1$ ensure that at most one program is recorded at each point in time.

This formulation can be tightened by considering the conflict graph and adding the corresponding clique constraints to the formulation [15].

Definition 1. *The set $C \subseteq V$ is called a conflict clique iff $I_i \cap I_j \neq \emptyset \; \forall \, i, j \in C$. A conflict clique C is called maximal iff $\forall \, D \subseteq V, D$ conflict clique: $C \subseteq D \Rightarrow C = D$. Let $M := \{C_0, \dots, C_{m-1}\} \subseteq 2^V$ the set of maximal conflict cliques.*

These clique constraints are obviously valid inequalities since, if $x_i + x_j \leq 1$ for all overlapping intervals, it is also true that $\sum_{i \in C_p} x_i \leq 1 \; \forall \, 0 \leq p \leq m$. We can therefore add the clique constraints to our original formulation.

$$\begin{array}{lll} Maximize & p^T x & \\ & w^T x \leq K & \\ & x_i + x_j \leq 1 & \forall \, 0 \leq i \leq j \leq n, I_i \cap I_j \neq \emptyset \\ & \sum_{i \in C_p} x_i \leq 1 & \forall \, 0 \leq p \leq m \\ & x \in \{0,1\}^n & \end{array} \qquad \text{(ARP 2)}$$

Though being NP-complete on general graphs, finding maximal cliques on the graph defined by our application is simple:

Definition 2. *A graph $G = (V, E)$ is called an interval graph if there exist intervals $I_1, \dots, I_{|V|} \subset \mathbb{R}$ such that $\forall v_i, v_j \in V : (v_i, v_j) \in E \iff I_i \cap I_j \neq \emptyset$.*

On interval graphs, the computation of maximal cliques can be performed in $O(n \log n)$ [7]. Hence, ARP 2 can be obtained in polynomial time.

3.3 Solution Counting for the ARP

We will now describe how we apply our counting algorithm to the ARP problem.

Initialization: The graph G for our ARP formulation is set up using the equation $w^T x \leq K$, where w_i represents the storage requirement of program i and K is the storage capacity.

Tree Search, and Generate and Test: To strengthen the quality of our bounds on the number of solutions, we employ a truncated tree search as described earlier. For branching, we select the variable with the highest knapsack efficiency p_i/w_i which is also selected in the shortest path in the DP according to the final multipliers λ. When we get total solution counts below 100 we generate all solutions and test them for feasibility.

Subgradient Optimization: At every choice point, we conduct the subgradient search using the object bundle optimization package from Frangioni [6]. On top of filtering with respect to the vectors λ that the subgradient optimizer visits, we also propagate the constraint $w^T x \leq K$ in the DP at every choice point. At leaf nodes, also choose randomly 3% of the original constraints in ARP 1 or ARP 2 and propagate them to prune the DPs one last time before counting the number of paths from source to sink.

3.4 Experimental Results

We used a benchmark set described in [15,14] which can downloaded at [18]. This benchmark set consists of randomly generated instances which are designed to mimic features of real-world instances for the automatic recording of TV content. For our

Table 1. Numerical Results for the ARP Problem. We present the upper bound on the number of solutions and the CPU-time in seconds for the binary constraint model (ARP-1) and the maximal clique model (ARP-2). The table on the left is for the small sized data set (20-720) with 20 channels and 720 minute time horizon, and the table on the right is for the large sized data set (50-1440) with 50 channels and 1440 minute time horizon. In this experiment, we do not generate and check solutions for feasibility.

		ARP-1		ARP-2	
Inst.	Gap	Count	Time	Count	Time
0	0%	2	20	2	3.2
1	0%	3	10	1	1.5
2	0%	1	16	1	2.8
0	1%	2.27E+10	90	1.60E+10	38.8
1	1%	3.26E+05	12	2.09E+05	3.2
2	1%	8.36E+07	33	3.69E+07	9.5
0	2%	7.51E+12	133	8.77E+11	73.8
1	2%	9.06E+05	13	4.56E+05	4.3
2	2%	2.87E+09	68	1.33E+09	24

		ARP-1		ARP-2	
Inst.	Gap	Count	Time	Count	Time
0	0%	39	1109	39	34
1	0%	203	933	35	20
2	0%	15	1146	15	22
0	1%	6.54E+43	2636	7.95E+35	353
1	1%	7.82E+10	1100	3.75E+10	73
2	1%	5.25E+23	314	1.05+23	294
0	2%	4.75E+59	5169	6.81E+52	992
1	2%	2.57E+13	3639	8.06E+12	221
2	2%	1.33E+26	6873	3.08E+24	893

experiments, we use the class usefulness (CU) instances. We consider a small sized data set which spans half a day (720 minutes) and consists of 20 channels, and a large sized data set which spans a full day (1440 minutes) and consists of 50 channels. Profits for each program are chosen based on the class that a program belongs to. This class also determines the parameters according to which its length is randomly chosen. On average, these instances have 300 and 1500 programs, respectively. All experiments in this paper were performed on a machine with Intel Core 2 Quad Q6600, 2.4GHz CPUs and 2GByte of RAM operating Linux Debian 5.0.3 32-bit. On all experiments, we enforced a time limit of 3 hours CPU time.

Our first evaluation compares the effectiveness of the models described by ARP 1 and ARP 2 in terms of the upper bound on the solution count that they provide and the time they take. Specifically, we are interested in the increase of the number of solutions as we move away from the optimal value. To this end, we introduce the *Gap* parameter which indicates the percentage gap between a threshold and the optimal value. We only consider solutions that achieve an objective value above the threshold. We experiment with objective gaps of 0%, 1% and 2% and truncate the search at depth 5. Table 1 shows that the ARP 2 formulation which includes the clique cuts provides much better upper bounds than ARP 1 in substantially less time. This indicates that exploiting the common methods for strengthening LP relaxations can also be exploited effectively to compute superior upper bounds on the number of solutions of BIPs. The fact that ARP 2 actually runs faster can be attributed to the fact that the cutting planes allow much better edge-filtering effectiveness. Therefore, the DP contains much fewer edges higher up in the tree, which leads to much faster times per choice point.

We next compare our approach (UBound) with the Cplex IP solution counter which enumerates all solutions [10,5] and the Scip solution counter which collects several solutions at a time. Note that Cplex and Scip provide only a lower bound in case they time out or reach the memory limit. We again consider objective gaps 0%, 1% and 2%.

Table 2. Numerical Results for the ARP Problem with 0% objective gap. We present the upper bound on the number of solutions and the CPU-time in seconds at depth 5. The table on the left is for the small sized data set (20-720) with 20 channels and 720 minute time horizon, and the table on the right is for the large sized data set (50-1440) with 50 channels and 1440 minute time horizon. 'T' means that the time limit has been reached. The numbers in bold show exact counts and the numbers in parenthesis are our upper bounds before we generate and check solutions for feasibility.

	Cplex		Scip		Ubound	
Inst.	Count	Time	Count	Time	Count	Time
0	**2**	0.17	**2**	0.3	**2**	3.16
1	**1**	0.03	**1**	0.05	**1**	1.53
2	**1**	0.08	**1**		**1**	2.75
3	**1**	0.04	**1**	0.03	**1**	1.71
4	**1**	0.06	**1**	0.06	**1**	2.46
5	**12**	0.62	**12**	0.16	**12**	3.83
6	**6**	0.17	**6**		**6**	2.47
7	**1**	0.07	**1**	0.03	**1**	1.60
8	**1**	0.09	**1**	0.06	**1**	2.45
9	**3**	0.37	**3**	0.04	**3**	2.30

	Cplex		Scip		Ubound	
Inst.	Count	Time	Count	Time	Count	Time
0	**39**	182	**39**	1.91	**39**	34.3
1	35	T	**35**	100	**35**	20.7
2	**14**	0.98	**14**	1.54	**14** (15)	30.5
3	**6**	0.64	**6**	0.25	**6**	30.2
4	**20**	2.52	**20**	0.51	**20**	30.8
5	**1**	0.34	**1**	0.4	**1**	20.9
6	**33**	3.95	**33**	71	**33** (39)	27.5
7	**1**	0.49	**1**	0.31	**1**	58.0
8	**4**	2.16	**4**	1.95	**4**	69.6
9	**6**	27.1	**6**	1.81	**6**	43.8

For 0% gap, we run our method with depth 5 which is adequate to achieve the exact counts. For higher gaps, we present the results for depths 5, 10, and 15.

Our results are presented in Table 2, Table 3, and Table 4. For the optimal objective threshold, UBound provides exact counts for all test instances. In terms of running time, UBound does not perform as quickly as the IBM Cplex and the Scip solution counter. There is only one notable exception to this rule, instance 50-1440-1. On this instance, Scip takes 100 seconds and Cplex times out after three hours while our method could have provided the 35 solutions to the problem in 20 seconds.

This discrepancy becomes more evident when we are interested in the number of solutions that are with 1% or 2% of the optimum. As we can see from Table 3 and Table 4 the number of solutions increases very rapidly even for those small objective gaps. Not surprisingly, the counts obtained by Cplex and Scip are limited by the number of solutions they can enumerate within the memory and the time constraints, yielding a count of roughly 1E+5 to 1E+7 solutions in most cases. Due to the explosion in the number of solutions, Cplex and Scip are never able to give exact counts for the large instances but only give a lower bound. Cplex hits the time cutoff in 17 out of 20 large instances and reaches the memory limit for the remaining 3, and Scip times out in all large instances. In most cases where Cplex or Scip are able to find the exact counts, UBound is able to provide tight upper bounds that are not more than an order of magnitude bigger. In Figure 2, we show how the upper and lower bounds obtained by UBound, Cplex, and Scip progress as they approach the exact count.

We also compared our approach with the method from [8] which provides very good bounds on the number of solutions for constraint satisfaction problems. The method is based on the addition of random XOR-constraints. Unfortunately, we found that, in combination with an integer programming problem, the method does not perform well.

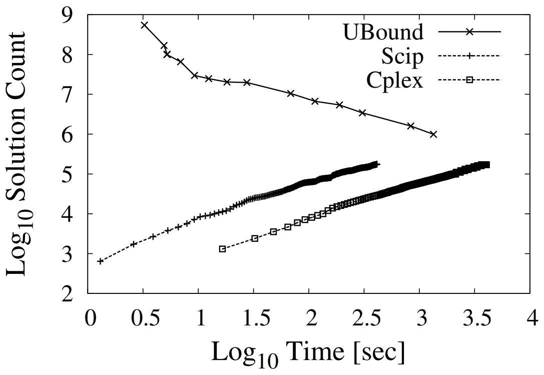


Fig. 2. Solution Count for the instance 20-720-2 with 1% objective gap. We present the progress of the upper bound obtained by UBound and the lower bounds obtained by Cplex and Scip as time progresses. The time and solution count are given on a logarithmic scale of base 10. We run UBound until depth 17 which is within the time that Cplex reaches the memory limit.

We tried using the vanilla code[1] which was designed for pure CSPs. It did not perform well for the ARP. So we modified the code, providing better branching variables for the tree search and using linear bounds to prune the search. That improved the performance. With this approach we are able to compute lower bounds, but computing these takes more time and the counts are worse than those provided by Cplex and Scip. Upper bounds take even more time as the XOR constraints involve more variables. We could not obtain upper bounds within the time limit of three hours. We conjecture that a tight integration between the XOR constraints and linear inequalities would be needed to make this approach, which gives very good results for CSPs, work well for optimization problems.

3.5 Market Split

We next consider the market split problem (MSP), a benchmark that was suggested for knapsack constraints in [20,21].

3.6 Problem Formulation

The original definition goes back to [4,22]: A large company has two divisions D_1 and D_2. The company supplies retailers with several products. The goal is to allocate each retailer to either division D_1 or D_2 so that D_1 controls A% of the company's market for each product and D_2 the remaining (100-A)%. Formulated as an integer program, the problem reads:

$$\begin{array}{cc} \sum_j a_{ij} x_j = \lfloor \frac{A}{100} \sum_j a_{ij} \rfloor & \forall\, 0 \le i < m \\ x_j \in \{0,1\} & \forall\, 0 \le j < n, \end{array}$$

[1] Many thanks to Ashish Sabharwal for providing us the source code!

Table 3. Numerical Results for the ARP Problem with 1% objective gap. We present the upper bound on the number of solutions and the CPU-time in seconds. 'T' means that the time limit has been reached and 'M' indicates a solver has reached the memory limit. The numbers in bold show exact counts and the numbers in square brackets denote the best count UBound could achieve within the time limit.

	Cplex		Scip		UBound					
					Depth 5		Depth 10		Depth 15	
Instance	Count	Time	Count	Time	Count	Time	Count	Time	Count	Time
20-720-0	5.20E+05	M	**1.01E+06**	2518	1.60E+10	38.8	1.97E+08	137	3.31E+07	1183
20-720-1	**3.15E+04**	175	**3.15E+04**	20.3	2.09E+05	3.16	1.48E+05	7.52	1.02E+05	40.9
20-720-2	1.77E+05	M	**1.77E+05**	414	3.69E+07	9.51	1.36E+07	45.2	2.87E+06	622
20-720-3	**2.09E+02**	3.39	**2.09E+02**	0.25	4.05E+02	4.19	2.99E+02	12.5	2.48E+02	40.5
20-720-4	**5.20E+03**	76	**5.20E+03**	6.7	1.13E+05	7.24	1.79E+04	23.1	1.02E+04	122
20-720-5	**2.00E+04**	174	**2.00E+04**	22.5	1.58E+12	22.2	6.81E+08	58.8	4.50E+04	228
20-720-6	**5.45E+04**	932	**5.45E+04**	153	2.00E+07	10.9	3.96E+06	46.5	1.68E+06	431
20-720-7	**9.80E+01**	1.68	**9.80E+01**	0.07	1.04E+02	2.82	1.04E+02	6.70	1.03E+02	16.7
20-720-8	**1.77E+05**	1386	**1.77E+05**	298	3.41E+09	40.7	3.42E+07	191	9.00E+06	899
20-720-9	**1.88E+03**	35.5	**1.88E+03**	1	3.66E+03	4.23	3.48E+03	17	2.99E+03	87.8
50-1440-0	1.95E+04	T	1.15E+07	T	7.95E+35	353	1.21E+34	2572	[1.21E+34]	T
50-1440-1	5.59E+04	T	1.11E+07	T	3.75E+10	73.8	2.21E+10	305	1.85E+10	3025
50-1440-2	7.63E+04	T	1.77E+06	T	1.05E+23	293	1.76E+21	2635	[1.76E+21]	T
50-1440-3	6.00E+04	T	9.48E+06	T	3.56E+16	149	2.34E+15	452	2.45E+14	3333
50-1440-4	7.13E+04	T	7.29E+05	T	4.15E+21	412	4.31E+19	1852	[4.31E+19]	T
50-1440-5	9.33E+04	M	1.04E+06	T	3.28E+10	90.4	7.06E+09	314	6.17E+09	4093
50-1440-6	1.20E+05	M	3.03E+06	T	7.53E+12	101	2.44E+12	350	4.12E+11	3483
50-1440-7	4.92E+04	T	1.96E+06	T	1.04E+20	396	6.03E+18	3037	[6.03E+18]	T
50-1440-8	8.90E+04	T	3.75E+05	T	5.56E+27	719	1.44E+25	3776	[1.44E+25]	T
50-1440-9	8.35E+04	M	9.55E+05	T	2.89E+14	259	2.01E+13	434	2.09E+06	578

whereby m denotes the number of products, n is the number of retailers, and a_{ij} is the demand of retailer j of product i. MSPs are generally very hard to solve, especially the randomly generated instances proposed by Cornuejols and Dawande where weight coefficients are randomly chosen in $[1, \ldots, 100]$ and $A = 50$. Special CP approaches for the MSP have been studied in [20,21,14,9].

3.7 Solution Counting for the MSP

Initialization: Our MSP formulation does not have an objective function, thus we construct the graph G using the equation $\lambda^T Ax \geq \lambda^T b$, where $\lambda_i = 5^{i-1}$ as proposed in [20,21].

Compatibility Labels, and Generate and Test: For the MSP, we strengthen the solution counts by employing the compatibility labels introduced in [9]. We additionally

Table 4. Numerical Results for the ARP Problem with 2% objective gap. We present the upper bound on the number of solutions and the CPU-time in seconds. 'T' means that the time limit has been reached and 'M' indicates a solver has reached the memory limit. The numbers in bold show exact counts and the numbers in square brackets denote the best count UBound could achieve within the time limit.

	Cplex		Scip		UBound					
					Depth 5		Depth 10		Depth 15	
Instance	Count	Time	Count	Time	Count	Time	Count	Time	Count	Time
20-720-0	6.80E+05	M	1.14E+07	T	8.77E+11	73.8	9.23E+09	326	2.30E+09	4002
20-720-1	**1.87E+05**	969	**1.87E+05**	49.5	4.56E+05	4.31	4.01E+05	8.76	3.24E+05	46.2
20-720-2	3.00E+05	T	**8.77E+06**	6528	1.33E+09	24.3	1.65E+08	218	5.07E+07	2276
20-720-3	**4.95E+02**	5.77	**4.95E+02**	0.42	6.60E+02	5.80	5.26E+02	24.2	5.21E+02	75.8
20-720-4	**8.89E+04**	1335	**8.89E+04**	73.5	3.94E+06	10.3	3.26E+05	53.3	2.36E+05	274
20-720-5	3.30E+05	M	**3.32E+05**	618	1.27E+15	43.9	1.15E+13	277	1.86E+09	1540
20-720-6	2.80E+05	M	**3.12E+06**	1966	3.80E+08	19.3	9.20E+07	84.9	6.24E+07	911
20-720-7	**1.35E+02**	2.09	**1.35E+02**	0.07	1.38E+02	3.70	1.38E+02	9.94	1.37E+02	27.2
20-720-8	3.00E+05	M	1.39E+07	T	7.16E+11	82.3	4.91E+09	727	[4.91E+09]	T
20-720-9	**4.17E+03**	63.9	**4.17E+03**	2.33	4.88E+03	7.38	4.71E+03	29.1	4.57E+03	135
50-1440-0	3.03E+04	T	3.11E+06	T	6.81E+52	992	[6.81E+52]	T	[6.81E+52]	T
50-1440-1	5.58E+04	T	3.43E+06	T	8.06E+12	221	1.01E+12	1240	[1.01E+12]	T
50-1440-2	1.40E+05	M	8.97E+06	T	3.08E+24	893	[3.08E+24]	T	[3.08E+24]	T
50-1440-3	7.89E+04	T	1.52E+07	T	2.5E+43	460	2.25E+32	1802	[2.25E+32]	T
50-1440-4	1.00E+05	M	1.35E+06	T	8.89E+22	996	[8.89E+22]	T	[8.89E+22]	T
50-1440-5	9.28E+04	M	1.62E+06	T	3.03E+12	252	1.82E+11	1679	[1.82E+11]	T
50-1440-6	1.50E+05	M	1.68E+06	T	2.1E+34	341	1.53E+29	1607	[1.53E+29]	T
50-1440-7	7.66E+04	T	6.18E+06	T	6.9E+37	1281	[6.9E+37]	T	[6.9E+37]	T
50-1440-8	1.10E+05	M	6.73E+05	T	4.87E+30	2264	[4.87E+30]	T	[4.87E+30]	T
50-1440-9	4.65E+04	T	1.12E+07	T	6.91E+46	1075	2.74E+29	3482	[2.74E+29]	T

set up the DPs for the original equations in the problem. If there are $m > 3$ constraints in the MSP, we set up $m - 2$ DPs where the kth DP is defined by the sum of the kth constraint plus five times the k+first constraint plus 25 times the k+second constraint.

Often, the number of solutions to MSP instances is comparably low, and checking feasibility is very fast. In case that we find an upper bound of less than 50,000 we simply generate and check those solutions for feasibility. Therefore, each number that is less than 50,000 is actually an exact count.

3.8 Experimental Results

For the purpose of solution counting, we consider the Cornuejols-Dawande instances as described before. Many of these instances are actually infeasible. When there are m constraints, Cornuejols and Dawande introduce $10(m - 1)$ binary variables. We introduce more variables to create less and less tightly constrained instances which have

Table 5. Numerical Results for the MSP Problem. We present the upper bound on the number of solutions found and the CPU-time taken in seconds for the binary constraint model and the maximal clique model. 'T' means that the time limit has been reached. The numbers in bold show exact counts. The numbers in parenthesis are our upper bounds before we generate and check solutions for feasibility.

			Cplex		Scip		Ubound	
Ins	Order	#Vars	Count	Time	Count	Time	Count	Time
1	3	24	**2**	1.78	**2**	5.7	**2**	3.92
2	3	24	**0**	0.91	**0**	3.76	**0**	0.53
3	3	24	**0**	1.24	**0**	2.94	**0**	0.51
4	3	30	**32**	39	**32**	107	**32** (36)	13
5	3	30	**70**	70	**70**	117	**70** (82)	21
6	3	30	**54**	78	**54**	174	**54** (58)	25
7	3	36	**2.3K**	1962	**2.3K**	5118	**2.3K** (32K)	176
8	3	36	292	T	**2.3K**	9203	**2.3K** (23K)	164
9	3	36	569	T	**2K**	5656	**2K** (14K)	130
10	4	34	**2**	5707	**2**	1087	**2**	198
11	4	34	**0**	396	**0**	1088	**0**	189
12	4	34	**2**	109	**2**	955	**2**	190
13	4	36	**6**	1227	**6**	4175	**6**	301
14	4	36	**2**	753	**2**	2400	**2**	266
15	4	36	**6**	366	**6**	2470	**6**	278
16	4	38	**12**	4422	11	T	**12**	412
17	4	38	9	T	29	T	**36**	405
18	4	38	**44**	3391	43	T	**44**	401

more solutions. We compare UBound again with the counts provided by IBM Cplex and Scip. As before, Cplex and Scip provide a lower bound in case they time out. We consider MSPs of orders 3 and 4 with an increasing number of variables between 24 and 38.

We present our results in Table 5. As we can see, UBound provides high quality upper bounds very quickly as shown in the counts given in brackets. Using the generate and test technique, on all instances we are able to provide exact counts in considerably less time than Cplex and Scip.

Again, we compared our results also with the XOR approach from [8]. After the vanilla implementation from [8] did not provide competitive results, we devised an efficient code that can solve pure MSPs efficiently and added XOR constraints to it. Again, we found that the problems augmented by XORs are much harder to solve which resulted in the approach timing out on our entire benchmark. We attribute this behavior to our inability to integrate the XOR constraints tightly with the subset-sum constraints in the problem.

4 Conclusions

We presented a new method for computing upper bounds on the number of solutions of BIPs. We demonstrated its efficiency on automatic recording and market split problems. We showed that standard methods for tightening the LP relaxation by means of cutting planes can be exploited also to provide better bounds on the number of solutions. Moreover, we showed that a recent new method for integrating graph-based constraints more tightly via so-called compatibility labels can be exploited effectively to count solutions for market split problems.

We do not see this method so much as a competitor to the existing solution counting methods that are parts of IBM Cplex and Scip. Instead, we believe that these solvers could benefit greatly from providing upper bounds on the number of solutions. This obviously makes sense when the number of solutions is very large and solution enumeration must fail. However, as we saw on the market split problem, considering upper bounds can also boost the performance dramatically on problems that have few numbers of solutions. In this case, our method can be used to give a super-set of potential solutions whose feasibility can be checked very quickly.

References

1. Achterberg, T.: SCIP - A Framework to Integrate Constraint and Mixed Integer Programming, http://www.zib.de/Publications/abstracts/ZR-04-19/
2. Ahuja, R.K., Magnati, T.L., Orlin, J.B.: Network Flows. Prentice Hall, Englewood Cliffs (1993)
3. Birnbaum, E., Lozinskii, E.L.: The Good Old Davis-Putnam Procedure Helps Counting Models. Journal of Artificial Intelligence Research 10, 457–477 (1999)
4. Cornuejols, G., Dawande, M.: A class of hard small 0-1 programs. In: Bixby, R.E., Boyd, E.A., Ríos-Mercado, R.Z. (eds.) IPCO 1998. LNCS, vol. 1412, pp. 284–293. Springer, Heidelberg (1998)
5. Danna, E., Fenelon, M., Gu, Z., Wunderling, R.: Generating Multiple Solutions for Mixed Integer Programming Problems. In: Fischetti, M., Williamson, D.P. (eds.) IPCO 2007. LNCS, vol. 4513, pp. 280–294. Springer, Heidelberg (2007)
6. Frangioni, A.: Object Bundle Optimization Package, www.di.unipi.it/optimize/Software/Bundle.html
7. Golumbic, M.C.: Algorithmic Graph Theory and Perfect Graphs. Academic Press, New York (1991)
8. Gomes, C.P., Hoeve, W., Sabharwal, A., Selman, B.: Counting CSP Solutions Using Generalized XOR Constraints. In: 22nd Conference on Artificial Intelligence (AAAI), pp. 204–209 (2007)
9. Hadzic, T., O'Mahony, E., O'Sullivan, B., Sellmann, M.: Enhanced Inference for the Market Split Problem. In: 21st IEEE International Conference on Tools with Artificial Intelligence (ICTAI), pp. 716–723 (2009)
10. IBM. IBM CPLEX Reference manual and user manual. V12.1, IBM (2009)
11. Kroc, L., Sabharwal, A., Selman, B.: Leveraging Belief Propagation, Backtrack Search, and Statistics for Model Counting. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 278–282. Springer, Heidelberg (2008)
12. Sellmann, M.: Approximated Consistency for Knapsack Constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 679–693. Springer, Heidelberg (2003)
13. Sellmann, M.: Cost-Based Filtering for Shorter Path Constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 694–708. Springer, Heidelberg (2003)
14. Sellmann, M.: Theoretical Foundations of CP-based Lagrangian Relaxation. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 634–647. Springer, Heidelberg (2004)
15. Sellmann, M., Fahle, T.: Constraint Programming Based Lagrangian Relaxation for the Automatic Recording Problem. Annals of Operations Research (AOR), 17–33 (2003)
16. Sellmann, M.: Approximated Consistency for the Automatic Recording Constraint. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 822–826. Springer, Heidelberg (2005)
17. Sellmann, M.: Approximated Consistency for the Automatic Recording Constraint. Computers and Operations Research 36(8), 2341–2347 (2009)
18. Sellmann, M.: ARP: A Benchmark Set for the Automatic Recording Problem maintained, http://www.cs.brown.edu/people/sello/arp-benchmark.html
19. TIVO$^{\text{tm}}$ System, http://www.tivo.com
20. Trick, M.: A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. In: 3rd Int. Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR), pp. 113–124 (2001)

21. Trick, M.: A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. Annals of Operations Research 118, 73–84 (2003)
22. Williams, H.P.: Model Building in Mathematical Programming. Wiley, Chicester (1978)
23. Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 743–757. Springer, Heidelberg (2007)
24. Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics. Constraints 14(3), 392–413

Matrix Interdiction Problem

Shiva Prasad Kasiviswanathan[1] and Feng Pan[2]

[1] CCS-3, Los Alamos National Laboratory
kasivisw@lanl.gov
[2] D-6, Los Alamos National Laboratory
fpan@lanl.gov

Abstract. In the matrix interdiction problem, a real-valued matrix and an integer k is given. The objective is to remove a set of k matrix columns that minimizes in the residual matrix the sum of the row values, where the value of a row is defined to be the largest entry in that row. This combinatorial problem is closely related to bipartite network interdiction problem that can be applied to minimize the probability that an adversary can successfully smuggle weapons. After introducing the matrix interdiction problem, we study the computational complexity of this problem. We show that the matrix interdiction problem is NP-hard and that there exists a constant γ such that it is even NP-hard to approximate this problem within an n^γ additive factor. We also present an algorithm for this problem that achieves an $(n-k)$ multiplicative approximation ratio.

1 Introduction

In this paper, we introduce a combinatorial optimization problem, named *matrix interdiction*. The input to a matrix interdiction problem consists of a real valued matrix of dimension $m \times n$ and an integer $k \leq n$. The objective is to remove a set of k columns (from the matrix) that minimizes in the residual matrix the sum of the row values, where the value of a row is defined to be the largest entry in that row. This combinatorial problem is closely related to a bipartite network interdiction problem that can be used to reduce the probability of nuclear material trafficking. The matrix interdiction problem turns out to be NP-hard. In fact, it turns out that it is even NP-hard to approximate this problem within an n^γ additive approximation factor (for a fixed constant $\gamma > 0$). On the positive side, we present a simple greedy algorithm that runs in time linear in the size of the input matrix and guarantees an $(n-k)$ multiplicative approximation ratio for the matrix interdiction problem.

The setting of a matrix interdiction problem is closely related to settings encountered in *resource allocation problems* [18]. One of those problems is the *network interdiction* with which the matrix interdiction shares a common name. Network interdiction is an active research area in operations research. It is easiest to view a network interdiction problem as a *Stackelberg game* on a network. There are two competitors, an evader and an interdictor, and the two competitors compete on an objective with opposing interests. The interdictor interdicts the network by modifying node and edge attributes on a network, and these modifications are usually constrained by limited resources. The evader then optimizes over the residual network. The origins of the network interdiction research can be traced back to 1970s when *minimizing maximum flow* models [13, 19] were developed to disrupt flow of enemy troops and supplies in the Vietnam War. A discrete version of maximum flow interdiction considers removing edges [27, 28] and is NP-hard. Another type of network interdiction problem is the *shortest path interdiction* where the goal is, given that only a fixed number of edges (or nodes) can be removed, to decide which set of edges (or nodes) to be removed so as to produce the largest increase in the shortest path between a source and a destination [12, 14, 17]. This problem is also known as the *most vital edges* (also *most vital nodes*) problem [7] and is also NP-hard [3].

Network interdiction problems are often inspired by different applications, e.g., detecting drug smuggling [25], analyzing power grid vulnerability [24], and fighting infectious disease in hospital [2]. Some recent network interdiction research has been motivated by homeland security applications. Researchers [26, 5] investigated how to allocate resources at individual ports to stop the illegal trafficking of weapons of mass destruction. For the application of detecting smuggled nuclear material, interdiction models that *minimize maximum-reliability paths* on a transportation network [22, 20, 21] have been used to select border crossings for the installation of radiation monitors. Stochastic models were developed to capture the uncertainties in the source-destination pairs [20], smuggling paths [15, 11], and physical characteristics of the detectors [10]. Minimizing maximum reliability path on general network can be applied to the cases where there are multiple layers of borders, but again this problem is NP-hard [21, 22]. For a single layer of border, the problem can be formulated as interdiction on bipartite network, which as we show in Section 2 is closely related to the matrix interdiction problem.

In network interdiction applications, the underlying networks are often large scale, e.g., U.S. power grids and global transportation networks. Efficient algorithms are required for these real world applications. Currently, network interdiction problems are usually formulated as stochastic integer programs, and solution methods mainly involve the techniques from mixed integer programming. Benders decomposition is an efficient method to decompose the interdiction problem to smaller subproblems [9, 17], and valid inequalities are derived to strengthen the formulation [17, 23]. Fast approximation algorithms have been developed for some special types of the network interdiction problem like the maximum flow interdiction problem [6] and the graph matching interdiction problem [29].

In this paper, we introduce a concisely defined combinatorial optimization problem that we refer to as the matrix interdiction problem. Our main contributions are the following: (a) we provide a theoretical analysis of the complexity of this problem, and (b) we design a fast approximation algorithm for it. The outline for the remaining paper is a follows. In Section 2, we will formally define the matrix interdiction problem and show that the matrix interdiction is an abstraction from a class of *stochastic resource allocation* problem. To illustrate this relation, we will show explicitly a transformation from the bipartite network interdiction to the matrix interdiction problem. A proof of NP-hardness is given in Section 3. In Section 4, we show the inapproximability result for the matrix interdiction problem, and in Section 5, we describe a greedy approximation algorithm for the matrix interdiction problem.

2 Matrix Interdiction

Let $[n]$ denote the set $\{1,\ldots,n\}$. For a matrix M of dimension $m \times n$, let $M_{i,j}$ denote the (i,j)th (ith row and jth column) entry of M. For a set $J \subseteq [n]$, let $M|_J$ denote the submatrix of M obtained by picking only the columns of M indexed by J. Define,

$$val(M|_J) = \sum_{i=1}^{m} \max_{j \in J}\{M_{i,j}\}.$$

Definition 1 (Matrix Interdiction Problem). *Let M be an $m \times n$ matrix with entries from $\mathbb{R}$. Let $\mathcal{M}_s$ be the set of all submatrices of M with dimension $m \times n - k$. The matrix interdiction problem is to select a submatrix $M^* \in \mathcal{M}_s$ such that*

$$M^* = M|_{J^*}, \text{ and } J^* = \operatorname{argmin}_{J \subseteq [n], |J| = n-k} \left\{ \sum_{i=1}^{m} \max_{j \in J} \{M_{i,j}\} \right\}.$$

In other words, the matrix interdiction problem is to find an element M^* from $\mathcal{M}_s$ with the property that

$$val(M^*) = \min_{M_z \in \mathcal{M}_s} \{val(M_z)\}.$$

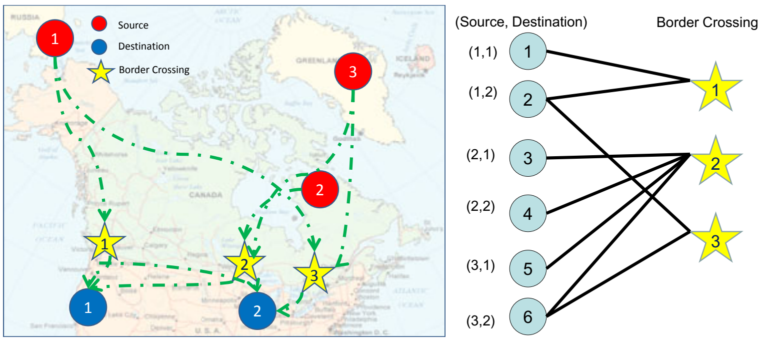


Fig. 1. Bipartite network interdiction for border control with 3 sources, 2 destinations, and 3 border crossings. The figure to the right is the corresponding bipartite network.

Next, we show a connection between the matrix interdiction problem and a special class of stochastic resource allocation problem (which we refer to as the Min-Expected-Max SRA problem). A stochastic resource allocation problem can be formulated as a two-stage stochastic program [4]. In the first stage, the resources have to be allocated without knowing which future scenarios will be realized in the second stage. In the Min-Expected-Max SRA problem, there is a process with n components. The processing time of the component j is denoted as a_j. We refer to $\mathbf{a} = (a_1, \dots, a_n)$ as the performance vector. The n components can be processed in parallel. The overall performance (the makespan of the process) is the maximum a_j over all components. In reality, the processing times of the components are probabilistic. There is a set Ω of scenarios, and for a scenario $\omega \in \Omega$ the performance vector is $\mathbf{a}^\omega$ with probability p^ω. Now, assume that we have resources to improve the performance of any k ($k \le n$) components. Improving the performance of component j results in the processing time a_j^ω decreasing by b_j^ω for all $\omega \in \Omega$. The resource allocation problem is decide which k components to improve so as to minimize the expected makespan. The Min-Expected-Max SRA problem can be formulated as a two-stage stochastic program,

$$\min_x \sum_{\omega \in \Omega} p^\omega h(x, \omega), \;\; \sum_i x_i = k, \;\; x \in \{0,1\}^n, \tag{1}$$

where

$$h(x, \omega) = \max_{i \in [n]} \{[a_i^\omega - b_i^\omega x_i]^+\}. \tag{2}$$

Here, $[\cdot]^+ = \max\{\cdot, 0\}$. This is a stochastic resource allocation problem with a simple structure. The assumption is that the changing a component performance does not effect the performance of any other component. It is this simple structure that allows the conversion from a min-expected-max two-stage stochastic program to a matrix interdiction problem. At each scenario, by subtracting a constant $c^\omega = \max_i\{[a_i^\omega - b_i^\omega]^+\}$ from each constraint in (2), we are able to simplify the second-stage optimization to

$$h(x, \omega) = \max_{i \in [n]} \{[\hat{a}_i^\omega - \hat{a}_i^\omega x_i]^+\}, \tag{3}$$

where $\hat{a}_i^\omega = a_i^\omega - c^\omega$.

The optimization problems (2) and (3) are equivalent in the sense that they have the same optimal solutions and they are different by a constant at their optimal value. The full process of this simplification step involves elaborate algebraic manipulations, and we refer the reader to [21, 20] which discusses this in the context of interdiction. After the simplification, the two-stage stochastic program

$$\begin{gathered} \min_x \textstyle\sum_{\omega \in \Omega} p^\omega h(x, \omega), \;\; \sum_i x_i = k, \;\; x \in \{0,1\}^n, \text{ where} \\ h(x, \omega) = \max_{i \in [n]} \{[\hat{a}_i^\omega - \hat{a}_i^\omega x_i]^+\}. \end{gathered} \tag{4}$$

can be converted to a matrix interdiction problem whose input is a matrix M of dimension $|\Omega| \times n$. Each row of M represents a scenario ω. Let $\omega_i \in \Omega$ be the scenario corresponding to the ith row, then $M_{i,j} = \hat{a}_j^{\omega_i}$.

Next, we define the bipartite interdiction problem that has a similar formulation as that of the optimization problem (4) defined above.

Bipartite Network Interdiction Problem. In the bipartite network interdiction problem, there is a set of border crossings (B) that separate the sources (S) from the destinations (T). An evader attempts to travel from a source to a destination, and any source-destination route will go through one and only one border crossing. Evasion may happen between any source-destination (s-t) pair, and every s-t pair has a probabilistic weight p_{st} on it. At each edge, there is the edge reliability defined as the probability of traversing the edge without being captured, and the evader will use a route with the maximum reliability. These edge reliabilities can be derived from travel time and distance (see [20, 10] for more details). For a triplet (s, b, t), where $s \in S$, $b \in B$ and $t \in T$, we can calculate the maximum reliability of a path from s through b to t and denote it as r_{sbt}. Interdiction of a border crossing b means to strengthen the security at that location, and as the result, $r_{sbt} = 0$ for all $s \in S$ and $t \in T$. The bipartite network structure is formed by representing source-destination pairs as one set of nodes and border crossings as the other set of nodes. An edge in the bipartite network implies that there exists a path connecting the triplet. For example, in Figure 1, there are three sources, three border crossings (forming a single layer of border crossings), and two destinations. To go from source 1 to destination 1 the evader has to go through crossing 1, therefore, there is an edge between source-destination pair (1,1) and crossing 1. While, between source 1 and destination 2 the evader has the option of using either crossing 1 or 3, therefore, there are edges between source-destination pair (1,2) and crossings 1 and 3. Figure 1(b) shows the bipartite network. The maximum reliability for the triplet (1,1,2) is calculated by multiplying the maximum reliability between source 1 and crossing 1 and between crossing 1 and destination 2. A budgetary constraint limits the interdiction to k crossings, and the objective of the interdiction is to select k crossings that minimizes the expected maximum probability of successful evasion between any pair of source and destination. This problem can be formulated as a bi-level integer program:

$$\min_{|X|=k, X\in\{0,1\}^{|B|}} \sum_{(s,t)\in S\times T} p_{st} \cdot \max_{b\in B}\{r_{sbt} \cdot (1 - x_b)\}. \tag{5}$$

For more details on the bipartite network interdiction, see [21, 20].

We can construct a matrix M from bipartite network interdiction problem as follows. The dimension of M is set as $n = |B|$ and $m = |S| \times |T|$, and the entry $M_{ij} = r_{sjt}p_{st}$, where i is the node index in the bipartite network for source-destination pair (s, t). With this construction, the entries of M are positive real values between 0 and 1, and the optimal solution of the matrix interdiction problem for input M is exactly the k optimal border crossings to be interdicted in Equation (5). Also, the two problems will also have the same optimal objective values. This leads to the following theorem.

Theorem 1. *Bipartite network interdiction problem is a special case of the matrix interdiction problem.*

The above theorem can be summarized as saying that every instance of the bipartite interdiction problem is also an instance of the matrix interdiction problem. The NP-hardness (Section 3) and the hardness of approximation (Section 4) results that we obtain for the matrix interdiction problem also hold for the bipartite network interdiction problem. Also, since the approximation guarantee of the greedy algorithm (Section 5) holds for every instance of the matrix interdiction problem, it also holds for every instance of the bipartite network interdiction problem. In the following sections, we concentrate only on the matrix interdiction problem.

3 NP-Hardness Result

In this section, we show that the matrix interdiction problem is NP-hard. Thus, assuming P $\neq$ NP there exists no polynomial time algorithm that can exactly solve the matrix interdiction problem. For establishing the NP-hardness we reduce the clique problem to the matrix interdiction problem. The clique problem is defined as follows.

Definition 2 (Clique Problem [8]). *Let $G = (V, E)$ be an undirected graph, where V is the set of vertices and E is the set of edges of G. For a subset $S \subseteq V$, we let $G(S)$ denote the subgraph of G induced by S. A clique C is a subset of V such that the induced graph $G(C)$ is complete (i.e., $\forall u, v \in C$ an edge exists between u and v). The clique problem is the optimization problem of finding a clique of maximum size in the graph. As a decision problem, it requires us to decide whether there exists a clique of a given size k in the graph.*

Reduction from Clique to Matrix Interdiction. Consider a graph $G = (V, E)$ with $|E| = m$ and $|V| = n$. We construct a matrix $M = M(G)$ of dimension $m \times n$ as follows: The rows of M correspond to the edges of G and columns of M correspond to the vertices of G. Let $e_1, \ldots, e_m$ be the edges of G. Now for every $l \in [m]$ consider the edge e_l, and let u and v be the end points of e_l. In the lth row of M add 1 in the columns corresponding to u and v, all other entries of the lth row are 0.

Notice that M has exactly two 1's in each row, and all the remaining entries of M are 0. Now,

$$val(M) = \sum_{i=1}^{m} \max_{j \in [n]} \{M_{i,j}\} = \sum_{i=1}^{m} 1 = m.$$

That is each edge in G contributes 1 to $val(M)$. Now if there exists a clique C of size k in G, then we can delete the columns of M corresponding to vertices in

C and we obtain a submatrix M^* with $val(M^*) = m - \binom{k}{2}$ (because by deleting the columns corresponding to C, contribution to $val(M^*)$ will be 0 for the $\binom{k}{2}$ rows of M corresponding to the $\binom{k}{2}$ edges in C). Similarly, if the output to the matrix interdiction problem is a matrix M^* and if $val(M^*) > m - \binom{k}{2}$ then there exists no clique of size k in G and if $val(M^*) = m - \binom{k}{2}$ then there exists a clique of size k in G. The following two lemmas formalize the observations explained above.

Lemma 1. *Consider a graph G, and let $M = M(G)$ be the matrix as defined above. If there exists a clique C of size k in G, then there exists a submatrix M^* of M such that $val(M^*) = val(M) - \binom{k}{2}$ and M^* is a (optimum) solution to the matrix interdiction problem. Otherwise, if there exists no clique of size k in G then any (optimum) solution to the matrix interdiction problem will have a value strictly greater than $val(M) - \binom{k}{2}$.*

Proof. To show the first part of the lemma notice that each row of M contributes 1 to $val(M)$, or in other words each edge in G contributes 1 to $val(M)$. Consider a row of M, let us assume it corresponds to some edge (u, v) in G. Now notice that to obtain M^* if one only deletes the column corresponding to u or the column corresponding to v then the contribution of this row to $val(M^*)$ still remains 1 (because the row has two 1's and only one of these gets removed). So to reduce the contribution of this row to 0 one needs to delete columns corresponding to both u and v.

A clique C of size k has exactly $\binom{k}{2}$ edges between the vertices in C. Therefore, by deleting the columns corresponding to vertices in C, one can create a submatrix M^* of dimension $m \times n - k$ with $val(M^*) = val(M) - \binom{k}{2}$. We now argue that M^* is a (optimum) solution to the matrix interdiction problem. Consider a set $J \subseteq [n], |J| = n - k$ and let $\bar{J} = [n] - J$. Deleting the columns of M in $\bar{J}$ creates $M|_J$ in which the number of rows with all zero entries is the same as the number of edges present between the vertices corresponding to entries in $\bar{J}$. In other words, $val(M|_J) = val(M) - e(\bar{J})$, where $e(\bar{J})$ is the number of edges in G that are present between the vertices corresponding to entries in $\bar{J}$. Since, for any $\bar{J}$, $e(\bar{J}) \leq \binom{k}{2}$, therefore for all J,

$$val(M|_J) \geq val(M) - \binom{k}{2}.$$

Therefore, M^* whose value equals $val(M) - \binom{k}{2}$ is a (optimum) solution to the matrix interdiction problem.

To show the second part of the lemma, notice that if there exists no clique of size k in G, then for all $\bar{J}$,

$$val(M|_J) = val(M) - e(\bar{J}) > val(M) - \binom{k}{2},$$

as in the absence of a clique of size k, $e(\bar{J})$ is always less than $\binom{k}{2}$.

Lemma 2. *Consider a graph G, and let $M = M(G)$ be the matrix as defined above. Let M^* be a (optimum) solution to the matrix interdiction problem with input M. Then if $val(M^*) = val(M) - \binom{k}{2}$ then there exists a clique of size k in G, and otherwise there exists no clique of size k in G.*

Proof. From Lemma 1, we know that $val(M^*) \geq val(M) - \binom{k}{2}$. Let $\bar{J}$ be the set of columns deleted from M to obtain M^*. If $val(M^*) = val(M) - e(\bar{J}) = val(M) - \binom{k}{2}$, then the vertices corresponding to entries in $\bar{J}$ form a clique of size k (as $e(\bar{J}) = \binom{k}{2}$). If $val(M^*) > val(M) - \binom{k}{2}$, then there exists no clique of size k in G because if there did exist a clique of size k in G then one can delete the columns corresponding to the vertices in the clique to obtain a matrix M_z with

$$val(M_z) = val(M) - \binom{k}{2} < val(M^*),$$

a contradiction to the optimality of M^*.

Theorem 2. *The matrix interdiction problem is NP-hard.*

Proof. The clique problem is NP-complete [8]. Lemmas 1 and 2 show a polynomial time reduction from the clique problem to the matrix interdiction problem. Therefore, the matrix interdiction problem is NP-hard.

4 Inapproximability Result

In this section, we show that there exists a fixed constant γ such that the matrix interdiction problem is NP-hard to approximate to within an n^γ additive factor. More precisely, we show that assuming P $\neq$ NP there exists no polynomial time approximation algorithm for the matrix interdiction problem that can achieve better than an n^γ additive approximation. Note that this statement is stronger than Theorem 2. Whereas, Theorem 2 shows that assuming P $\neq$ NP there exists no polynomial time algorithm that can solve the matrix interdiction problem exactly, this inapproximability statement shows that unless P = NP it is not even possible to design a polynomial time algorithm which gives close to an optimum solution for the matrix interdiction problem.

To show the inapproximability bound we reduce a problem with known inapproximability bound to the matrix interdiction problem. We will use a reduction that is similar to that in the previous section. It will be convenient to use a variant of the clique problem known as the k-clique.

Definition 3 (k-clique Problem). *In the k-clique problem the input consists of a positive integer k and a k-partite graph G (that is a graph that can be partitioned into k disjoint independent sets) along with its k-partition. The goal is to find the largest clique in G. Define a function k-clique(G) as $\ell(G)/k$, where $\ell(G)$ is the size of the largest clique in G.*

Since in a k-partite graph G a clique can have at most one vertex in common with an independent set, the size of the largest clique in G is at most k. Therefore, k-clique$(G) \leq 1$.

Theorem 3 (Arora *et al.* [1]). *There exists a fixed $0 < \delta < 1$ such that approximating the k-clique problem to within an n^δ multiplicative factor is NP-hard.*

Proof Sketch. The proof presented in [1] (see also Chapter 10 in [16]) proceeds by showing a polynomial time reduction τ from the *SAT* problem (the problem of determining whether the variables of a Boolean formula can be assigned in a way that makes the formula satisfiable) to the k-clique problem. The reduction τ ensures for all instances I of SAT:

$$\text{If } I \text{ is satisfiable} \Rightarrow k\text{-clique}(\tau(I)) = 1,$$
$$\text{If } I \text{ is not satisfiable} \Rightarrow k\text{-clique}(\tau(I)) \leq \tfrac{1}{n^\delta}.$$

Since, SAT is a NP-complete problem, therefore, approximating the k-clique problem to within an n^δ multiplicative factor is NP-hard (because if one can approximate the k-clique problem to within an n^δ multiplicative factor, then one can use τ to solve the SAT problem in polynomial time). □

The following lemma relates the problem of approximating the k-clique to approximating the matrix interdiction problem.

Lemma 3. *Let G be a k-partite graph and $0 < \delta < 1$ be the constant from Theorem 3. Let $M = M(G)$ be a matrix created from G as defined in Section 3. Let M^* be a (optimum) solution to the matrix interdiction problem with input M. Then*

$$\textit{If } k\textit{-clique}(G) = 1 \Rightarrow val(M^*) = val(M) - \tbinom{k}{2},$$
$$\textit{If } k\textit{-clique}(G) \leq \tfrac{1}{n^\delta} \Rightarrow val(M^*) \leq val(M) - n^\delta \tbinom{k/n^\delta}{2} - \tbinom{n^\delta}{2}\left(\left(\tfrac{k}{n^\delta}\right)^2 - 1\right).$$

Proof. If k-clique$(G) = 1$, then the size of the largest clique in G is k, and by deleting the columns corresponding to the vertices in this clique we get a submatrix M^* with $val(M^*) = val(M) - \binom{k}{2}$.

If the k-clique$(G) \leq 1/n^\delta$, then the size of the largest clique in G is at most k/n^δ. The maximum reduction to $val(M)$ occurs when there are n^δ cliques each of size k/n^δ and one deletes the k columns corresponding to the vertices appearing in all these cliques. Each clique has $\binom{k/n^\delta}{2}$ edges within itself. Since there are n^δ such cliques, this gives a total of $n^\delta\binom{k/n^\delta}{2}$ edges within the cliques. There are also edges across these n^δ cliques. Now across any two cliques there are at most $(k/n^\delta)^2 - 1$ edges, and since there are at most $\binom{n^\delta}{2}$ such pairs of cliques, this gives a total of $\binom{n^\delta}{2}((k/n^\delta)^2 - 1)$ edges across the cliques. Accounting for all edges within and across cliques, we get

$$val(M^*) \leq val(M) - n^\delta \binom{k/n^\delta}{2} - \binom{n^\delta}{2}\left(\left(\frac{k}{n^\delta}\right)^2 - 1\right).$$

Theorem 4. *There exists a fixed constant $\gamma > 0$, such that the matrix interdiction problem is NP-hard to approximate within an additive factor of n^γ.*

Proof. From Theorem 3, we know there exists a constant δ such that it is NP-hard to approximate the k-clique problem to within an n^δ multiplicative factor. From Lemma 3, we know that for a k-partite graph G, there exists a matrix $M = M(G)$ such that

$$\text{If } k\text{-clique}(G) = 1 \Rightarrow val(M^*) = val(M) - \left(\tfrac{k^2}{2} - \tfrac{k}{2}\right),$$

$$\text{If } k\text{-clique}(G) \leq \tfrac{1}{n^\delta} \Rightarrow val(M^*) \leq val(M) - \left(\tfrac{k^2}{2n^\delta} - \tfrac{k}{2}\right) - \left(\tfrac{k^2}{2} - \tfrac{k^2}{2n^\delta} - \tfrac{n^{2\delta}}{2} + \tfrac{n^\delta}{2}\right).$$

By comparing the above two equations, we see that if we can approximate the matrix interdiction problem within an $n^{2\delta}/2 - n^\delta/2$ additive factor, then we can approximate the k-clique problem to within an n^δ multiplicative factor. Since, the latter is NP-hard, it implies that an $n^{2\delta}/2 - n^\delta/2$ additive approximation of the matrix interdiction problem is also NP-hard. Setting γ such that, $n^\gamma = n^{2\delta}/2 - n^\delta/2$ proves the theorem.

5 Greedy Approximation Algorithm

In this section, we present a greedy algorithm for the matrix interdiction problem that achieves an $(n - k)$ multiplicative approximation ratio. The input to the greedy algorithm is a matrix M of dimension $m \times n$ with real entries. The output of the algorithm is a matrix M_g. The running time of the algorithm is $O(nm + n \log n)$.

```
ALGORITHM GREEDY(M)

1. For every $j \in [n]$, compute $c_j = \sum_{i=1}^{m} M_{i,j}$, i.e., $c_j$ is the sum of the entries in the $j$th column.
2. Pick the top $k$ columns ranked according to the column sums.
3. Delete the $k$ columns picked in Step 2 to create a submatrix $M_g$ of $M$.
4. Output $M_g$.
```

Theorem 5. *Algorithm Greedy is an $(n - k)$ multiplicative approximation algorithm for the matrix interdiction problem. More precisely, the output M_g of Greedy(M) satisfies the following*

$$val(M_g) \leq (n - k)val(M^*),$$

where M^* is a (optimum) solution to the matrix interdiction problem with input M.

Proof. Let $Soln \subseteq [n], |Soln| = n - k$ be the set of $n - k$ columns present in M_g. Let $Opt \subseteq [n], |Opt| = n - k$ be the set of $n - k$ columns present in M^*. Now,

$$
\begin{aligned}
val(M_g) &= \sum_{i=1}^{m} \max_{j\in Soln} \{M_{i,j}\} \leq \sum_{i=1}^{m} \sum_{j\in Soln} M_{i,j} \\
&= \sum_{j\in Soln} \sum_{i=1}^{m} M_{i,j} \leq \sum_{j\in Opt} \sum_{i=1}^{m} M_{i,j} \\
&= \sum_{i=1}^{m} \sum_{j\in Opt} M_{i,j} \leq \sum_{i=1}^{m} (n-k) \max_{j\in Opt} \{M_{i,j}\} \\
&= (n-k) \sum_{i=1}^{m} \max_{j\in Opt} \{M_{i,j}\} \\
&= (n-k) val(M^*).
\end{aligned}
$$

The second inequality follows because the Greedy algorithm deletes the k columns with the largest column sums. The third inequality follows because for any real vector $v = (v_1, \ldots, v_{n-k})$, $\sum_{p=1}^{n-k} v_p \leq (n-k)\max\{v\}$.

The above argument shows that the Greedy algorithm achieves an $(n-k)$ multiplicative approximation ratio for the matrix interdiction problem.

6 Conclusion

Motivated by security applications, we introduced the matrix interdiction problem. Our main contribution is in providing a complexity analysis and an approximation algorithm for this problem. We proved that the matrix interdiction problem is NP-hard, and furthermore, unless P = NP there exists no n^γ additive approximation algorithm for this problem. We then presented a simple greedy algorithm for the matrix interdiction problem and showed that this algorithm has an $(n-k)$ multiplicative approximation ratio. It is also possible to design a dynamic programming based algorithm that achieves the same approximation ratio. An interesting open question would be to either design a better approximation algorithm or to show a better hardness of approximation result.

References

[1] Arora, S., Lund, C., Motwani, R., Sudan, M., Szegedy, M.: Proof verification and the hardness of approximation problems. Journal of the ACM (JACM) 45(3), 555 (1998)

[2] Assimakopoulos, N.: A network interdiction model for hospital infection control. Comput. Biol. Med. 17(6), 413–422 (1987)

[3] Bar-Noy, A., Khuller, S., Schieber, B.: The complexity of finding most vital arcs and nodes. Technical report, University of Maryland (1995)

[4] Birge, J.R., Louveaux, F.: Introduction to stochastic programming. Springer, New York (1997)

[5] Boros, E., Fedzhora, L., Kantor, P.B., Saeger, K., Stroud, P.: Large scale lp model for finding optimal container inspection strategies. Naval Research Logistics Quarterly 56(5), 404–420 (2009)

[6] Burch, C., Carr, R., Krumke, S., Marathe, M., Phillips, C., Sundberg, E.: A decomposition-based pseudoapproximation algorithm for network flow inhibition. In: Network Interdiction and Stochastic Integer Programming. Operations Research/Computer Science Interfaces, vol. 22, pp. 51–68. Springer, US (2003)

[7] Corley, H.W., Sha, D.Y.: Most vital links and nodes in weighted networks. Operations Research Letters 1(4), 157–160 (1982)

[8] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT Press, Cambridge (2001)

[9] Cormican, K.J., Morton, D.P., Wood, K.R.: Stochastic network interdiction. Operations Research 46(2), 184–197 (1998)

[10] Dimitrov, N., Michalopoulos, D.P., Morton, D.P., Nehme, M.V., Pan, F., Popova, E., Schneider, E.A., Thoreson, G.G.: Network deployment of radiation detectors with physics-based detection probability calculations. Annals of Operations Research (2009)

[11] Dimitrov, N.B., Morton, D.P.: Combinatorial design of a stochastic markov decision process. In: Operations Research and Cyber-Infrastructure. Operations Research/Computer Science Interfaces, vol. 47, pp. 167–193. Springer, Heidelberg (2009)

[12] Fulkerson, D.R., Harding, G.C.: Maximizing the minimum source-sink path subject to a budget constraint. Mathematical Programming 13(1), 116–118 (1977)

[13] Ghare, P.M., Montgomery, D.C., Turner, W.C.: Optimal interdiction policy for a flow network. Naval Research Logistics Quarterly 18(1), 37 (1971)

[14] Golden, B.: A problem in network interdiction. Naval Research Logistics Quarterly 25, 711–713 (1978)

[15] Gutfraind, A., Hagberg, A., Pan, F.: Optimal interdiction of unreactive markovian evaders. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 102–116. Springer, Heidelberg (2009)

[16] Hochbaum, D.S. (ed.): Approximation algorithms for NP-hard problems. PWS Publishing Co., Boston (1997)

[17] Israeli, E., Kevin Wood, R.: Shortest-path network interdiction. Networks 40(2), 97–111 (2002)

[18] Karabati, S., Kouveils, P.: A min-sum-max resource allocation problem. IEE Transactions 32(3), 263–271 (2000)

[19] McMasters, A.W., Mustin, T.M.: Optimal interdiction of a supply network. Naval Research Logistics Quarterly 17(3), 261 (1970)

[20] Morton, D.P., Pan, F., Saeger, K.J.: Models for nuclear smuggling interdiction. IIE Transactions 39(1), 3–14 (2007)

[21] Pan, F.: Stochastic Network Interdiction: Models and Methods. PhD dissertation, University of Texas at Austin, Operations Research (2005)

[22] Pan, F., Charlton, W., Morton, D.P.: Interdicting smuggled nuclear material. In: Woodruff, D.L. (ed.) Network Interdiction and Stochastic Integer Programming, pp. 1–19. Kluwer Academic Publishers, Boston (2003)

[23] Pan, F., Morton, D.P.: Minimizing a stochastic maximum-reliability path. Networks 52(3), 111–119 (2008)

[24] Salmeron, J., Wood, K., Baldick, R.: Worst-case interdiction analysis of large-scale electric power grids. IEEE Transactions on Power Systems 24(1), 96–104 (2009)

[25] Washburn, A., Wood, K.R.: Two-person zero-sum games for network interdiction. Operations Research 43(2), 243–251 (1995)

[26] Wein, L.M., Wilkins, A.H., Baveja, M., Flynn, S.E.: Preventing the importation of illicit nuclear materials in shipping containers. Risk Analysis 26(5), 1377–1393 (2006)
[27] Wollmer, R.: Removing Arcs from a Network. Operations Research 12(6), 934–940 (1964)
[28] Wood, R.K.: Deterministic network interdiction. Mathematical and Computer Modeling 17, 1–18 (1997)
[29] Zenklusen, R.: Matching interdiction. Arxiv preprint arXiv:0804.3583 (2008)

Strong Combination of Ant Colony Optimization with Constraint Programming Optimization

Madjid Khichane[1,2], Patrick Albert[1], and Christine Solnon[2,⋆]

[1] IBM
9, rue de Verdun, Gentilly 94253, France
{madjid.khichane,palbert}@fr.ibm.com
[2] Université de Lyon
Université Lyon 1, LIRIS CNRS UMR5205, France
christine.solnon@liris.cnrs.fr

Abstract. We introduce an approach which combines ACO (Ant Colony Optimization) and IBM ILOG CP Optimizer for solving COPs (Combinatorial Optimization Problems). The problem is modeled using the CP Optimizer modeling API. Then, it is solved in a generic way by a two-phase algorithm. The first phase aims at creating a hot start for the second: it samples the solution space and applies reinforcement learning techniques as implemented in ACO to create pheromone trails. During the second phase, CP Optimizer performs a complete tree search guided by the pheromone trails previously accumulated. The first experimental results on knapsack, quadratic assignment and maximum independent set problems show that this new algorithm enhances the performance of CP Optimizer alone.

1 Introduction

Combinatorial Optimization Problems (COPs) are of high importance for the scientific world as well as for the industrial world. Most of these problems are *NP*-hard so that they cannot be solved exactly in polynomial time (unless $P = NP$). Examples of *NP*-hard COPs include timetabling, telecommunication network design, traveling salesman problems, Multi-dimensional Knapsack Problems (MKPs), and Quadratic Assignment Problems (QAPs). To solve COPs, two main dual approaches may be considered, i.e., Branch and Propagate and Bound (B&P&B) approaches, and metaheuristic approaches.

B&P&B approaches combine an exhaustive tree-based exploration of the search space with constraint propagation and bounding techniques which reduce the search space. These approaches ensure to find an optimal solution in bounded time. As a counterpart, they might need exponential computation time in the worst case [1,2]. Constraint Programming (CP) is one of the most popular generic B&P&B approaches for solving COPs modeled by means of constraints:

⋆ This work has been partially financed by IBM ILOG under the research collaboration contract IBM ILOG/UCBL-LIRIS.

it offers high-level languages for modeling COPs and it integrates constraint propagation and search algorithms for solving them in a generic way. Hence, solving COPs with CP does not require a lot of programming work. CP is usually very effective when constraints are tight enough to eliminate large infeasible regions by propagating constraints. However, as it is generally based on B&P&B approach, it fail to found a high solution quality with an acceptable computational time limit.

Metaheuristics have shown to be very effective for solving many COPs. They explore the search space in an incomplete way and sacrifice optimality guarantees, but gets good solutions in reasonable computational times. However, solving a new problem with a metaheuristic usually requires a lot of programming work. In particular, handling constraints is not an easy task and requires designing appropriate incremental data structures for quickly evaluating constraint violations before making a decision.

Many metaheuristics are based on a local search framework such that the search space is explored by iteratively perturbing combinations. Among others, local search-based metaheuristics include simulated annealing [3], tabu search [4], iterated local search [5], and variable neighborhood search [6]. To ease the implementation of local search-based algorithms for solving COPs, Van Hentenryck and Michel have designed a high level constraint-based language, named Comet [7]. In particular, Comet introduces incremental variables, thus allowing the programmer to declaratively design data structures which are able to efficiently evaluate neighborhoods.

In this paper, we propose a generic approach for solving COPs which combines CP Optimizer —a B&P&B-based solver developed by IBM ILOG— with the Ant Colony Optimization (ACO) metaheuristic [8]. ACO is a constructive approach (and not a local search-based one): it explores the search space by iteratively constructing new combinations in a greedy randomized way. ACO has shown to be very effective to quickly find good solutions to COPs, but it suffers from the same drawbacks as other metaheuristics, i.e., there are no optimality guarantees and quite a lot of programming is required to solve new COPs with ACO.

By combining ACO with CP Optimizer, we take the best of both approaches. In particular, we use the CP Optimizer modeling API to describe the problem to solve. Hence, to solve a new COP with our approach, one only has to model the problem to solve by means of a set of constraints and an objective function to optimize, and then ask the solver to search for the optimal solution. This search is decomposed in two phases. In a first phase, ACO is used to sample the space of feasible solutions and gather useful information about the problem by means of pheromone trails. During this first phase, CP Optimizer is used to propagate constraints and provide feasible solutions to ACO, while in a second phase, it performs a complete B&P&B to search for the optimal solution. During this second phase, pheromone trails collected during the first phase are used by CP Optimizer as value ordering heuristics, allowing it to quickly focus on the most promising areas of the space of feasible solutions. In both phases, we also use impacts [9] as an ordering heuristic.

Let us point out that our main objective is not to compete with state-of-the-art algorithms which are dedicated to solving specific problems, but to show that sampling the search space with ACO can significantly improve the solution process of a generic B&P&B approach for solving COPs. For this, we chose CP Optimizer as our reference.

The rest of this paper is organized as follows. In Section 2, we recall some definitions about COP, CP, and ACO. Section 3 describes the *CPO-ACO* algorithm. In section 4, we give some experimental results on the multidimensional knapsack problem, the quadratic assignment problem and the maximum independent set problem. We conclude with a discussion on some other related work and further work.

2 Background

2.1 COP

A COP is defined by a tuple $P = (X, D, C, F)$ such that $X = \{x_1, \ldots, x_n\}$ is a set of n decision variables; for every variable $x_i \in X$, $D(x_i)$ is a finite set of integer values defining the domain of x_i; C is a set of constraints; and $F : D(x_1) \times \ldots \times D(x_n) \longrightarrow \mathbb{R}$ is an objective function to optimize.

An assignment $\mathcal{A}$ is a set of variable-value couples denoted $< x_i, v_i >$ which correspond to the assignment of a value $v_i \in D(x_i)$ to a variable x_i. An assignment $\mathcal{A}$ is complete if all variables of X are assigned in $\mathcal{A}$; it is partial otherwise. An assignment is inconsistent if it violates a constraint and it is consistent otherwise. A feasible solution is a complete consistent assignment. A feasible solution $\mathcal{A}$ of P is optimal if for every other feasible solution $\mathcal{A}^{'}$ of P, $F(\mathcal{A}) \leq F(\mathcal{A}^{'})$ if P is a minimization problem or $F(\mathcal{A}) \geq F(\mathcal{A}^{'})$ if P is a maximization problem.

2.2 Complete B&P&B Approaches

B&P&B approaches solve COPs by building search trees: at each node, one chooses a non-assigned variable x_i and, for each value $v_i \in D(x_i)$, one creates a new node corresponding to the assignment of x_i to v_i. This tree search is usually combined with constraint propagation and bounding techniques. Constraint propagation filters variable domains by removing inconsistent values with respect to some local consistency such as, for example, arc consistency. Bounding techniques compute bounds on the objective function and prune the nodes for which this approximation is worse than the best feasible solution found so far. When constraint propagation or bounding techniques detect a failure, one backtracks to the last choice point to explore another branch. This method is effective and generic, although it fails to solve some COPs for which constraint propagation and bounding techniques are not able to reduce the search space to a reasonable size.

2.3 Impact-Based Search Strategies

In constraint programming, as soon as a value v_i is assigned to a variable x_i, constraint propagation removes part of the infeasible space by reducing the

Algorithm 1. Generic ACO framework for solving a COP (X, D, C, F)

```
1 Initialize pheromone trails
2 while Stopping criteria not reached do
3     for Each ant do Construct a complete assignment
4     Update pheromone trails
```

domains of some variables. Refalo [9] has defined the *impact* of the assignment $x_i = v_i$ as the proportion of search space removed. He has defined the impact of a value as the average of its observed impacts and the impact of a variable as the average of the impact of its remaining values. He has shown that these impacts may be used to define valuable ordering heuristics.

2.4 Ant Colony Optimization (*ACO*)

There exist two main kinds of heuristic approaches, i.e., perturbative and constructive approaches. Perturbative heuristic approaches (such as local search) explore the search space by iteratively perturbating combinations. Constructive heuristic approaches sample the search space by iteratively constructing combinations in a greedy randomized way: starting from an empty combination, combination components are iteratively added until the combination is complete. At each step of these constructions, the combination component to be added is randomly chosen with respect to some probability.

Ant Colony Optimization (ACO) [8] is a constructive heuristic approach which borrows features from the collective behavior of ants to define the probability of adding a component to a combination: this probability depends on a quantity of pheromone which represents the past experience with respect to the choice of this component.

Algorithm 1 describes the generic ACO framework: at each cycle, each ant builds an assignment in a greedy randomized way using pheromone trails to progressively bias probabilities with respect to previous constructions; then pheromone trails are updated. We describe the main steps of this algorithm in the next paragraph, with a focus on COPs described by means of a tuple (X, D, C, F) so that the goal is to find the best feasible assignment.

Pheromone trails: Pheromone is used to guide the search and a key point lies in the choice of the components on which pheromone is laid. When the COP is defined by a tuple (X, D, C, F), one may associate a pheromone trail $\tau(x_i, v_i)$ with every variable $x_i \in X$ and every value $v_i \in D(x_i)$. Intuitively, this pheromone trail represents the desirability of assigning x_i to v_i. Such a pheromone structure has shown to be effective to solve, for example, QAPs [10], CSPs [11], and car sequencing problems [12].

Pheromone trails are used to intensify the search around the best assignments built so far. In order to balance intensification and diversification, Stützle and Hoos have proposed in [10] to bound pheromone trails between two parameters τ_{min} and τ_{max} so that the relative difference between pheromone trails is limited. Also, pheromone trails are initialized to τ_{max} at the beginning of an ACO search.

Construction of assignments by ants (line 3): Each assignment is built in a greedy randomized way: starting from an empty assignment, one iteratively chooses a non assigned variable and a value to assign to this variable, until all variables have been assigned. The next variable to assign is usually chosen with respect to some given ordering heuristic (e.g., in increasing order for the QAP or the car sequencing problem, or with respect to the min-domain heuristic for CSPs). Once a non assigned variable x_i has been chosen, the value $v_i \in D(x_i)$ to assign to x_i is chosen with respect to probability:

$$p(x_i, v_i) = \frac{[\tau(x_i, v_i)]^\alpha \cdot [\eta(x_i, v_i)]^\beta}{\sum_{v_j \in D(x_i)} [\tau(x_i, v_j)]^\alpha \cdot [\eta(x_i, v_j)]^\beta} \tag{1}$$

where $\eta(x_i, v_i)$ is the heuristic factor associated with the assignment of x_i to v_i. The definition of this factor depends on the considered application, and usually evaluates the impact of this assignment on the objective function. α and β are two parameters that allow the user to balance the influence of pheromone and heuristic factors in the transition probability.

The way constraints are handled may be different from a COP to another. For loosely constrained COPs, such as QAPs [10], maximum clique problems [13], or MKPs [14], constraints are propagated after each variable assignment in order to remove inconsistent values from the domains of non assigned variables, so that ants always build feasible solutions. However, when constraints are tighter so that it is actually difficult to build feasible solutions, constraint violations may be integrated in the heuristic factor and in the objective function so that ants may build inconsistent assignments [11].

Pheromone updating step (line 4): Once each ant has constructed an assignment, pheromone trails are updated. In a first step, all pheromone trails are decreased by multiplying them by a factor $(1 - \rho)$, where $\rho \in [0; 1]$ is the evaporation rate. This evaporation process allows ants to progressively forget older constructions and to emphasize more recent ones. In a second step, some assignments are rewarded by laying pheromone trails. These assignments may be the best of the cycle and/or the best since the beginning of the search. The goal is to increase the probability of selecting the components of these assignments during the next constructions. The pheromone is laid on the trails associated with the rewarded assignments. When pheromone trails are associated with variable/value couples, pheromone is laid on the variable/value couples of the assignment to reward. The quantity of pheromone laid usually is proportional to the quality of the rewarded assignment. This quantity is often normalized between 0 and 1 by defining it as a ratio between the value of the assignment to reward and the optimal value (if it is known) or the best value found since the beginning of the search.

3 Description of $CPO - ACO$

ACO has shown to be very effective for quickly finding good solutions to many COPs. However, designing ACO algorithms for new COPs implies a lot of programming: if procedures for managing and exploiting pheromone are very similar from a COP to another so that one can easily reuse them, solving a new COP implies to write procedures for propagating and checking problem dependent constraints. Hence, a first motivation for combining ACO with CP is to reuse the numerous available procedures for managing constraints. Moreover, combining ACO with CP optimizer allows us to take the best of these two approaches:

– During a first phase, CP Optimizer is used to sample the space of feasible solutions, and pheromone trails are used to progressively intensify the search around the best feasible solutions.
– During a second phase, CP Optimizer is used to search for an optimal solution, and the pheromone trails collected during the first phase are used to guide CP Optimizer in this search.

3.1 First Phase of $CPO - ACO$

Algorithm 2 describes the first phase of $CPO - ACO$, the main steps of which are described in the next paragraphs.

Pheromone structure: The pheromone structure is used in order to progressively intensify the search around the most promising areas, i.e., those that

Algorithm 2. Phase 1 of $CPO - ACO$

```
Input: a COP $P = (X, D, C, F)$ and a set of parameters
        $\{t_{max1}, d_{min}, it_{max}, \alpha, \beta, \rho, \tau_{min}, \tau_{max}, nbAnts\}$
Output: A feasible solution $\mathcal{A}_{best}$ and a pheromone matrix
        $\tau : X \times D \rightarrow [\tau_{min}; \tau_{max}]$
1  foreach $x_i \in X$ and foreach $v_i \in D(x_i)$ do $\tau(x_i, v_i) \leftarrow \tau_{max}$
2  repeat
       /* Step 1: Construction of $nbAnts$ feasible solutions */
3      foreach $k \in \{1, \dots, nbAnts\}$ do
4          Construct a feasible solution $\mathcal{A}_k$ using CP Optimizer
       /* Step 2: Evaporation of all pheromone trails */
5      foreach $x_i \in X$ and foreach $v_i \in D(x_i)$ do
6          $\tau(x_i, v_i) \leftarrow max(\tau_{min}, (1 - \rho) \cdot \tau(x_i, v_i))$
       /* Step 3: Pheromone laying on good feasible solutions */
7      Let $\mathcal{A}_{best}$ be the best assignment built so far (including the current cycle)
8      foreach $k \in \{1, \dots, nbAnts\}$ do
9          if $\forall l \in \{1, \dots, nbAnts\}, \mathcal{A}_k$ is at least as good as $\mathcal{A}_l$ then
10             foreach $< x_i, v_i > \in \mathcal{A}_k$ do
11                 $\tau(x_i, v_i) \leftarrow min(\tau_{max}, \tau(x_i, v_i) + \frac{1}{1+|F(\mathcal{A}_k) - F(\mathcal{A}_{best})|})$
12     if $\mathcal{A}_{best}$ is strictly better than all feasible solutions of $\{\mathcal{A}_1, ..., \mathcal{A}_{nbAnts}\}$ then
13         foreach $< x_i, v_i > \in \mathcal{A}_{best}$ do $\tau(x_i, v_i) \leftarrow min(\tau_{max}, \tau(x_i, v_i) + 1)$
14 until time spent $\geq t_{max1}$ or number of cycles without improvement of
   $\mathcal{A}_{best} \geq it_{max}$ or average distance of $\{\mathcal{A}_1, ..., \mathcal{A}_{nbAnts}\} \leq d_{min}$ ;
15 return $\mathcal{A}_{best}$ and $\tau$
```

contain the best feasible solutions with respect to the objective function. This pheromone structure associates a pheromone trail $\tau(x_i, v_i)$ with each variable $x_i \in X$ and each value $v_i \in D(x_i)$. Each pheromone trail is bounded between two given bounds τ_{min} and τ_{max}, and is initialized to τ_{max} (line 1) as proposed in [10]. At the end of the first phase, the pheromone structure τ is returned so that it can be used in the second phase as a value ordering heuristic.

Construction of assignments: At each cycle (lines 2-14), each ant calls CP Optimizer in order to construct a feasible solution (line 4). Note that during this first phase, we do not ask CP Optimizer to optimize the objective function, but simply to find feasible solutions that satisfy all the constraints.

CP Optimizer is used as a black-box with its default search parameters and each new call corresponds to a restart. In particular, the variable ordering heuristic is based on the variable impact heuristic of [9], i.e., at each step of the search tree, CP Optimizer chooses the variable with the highest impact. CP Optimizer propagates constraints using predefined procedures, and when an inconsistency is detected, it backtracks until finding a feasible solution that satisfies all constraints. Also, if the given cutoff in the number of backtracks is met without having found a solution, CP Optimizer automatically restarts the search, as described in [9].

However, the value ordering heuristic procedure is given to CP Optimizer and it is defined according to ACO: let x_i be the next variable to be assigned; v_i is randomly chosen in $D(x_i)$ w.r.t. probability

$$p(v_i) = \frac{[\tau(x_i, v_i)]^\alpha \cdot [1/\mathit{impact}(v_i)]^\beta}{\sum_{v_j \in D(x_i)} [\tau(x_i, v_j)]^\alpha \cdot [1/\mathit{impact}(v_j)]^\beta}$$

where $\mathit{impact}(v_i)$ is the observed impact of value v_i as defined in [9], and α and β are two parameters that weight the pheromone and impact factors respectively. Hence, during the first cycle, values are randomly chosen with respect to impacts only as all pheromone trails are initialized to the same value (i.e., τ_{max}). However, at the end of each cycle, pheromone trails are updated so that these probabilities are progressively biased with respect to past constructions.

It is worth mentioning here that our CPO-ACO framework is designed to solve underconstrained COPs that have a rather large number of feasible solutions (such as, for example, MKPs or QAPs): when solving these problems, the difficulty is not to build a feasible solution, but to find the feasible solution that optimizes the objective function. Hence, on these problems CP Optimizer is able to build feasible solutions very quickly, with very few backtracks. Our CPO-ACO framework may be used to solve more tightly constrained COPs. However, for these problems, CP Optimizer may backtrack a lot (and therefore need more CPU time) to compute each feasible solution. In this case, pheromone learning will be based on a very small set of feasible solutions so that it may not be very useful and CPO-ACO will simply behave like CP Optimizer.

Pheromone evaporation: Once every ant has constructed an assignment, pheromone trails are evaporated by multiplying them by $(1-\rho)$ where $\rho \in [0;1]$ is the pheromone evaporation rate (lines 5-6).

Pheromone laying step: At the end of each cycle, good feasible solutions (with respect to the objective function) are rewarded in order to intensify the search around them. Lines 8-11, the best feasible solutions of the cycle are rewarded. Lines 12-13, the best feasible solution built so far is rewarded if it is better than the best feasible solutions of the cycle (otherwise it is not rewarded as it belongs to the best feasible solutions of the cycle that have already been rewarded). In both cases, a feasible solution $\mathcal{A}$ is rewarded by increasing the quantity of pheromone laying on every couple $< x_i, v_i >$ of $\mathcal{A}$, thus increasing the probability of assigning x_i to v_i. The quantity of pheromone added is inversely proportional to the gap between $F(\mathcal{A})$ and $F(\mathcal{A}_{best})$.

Termination conditions: The first phase is stopped either if the CPU time limit of the first phase t_{max1} has been reached, or if $\mathcal{A}_{best}$ has not been improved since it_{max} iterations, or if the average distance between the assignments computed during the last cycle is smaller than d_{min}, thus indicating that pheromone trails have allowed the search to converge. We define the distance between two assignments with respect to the number of variable/value couples they share, i.e., the distance between $\mathcal{A}_1$ and $\mathcal{A}_2$ is $\frac{|X|-|\mathcal{A}_1 \cap \mathcal{A}_2|}{|X|}$

3.2 Second Phase of $CPO - ACO$

At the end of the first phase, the best constructed feasible solution $\mathcal{A}_{best}$ and the pheromone structure τ are forwarded to the second phase. $\mathcal{A}_{best}$ is used to bound the objective function with its cost. Then, we ask CP Optimizer to find a feasible solution that optimizes the objective function F: in this second phase, each time CP Optimizer finds a better feasible solution, it adds a constraint to bound the objective function with respect to its cost, and it backtracks to find better feasible solutions, or prove the optimality of the last computed bound.

Like in the first phase, CP Optimizer is used as a black-box with its default search parameters: the restart of [9] is used as the search type parameter, and impacts are used as variable ordering heuristic. However, the value ordering heuristic is defined by the pheromone structure τ: given a variable x_i to be assigned, CP Optimizer chooses the value $v_i \in D(x_i)$ which maximizes the formula: $[\tau(x_i, v_i)]^{\alpha} \cdot [1/impact(v_i)]^{\beta}$.

Note that we have experimentally compared different other frameworks which are listed below:

– We have considered a framework where, at the end of the first phase, we only return $\mathcal{A}_{best}$ (which is used to bound the objective function at the beginning of the second phase) and we do not return the pheromone structure (so that the value ordering heuristic used in the second phase is only defined with respect to impacts). This framework obtains significantly worse results, showing us that the pheromone structure is a valuable ordering heuristic.

- We have considered a framework where, during the first phase, the sampling is done randomly, without using pheromone for biasing probabilities (i.e., α is set to 0). This framework also obtains significantly worse results, showing us that it is worth using an ACO learning mechanism.
- We have considered a framework where, during the second phase, the value ordering heuristic is defined by the probabilistic rule used in the first phase, instead of selecting the value that maximizes the formula. This framework obtains results that are not significantly different on most instances.

4 Experimental Evaluation of CPO-ACO

4.1 Considered Problems

We evaluate our CPO-ACO approach on three well known COPs, i.e., Multidimensional Knapsack problem (MKP), Quadratic Assignment Problem (QAP) and the Maximum Independent Set (MIS).

The Multidimensional Knapsack problem (MKP) involves selecting a subset of objects in a knapsack so that the capacity of the knapsack is not exceeded and the profit of the selected objects is maximized. The associated CP model is such that

- $X = \{x_1, \ldots, x_n\}$ associates a decision variable x_i with every object i;
- $\forall x_i \in X$, $D(x_i) = \{0, 1\}$ so that $x_i = 0$ if i is not selected, and 1 otherwise;
- $C = \{C_1, \ldots, C_m\}$ is a set of m capacity constraints such that each constraint $C_j \in C$ is of the form $\sum_{i=1}^{n} c_{ij} \cdot x_i \leq r_j$ where c_{ij} is the amount of resource j required by object i and r_j is the available amount of resource j;
- the objective function to maximize is $F = \sum_{i=1}^{n} u_i \cdot x_i$ where u_i is the profit associated with object i.

We have considered academic instances with 100 objects which are available at http://people.brunel.ac.uk/˜mastjjb/jeb/orlib/mknapinfo.html. We have considered the first 20 instances with 5 resource constraints (5-100-00 to 5-100-19); the first 20 instances with 10 resource constraints (10-100-00 to 10-100-19) and the first 20 instances with 30 resource constraints (30-100-00 to 30-100-19).

The Quadratic Assignment Problem (QAP) involves assigning facilities to locations so that a sum of products between facility flows and location distances is minimized. The associated CP model is such that

- $X = \{x_1, \ldots, x_n\}$ associates a decision variable x_i with every facility i;
- $\forall x_i \in X$, $D(x_i) = \{1, \ldots, n\}$ so that $x_i = j$ if facility i is assigned to location j;
- C only contains a global all different constraint among the whole set of variables, thus ensuring that every facility is assigned to a different location;
- the objective function to minimize is $F = \sum_{i=1}^{n} \sum_{j=1}^{n} a_{x_i x_j} b_{ij}$ where $a_{x_i x_j}$ is the distance between locations x_i and x_j, and b_{ij} is the flow between facilities i and j.

We have considered instances of the QAPLIB which are available at
http://www.opt.math.tu-graz.ac.at/qaplib/inst.html.

The Maximum Independent Set (MIS) involves selecting the largest subset of vertices of a graph such that no two selected vertices are connected by an edge (this problem is equivalent to searching for a maximum clique in the inverse graph). The associated CP model is such that

- $X = \{x_1, \ldots, x_n\}$ associates a decision variable x_i with every vertex i;
- $\forall x_i \in X$, $D(x_i) = \{0, 1\}$ so that $x_i = 0$ if vertex i is not selected, and 1 otherwise;
- C associates a binary constraint c_{ij} with every edge (i, j) of the graph. This constraint ensures that i and j have not been both selected, i.e., $c_{ij} = (x_i + x_j < 2)$.
- the objective function to maximize is $F = \sum_{i=1}^{n} x_i$.

We have considered instances of the MIS problem which are available at http://www.nlsde.buaa.edu.cn/∼kexu/benchmarks/graph-benchmarks.htm.

4.2 Experimental Settings

For each problem, the CP model has been written in C++ using the CP Optimizer modeling API. It is worth mentionning here that this CP model is straightforwardly derived from the definition of the problem as given in the previous section: it basically declares the variables together with their domains, the constraints, and the objective function.

We compare CPO-ACO with CP Optimizer (denoted CPO). In both cases, we used the version V2.3 of CP Optimizer with its default settings. However, for CPO-ACO, the value ordering heuristic is given to CPO (see Section 3). For CPO, the default value ordering heuristic is defined with respect to impacts as proposed in [9].

For all experiments, the total CPU time has been limited to 300 seconds on a 2.2 Gz Pentium 4. For CPO-ACO, this total time is shared between the two phases as follows: $t_{max1} = 25\%$ of the total time (so that phases 1 cannot spend more than 75s); $d_{min} = 0.05$ (so that phase 1 is stopped as soon as the average distance is smaller than 5%); and $it_{max} = 500$ (so that phase 1 is stopped if $\mathcal{A}_{best}$ has not been improved since 500 cycles). The number of ants is $nbAnts = 20$; pheromone and impact factors are respectively weighted by $\alpha = 1$ and $\beta = 2$; and pheromone trails are bounded between $\tau_{min} = 0.01$ and $\tau_{max} = 1$.

However, we have not considered the same pheromone evaporation rate for all experiments. Indeed, both MKP and MIS have $0 - 1$ variables so that, at each cycle, one value over the two possible ones is rewarded, and ACO converges rather quickly. For the QAP, all domains contain n values (where n is equal to the number of variables) so that, at each cycle, one value over the n possible ones is rewarded. In this case, we speed-up the convergence of ACO by increasing the evaporation rate. Hence, $\rho = 0.01$ for MKP and MIS whereas $\rho = 0.1$ for QAP.

Note that we have not yet extensively studied the influence of parameters on the solution process so that the parameter setting considered here is probably not optimal. Further work will include the use of a racing algorithm based on statistical tests in order to automatically tune parameters [15].

For both CPO and CPO-ACO, we performed 30 runs per problem instance with a different random-seed for each run.

4.3 Experimental Results

Table 1 gives experimental results obtained by CPO and CPO-ACO on the MKP, the QAP and the MIS. For each class of instances and each approach, this table gives the percentage of deviation from the best known solution. Let

Table 1. Comparison of CPO and CPO-ACO on the MKP, the QAP and the MIS. Each line successively gives: the name of the class, the number of instances in the class ($\#I$), the average number of variables in these instances ($\#X$), the results obtained by CPO (resp. CPO-ACO), i.e., the percentage of deviation from the best known solution (average (avg) and standard deviation (sd)), the percentage of instances for which CPO (resp. CPO-ACO) has obtained strictly better average results ($>_{avg}$), and the percentage of instances for which CPO (resp. CPO-ACO) is significantly better w.r.t. the statistical test.

Results for the MKP

			CPO			*CPO – ACO*		
Name	# I	# X	avg (sd)	$>_{avg}$	$>_{t-test}$	avg (sd)	$>_{avg}$	$>_{t-test}$
5.100-*	20	100	1.20 (0.30)	0%	0%	**0.46** (0.23)	**100%**	**100%**
10.100-*	20	100	1.53 (0.31)	0%	0%	**0.83** (0.34)	**100%**	**100%**
30.100-*	20	100	1.24 (0.06)	5%	0%	**0.86** (0.08)	**95%**	**85%**

Results for the QAP

			CPO			*CPO – ACO*		
Name	# I	# X	avg (sd)	$>_{avg}$	$>_{t-test}$	avg (sd)	$>_{avg}$	$>_{t-test}$
bur*	7	26	1.17 (0.43)	0%	0%	**0.88** (0.43)	**100%**	**57 %**
chr*	11	19	12.11 (6.81)	45%	9%	**10.99** (6.01)	**55 %**	**45 %**
had*	5	16	1.07 (0.89)	0%	0%	**0.54** (1.14)	**100%**	**60 %**
kra*	2	30	17.46 (3.00)	0%	0%	**14.99** (2.79)	**100%**	**100%**
lipa*	6	37	22.11 (0.82)	0%	0%	**20.87** (0.75)	**100%**	**100%**
nug*	15	20	8.03 (1.59)	7%	0%	**5.95** (1.44)	**93 %**	**80 %**
rou*	3	16	5.33 (1.15)	33%	0%	**3.98** (1.00)	**67 %**	**67 %**
scr*	3	16	**4.60** (2.4)	33%	0%	5.12 (2.60)	**67 %**	0 %
tai*	4	16	6.06 (1.35)	25%	25%	**4.84** (1.25)	**75 %**	**50 %**

Results for the MIS

			CPO			*CPO – ACO*		
Name	# I	# X	avg (sd)	$>_{avg}$	$>_{t-test}$	avg (sd)	$>_{avg}$	$>_{t-test}$
frb-30-15-*	5	450	9.83 (1.86)	0%	0%	**9.46** (2.00)	**80%**	**20%**
frb-35-17-*	5	595	**11.62** (2.05)	**60%**	0%	11.82 (2.31)	40%	0%
frb-40-19-*	5	760	13.47 (1.92)	20%	0%	**12.85** (2.22)	**80%**	**20%**
frb-45-21-*	5	945	15.40 (2.43)	0%	0%	**14.35** (1.82)	**100%**	**80%**
frb-50-23-*	5	1150	16.24 (2.32)	20%	0%	**15.84** (2.00)	**80%**	**20%**
frb-53-24-*	5	1272	18.15 (2.55)	0%	0%	**16.86** (1.84)	**100%**	**80%**
frb-56-25-*	5	1400	17.85 (2.37)	20%	0%	**16.89** (1.08)	**80%**	**40%**
frb-59-26-*	5	1534	18.40 (2.44)	40%	0%	**18.37** (2.16)	**60%**	**20%**

us first note that CPO and CPO-ACO (nearly) never reach the best known solution: indeed, best known solutions have usually been computed with state-of-the-art dedicated approaches. Both CPO and CPO-ACO are completely generic approaches that do not aim at competing with these dedicated approaches which have often required a lot of programming and tuning work. Also, we have chosen a reasonable CPU time limit (300 seconds) in order to allow us to perform a significant number of runs per instance, thus allowing us to use statistical tests. Within this rather short time limit, CPO-ACO obtains competitive results with dedicated approaches on the MKP (less than 1% of deviation from best known solutions); however, it is rather far from best known solutions on many instances of the QAP and the MIS.

Let us now compare CPO with CPO-ACO. Table 1 shows us that using ACO to guide CPO search improves the search process on all classes except two. However, this improvement is more important for the MKP than for the two other problems. As the two approaches have obtained rather close results on some instances, we have used statistical tests to determine if the results are significantly different or not: we have performed the Student's t-test with significance level of 0.05, using the R Stats Package available at http://sekhon.berkeley.edu/doc-/html/index.html. For each class, we report the percentage of instances for which an approach has obtained significantly better results than the other one (column $>_{t-test}$ of table 1). For the MKP, CPO-ACO is significantly better than CPO for 57 instances, whereas it is not significantly different for 3 instances. For the QAP, CPO-ACO is significantly better than CPO on a large number of instances. However, CPO is better than CPO-ACO on one instance of the class tai* of the QAP. For the MIS, CPO-ACO is significantly better than CPO on 35% of instances, but it is not significantly different on all other instances.

5 Conclusion

We have proposed CPO-ACO, a generic approach for solving COPs defined by means of a set of constraints and an objective function. This generic approach combines a complete B&P&B approach with ACO. One of the main ideas behind this combination is the utilization of the effectiveness of (i) ACO to explore the search space and quickly identify promising areas (ii) CP Optimizer to strongly exploit the neighborhood of the best solutions found by ACO. This combination allows us to reach a good balance between diversification and intensification of the search: diversification is mainly ensured during the first phase by ACO; intensification is ensured by CP optimizer during the second phase.

It is worth noting that thanks to the modular nature of IBM ILOG CP Optimizer that clearly separates the modeling part of the problem from its resolution part, the proposed combination of ACO and CP was made in natural way. Hence, the CPO-ACO program used was exactly the same for the experiments on the different problems used in this work.

We have shown through experiments on three different COPs that CPO-ACO is significantly better than CP Optimizer.

5.1 Related Works

Recent research has focused on the integration of ACO in classical branch and bound algorithms, but most of them were applied on specific problems and/or proposed a combination based on an incomplete search.

In particular, B.Meyer has proposed in [16] two hybrid algorithms where the metaheuristic Ant Colony Optimization (ACO) was coupled with CP. In his work, Meyer has proposed a loose coupling where both components run in parallel, exchanging only (partial) solutions and bounds. Then, he has proposed a tight coupling where both components collaborate in an interleaved fashion so that, the constraint propagation was embedded in ACO in order to allow an ant to backtrack when an association of a value v with a given variable fails. However, the backtrack procedure was limited at the level of the last chosen variable. This means that, if all the possible values of the last chosen variable have been tried without success, the search of an ant ends with failure. The results of this work show on the machine scheduling problem with sequence-dependent setup time that the tight coupling is better. But unfortunately, the proposed tight coupling is not based on a complete search and in the both proposed algorithms; the author has assumed that the variable ordering is fixed.

Also, we have proposed in [12] an hybrid approach, denoted Ant-CP, which combines ACO with a CP solver in order to solve constraint satisfaction problems (without objective function to optimize). Like CPO-ACO, Ant-CP uses the CP modeling language to define the problem, and ants use predefined CP procedures to check and propagate constraints. However, unlike CPO-ACO, Ant-CP performs an incomplete search which never backtracks.

5.2 Further Work

There are several points which are worth mentioning as further improvements to CPO-ACO. At the moment, CPO-ACO works well (if we compare it with CP Optimizer) on the problems for which finding a feasible solution is relatively easy. In this paper, we have applied CPO-ACO on three different problems without using problem-dependent heuristics. We plan to study the interest of adding problem-dependent heuristics, that may improve the efficiency of CPO-ACO and allow it to become competitve with state-of-the-art approches.

Parameter tuning is another interesting topic. For the moment the parameters of CPO-ACO are roughly tuned using our experience, but we believe that an adaptive version which dynamically tunes the parameters during the execution should significantly increase the algorithm's efficiency and robustness.

Acknowledgments

The authors thank Renaud Dumeur, Jérôme Rogerie, Philippe Refalo and Philippe Laborie for the fruitful and animated discussions about combinatorial optimization in general case and, particularly, the search strategies. Also, we thank Renaud Dumeur and Paul Shaw for proofreading drafts of this paper.

References

1. Nemhauser, G., Wolsey, A.: Integer and combinatorial optimization. Jhon Wiley & Sons, New York (1988)
2. Papadimitriou, C., Steiglitz, K.: Combinatorial optimization–Algorithms and complexity. Dover, New York (1982)
3. Kirkpatrick, S., Gellat, C., Vecchi, M.: Optimization by simulated annealing. Science 220, 671–680 (1983)
4. Glover, F., Laguna, M.: Tabu Search. Kluwer Academic Publishers, Dordrecht (1997)
5. Lourenço, H., Martin, O., Stützle, T.: Iterated local search. In: Golver, F., Kochenberger, G. (eds.) Handbook of Metaheuristics. International Series in Operations Research Management Science, vol. 57, pp. 321–353. Kluwer Academic Publisher, Dordrecht (2001)
6. Hensen, P., Mladenović, N.: An introduction to variable neighborhood search. In: Voβ, S., Martello, S., Osman, I., Roucairol, C. (eds.) Meta-heuristics: advances and trends in local search paradigms for optimization, pp. 433–438. Kluwer Academic Publisher, Dordrecht (1999)
7. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. MIT Press, Cambridge (2005)
8. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press, Cambridge (2004)
9. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)
10. Stützle, T., Hoos, H.: $\mathcal{MAX} - \mathcal{MIN}$ Ant System. Journal of Future Generation Computer Systems 16, 889–914 (2000)
11. Solnon, C.: Ants can solve constraint satisfaction problems. IEEE Transactions on Evolutionary Computation 6(4), 347–357 (2002)
12. Khichane, M., Albert, P., Solnon, C.: Integration of ACO in a constraint programming language. In: Dorigo, M., Birattari, M., Blum, C., Clerc, M., Stützle, T., Winfield, A.F.T. (eds.) ANTS 2008. LNCS, vol. 5217, pp. 84–95. Springer, Heidelberg (2008)
13. Solnon, C., Fenet, S.: A study of ACO capabilities for solving the Maximum Clique Problem. Journal of Heuristics 12(3), 155–180 (2006)
14. Alaya, I., Solnon, C., Ghedira, K.: Ant Colony Optimization for Multi-objective Optimization Problems. In: 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), pp. 450–457. IEEE Computer Society, Los Alamitos (2007)
15. Birattari, M., Stutzle, T., Paquete, L., Varrentrapp, K.: A Racing Algorithm for Configuring Metaheuristics. In: GECCO, pp. 11–18 (2002)
16. Meyer, B.: Hybrids of constructive meta-heuristics and constraint programming: A case study with ACO. In: Blesa, M.J., Blum, C., Cotta, C., Fernández, A.J., Gallardo, J.E., Roli, A., Sampels, M. (eds.) HM 2008. LNCS, vol. 5296. Springer, Heidelberg (2008)

Service-Oriented Volunteer Computing for Massively Parallel Constraint Solving Using Portfolios

Zeynep Kiziltan and Jacopo Mauro

Department of Computer Science, University of Bologna, Italy
{zeynep,jmauro}@cs.unibo.it

1 Introduction

Recent years have witnessed growing interest in parallelising constraint solving based on tree search (see [1] for a brief overview). One approach is search-space splitting in which different parts of the tree are explored in parallel (e.g. [2]). Another approach is the use of algorithm portfolios. This technique exploits the significant variety in performance observed between different algorithms and combines them in a portfolio [3]. In constraint solving, an algorithm can be a solver or a tuning of a solver. Portfolios have often been run in an interleaving fashion (e.g. [4]). Their use in a parallel context is more recent ([5], [1]).

Considering the complexity of the constraint problems and thus the computational power needed to tackle them, it is appealing to benefit from large-scale parallelism and push for a massive number of CPUs. Bordeaux et. al have investigated this in [1] . By using the portfolio and search-space splitting strategies, they have conducted experiments on constraint problems using a parallel computer with the number of processors up to 128. They reported that the parallel portfolio approach scales very well in SAT, in the sense that utilizing more processors consistently helps solving more instances in a fixed amount of time.

As done also in [1], most of the prior work in parallel constraint solving assumes a parallel computer with multiple CPUs. This architecture is fairly reliable and has low communication overhead. However, such a computer is costly and is not always at our disposal, especially if we want to push for massive parallelism. Jaffar et al addressed this problem in [2] by using a bunch of computers in a network (called "volunteer computing" in what follows). They employed 61 computers in a search-space splitting approach and showed that such a method is effectively scalable in ILP.

In this paper, we combine the benefits of [1] and [2] when solving constraint satisfaction problems (CSPs). We present an architecture in which massive number of volunteer computers can run several (tunings of) constraint solvers in parallel in a portfolio approach so as to solve many CSPs in a fixed amount of time. The architecture is implemented using the service-oriented computing paradigm and is thus modular, flexible for useful extensions and allows to utilise even the computers behind a firewall. We report experiments up until 100 computers. As the results confirm, the architecture is effectively scalable.

2 Service-Oriented Volunteer Computing

Volunteer computing is a type of distributed computing which brings together the computational resources that are often idle and available in a network (e.g. `distributed.net`). As it offers a cost effective and large computing capability, volunteer computing seems to be a good candidate for the basis of a massive parallel constraint solving architecture using portfolios.

Service-oriented computing (SoC) is an emerging paradigm in which services are autonomous computational entities that can be composed to obtain more complex services for developing massively distributed applications (see e.g. the Sensoria Project `http://www.sensoria-ist.eu/`). In the context of volunteer computing, a service can be for instance the functionality which distributes jobs to the computers. Our architecture is designed and implemented in Jolie (`http://www.jolie-lang.org/`) which is the first full-fledged programming language based on SoC paradigm. The reasons behind the choice of SoC and thus Jolie can be summarised as follows. First, it is scalable; massive number of communications with different computers can easily be handled. Second, it is modular; new services can easily be integrated and organised in a hierarcy. This is particulary important in an architecture like ours which has several sub services. Third, it allows us to deploy the framework in a number of different ways. Jolie provides interaction between heterogenous services, like in the case of web services (e.g integrating a google map application in a hotel-search application). We can therefore easily interact with other services (even graphical ones) in the future and make our architecture be part of a more complex system.

3 Our Architecture

Fig. 1 depicts our architecture using a notation similar to UML communication diagrams. When services are used, we can have two kinds of messages: one way message denoted by the string ⟨ message name ⟩(⟨ data sent ⟩) and a request response message denoted by ⟨ message name ⟩ (⟨ data sent ⟩)(⟨ data received ⟩). The figure is read as follows. The user utilises the *redirecting service* to get the location of the *preprocessing service* and then sends to the preprocessing service a problem instance i_k to be solved. Once i_k is sent, the preprocessing service contacts the *CBR service* which runs a case-based reasoning system to provide the expected solving time t_k of i_k. The preprocessing server then sends t_k and i_k to the *instance distributor service*. This service is responsible for scheduling the instances for different (tunings of) solvers and assigning the related jobs to the volunteer computers. This can be done in a more intelligent way thanks to t_k provided by the CBR service. This value can be used for instance to minimize the average solving time. Finally, the *volunteer service* asks the redirecting service the location of the instance distributor service and then requests a job from it using a request response message. This is the only way a job can be sent to the volunteer service. Note that the use of the redirecting service makes it possible to have multiple preprocessing and instance distributor services in the future.

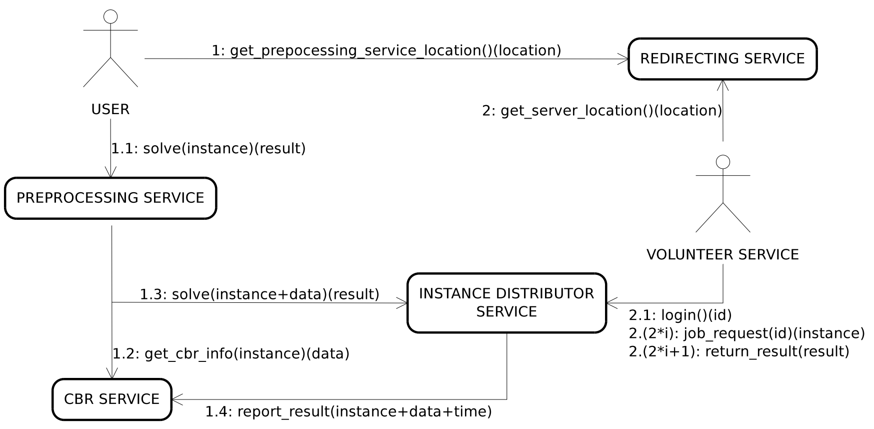

USER
1: get_prepocessing_service_location()(location)
REDIRECTING SERVICE
2: get_server_location()(location)
1.1: solve(instance)(result)
PREPROCESSING SERVICE
VOLUNTEER SERVICE
1.3: solve(instance+data)(result)
INSTANCE DISTRIBUTOR SERVICE
2.1: login()(id)
2.(2*i): job_request(id)(instance)
2.(2*i+1): return_result(result)
1.2: get_cbr_info(instance)(data)
CBR SERVICE
1.4: report_result(instance+data+time)

Fig. 1. Architecture

An input instance of the architecture is specified in XCSP which is a relatively new format to represent constraint networks using XML (`http://www.cril.univ-artois.fr/CPAI08/XCSP2_1.pdf`). The reason of this choice is that (i) XCSP format has been used in the last constraint solver competitions and thus many solvers have started to support it; (ii) such a low level representation is useful to extract the feature vectors needed by a CBR algorithm.

4 Preliminary Experimental Results

In these preliminary experiments, our concern is the scalability. We thus currently exclude the CBR service and observe how the architecture scales as the number of computers increases. In the experiments, Dell Optiplex computers of our labs running Linux with Intel core 2 duo and Pentium 4 processors are used. Up to 100 of them are employed for the volunteer service and only one for the remaining services. We consider the instances of the 2009 CSP Solver Competition (`http://www.cril.univ-artois.fr/CPAI09/`), six of its participating solvers (Abscon 112v4 AC, Abscon 112v4 ESAC, Choco2.1.1 2009-06-10, Choco2.1.1b 2009-07-16, Mistral 1.545, SAT4J CSP 2.1.1) and one solver (bpsolver 2008-06-27) from the 2008 competition (`http://www.cril.univ-artois.fr/CPAI08/`). These solvers are provided as black-box, their tunings is not possible. Hence, an instance is solved by 7 solvers on 7 different computers. The experiments focus on the following instances: (i) Easy SAT: 1607 satisfiable instances solved in less than 1 minute; (ii) Easy UNSAT: 1048 unsatisfiable instances solved in less than 1 minute; (iii) Hard SAT: 207 satisfiable instances solved in between 1 and 30 minutes; (iv) Hard UNSAT: 106 unsatisfiable instances solved in between 1 and 30 minutes. Such times refer to the best solving times of the competition.

In Table 1, we present the number of instances solved in 30 minutes for the easy instances and in 1 hour for the hard instances. As an experiment is affected by the current work load of the computers, we perform and report three

Table 1. Experimental results

n°	Easy SAT (30 min)			Easy UNSAT (30 min)			Hard SAT (1h)			Hard UNSAT (1h)		
20	15	14	15	17	18	18	3	3	6	7	7	9
40	132	128	135	150	150	150	8	8	7	16	17	13
60	141	140	140	320	318	322	19	15	14	23	23	22
80	144	145	151	335	323	328	25	21	25	29	30	30
100	179	179	192	336	345	334	25	25	25	44	33	36

runs. The results are promising. Even without the CBR service and the different tunings of solvers, the number of the instances solved in a fixed amount of time increases as the number of computers increases, no matter how busy the volunteer computers are. Note that only one computer is used to run the preprocessing and the instance distributor services, and yet the system can handle 100 computers without any problems. The main reason for not always obtaining a linear speed up is that some solvers cannot solve even the easy instances in less then 30 minutes. Hence, many computers are spending more than 30 minutes for solving an already solved instance. This has happened 104 times in the tests of easy SAT instances with 100 volunteer computers. In the same tests, we as well encountered 35 solver failures. These observations suggest we shall allow the interruption of a computation if the related instance is already solved.

5 Related Work

There is considerable amount of prior work on parallel constraint solving. We here discuss only those that use massive parallelism. Our work is similar to the one described in [1] in the sense that we too use the portfolio approach. However, there are a number of differences. First, we consider CSP instances and several different constraint solvers (including SAT and CP solvers), as opposed to SAT instances and one SAT solver. Second, we create portfolios by running each instance on several computers and several (tunings of) solvers at the same time. The solver-independent approach of [1] instead uses only different tunings of the same solver. Third, whilst we assume a group of independent computers available in a network, [1] assumes a dedicated cluster of computers.

Jaffar et al [2] as well propose an architecture based on volunteer computing. Unlike ours, this architecture uses the search-space splitting strategy and the experiments confirm scalability on ILP instances using 61 computers. There are however other substantial differences. In many environments like laboratories and home networks, computers stay behind a firewall or network address translation (NAT) which limit their access from outside. We are able to access such computers by using only the request response messages instead of using direct messages as done in [2]. This choice brings further advantages over [2] like smaller number of messages sent, the applicability to the majority of networks, and having only the server as a possible bottleneck. The price to pay is the impossibility of using certain protocols to create a tree of volunteer computers. This is however not needed in our architecture. Our volunteer computers never communicate with each other so as to avoid potential scalability problems. This is not the case in [2]. As the number of computers increases, the overhead of distribution becomes too high which can lead to significant slowdown.

Our architecture owes a lot to CPHYDRA [4] , the winner of 2008 CSP Solver Competition . It combines many CP solvers in a portfolio. CPHYDRA determines via CBR the subset of the solvers to use in an interleaved fashion and the time to allocate for each solver, given a CSP instance. Our work can thus be seen as the parallel version of CPHYDRA which eliminates the need of interleaving, giving the possibility of running several (tunings of) solvers at the same time. This brings the chance of minimising the expected solving time as there is no order among the solvers. Moreover, parallelism gives the opportunity of updating the base case of CBR even in a competition environment.

6 Conclusions and Future Work

We have presented an architecture in which massive number of volunteer computers can run several (tunings of) constraint solvers in parallel in a portfolio approach. The architecture is implemented in SoC which is becoming the choice of paradigm for the development of scalable and massively distributed systems. The initial experimental results confirm the scalibility. Our plans for future are to make the architecture more efficient, useful and realiable. As for efficiency, we are currently working on the CBR service and investigating how to best benefit from similar cases in a parallel context. As for usability, we aim at tackling two limitations. First, our architecture gets XCSP instance format which is too low level for a CP user. The good news is that the architecture can easily be integrated to a high level modelling & solving platform such as Numberjack (`http://4c110.ucc.ie/numberjack`) which will soon output to XCSP. In this way, we can obtain a system in which the user states her problem easily at a high-level of abstraction, then the problem gets converted to XCSP and then (CBR-based) parallel solver is invoked. Second, our architecture is focused on the portfolio approach. We intend to investigate how to exploit massive number of computers in search-space splitting when solving optimisation problems in CP. Finally, should the computers go off or malfunction, we might want to replicate the jobs assigned to the computers or redirect them to other computers. We will study methods to make the architecture more reliable from this perspective.

References

1. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 443–448 (2009)
2. Jaffar, J., Santosa, A.E., Yap, R.H.C., Zhu, K.Q.: Scalable distributed depth-first search with greedy work stealing. In: Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), pp. 98–103 (2004)

3. Gomes, C., Selman, B.: Algorithm portfolios. Artificial Intelligence 1-2, 43–62 (2001)
4. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Proceedings of the 19th Irish Conference on Artificial Intelligence, AICS 2008 (2008)
5. Hamadi, Y., Jabbour, S., Sais, L.: Manysat: Solver description. In: SAT-Race 2008 (2008)

Constraint Programming with Arbitrarily Large Integer Variables

Anna Moss

Intel Corporation, Haifa, Israel
anna.moss@intel.com

Abstract. In the standard Constraint Programming (CP) framework, an integer variable represents a signed integer and its domain is bounded by some minimal and maximal integer type values. In existing CP tools, the integer type is used to represent domain values, and hence domain bounds are inherently limited by the minimal and maximal signed integer values representable on a given platform. However, this implementation of integer variable fails to satisfy use cases where modeled integers can be arbitrarily large. An example of such CP application is the functional test generation where integer variables are used to model large architectural fields like memory addresses or operand data. In addition, in such applications, the set of standard arithmetic operations on an integer variable provided by the traditional CP framework does not represent the whole range of operations required for modeling. In this paper, we define a new type of integer variables with arbitrarily large domain size and a modified operation set. We show how this variable type can be realized on top of a traditional CP framework by means of global constraints over standard integer variables. The ideas presented in this paper can also be used to implement a native variable of the introduced type in a CP tool. The paper provides experimental results to demonstrate the effectiveness of the proposed approach.

1 Introduction

One of the basic components in constraint modeling is the integer constraint variable, which is used to represent signed integers with a finite domain. The existing CP tools use some programming language integer types, e.g. signed 32 or 64 bit integers, to represent domain values of an integer variable. In this approach, upper and lower bounds on integer variable domains are inherently limited by the maximal and minimal values that can be represented by the corresponding integer type. This limitation becomes in fact an implicit constraint on integer variables, reducing the expressive capabilities of integer modeling.

The limitation above does not impair the modeling in most combinatorial problems that constitute the conventional application domain of CP technology, since the integer types typically suffice to represent values in this kind of problems. However, in recent years the application domain of CP technology has expanded to additional areas. In particular, new CP applications emerged where integers modeled by integer variables can be arbitrarily large. An example of such application is the functional test generation [1], [2], [3], [4]. Functional tests are required in the task of simulation based approach to hardware design verification. In this approach, the design under test (DUT) is examined against a large amount of functional tests, which exercise numerous execution scenarios in order to expose potential bugs. The work of validation engineers on developing test suites is facilitated by the use of automated test generation tools. CP technology provides powerful means in performing automated functional test generation. In particular, constraint modeling provides the capability to declaratively describe the DUT specification that defines a valid test as well as to describe a specific test scenario. Moreover, advanced CP algorithms can be used by automated test generation tools to produce tests that answer the architecture and scenario requirements.

In the application described above, integer variables are used to model hardware architectural components of large bit width, e.g. memory addresses or operand data. The domain size of corresponding variables cannot be accommodated by common integer types. Moreover, as hardware designs become more complex, the sizes of such architectural components keep growing. This poses a requirement for integer variables of arbitrarily large domain size.

The CP application to functional test generation also poses additional requirements on operations supported by integer variables in CP modeling. In the examples above, integers are viewed as unsigned machine words of a fixed bit width. The common operations performed on this kind of integers are not confined to the set of standard arithmetic operations provided in the traditional CP framework. They include, for example, addition/subtraction modulo the bit width, left/right shift, and other operations that can be performed on machine words by a processor.

To the best of our knowledge, very little attempts have been made to extend the traditional CP framework to accommodate integer variables of arbitrarily large size. The only example we are aware of comes from the domain of Constraint Logic Programming (CLP) [5]. This work reports the implementation of a Prolog based solver supporting integer variables of arbitrarily large size. However, neither theoretical results describing the implemented algorithms nor experimental results demonstrating the performance have been published. Aside from the example above, the problem of domain size limitation for integer variables has never been addressed in the CP literature and none of the existing CP tools supports integer variables with arbitrarily large domain size. Such limitation impairs the applicability of the CP technology to a range of problems where modeling of large integers is required. For example, in tools for automated test generation, the problem of large integer variables and extended arithmetic operation set is typically addressed by developing specialized solvers, e.g. [6]. Such approach is not generic, incurs a large development cost and makes it hard to take advantage of the cutting edge CP technology. This paper comes to address this problem and extends the traditional CP framework by introducing a new CP modeling means, namely, an integer variable supporting arbitrarily large domain size as well as an extended operation set. We refer to this new version of integer variable as LiVar (standing for Large Integer VARiable). We show how LiVar abstraction can be realized within a traditional CP framework by means of global constraints over standard integer variables. The same ideas can be applied to implement a native variable of LiVar type in a CP tool.

To demonstrate the robustness of our approach, we implemented the proposed LiVar realization method. The paper presents experimental results demonstrating the effectiveness of global constraint propagation over LiVars.

The rest of the paper is organized as follows. In section 2 we provide the background required for presentation of our results. Section 3 presents the definition of LiVar including the set of supported operations. In Section 4 we propose LiVar representation in the standard CP framework and present implementations of comparison constraints on LiVars as well as a method for handling arithmetic expressions on LiVars by means of global constraints over integer variables. In order to provide the basis for comparison and quality evaluation of the latter method, we also describe a naïve method for handling arithmetic expression through formulating standard integer expressions. Section 5 demonstrates experimental results. We conclude in Section 6 with the summary of the presented results.

2 Background

For the sake of completeness, in this section we provide CP background required to facilitate the presentation of the rest of this paper. An in-depth survey of the traditional CP can be found in [7].

The CP paradigm comprises the modeling of a problem as a CSP, constraint propagation, search algorithms, and heuristics. A CSP is defined by:

- a set of constrained variables. Each variable is associated with a (finite) domain defined as a collection of values that the variable is allowed to take;
- a set of constraints. A constraint is a relation defined on a subset of variables which restricts the combinations of values that the variables can take simultaneously.

A *solution* to a CSP is an assignment of values to variables so that each variable is assigned a value from its domain and all the constraints are satisfied.

A CSP formulation of a problem is processed by a constraint solver, which attempts to find a solution using a search algorithm combined with reductions of variable domains based on constraint information. The latter mechanism is known as *constraint propagation*. During the constraint propagation, domains of the variables involved in the constraint are reduced until some type of consistency is achieved. For example, one of the possible types of consistency is the *generalized arc consistency* (GAC), also known as *domain consistency* [8], which implies that for each value in the domain of a variable, there exists a value in the domain of each of the other variables participating in the constraint so that the assignment of these values to the variables satisfies the constraint. Another commonly used consistency type is the *bounds consistency*, also known as *interval consistency* [8], which implies that for the minimal and maximal domain values of a variable there exists a value within the domain bounds of each of the other variables participating in the constraint so that the assignment of these values to the variables satisfies the constraint. To ensure the required type of consistency, a solver associates each constraint type with its specific propagation algorithm. There can be a number of propagation algorithms tied to one constraint type, where each algorithm is responsible for propagating a specific domain change to domains of other variables involved in the constraint. Such algorithms are known as *propagation demons*.

A constraint solver may implement a number of special constraints known as *global constraints*. These constraints are characterized by semantic efficiency in the sense that they would require a large number of simpler constraints or would be at all impossible to formulate using simple relations. Moreover, these constraints are typically associated with efficient propagation algorithms that achieve better filtering with respect to some consistency level than any alternative combination of simpler relations. See [9] for formal definitions related to this concept.

The *search space* is the Cartesian product of all the variable domains. Let Z be a search space. A *solution space* $S \subseteq Z$ is a set of all possible assignments to variables that are solutions to the CSP.

3 LiVar Definition

In this section we formally introduce the notion of LiVar.

We define LiVar as a variable type used to represent unsigned integers of a fixed bit width in a CSP model. LiVar is specified by a single natural number n indicating the bit width of the corresponding object. Let $A(n)$ be a LiVar, then the domain of A is the integer range $[0 \ldots 2^n - 1]$. An arithmetic operation on LiVars results in an expression also associated with a bit width. We refer to an arithmetic expression involving LiVars as *LiExpr*. LiVar can be viewed as a special case of LiExpr.

The following arithmetic operations are defined on a LiExpr $A(n)$:

- addition/subtraction modulo n of a constant c
- addition/subtraction of another LiExpr $B(m)$ modulo $max(n,m)$
- shift left by k bits
- shift right by k bits
- multiplication by a constant 2^c
- division by a constant 2^c
- sub-range extraction; given the range start bit index b_{start} and the range end bit index b_{end}, this operation returns the expression equal to the value of the sub-range of A within the specified range bounds, $A[b_{start}:b_{end}]$

In addition, a LiExpr $A(n)$ supports the arithmetic comparison constraints ($=, \neq, >, \geq, <, \leq$) defined on the standard integer variables, where the comparison can be performed with either a constant value or another LiExpr.

We observe that the domain size of a LiVar is 2^n for any specified bit width n and it is not limited by the size of an integer type.

We also note that the definition of LiVar presented above is motivated by applications where integer variables are used to model unsigned machine words. However, the ideas presented in this paper can be also applied to implement arbitrarily large signed integer variables.

4 LiVar Realization in the Standard CP Framework

In this section we propose LiVar representation in the standard CP framework. We show how the arithmetic comparison constraints on LiVar can be implemented for the proposed LiVar representation. Finally, we define a set of global constraints to support the operations on LiVars specified in Section 3, and propose propagation algorithms for these global constraints.

4.1 LiVar Representation

We represent a LiVar (or a LiExpr) $A(n)$ as an array of standard integer CSP variables $A_1, A_2, \ldots, A_m$ so that each variable A_i, $1 \le i \le m$, represents the corresponding sub-range of $A(n)$. Specifically, we choose a granularity parameter k so that the value 2^k-1 is small enough to be representable by a standard integer type. Then the number of integer variables required for representation of $A(n)$ is $m=\lceil n/k \rceil$. Assuming the bits of $A(n)$ are numbered from 0 (the least significant bit) to $n-1$ (the most significant bit), each variable A_i, with the possible exception of A_m, represents a sub-range of $A(n)$ of size k between the bits $k\cdot(i-1)$ and $k\cdot i-1$, for $1 \le i \le m-1$. The last variable A_m represents a possibly smaller sub-range of the most significant bits of $A(n)$, namely, the range between the bits $k\cdot(m-1)$ and $n-1$. Consequently, the variables A_i, $1 \le i \le m-1$, have the domain $[0 \ldots 2^k-1]$, and A_m has the domain $[0 \ldots 2^{n+k-k\cdot m}-1]$.

To facilitate the understanding, the proposed representation can be thought of as a "byte" representation of $A(n)$, with the "byte" size of k bits. We will refer to the integer variables in the representation of LiVar A as the *byte variables* of A.

To illustrate the proposed LiVar representation, we consider an example of a LiVar $A(20)$ for $k=8$. For this value of k, $A(20)$ is represented by three standard integer variables $A_1[0 \ldots FFh]$, $A_2[0 \ldots FFh]$ and $A_3[0 \ldots Fh]$. The state in which there remain two possible values in the domain of A, namely `91BE2h` and `A1B3Ch` (hexadecimal representation of values is used for the ease of transformation), corresponds to the state of domains of A_1, A_2, A_3 as shown in Fig. 1.

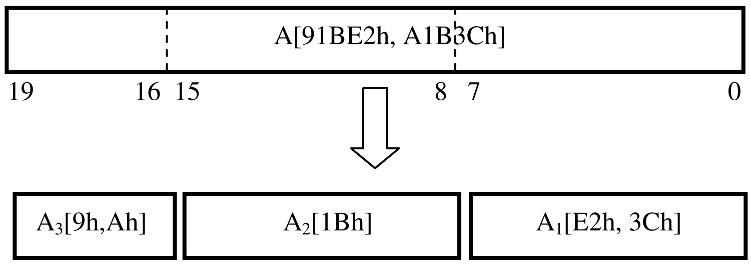


Fig. 1. LiVar representation as an array of integer variables

Observe that in the representation of LiVar described above there is no limitation on the domain size 2^n of LiVar $A(n)$ other than the computer memory size limitation in storing $m=\lceil n/k \rceil$ integer variables.

We note that the proposed LiVar representation is associated with an inherent precision loss. For example, in Fig. 1 the true variable A has two values in its domain whereas the proposed representation allows four possible values. However, since the propagation algorithms described in the sequel of this paper for constraints defined on LiVars do not maintain GAC, the impact of this precision loss is decreased.

4.2 Arithmetic Comparison Constraints on LiVars

In this subsection we show how the arithmetic comparison constraints (=, ≠, >, ≥, <, ≤) can be implemented for LiVar representation defined in Section 4.1. The proposed implementation for all of these constraints is by means of standard comparison constraints on integer variables.

We start with constraint formulations for comparisons between two LiVars. The equality and inequality constraint implementation is straightforward. Let $A(n_1)$ and $B(n_2)$ be two LiVars, and let A_i, $1 \le i \le m_1$ and B_j, $1 \le j \le m_2$ be the byte variables in the representations of these LiVars as defined in Section 4.1. We assume without the loss of generality that $m_1 \ge m_2$. Then the equality constraint $A=B$ is equivalent to:

$$\bigwedge_{1 \le i \le m_2} (A_i = B_i) \wedge \bigwedge_{m_2 < j \le m_1} (A_i = 0)$$

Similarly, the inequality constraint $A \ne B$ corresponds to:

$$\bigvee_{1 \le i \le m_2} (A_i \ne B_i) \vee \bigvee_{m_2 < j \le m_1} (A_i \ne 0)$$

We proceed with presenting the implementation of *greater than* constraint between two LiVars. First, assume for the simplicity of presentation that $A(n)$ and $B(n)$ are two LiVars of the same bit width n (we will drop this assumption later), and let A_i, $1 \le i \le m$ and B_j, $1 \le j \le m$ be the byte variables of A and B, respectively. The mathematical formulation of the corresponding relation is not difficult, but the challenge in implementing this constraint is to create an efficient formulation in terms of constraint propagation. For example, the following formulation

$$\bigvee_{1 \le i \le m} \left(\bigwedge_{m \ge j > i} (A_j = B_j) \wedge (A_i > B_i) \right)$$

is weak due to poor propagation of the disjunction constraint. For instance, a combination of *greater/less than* relations that uniquely determines the values for some of the byte variables of A or B, would not be propagated to those variables, leaving their domains unchanged.

Instead we propose the following formulation that achieves efficient propagation between integer variables in LiVar representation. We define for each $2 \le i \le m$ the following constraint denoted as *EqualPrefix(i)*: $\wedge_{m \ge j \ge i}(A_j = B_j)$. Then the constraint $A>B$ is formulated as follows:

$$(A_m \ge B_m) \wedge \bigwedge_{m \ge i \ge 3} \left(Equal\mathrm{Pr}efix(i) \Rightarrow (A_{i-1} \ge B_{i-1}) \right) \wedge \left(Equal\mathrm{Pr}efix(2) \Rightarrow (A_1 > B_1) \right) \quad (4.1)$$

The formulation above is conjunction based and propagates any relevant change in domains of byte variables. If A and B have different representation lengths m_A and m_B, then the constraint above should be augmented as follows. If $m_A > m_B$, then the resulting constraint is the disjunction of the constraint (4.1) with $\vee_{m_B < i \le m_A} (A_i > 0)$. Otherwise, the resulting constraint is the conjunction of the constraint (4.1) with $\vee_{m_A < i \le m_B} (B_i = 0)$.

The implementation of *greater than or equal* constraint $A \geq B$ is similar to that of $A > B$. Here, the constraint (4.1) should be replaced by:

$$(A_m \geq B_m) \wedge \bigwedge_{m \geq i \geq 2} \left(EqualPrefix(i) \Rightarrow (A_{i-1} \geq B_{i-1}) \right) \tag{4.2}$$

The implementations of *less than* and *less than or equal* constraints are analogous to those of *greater than* and *greater than or equal* constraints presented above.

We observe that when LiVar representations of A and B have the same length, the comparison constraints between A and B are equivalent to the lexicographic ordering constraints on vectors of variables presented in [10]. The constraint formulations above are similar to one of the alternative formulations for the lexicographic ordering constraint given in [10]. The latter work also presents a GAC propagation algorithm for lexicographic ordering and demonstrates experimentally that for selected combinatorial problems this algorithm outperforms the alternative formulation. However, in the context of this paper where vectors of variables represent LiVars and allowed constraint types are confined to the set defined in Section 3, the consistency level achieved by the formulation shown above is typically sufficient to eliminate backtracking and GAC enforcement is not required.

Finally, the comparison constraints of LiVar A with a constant c are implemented similarly to comparisons of LiVar A with LiVar B. In this case, the values of c in the corresponding sub-ranges of size k participate in the constraint formulations in place of the variables B_j.

4.3 Arithmetic Expressions on LiVars

As mentioned earlier, we represent an expression involving LiVars, or LiExpr, in the same way as LiVar, i.e. as an array of integer variables. Let LiExpr E be a result of an operation on LiVars, e.g. *op(A)* in case of an unary operation (such as left/right shift) or *(A op B)* in case of a binary operation (such as addition). In this section we describe a method of expressing the relation between LiVars involved in the operation and the resulting expression E by means of global constraints on the byte variables of A, B, and E. In order to provide a comparison basis for the evaluation of the proposed method, we start by presenting a naïve approach to handling arithmetic expressions on LiVars. The latter approach is to express byte variables of E as standard integer expressions involving byte variables of A and B. The comparison of the methods based on experimental results is done in Section 5.

4.3.1 Naïve Implementation by Means of Standard Integer Expressions

In this subsection we show how the byte variables of LiExpr E resulting from an arithmetic operation on LiVars can be expressed through standard integer expressions on byte variables of LiVars involved in the operation.

We start with the addition between two LiVars, $E=A+B \pmod{2^n}$. In the expressions below we will use the operation *div* to denote the integer division operation. We define an auxiliary array *Carry* of m standard integer expressions, $m=\lceil n/k \rceil$, representing the carry bits participating in addition of each pair of sub-ranges of size k. The carry bit expressions are defined as follows:

$$Carry[1] = 0,\ Carry[i+1] = div(A_i + B_i + Carry[i],\ 2^k),\ for\ 1 \leq i \leq m-1$$

Then the byte variables of E can be expressed as follows:

$$E_i = A_i+B_i+Carry[i]-div(A_i+B_i+Carry[i],2^k)\cdot 2^k$$

A special treatment is required when the number of byte variables in the representation of A or B is less than m. In this case, the zero constant should be used in the expression above in place of the "missing" byte variables.

We proceed with the subtraction operation, $E=A-B \ (mod\ 2^n)$. In this case, the *Carry* expression array is defined as follows:

$$Carry[1] = 0,\ Carry[i+1] = 1-div(A_i - B_i - Carry[i]+2^k,\ 2^k),\ for\ 1\le i\le m-1$$

The expressions for the byte variables of E are defined as shown below:

$$E_i = A_i-B_i-Carry[i]+2^k - div(A_i-B_i-Carry[i]+2^k,2^k)\cdot 2^k$$

Similarly to the addition operation, a special treatment is required when the representation of A or B has less than m byte variables.

The addition and subtraction of a constant to a LiVar can be expressed in the similar way as the operations described above. Here, the corresponding sub-range values of the constant should be used in place of the byte variables B_i.

Next consider the operation of *shift left by t bits* of a LiVar $A(n)$. Let $\delta=t \ (mod\ k)$, and let $p=t/k$. Then a byte variable of the resulting expression E can be defined by

$$E_i = (A_{i-p} - div(A_{i-p},2^{k-\delta})\cdot 2^{k-\delta})\cdot 2^{\delta} + div(A_{i-p-1},2^{k-\delta})$$

We proceed with the operation of *shift right by t bits* of a LiVar $A(n)$. Using the same notation as above, the expression for a byte variable of E can be defined as

$$E_i = (A_{i+p+1} - div(A_{i+p+1},2^{\delta})\cdot 2^{\delta})\cdot 2^{k-\delta} + div(A_{i+p},2^{\delta})$$

In the shift operations above, E_i equals 0 when the indexes of both corresponding byte variables of A are out of the range of A representation, and special treatment is required for border cases where only one of these indexes is out of the range.

The multiplication/division by a constant $c=2^t$ can be expressed similarly to the shift left/right operations. Multiplication expressions are defined in the same way as for the shift left operation, but in case of multiplication there are more byte variables in the representation of E as the bits of A are not shifted out. The expressions for the division operation are defined in the same way as for the shift right operation, but in case of division E has fewer byte variables as there are no leading zeros.

Finally, consider the sub-range extraction operation $E=A[b_{start}:b_{end}]$. This operation can be expressed as a combination of shift left and division by a power-of-two constant. Indeed, let n be the bit width of A and let $p=b_{end}-b_{start}+1$ denote the sub-range width. We can apply the shift left by $(n-b_{end}-1)$ bits to shift out the most significant bits of A starting from $b_{end}+1$, and then to divide the result by 2^{n-p} to truncate the least significant zeros and bits of A up to $b_{start}-1$.

4.3.2 Implementation by Means of Global Constraints

In this subsection we define global constraints to express the relations corresponding to arithmetic expressions on LiVars and present their propagation algorithms.

We start with a global constraint *SumConstraint(A,B,C)* describing the relation $A+B=C \pmod{2^n}$ for three LiExprs *A*, *B*, and *C*, where *n* is the maximal bit width between those of *A* and *B*. Similarly to the representation with integer expressions described in the previous subsection, the constraint defines an auxiliary array *Carry* of *m* standard integer expressions representing the carry bits participating in addition of each pair of sub-ranges of size *k*, where $m=\lceil n/k \rceil$. Here, we propose an alternative definition of carry expressions. The carry bit expressions are defined as follows:

$$Carry[1] = 0$$

$$Carry[i+1] = (A_i+B_i+Carry[i] \geq 2^k), \text{ for } 1 \leq i \leq m-1$$

Observe that the definition of *Carry* above applies reification.

The propagation algorithm that we propose for *SumConstraint(A,B,C)* achieves the bounds consistency on each of the byte variables of *A*, *B*, and *C*. Actually, the performed filtering is stronger than required to achieve the bounds consistency, removing internal domain sub ranges in certain cases. The proposed algorithm has four propagation demons associated with domain changes in *A*, *B*, *C*, and *Carry*. Each of the propagation demons is activated when domain bounds of one of the corresponding byte variables change. When the domain bound of a byte variable *i* changes, the algorithm of the corresponding demon performs filtering required to achieve the bounds consistency on the relation $A_i+B_i+Carry[i] = C_i \pmod{2^k}$. The filtering works in the following way. Let $op1 \in \{A_i, B_i, Carry[i]\}$ denote the term whose domain is being reduced, and let *op2* and *op3* denote the other two terms in this set. Then the domain reduction for $op1_i$ is described in the following procedure:

```
ReduceOperandDomain(op1,op2,op3,i):
    op1Min ← DomainMin(C_i)–(DomainMax(op2_i)+DomainMax(op3_i))
    op1Max ← DomainMax(C_i)–(DomainMin(op2_i)+DomainMin(op3_i))
    if (op1Max–op1Min <2^k–1)
        op1MinMod ← op1Min mod 2^k
        op1MaxMod ← op1Max mod 2^k
        if (op1MinMod ≤ op1MaxMod)
            setDomainMin(op1_i, op1MinMod)
            setDomainMax(op1_i, op1MaxMod)
        else
            removeDomainRange(op1_i,op1MaxMod+1,op1MinMod–1)
```

The domain reduction for the byte variables C_i is similar and is performed as follows:

```
ReduceSumDomain(i):
    CMin ← DomainMin(A_i)+DomainMin(B_i)+DomainMin(Carry[i])
    CMax ← DomainMax(A_i)+DomainMax(B_i)+DomainMax(Carry[i])
    if (CMax–CMin < 2^k–1)
        CMinMod ← CMin mod 2^k
        CMaxMod ← CMax mod 2^k
        if (CMinMod ≤ CMaxMod)
            setDomainMin(C_i, CMinMod)
            setDomainMax(C_i, CMaxMod)
        else
            removeDomainRange(C_i,CMaxMod+1,CMinMod–1)
```

In the case when the procedures *ReduceOperandDomain* and *ReduceSumDomain* are performed for the most significant byte variables with the index $i=m=\lceil n/k \rceil$, k in the procedures should be replaced by the actual width of the most significant byte variable, namely, $n+k-k \cdot m$.

Each of the propagation demons for A, B, and *Carry* performs the *ReduceOperandDomain* procedure for the other two sum terms, and *ReduceSumDomain* procedure for the corresponding C_i. The propagation demon for C performs the *Reduce OperandDomain* procedure for each of the sum terms A, B, and *Carry*.

Next we observe that the implementation of the global constraint *DiffConstraint(A,B,C)* expressing the arithmetic difference relation $A-B=C \ (mod\ 2^n)$ can be obtained straightforwardly from the implementation of SumConstraint. Indeed, observe that *DiffConstraint(A,B,C)* is equivalent to *SumConstraint(C,B,A)*.

Now consider the global constraint *SumConstraint(A,Const,C)* describing the relation $A+Const=C \ (mod\ 2^n)$ for LiVars A and C, and a constant value *Const*. We observe that the propagation algorithm for this constraint works along the same lines as that of *SumConstraint(A,B,C)*, where values of the constant *Const* in the sub-ranges of size k are used in calculating new domain bounds in place of the minimal and maximal values of the byte variables B_i. Again, similarly to the observation above, the constraint *DiffConstraint(A,Const,C)* is equivalent to *SumConstraint(C,Const,A)*.

We proceed with defining a global constraint required to express the remaining arithmetic operations on LiVars as defined in Section 3, namely, *shift left/right*, *multiplication/division by a power-of-two constant* and *sub-range extraction*. We implement all of these operations by means of a single generic global constraint $SubRangeConstraint(A,C,b_{start},b_{end})$. Here $b_{start} \le b_{end}$ indicate the start and the end bit indices of a sub-range of A and can be any integer values. The bits of the sub-range $[b_{start}: b_{end}]$ that fall outside the boundaries of A, that is, have indices less than 0 or greater than $n-1$ are considered to be 0. This constraint expresses the relation between LiExprs $A(n)$ and $C(b_{end}-b_{start}+1)$ implying that the value of C is equal to the value of A in the sub-range $[b_{start}: b_{end}]$.

Next we show how the constraint $SubRangeConstraint(A,C,b_{start},b_{end})$ defined above can be applied to express the required operations on LiVars. The *shift left by t bits* operation on LiExpr $A(n)$ results in LiExpr $C(n)$ so that the bits of C are the bits of A shifted left by t positions (and thus the t most significant bits of A are shifted out) and the t least significant bits of C are zeros. This relation can be expressed as $SubRangeConstraint(A,C,-t,-t+n-1)$. Similarly, the shift right by t bits can be expressed as $SubRangeConstraint(A,C,t,t+n-1)$. We proceed with the multiplication of LiExpr $A(n)$ by a constant 2^t. This operation results in LiExpr $C(n+t)$ such that the n most significant bits of C equal to the bits of A and the t least significant bits are zeros. This relation can be represented as $SubRangeConstraint(A,C,-t,n-1)$. Similarly, the integer division of A by a constant 2^t results in LiExpr $C(n-t)$ such that the bits of C equal to the $n-t$ most significant bits of A. This operation can be expressed by $SubRangeConstraint(A,C,t,n-1)$. Finally, the implementation of sub-range extraction of $A(n)$ for a sub-range $[b_{start}: b_{end}]$ is straightforward; the latter relation can be expressed as $SubRangeConstraint(A,C,b_{start},b_{end})$.

We proceed by presenting the propagation algorithm for the global constraint $SubRangeConstraint(A,C,b_{start},b_{end})$. Like the algorithm presented for SumConstraint, the algorithm for *SubRangeConstraint* maintains bounds consistency, however, it performs more filtering than required for maintaining this consistency level. The constraint has two propagation demons associated with domain changes in *A* and in *C*, respectively. Like in *SumConstraint*, these demons are parameterized by *i*, the index of the byte variable the domain of which has been changed. Each of the demons is called on domain boundary changes of one of the corresponding byte variables.

Suppose a domain boundary change occurred in a byte variable A_i. Let C_j and C_{j+1} be the byte variables of *C* such that their sub-ranges overlap with the sub-range of A_i. For convenience, denote C_j by $C_{high}(i)$ to indicate that C_j has the overlapping with A_i in its most significant bits. Similarly, denote C_{j+1} by $C_{low}(i)$. Furthermore, let $\delta = b_{start}\ (mod\ 2^k)$ denote the relative shift of $C_{low}(i)$ with respect to A_i. Fig. 2 illustrates the proposed notation.

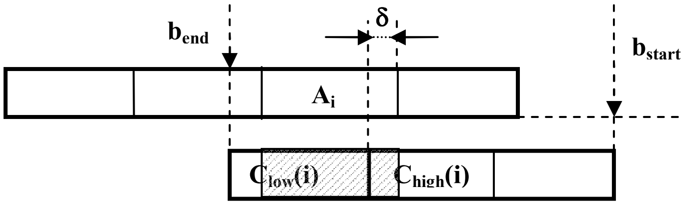


Fig. 2. Range overlapping in the sub-range constraint

The following filtering procedures for $C_{high}(i)$ and $C_{low}(i)$ are performed when the domain boundaries of A_i change. The filtering procedure for $C_{high}(i)$ bases upon the equality between the least significant bits (the suffix) of A_i and the most significant bits (the prefix) of $C_{high}(i)$ to reduce the allowed range of the latter. Due to the modulo operation, this reduction can lead to removing an internal sub range of $C_{high}(i)$.

```
UpdateHigh(i):
      d ← δ
      if (d=0)
            d ← k
      if (DomainMax(A_i)−DomainMin(A_i)<2^d−1)
            SuffixMax ← DomainMax(A_i) (mod 2^d)
            SuffixMin ← DomainMin(A_i) (mod 2^d)
            if (SuffixMax < SuffixMin)
                  removeDomainRange(C_high(i),(SuffixMax+1)·2^(k−d),
                        (SuffixMin−1)·2^(k−d) + 2^(k−d)−1)
            else
                  setDomainMin(C_high(i),SuffixMin·2^(k−d))
                  setDomainMax(C_high(i),SuffixMax·2^(k−d) + 2^(k−d)−1)
```

The filtering procedure for $C_{low}(i)$ bases upon the equality between the prefix of A_i and the suffix of $C_{low}(i)$ to remove the sub ranges corresponding to forbidden suffixes from the domain of the latter.

```
UpdateLow(i):
    if (δ>0)
        PrefixMax ← DomainMax(A_i)/2^{k-δ}
        PrefixMin ← DomainMin(A_i)/2^{k-δ}
        forbiddenMin1 ← 0;
        forbiddenMax1 ← PrefixMin-1
        forbiddenMin2 ← PrefixMax+1
        forbiddenMax2 ← 2^{k-δ}-1
        for each prefix: PrefixMin≤prefix≤PrefixMax
            removeDomainRange(C_low(i),
                prefix·2^{k-δ}+forbiddenMin1,
                prefix·2^{k-δ}+forbiddenMax1)
            removeDomainRange(C_low(i),
                prefix·2^{k-δ}+forbiddenMin1,
                prefix·2^{k-δ}+forbiddenMax1)
```

We observe that the procedures presented above refer to the mainstream case when all the byte variables involved in the filtering algorithm are of size k and fall inside the range of A. There is a substantial number of corner cases to be considered where the most significant byte variables of shorter size or the byte variables of C that are not fully contained in the range of A are involved in the filtering procedure. The filtering algorithms for these cases are performed according to the similar principles as those presented above and are not presented here for the sake of conciseness.

The same filtering procedures are performed in the opposite direction when the domain boundary change occurs in a byte variable C_i of C. In this case, the filtering is done for the byte variables $A_j=A_{high}(i)$ and $A_{j+1}=A_{low}(i)$ that overlap with C_i.

To summarize the propagation algorithm for *SubRangeConstraint*(A,C,b_{start},b_{end}), the propagation demon for A_i performs the filtering procedures *UpdateHigh(i)* and *UpdateLow(i)* for the corresponding byte variables of C, and the propagation demon for C_i performs the symmetric filtering procedures for the corresponding byte variables of A.

5 Experimental Results

To demonstrate the effectiveness of the proposed LiVar realization, we implemented the approach described in this paper within a preset traditional CP environment and compared the performance and constraint propagation quality of the two methods for arithmetic expression implementation presented in Section 4.3.

We observe that, commonly for CP, there is a tradeoff between the time spent on domain filtering and the time spent on variable assignment enumeration. We experimented with different levels of consistency for the global constraint presented in Section 4.3.2. In our experiments, we found out that while increasing the consistency levels to GAC achieves less search fails, it incurs too much time cost leading in most cases to the performance slowdown. Based on our experiments, we believe that the consistency level of algorithms presented in this paper achieves a good tradeoff between the search phases, optimizing the overall search time.

We organized our experiments as follows. In all the experiments, we applied random search, specifically, the value selection for each variable was performed uniformly at random from the values remaining in the domain of the variable. The motivation for applying random search has been to cover a representative sample of search invocations as filtering quality depends on specific value selection and may differ significantly for different value choices. We observe that since the total number of possible assignments to LiVars in our test cases is huge, and our algorithms do not achieve GAC on all of the involved integer variables, there can be particular random search runs that take impractically high time to complete. For this reason, we performed our experiments imposing limits on the search. For each test case, we performed as many as needed searches to obtain 1000 random solutions, and counted the number of discards due to the search reaching the specified limit. We performed experiments with different kinds of search limits, specifically, the limit of 100, 500, and 5000, respectively, on the number of fails, and the time limit of 1 second.

We compared the proposed methods on three test cases, two involving operations of addition/subtraction only, and the third one involving a combination of addition/subtraction and sub-range operations. All of the three test cases involve 128 bit width LiVars, which cannot be accommodated by standard CP engines. In Test Case 1, we considered an expression $E=A+B-C$ where A, B, and C are LiVars, and constrained this expression to fall within a “loose” range containing $3{\cdot}2^{53}$ values out of 2^{128} in the domain of E. Test Case 2 is identical to Test Case 1 but for the size of the value range for E. Here, we required E to fall within a “tight” range of $4{\cdot}2^{16}$ values. In Test Case 3, we considered an expression $(A+B)[11:0]$ (the 12 least significant bits of $A+B$) and constrained it to equal zero.

The tables below present the results of our experiments. Each table entry indicates the running time until 1000 solutions are obtained (the first line), the average number of search fails in each run (the second line) and the number of discards due to search limits (the third line). The empty entries indicate that the specific method was not able to obtain solutions for the specific test case within a practical running time. Tables 1, 2, and 3 present the results for Test Case 1, 2, and 3, respectively. We performed the evaluation on Pentium M 2.5 GHz processor with 2.96 GB of RAM.

Table 1. Experimental results for Test Case 1

	Fail Limit 100	Fail Limit 500	Fail Limit 5000	Time limit 1sec
Integer Expressions Method	83.6 sec 2.98 fails 451 discards	62.5 sec 14.7 fails 325 discards	82.9 sec 24.0 fails 331 discards	368 sec 869 fails 313 discards
Global Constraints Method	1.98 sec 0.48 fails 1 discard	2.89 sec 0.65 fails 5 discards	3.59 sec 0.49 fails 2 discards	4.80 sec 1.08 fails 3 discards

Table 2. Experimental results for test case 2

	Fail Limit 100	Fail Limit 500	Fail Limit 5000	Time limit 1sec
Integer Expressions Method	–	–	–	–
Global Constraints Method	2.67 sec 0.28 fails 2 discard	2.20 sec 0.45 fails 1 discard	3.11 sec 0.21 fails 1 discard	4.38 sec 0.99 fails 2 discards

Table 3. Experimental results for test case 3

	Fail Limit 100	Fail Limit 500	Fail Limit 5000	Time limit 1sec
Integer Expressions Method	4.03 sec 40.6 fails 4232 discards	5.63 sec 102 fails 2200 discards	35.6 sec 141 fails 2147 discards	2061 sec 4644 fails 2042 discards
Global Constraints Method	1.39 sec 7.60 fails 0 discards	1.31 sec 7.70 fails 0 discards	1.27 sec 7.48 fails 0 discards	1.38 sec 7.77 fails 0 discards

The presented experimental results show that the expression implementation based on global constraints has a clear advantage, sometimes in several orders of magnitude, over the naïve method based on integer expressions. This result can be explained by the fact that the global constraints method with custom domain filtering algorithms achieves better propagation and therefore fewer search fails than the standard integer expression propagation algorithms which are not tuned for this specific problem.

6 Conclusion

The main contribution of this paper is the extension of the traditional CP framework to accommodate integer constraint variables of arbitrarily large domain size. Such extension makes it possible to apply CP to a range of problems where such application has not been possible due to integer variable size limitations of the existing CP tools.

We proposed a method to represent such variables on top of the traditional CP framework. The paper shows how constraints and expressions on integer variables can be implemented through standard CP means for the proposed representation. The set of arithmetic operations on integer variables considered in this paper was defined to accommodate common requirements on large integers. We proposed a method to implement expressions on large integer variables by means of customized global constraints.

We presented experimental results to demonstrate the effectiveness of the proposed method. The results show that the proposed extension can be efficiently integrated into the standard CP framework by means of global constraints.

Finally, we observe that the ideas presented in this paper can be used to implement a native large integer variable within a CP tool. For this purpose, one can represent domains of such variables as a vector of bytes (or a partition with any sufficiently small granularity) and perform domain reductions based on the algorithms presented in the paper.

Acknowledgements

The author would like to thank Boris Gutkovich and Yevgeny Schreiber for their valuable comments and helpful discussions.

References

1. Bin, E., Emek, R., Shurek, G., Ziv, A.: Using a constraint satisfaction formulation and solution techniques for random test program generation. IBM Systems Journal 41(3), 386–402 (2002)
2. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. AI Magazine 28(3), 13–18 (2007)
3. Gutkovich, B., Moss, A.: CP with architectural state lookup for functional test generation. In: 11th Annual IEEE International Workshop on High Level Design Validation and Test, pp. 111–118 (2006)
4. Moss, A.: Constraint patterns and search procedures for CP-based random test generation. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 86–103. Springer, Heidelberg (2008)
5. Triska, M.: Generalising Constraint Solving over Finite Domains. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 820–821. Springer, Heidelberg (2008)
6. SystemVerilog, IEEE Std. 1800^{TM} (2005)
7. Smith, B.M.: Modeling for constraint programming. In: The 1st Constraint Programming Summer School (2005)
8. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation and evaluation of the constraint language cc(FD). Journal of Logic Programming 31(1-3), 139–164 (1998)
9. Bessière, C., Van Hentenryck, P.: To be or not to be ... a global constraint. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 789–794. Springer, Heidelberg (2003)
10. Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Global constraints for lexicographic orderings. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 93–108. Springer, Heidelberg (2002)

Constraint-Based Local Search for Constrained Optimum Paths Problems

Quang Dung Pham[1], Yves Deville[1], and Pascal Van Hentenryck[2]

[1] Université catholique de Louvain B-1348 Louvain-la-Neuve, Belgium
{quang.pham,yves.deville}@uclouvain.be
[2] Brown University, Box 1910 Providence, RI 02912, USA
pvh@cs.brown.edu

Abstract. Constrained Optimum Path (COP) problems arise in many real-life applications and are ubiquitous in communication networks. They have been traditionally approached by dedicated algorithms, which are often hard to extend with side constraints and to apply widely. This paper proposes a constraint-based local search (CBLS) framework for COP applications, bringing the compositionality, reuse, and extensibility at the core of CBLS and CP systems. The modeling contribution is the ability to express compositional models for various COP applications at a high level of abstraction, while cleanly separating the model and the search procedure. The main technical contribution is a connected neighborhood based on rooted spanning trees to find high-quality solutions to COP problems. The framework, implemented in `COMET`, is applied to Resource Constrained Shortest Path (RCSP) problems (with and without side constraints) and to the edge-disjoint paths problem (EDP). Computational results show the potential significance of the approach.

1 Introduction

Constrained Optimum Path (COP) problems appear in many real-life applications, especially in communication and transportation networks (e.g., [5]). They aim at finding one or more paths from some origins to some destinations satisfying some constraints and optimizing an objective function. For instance, in telecommunication networks, routing problems supporting multiple services involve the computation of paths minimizing transmission costs while satisfying bandwidth and delay constraints [3,6]. Similarly, the problem of establishing routes for connection requests between network nodes is one of the basic operations in communication networks and it is typically required that no two routes interfere with each other due to quality-of-service and survivability requirements. This problem can be modeled as edge-disjoint paths problem [4]. Most of COP problems are NP-hard. They are often approached by dedicated algorithms, such as the Lagrangian-based branch and bound in [3] and the vertex labeling algorithm from [7]. These techniques exploit the structure of constraints and objective functions but are often difficult to extend and reuse.

This paper proposes a constraint-based local search (CBLS) [10] framework for COP applications to support the compositionality, reuse, and extensibility at the core of CBLS and CP systems. It follows the trend of defining domain-specific CBLS frameworks, capturing modeling abstractions and neighborhoods for classes of applications exhibiting significant structures. The COP framework can also be viewed as an extension of the `LS(Graph & Tree)` framework [8] for those applications where the output of the optimization model is one or more elementary paths (i.e., paths with no repeated nodes). As is traditional for CBLS, the resulting COP framework allows the model to be compositional and easy to extend, and provides a clean separation of concerns between the model and the search procedure. Moreover, the framework captures structural moves that are fundamental in obtaining high-quality solutions for COP applications. The key technical contribution underlying the COP framework is a novel connected neighborhood for COP problems based on rooted spanning trees. More precisely, the COP framework incrementally maintains, for each desired elementary path, a rooted spanning tree that specifies the current path and provides an efficient data structure to obtain its neighboring paths and their evaluations.

The availability of high-level abstractions (the "what") and the underlying connected neighborhood for elementary paths (the "how") make the COP framework particularly appealing for modeling and solving complex COP applications. The COP framework, implemented in `COMET`, was evaluated experimentally on two classes of applications: Resource-Constrained Shortest Path (RCSP) problems with and without side constraints and Edge-Disjoint Path (EDP) problems. The experimental results show the potential of the approach.

The rest of this paper is organized as follows. Section 2 gives the basic definitions and notations. Section 3 specifies our novel neighborhoods for COP applications and Section 4 presents the modeling framework. Section 5 applies the framework to two various COP applications and Section 6 concludes the paper.

2 Definitions and Notations

Graphs Given an undirected graph g, we denote the set of nodes and the set of edges of g by $V(g)$, $E(g)$ respectively. A path on g is a sequence of nodes $< v_1, v_2, ..., v_k >$ $(k > 1)$ in which $v_i \in V(g)$ and $(v_i, v_{i+1}) \in E(g), (i = 1, \ldots, k-1$. The nodes v_1 and v_k are the origin and the destination of the path. A path is called *simple* if there is no repeated edge and *elementary* if there is no repeated node. A cycle is a path in which the origin and the destination are the same. This paper only considers elementary paths and hence we use "path" and "elementary path" interchangeably if there is no ambiguity. A graph is connected if and only if there exists a path from u to v for all $u, v \in V(g)$.

Trees. A tree is an undirected connected graph containing no cycles. A spanning tree tr of an undirected connected graph g is a tree spanning all the nodes of g: $V(tr) = V(g)$ and $E(tr) \subseteq E(g)$. A tree tr is called a rooted tree at r if the node r has been designated the root. Each edge of tr is implicitly oriented towards the root. If the edge (u, v) is oriented from u to v, we call v the father of u in

tr, which is denoted by $fa_{tr}(u)$. Given a rooted tree tr and a node $s \in V(tr)$, we use the following notations:

- $root(tr)$ for the root of tr,
- $path_{tr}(v)$ for the path from v to $root(tr)$ on tr. For each node u of $path_{tr}(v)$, we say that u *dominates* v in tr (u is a dominator of v, v is a descendant of u) which we denote by $u\ Dom_{tr}\ v$.
- $path_{tr}(u, v)$ for the path from u to v in t ($u, v \in V(tr)$).
- $nca_{tr}(u, v)$ for the nearest common ancestor of two nodes u and v. In other words, $nca_{tr}(u, v)$ is the common dominator of u and v such that there is no other common dominator of u and v that is a descendant of $nca_{tr}(u, v)$.

3 The COP Neighborhoods

A neighborhood for COP problems defines the set of paths that can be reached from the current solution. To obtain a reasonable efficiency, a local-search algorithm must maintain incremental data structures that allow a fast exploration of this neighborhood and a fast evaluation of the impact of the moves (differentiation). The key novel contribution of our COP framework is to use a rooted spanning tree to represent the current solution and its neighborhood. It is based on the observation that, given a spanning tree tr whose root is t, the path from a given node s to t in tr is unique. Moreover, the spanning tree implicitly specifies a set of paths that can be reached from the induced path and provides the data structure to evaluate their desirability. The rest of this section describes the neighborhood in detail. Our COP framework considers both directed and undirected graphs but, for space reasons, only undirected graphs are considered.

Rooted Spanning Trees. Given an undirected graph g and a target node $t \in V(g)$, our COP neighborhood maintains a spanning tree of g rooted at t. Moreover, since we are interested in elementary paths between a source s and a target t, the data structure also maintains the source node s and is called a rooted spanning tree (RST) over (g, s, t). An RST tr over (g, s, t) specifies a unique path from s to t in g: $path_{tr}(s) =< v_1, v_2, ..., v_k >$ in which $s = v_1, t = v_k$ and $v_{i+1} = fa_{tr}(v_i)$, $\forall i = 1, \ldots, k-1$. By maintaining RSTs for COP problems, our framework avoids an explicit representation of paths and enables the definition of an connected neighborhood that can be explored efficiently. Indeed, the tree structure directly captures the path structure from a node s to the root and simple updates to the RST (e.g., an edge replacement) will induce a new path from s to the root.

The Basic Neighborhood. We now consider the definition of our COP neighborhood. We first show how to update an RST tr over (g, s, t) to generate a new rooted spanning tree tr' over (g, s, t) which induces a new path from s to t in g: $path_{tr'}(s) \neq path_{tr}(s)$.

Given an RST over (g, s, t), an edge $e = (u, v)$ such that $e \in E(g) \setminus E(tr)$ is called a *replacing* edge of tr and we denote by $rpl(tr)$ the set of *replacing* edges of tr. An edge e' belonging to $path_{tr}(u,v)$ is called a *replacable* edge of e and we denote by $rpl(tr,e)$ the set of *replacable* edges of e. Intuitionally, a *replacing* edge e is an edge that is not in the tree tr but that can be added to tr. This edge insertion creates a cycle C and all the edges of this cycle except e are *replacable* edges of e. Let tr be an RST over (g,s,t), e a *replacing* edge of tr and e' a *replacable* edge of e. We consider the following traditional edge replacement action [1]:

1. Insert the edge $e=(u,v)$ to tr. This creates an undirected graph g' with a cycle C containing the edge e'.
2. Remove e' from g'.

The application of these two actions yields a new rooted spanning tree tr' of g, denoted $tr' = rep(tr,e',e)$. The neighborhood of tr could then be defined as

$$N(tr) = \{tr' = rep(tr,e',e) \mid e \in rpl(tr), e' \in rpl(tr,e)\}.$$

It is easy to observe that two RSTs tr_1 and tr_2 over (g,s,t) may induce the same path from s to t. For this reason, we now show how to compute a subset $N^k(tr) \subseteq N(tr)$ such that $path_{tr'}(s) \neq path_{tr}(s), \forall tr' \in N^k(tr)$.

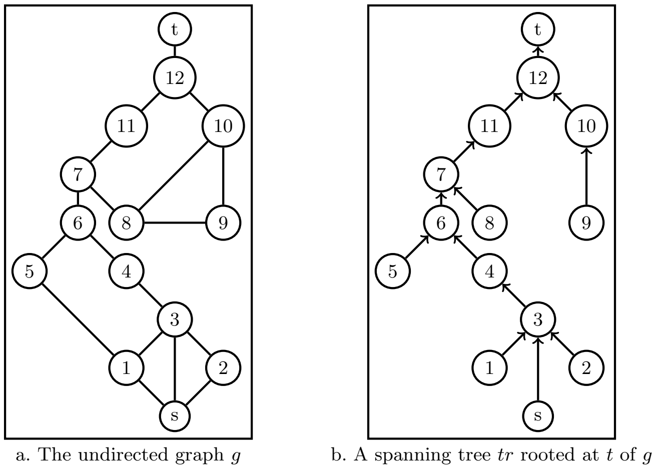
We first give some notations to be used in the following presentation. Given an RST tr over (g,s,t) and a *replacing* edge $e=(u,v)$, the nearest common ancestors of s and the two endpoints u, v of e are both located on the path from s to t. We denote by $lownca_{tr}(e,s)$ and $upnca_{tr}(e,s)$ the nearest common ancestors of s on the one hand and one of the two endpoints of e on the other hand, with the condition that $upnca_{tr}(e,s)$ dominates $lownca_{tr}(e,s)$. We denote by $low_{tr}(e,s)$, $up_{tr}(e,s)$ the endpoints of e such that $nca_{tr}(s, low_{tr}(e,s)) = lownca_{tr}(e,s)$ and $nca_{tr}(s, up_{tr}(e,s)) = upnca_{tr}(e,s)$. Figure 1 illustrates these concepts. The left part of the figure depicts the graph g and the right side depicts an RST tr over (g,s,r). Edge (8,10) is a *replacing* edge of tr; $nca_{tr}(s,10) = 12$ since 12 is the common ancestor of s and 10. $nca_{tr}(s,8) = 7$ since 7 is the common ancestor of s and 8. $lownca_{tr}((8,10)) = 7$ and $upnca_{tr}((8,10)) = 12$ because 12 Dom_{tr} 7; $low_{tr}((8,10)) = 8$; $up_{tr}((8,10)) = 10$.

We now specify the replacements that induce new path from s to t.

Proposition 1. *Let tr be an RST over (g,s,t), $e=(u,v)$ be a replacing edge of tr, let e' be a replacable edge of e, and let $tr^1 = rep(tr,e',e)$. We have that $path_{tr^1}(s) \neq path_{tr}(s)$ if and only if (1) $su \neq sv$ and (2) $e' \in path_{tr}(sv,su)$, where $su = upnca_{tr}(e,s)$ and $sv = lownca_{tr}(e,s)$.*

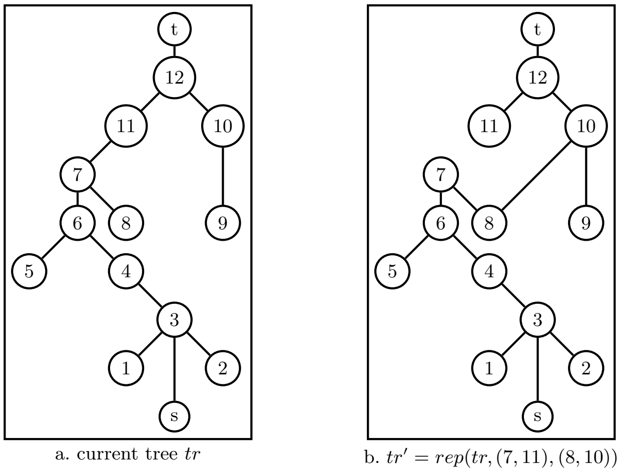
A *replacing* edge e of tr satisfying condition 1 is called a *preferred replacing* edge and a *replacable* edge e' of e in tr satisfying condition 2 is called *preferred replacable* edge of e. We denote by $prefRpl(tr)$ the set of *preferred replacing* edges of tr and by $prefRpl(tr,e)$ the set of *preferred replacable* edges of the *preferred replacing* edge e on tr. The basic COP neighborhood of an RST tr is defined as

$$N^1(tr) = \{tr' = rep(tr,e',e) \mid e \in prefRpl(tr), e' \in prefRpl(tr,e)\}.$$



a. The undirected graph g b. A spanning tree tr rooted at t of g

Fig. 1. An Example of Rooted Spanning Tree



a. current tree tr b. $tr' = rep(tr, (7, 11), (8, 10))$

Fig. 2. Ilustrating a Basic Move

The action $rep(tr, e', e)$ is called a basic move and is illustrated in Figure 2. In the current tree tr (see Figure 2a), the edge (8,10) is a *preferred replacing* edge, $nca_{tr}(s, 8) = 7$, $nca_{tr}(s, 10) = 12$, $lownca_{tr}((8, 10), s) = 7$, $upnca_{tr}((8, 10), s) = 12$, $low_{tr}((8, 10), s) = 8$ and $up_{tr}((8, 10), s) = 10$. The edges (7,11) and (11,12) are *preferred replacable* edges of (8,10) because these edges belong to $path_{tr}$ $(7, 12)$. The path induced by tr is: $<$ s, 3, 4, 6, 7, 11, 12, t $>$. The path induced by tr' is: $<$ s, 3, 4, 6, 7, 8, 10, 12, t $>$ (see Figure 2b).

Basic moves ensure that the neighborhood is connected.

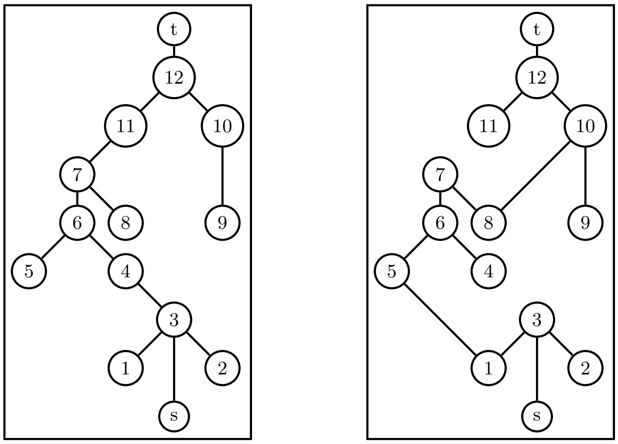
Proposition 2. *Let tr^0 be an RST over (g, t, s) and $\mathcal{P}$ be a path from s to t. An RST tr^k inducing $\mathcal{P}$ can be reached from tr^0 in $k \leq l$ basic moves, where l is the length of $\mathcal{P}$.*

Proof. The proposition is proved by showing how to generate that instance tr^k. This can be done by Algorithm 1. The idea is to travel the sequence of nodes of $\mathcal{P}$ on the current tree tr. Whenever we get stuck (we cannot go from the current node x to the next node y of $\mathcal{P}$ by an edge (x, y) on tr because (x, y) is not in tr), we change tr by replacing (x, y) by a replacable edge of (x, y) that is not traversed. The edge (x, y) in line 7 is a *replacing* edge of tr because this edge is not in tr but it is an edge of g. Line 8 chooses a *replacable* edge eo of ei that is not in S. This choice is always done because the set of *replacable* edges of ei that are not in S is not null (at least an edge $(y, fa_{tr}(y))$ belongs to this set). Line 9 performs the move that replaces the edge eo by the edge ei on tr. So Algorithm 1 always terminates and returns a rooted spanning tree tr inducing $\mathcal{P}$. Variable S (line 1) stores the set of traversed edges.

Algorithm 1. Moves

```
Input: An instance $tr^0$ of $RST$ on $(g, s, t)$ and a path $\mathcal{P}$ on $g$, $s =$
       firstNode($\mathcal{P}$), $t =$ lastNode($\mathcal{P}$)
Output: A tree inducing $\mathcal{P}$ computed by taking $k \leq l$ basic moves ($l$ is the
        length of $\mathcal{P}$)
1  $S \leftarrow \oslash$;
2  $tr \leftarrow tr^0$;
3  $x \leftarrow$ firstNode($\mathcal{P}$);
4  while $x \neq lastNode(\mathcal{P})$ do
5      $y \leftarrow$ nextNode($x, \mathcal{P}$);
6      if $(x, y) \notin E(tr)$ then
7          $ei \leftarrow (x, y)$;
8          $eo \leftarrow$ replacable edge of $ei$ that is not in $S$;
9          $tr \leftarrow rep(tr, eo, ei)$;
10     $S \leftarrow S \cup \{(x, y)\}$;
11     $x \leftarrow y$ ;
12 return $tr$;
```

Neighborhood of Independent Moves. It is possible to consider more complex moves by applying a set of independent basic moves. Two basic moves are independent if the execution of the first one does not affect the second one and vice versa. The sequence of basic moves $rep(tr, e'_1, e_1), \ldots, rep(tr, e'_k, e_k)$, denoted by $rep(tr, e'_1, e_1, e'_2, e_2, ..., e'_k, e_k)$, is defined as the application of the actions $rep(tr_j, e'_j, e_j)$, $j = 1, 2, ..., k$, where $tr_1 = tr$ and $tr_{j+1} = rep(tr_j, e'_j, e_j)$, $j = 1, 2, ..., k-1$. It is feasible if the basic moves are feasible, i.e., $e_j \in$ *prefRpl*(tr_j) and $e'_j \in$ *prefRpl*(tr_j, e_j), $\forall j = 1, 2, ..., k$.



a. The current tree tr b. $tr' = rep(tr, (7, 11), (8, 10), (3, 4), (1, 5))$

Fig. 3. Illustrating a Complex Move

Proposition 3. *Consider k basic moves $rep(tr, e'_1, e_1), \ldots, rep(tr, e'_k, e_k)$. If all possible execution sequences of these basic moves are feasible and the edges $e'_1, e_1, e'_2, e_2, ..., e'_k, e_k$ are all different, then these k basic moves are independent.*

We denote by $N^k(tr)$ the set of neighbors of tr obtained by applying k independent basic moves. The action of taking a neighbor in $N^k(tr)$ is called k-move.

It remains to find some criterion to determine whether two basic moves are independent. Given an RST tr over (g, s, t) and two *preferred replacing* edges e_1, e_2, we say that e_1 dominates e_2 in tr, denoted by e_1 Dom_{tr} e_2, if $lownca_{tr}(e_1, s)$ dominates $upnca_{tr}(e_2, s)$. Then, two *preferred replacing* edges e_1 and e_2 are independent w.r.t. tr if e_1 dominates e_2 in tr or e_2 dominates e_1 in tr.

Proposition 4. *Let tr be an RST over (g, s, t), e_1 and e_2 be two preferred replacing edges such that e_2 Dom_{tr} e_1, $e'_1 \in prefRpl(tr, e_1)$, and $e'_2 \in prefRpl(tr, e_2)$. Then, $rep(tr, e'_1, e_1)$ and $rep(tr, e'_2, e_2)$ are independent and the path induced by $rep(tr, e'_1, e_1, e'_2, e_2)$ is $path_{tr}(s, v_1)$ + $path_{tr}(u_1, v_2)$ + $path_{tr}(u_2, t)$, where + denotes path concatenation and $v_1 = low_{tr}(e_1, s)$, $u_1 = up_{tr}(e_1, s)$, $v_2 = low_{tr}(e_2, s)$, and $u_2 = up_{tr}(e_2, s)$.*

Figure 3 illustrates a complex move. In tr, two *preferred replacing* edges (1,5) and (8,10) are independent because $lownca_{tr}((8, 10), s) = 7$ which dominates $upnca_{tr}((1, 5), s) = 6$ in tr. The new path induced by tr' is: $< s$, 3 ,1, 5, 6, 7, 8, 10, 12, $t >$ which is actually the path: $path_{tr}(s, 1) + path_{tr}(5, 8) + path_{tr}(10, t)$.

4 The COP Modeling Framework

Our COP modeling framework is implemented in `COMET` as an extension of the `LS(Graph & Tree)` framework which provides graph invariants, graph

```
1. LSGraphSolver ls();
2. VarPath path(ls,g,s,t);
3. PreferredReplacingEdges prefReplacing(path);
4. PreferredReplacableEdges prefReplacable(path);
...
9. int d = MAXINT;
10. forall(ei in prefReplacing.getSet())
11.     forall(eo in prefReplacable.getSet(ei))
12.         d = min(d,C.getReplaceEdgeDelta(path,eo,ei));
```

Fig. 4. Exploring the Basic Neighborhood

constraints, and graph objectives [8]. Graph invariants maintain properties of dynamic graphs, such as the sum of weights of all the edges and the diameter of a tree, etc. Graph constraints and graph objectives are differentiable objects which maintain some properties of a dynamic graphs (for instance, the number of violations of a constraint or the value of an objective function) but also allow to determine the impact of local moves on these properties, a feature known as differentiation.

Our COP modeling framework introduces a new type of variable `VarPath` to model elementary paths. A path variable *path(g,s,t)* encapsulates an RST over (g, s, t) and may appear in a variety of constraints and objectives. For instance, `PathCostOnEdges(path,k)`, where `k` is the index of a considered weight on the edges of a graph, maintains the total weight accumulated along the path *path* from s to t, `PathEdgeDisjoint(Paths)` is a graph constraint defined over an array of paths that specifies that these paths are mutually edge-disjoint, while `MinEdgeCost(path,k)`, `MaxEdgeCost(path,k)` maintain the minimal and maximal weight of edges on the same path. `NodesVisited(path,S)` maintains the number of nodes of `S` visited by *path*. These abstractions are examples of graph objectives which are fundamental when modeling COP problems. For example, in QoS, we consider shortest path from an origin to a destination with constraints over bandwidth which is defined to be the minimum weight of edges on the specified path. As usual in CBLS, the objectives can be combined with traditional arithmetic operators (with +,-,* operators) and used in constraints expressed with relational operators.

Figure 4 illustrates the COP framework with a simple snippet to explore the basic neighborhood. Line 1 initializes a `LSGraphSolver` object `ls` which manages all the *VarGraph*, *VarPath*, graph invariants, graph constraints and graph objectives objects. Line 2 declares and initializes randomly a `VarPath` variable. This variable encapsulates an RST over (g, s, t) which evolves during the local search. `prefReplacing` and `prefReplacable` are graph invariants which maintain the set of *preferred replacing* edges and *preferred replacable* edges (lines 3-4). Lines 9–12 explore the basic neighborhood to find the best basic moves with respect to a graph constraint `C`. The `getReplaceEdgeDelta` (line 12) method returns the variation of the number of violations of `C` when the *preferred replacable* edge `eo` is replaced by the *preferred replacing* edge `ei` on the RST representing `path`.

5 Applications

5.1 The Resource Constrained Shortest Path (RCSP) Problem

The Resource constrained shortest path problem (RCSP) [3] is the problem of finding the shortest path between two vertices on a network satisfying the constraints over resources consumed along the path. There are some variations of this problem, but we first consider a simplified version introduced and evaluated in [3] over instances from the OR-Library [2]. Given a directed graph $G = (V, E)$, each arc e is associated with a length $c(e)$ and a vector $r(e)$ of resources consumed in traversing the arc e. Given a source node s, a destination node t and two vectors L, U of resources corresponding to the minimum and maximum amount that can be used on the chosen path (i.e., a lower and an upper limit on the resources consumed on the path). The length of a path $\mathcal{P}$ is defined as $f(\mathcal{P}) = \sum_{e \in \mathcal{P}} c(e)$. The resources consumed in traversing $\mathcal{P}$ is defined as $r(\mathcal{P}) = \sum_{e \in \mathcal{P}} r(e)$ The formulation of RCSP is then given by:

min $f(\mathcal{P})$
s.t. $L \leq r(\mathcal{P}) \leq U$
$\mathcal{P}$ is elementary path from s to t on G.

The RCSP problem with only constraints on the maximum resources consumed is also considered in [5,7]. The algorithm based on Lagrangian relaxation and enumeration from [5] and the vertex-labeling algorithm combining with several preprocessing techniques in [7] are known to be state-of-the-art for this problem. We give a Tabu search model (RCSP_TABU) for solving the RCSP problem considering both constraints of minimum and maximum resources consumed. This problem is rather pure and does not highlight the benefits of our framework but it is interesting as a starting point and a basis for comparison.

The RCSP Modeling. The model using the COP framework is as follows:

```
void stateModel{
    LSGraphSolver ls();
    VarPath path(ls,g,s,t);
    range Resources = 1..K;
    GraphObjective go[Resources];
    forall(k in Resources)
        go[k] = PathCostOnEdges(path,k);
    PathCostOnEdges len(path,0);
    GraphConstraintSystem gcs(ls);
    forall(k in Resources){
        gcs.post(L[k] <= go[k]);
        gcs.post(go[k] <= U[k]);
    }
    gcs.close();
    GraphObjectiveCombinator goc(ls);
    goc.add(len,1);
    goc.add(gcs,1000);
17.  goc.close();
18.  PreferredReplacingEdges prefReplacing(path);
19.  PreferredReplacableEdges prefReplacable(path);
20.  ls.close();
21.}
```

Line 1 declares a `LSGraphSolver ls`. A `VarPath` variable representing an *RST* over (g, s, t) is declared and initialized in line 2. Lines 3–6 create `K` graph objectives representing resources consumed in traversing the path from `s` to `t` where `PathCostOnEdges(ls,path,k)` (line 6) is a modeling abstraction representing the total weights of type k accumulated along the path from `s` to `t`[1]. Variable `len` represents the length of the path from `s` to `t` (line 7). Lines 9–12 initialize and post to the `GraphConstraintSystem gcs` (line 8) the constraints over resources consumed in traversing the path from `s` to `t`.

In this model, we combine the graph constraint `gcs` with coefficient 1000 and the graph objective `len` with coefficient 1 in a global `GraphObjectiveCombinator goc` to be minimized (lines 14–17). We introduce two graph invariants `prefReplacing` and `prefReplacable` to represent the set of *preferred replacing* edges of `path` and the sets *preferred replacable* edges of all *preferred replacing* edges of `path` (lines 18–19).

The search procedure not described here is based on tabu search on the neighborhood N^2 because the exploration on basic neighborhood N^1 gave poor results. At each local move, we consider the best neighbor which minimizes `goc`. We take this neighbor if it improves the current solution. Otherwise, a random neighbor which is not tabu will be taken. Solutions are made tabu by putting the edges appearing in the replacements into two tabu lists: one list for storing the edges to be added and another one for storing the edges to be removed. The length of these lists are set to be the number of vertices divided by 5.

Experimental Results. We compare the model with the algorithm described in [3] over the benchmarks from OR-Library [2] and over a modification of these benchmarks. The original benchmarks contains 24 instances whose orders vary from 100 to 500 and the number of resources are 1 or 10. Instance 14 does not have any feasible solution. On instances from the original benchmarks, the upper limit values of the resources consumed is small such that the pruning techniques used in [3] reduce substantially the problem sizes. The Algorithm in [3] is thus particularly efficient on these instances. The second benchmark is generated by modifying some upper limit values as follows. We chose the large instances of order 500 (instances numbered from 17 to 24). For instances numbered 17–20 (number of resources is equal to 1), we slightly decrease the upper limit values. For instances numbered 21–24 (number of resources is equal to 10), we multiply some upper limit values by 10. In order to compare the model with the algorithm from [3], we implemented that algorithm in `COMET` (denoted by RCSP_BEA) following the description in the paper.

[1] Each arc has multiple properties, the property indexed by 0 is the length and properties indexed from 1 to `k` are resources consumed in traversing this arc.

Table 1. First Experimental Results of RCSP_TABU: Original Instances

instances	opt	t*	min	max	%	avr_t	min_t	max_t	std_dev	min'	max'	%
rcsp1	131	0.62	131	131	100	**0.26**	**0.25**	**0.28**	0.01	131	131	100
rcsp2	131	0.05	131	131	100	0.26	0.24	0.26	0.01	131	131	100
rcsp3	2	0.60	2	2	100	2.11	**0.48**	5.82	1.29	2	2	100
rcsp4	2	0.09	2	2	100	3.82	0.82	10.19	2.96	2	7	100
rcsp5	100	0.84	100	100	100	**0.83**	**0.6**	0.97	0.1	100	100	100
rcsp6	100	0.84	100	100	100	**0.75**	**0.6**	0.95	0.1	100	119	100
rcsp7	6	1.40	6	6	100	21.44	3.48	55.08	15.17	6	9	100
rcsp8	14	1.58	14	14	100	51.28	1.22	183.94	47.03	14	∞	80
rcsp9	420	0.04	420	420	100	122.5	2.14	483.43	115.26	420	∞	60
rcsp10	420	0.03	420	420	100	71.04	4.14	416.6	92.89	420	∞	90
rcsp11	6	0.11	6	6	100	7.75	1.84	18.44	3.81	6	7	100
rcsp12	6	0.09	6	6	100	9.12	2.34	25.12	6.52	6	6	100
rcsp13	448	0.44	448	448	100	90.06	7.94	276.02	66.81	448	∞	70
rcsp14	-	-	-	-	-	-	-	-	-	-	-	-
rcsp15	9	9.28	9	9	100	93.43	31.89	284.25	60.53	9	∞	70
rcsp16	17	8.84	17	17	100	279.89	33.43	1049.57	253.27	17	∞	30
rcsp17	652	55.91	652	652	100	56.64	**19.9**	106.07	23.65	652	652	100
rcsp18	652	56.45	652	652	100	57.27	**25.61**	116.98	22.56	652	652	100
rcsp19	6	0.59	6	6	100	28.15	7.92	66.72	13.32	6	6	100
rcsp20	6	1.07	6	6	100	46.85	12.79	118.63	31.08	6	15	100
rcsp21	858	3.20	858	858	100	242.3	61.68	679.96	190.7	858	∞	50
rcsp22	858	1.86	858	858	100	294.94	108.41	827.04	186.13	858	∞	50
rcsp23	4	50.74	4	4	100	280.36	**11.92**	1053.61	279.03	4	∞	90
rcsp24	5	54.05	5	∞	80	719.92	**24.13**	1737.43	574.94	5	∞	20

The RCSP_TABU model is executed 20 times with a time window of 30 minutes for each instance. The first experimental results are shown in Table 1 (columns 1–10). The structure of the table is described as follows. Column 2 is the optimal value of the objective function and column 3 is the execution time (in seconds) of the RCSP_BEA model. Columns 4 and 5 present the minimal and maximal value of the objective function in 20 runs of RCSP_TABU. Column 6 is the rate of finding the optimal solution. Columns 7–10 show the average, the minimal, maximal, and the standard deviation of the execution time necessary to find the optimal solution. The results show that the RCSP_TABU model found the optimal solutions in all 20 runs over all instances except the instance rcsp24 (only 15 runs found the optimal solution) and the instance rcsp14 (a feasible solution does not exist). The table also shows that on the original benchmark, the RCSP_BEA model found optimal solutions faster than our RCSP_TABU model except for some instances (see lines 1, 3, 5, 6, 17, 18, 23, 24).

The experimental results for the second set of benchmarks are shown on Table 2 (Columns 11–13 should be ignored for now). Column 2 presents the execution times of the RCSP_BEA model for finding optimal solutions. The remaining columns report results of the RCSP_TABU model. Columns 3–6 show the average, minimal, maximal, and standard deviation of execution times to find optimal

Table 2. Second Experimental Results of RCSP_TABU: More Difficult Instances

instance	t*	avr_t	min_t	max_t	std_dev	%solved	avr_it
rcsp17_01	57.58	**54.16**	**26.18**	159.76	30.63	100	12.95
rcsp17_02	58.88	**56.26**	**30.1**	138.43	26.95	100	18.25
rcsp18_01	56.57	57.4	**27.35**	120.06	23.19	100	17.55
rcsp18_02	56.64	60.32	**16.98**	266.61	52.56	100	16.95
rcsp19_01	75.56	**40.65**	**11.78**	108.21	25.02	100	41.05
rcsp19_02	59.36	**56.9**	**8.74**	134.95	35.54	100	56.95
rcsp20_01	74.98	**49.32**	**6.66**	136.49	38.17	100	57.4
rcsp20_02	58.34	**51.6**	**8.72**	177.01	36.72	100	55.9
rcsp21_01	164.5	**72.91**	**31.74**	108.89	22.07	100	11.15
rcsp21_02	154.67	**63.88**	**34.02**	128.04	21.13	100	12.15
rcsp22_01	157.6	**73.85**	**28.28**	118.14	24.48	100	11.1
rcsp22_02	150.95	**72.42**	**33.07**	180.05	30.97	100	11.7
rcsp23_01	130.08	**76.99**	**21.58**	216.21	51.99	100	38.9
rcsp23_02	129.34	**70.3**	**21.64**	250.5	54.97	100	30.65
rcsp24_01	129.09	153.04	**34.35**	418.23	113.86	100	75.7
rcsp24_02	129.44	**94.54**	**24.13**	402.08	80.31	100	49.25

solutions. Column 7 present the percentage for finding optimal solutions. The final column reports the average of the number of moves. Experimental results show that our RCSP_TABU model found optimal solutions faster than the RCSP_BEA model in most cases. The reason is that, on these instances, the constraints over resources consumed are easy to satisfy but the search space is much larger. The reduction techniques in [3] do not reduce the search space much and the search procedure of the RCSP_BEA model is thus much slower.

5.2 The RCSP Problem with Multiple Choice Side Constraints

In order to illustrate the flexibility of our modeling approach, we consider the RCSP problem with the following side constraint over nodes on the path: The set of nodes V is partitioned into Q subsets $S_1, S_2, ..., S_Q$ and the path is required to visit at most one node from each subset. This constraint arises when solving a subproblem in a branch-and-price method for a variation of the vehicle routing problem, known as Multi-Resource Routing Problem (MRRP) [9]. This problem cannot be solved with RCSP_BEA without a substantial programming effort.

The Modeling The MC_RCSP problem is modeled by extending the RCSP model: the Multiple Choice constraints are stated and posted into the `GraphConstraintSystem gcs`. This can be done by simply adding the following snippet to the RCSP model:

```
1. GraphObjective nv[1..Q];
2. forall(q in 1..Q){
3.   nv[q] = NodesVisited(path,S[q]);
4.   gcs.post(nv[q] <= 1);
5. }
```

where `NodesVisited(ls,path,S[q])` is an abstraction representing the number of nodes in `S[q]` visited by the path. Notice that such a side constraint cannot be handled by the algorithm of [3].

Experimental Results. We experiment the model over the benchmark from the OR-library where the set of subsets $S_1, S_2, ..., S_Q$ is generated as follows. We take a random feasible solution to the RCSP problem which is an elementary path $v_1, v_2, ..., v_q$ satisfying the resource constraints. Then, we partition V into $Q = 3 * q$ sets $S_1, S_2, ..., S_Q$ where $v_j \in S_j, \forall j \in \{1, 2, ..., q\}$ and the size differences are at most one. This ensures that there exists at least one feasible solution $v_1, v_2, ..., v_q$ to the MC_RCSP problem. The model is executed 10 times with 10 minutes of time window for each instance. Experimental results are shown in Table 1 (columns 11–13) where column 13 presents the rate of finding feasible solutions. Columns 11–12 present the minimal and maximal value of the best solution in different executions. In most cases, the rate for finding feasible solutions is high except instances 16 and 24. In some cases, the model finds optimal solutions.

5.3 The EDP Problem

We are given an undirected graph $G = (V, E)$ and a set $T = \{< s_i, t_i > | \ s_i \neq t_i \in V\}$ representing a list of commodities ($\sharp T = k$). The EDP problem consists of finding a maximal cardinality set of mutually edge-disjoint paths from s_i to t_i on G ($< s_i, t_i > \in T$). In [4], a Multi-start Simple Greedy and an ACO algorithms are presented. The ACO is known to be state-of-the-art for this problem. We propose a local search algorithm using the following model:

```
void stateModel{
1.   LSGraphSolver ls();
2.   VarPath path[j in 1..k](ls,g,s[j],t[j]);
3.   PathEdgeDisjoint ed(path);
4.   ls.close();
5.}
```

where line 2 initializes an array of `k` `VarPath`s representing `k` paths between commodities. The edge-disjoint constraint `ed` is defined over paths from `s[i]` to `t[i]` and is stated in line 3.

In [4], the following criterion is introduced which quantifies the degree of non-disjointness of a solution $S = \{P_1, P_2, ...P_k\}$ (P_j is a path from s_j to t_j):

$$C(S) = \sum_{e \in E} (max\{0, \sum_{P_j \in S} \rho^j(S, e) - 1\})$$

where $\rho^j(S, e) = 1$, if $e \in P_j$ and $\rho^j(S, e) = 0$ otherwise. The number of violations of the $PathEdgeDisjoint(P_1, P_2, ..., P_k)$ constraint in the framework is defined to be $C(\{P_1, P_2, ..., P_k\})$ and the proposed local search algorithm tries to minimize this criterion.

From a solution which is normally a set of k non-disjoint path, a feasible solution to the EDP problem can be extracted by iteratively removing the path

Table 3. Experimental results for the EDP problem

instance	com.	MSGA		ACO		Local search	
		$\overline{q}$	$\overline{t}$	$\overline{q}$	$\overline{t}$	$\overline{q}$	$\overline{t}$
mesh25x25.bb	62	36.95	546.854	31.1	880.551	**38.85**	1165.47
	156	44.65	863.007	47.5	965.921	**55.5**	1082.78
	250	50.5	672.962	60.5	972.396	**67.95**	967.087
mesh15x15.bb	22	20.55	517.601	18.6	500.812	**21**	384.828
	56	27.15	651.27	28.35	988.782	**30.3**	485.693
	90	31	797.534	34.55	746.96	**36.05**	435.308
bl-wr2-wht2.10-50.rand.bb	50	18.7	688.651	19.6	201.235	**20.05**	228.382
	125	27.2	643.51	31.15	338.446	**31.2**	241.047
	200	36.6	625.138	41.55	164.783	**41.7**	202.186
bl-wr2-wht2.10-50.sdeg.bb	50	18.65	470.26	19.75	223.396	**20.1**	311.887
	125	28.1	662.916	31.55	163.151	**31.85**	357.25
	200	33.3	487.999	38.05	217.362	**38.25**	178.417

which has most edges in common with other paths until all remaining paths are mutually edge-disjoint as suggested in [4]. In our local search model, we extend this idea by taking a simple greedy algorithm over the remaining paths after that extraction procedure in hope of improving the number of edge-disjoint paths.

Experimental Results. For the experimentation, we re-implemented in COMET the Multi-start Greedy Algorithm (MSGA) and the ACO (the extended version) algorithm described in [4] and compare them with our local search model. The instances in the original paper [4] are not available (except some graphs). As a result, we use the instance generator described in [4] and generate new instances as follows. We take 4 graphs from [4]. For each graph, we generate randomly different sets of commodities with different sizes depending on the size of the graph: for each graph of size n, we generate randomly 20 instances with 0.10*n, 0.25*n and 0.40*n commodities. In total, we have 240 problems instances. Due to the high complexity of the problem, we execute each problem instance once with a time limit of 30 minutes for each execution. Experimental results are shown in Table 3. The time window for the MSGA and the ACO algorithms are also 30 minutes. The Table reports the average values of the objective function and the average execution times for obtaining the best solutions of 20 instances (a graph $G = (V, E)$ and a set of $r * |V|$ commodities, $r = 0.10, 0.25, 0.40$). Table 3 shows that our local search model gives very competitive results. It finds better solutions than MGSA in 217/240 instances, while MSGA find better solutions in 4/240 instances. On the other hand, in comparison with the ACO model, our model finds better solutions in 144/240 instances, while the ACO model find better solutions in 11/240 instances. This clearly demonstrates the potential benefits of our COP framework, from a modeling and computational standpoint.

6 Conclusion

This paper considered Constrained Optimum Path (COP) problems which arise in many real-life applications. It proposes a domain-specific constraint-based local search (CBLS) framework for COP applications, enabling models to be high level, compositional, and extensible and allowing for a clear separation between model and search. The key technical contribution to support the COP framework is a novel neighborhood based on a rooted spanning tree that implicitly defines a path between the source and the target and its neighbors, and provides an efficient data structure for differentiation. The paper proved that the neighborhood obtained by swapping edges in this tree is connected and presented a larger neighborhood involving multiple independent moves. The COP framework, implemented in `COMET`, was applied to Resource Constrained Shortest Path problems (with and without side constraints) and to the edge-disjoint paths problem. Computational results showed the potential significance of the approach, both from a modeling and computational standpoint.

Acknowledgments. We would like to thank Maria José Blesa Aguilera who has kindly provided some graphs for the experimentation. This research is partially supported by the Interuniversity Attraction Poles Programme (Belgian State, Belgian Science Policy) and the FRFC project 2.4504.10 of the Belgian FNRS (National Fund for Scientific Research).

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall, New Jersey (1993)
2. Beasley, J.E.: Or-library, `http://people.brunel.ac.uk/~mastjjb/jeb/info.html`
3. Beasley, J.E., Christofides, N.: An algorithm for the resource constrained shortest path problem. Network 19, 379–394 (1989)
4. Blesa, M., Blum, C.: Finding edge-disjoint paths in networks: An ant colony optimization algorithm. Journal of Mathematical Modelling and Algorithms 6(3), 361–391 (2007)
5. Carlyle, W.M., Wood, R.K.: Lagrangian relaxation and enumeration for solving constrained shortest-path problems. In: Proceedings of the 38th Annual ORSNZ Conference (2003)
6. Clímaco, J.C.N., Craveirinha, J.M.F., Pascoal, M.M.B.: A bicriterion approach for routing problems in multimedia networks. Network 41, 206–220 (2003)
7. Dumitrescu, I., Boland, N.: Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. Networks 42, 135–153 (2003)
8. Pham, Q.D., Deville, Y., Van Hentenryck, P.: Ls(graph & tree): A local search framework for constraint optimization on graphs and trees. In: Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC 2009) (2009)
9. Smilowitz, K.: Multi-resource routing with flexible tasks: an application in drayage operation. IIE Transactions 38(7), 555–568 (2006)
10. Van Hentenryck, P., Michel, L.: Constraint-based local search. The MIT Press, London (2005)

Stochastic Constraint Programming by Neuroevolution with Filtering[⋆]

Steve D. Prestwich[1], S. Armagan Tarim[2], Roberto Rossi[3], and Brahim Hnich[4]

[1] Cork Constraint Computation Centre, University College Cork, Ireland
[2] Department of Management, Hacettepe University, Ankara, Turkey
[3] Logistics, Decision and Information Sciences Group, Wageningen UR, The Netherlands
[4] Faculty of Computer Science, Izmir University of Economics, Turkey
s.prestwich@cs.ucc.ie, armtar@yahoo.com,
roberto.rossi@wur.nl, brahim.hnich@ieu.edu.tr

Abstract. Stochastic Constraint Programming is an extension of Constraint Programming for modelling and solving combinatorial problems involving uncertainty. A solution to such a problem is a policy tree that specifies decision variable assignments in each scenario. Several complete solution methods have been proposed, but the authors recently showed that an incomplete approach based on neuroevolution is more scalable. In this paper we hybridise neuroevolution with constraint filtering on hard constraints, and show both theoretically and empirically that the hybrid can learn more complex policies more quickly.

1 Introduction

Stochastic Constraint Programming (SCP) is an extension of Constraint Programming (CP) designed to model and solve complex problems involving uncertainty and probability [7]. An m-stage SCSP is defined as a tuple $(V, S, D, P, C, \theta, L)$ where V is a set of decision variables, S a set of stochastic variables, D a function mapping each element of $V \cup S$ to a domain of values, P a function mapping each variable in S to a probability distribution, C a set of constraints on $V \cup S$, θ a function mapping each constraint in C to a threshold value $\theta \in (0, 1]$, and $L = [\langle V_1, S_1\rangle, \ldots, \langle V_m, S_m\rangle]$ a list of *decision stages* such that the V_i partition V and the S_i partition S. Each constraint must contain at least one V variable, a constraint $h \in C$ containing only V variables is a *hard constraint* with threshold $\theta(h) = 1$, and one containing at least one S variable is a *chance constraint*.

To solve an SCSP we must find a *policy tree* of decisions, in which each node represents a value chosen for a decision variable, and each arc from a node represents the value assigned to a stochastic variable. Each path in the tree represents a different possible *scenario* and the values assigned to decision variables in that scenario. A *satisfying*

[⋆] S. A. Tarim and B. Hnich are supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No. SOBAG-108K027. S. A. Tarim is also supported by Hacettepe University (BAB). A version of this algorithm will used to further research in risk management as part of a collaboration with IBM Research, with partial support from the Irish Development Association and IRCSET.

policy tree is a policy tree in which each chance constraint is satisfied with respect to the tree. A chance constraint $h \in C$ is satisfied with respect to a policy tree if it is satisfied under some fraction $\phi \geq \theta(h)$ of all possible paths in the tree.

Most current SCP approaches are complete and do not seem practicable for large multi-stage problems, but the authors recently proposed a more scalable method called *Evolved Parameterised Policies* (EPP) [3]. In this paper we hybridise EPP with constraint filtering, and show theoretically and empirically that this improves learning. An upcoming technical report will contain details omitted from this short paper.

2 Filtered Evolved Parameterised Policies

EPP [3] uses an evolutionary algorithm to find an artificial neural network (ANN) whose input is a representation of a policy tree node, and whose output is a domain value for the decision variable to be assigned at that node. The ANN describes a *policy function*: it is applied whenever a decision variable is to be assigned, and can be used to represent or recreate a policy tree. The evolutionary fitness function penalises chance constraint violations, and is designed to be optimal for ANNs representing satisfying policy trees. In experiments on random SCSPs, EPP was orders of magnitude faster than state-of-the-art complete algorithms [3]. Because it evolves an ANN it is classed as a *neuroevolutionary* method (see for example [6]).

A drawback with EPP is that it treats hard constraints in the same way as chance constraints. This is not incorrect, but a problem containing many hard constraints may require a complex ANN with more parameters to tune, leading to longer run times. We now describe a constraint-based technique for the special case of finite domain SCSPs that allows more complex policies to be learned by simpler ANNs.

We modify EPP so that the ANN output is not used to compute a decision variable value directly, but instead to compute a *recommended value*. As we assign values to the decision and stochastic variables under some scenario ω, we apply constraint filtering algorithms using only the hard constraints, which may remove values from both decision and stochastic variable domains. If domain wipe-out occurs on any decision or stochastic variable then we stop assigning variables under ω and every constraint is artificially considered to be violated in ω; otherwise we continue. On assigning a stochastic variable s we choose $\omega(s)$, but if $\omega(s)$ has been removed from $\text{dom}(s)$ then we stop assigning variables under ω and every constraint h is artificially considered to be violated in ω; otherwise we continue. On assigning a decision variable x we compute the recommended value then choose the first remaining domain value after it in cyclic order. For example suppose that initially $\text{dom}(x) = \{1, 2, 3, 4, 5\}$ but this has been reduced to $\{2, 4\}$, and the recommended value is 5. This value is no longer in $\text{dom}(x)$ so we choose the cyclically next remaining value 2. If all variables are successfully assigned in ω then we check by inspection whether each constraint is violated or satisfied.

Some points should be clarified here. Firstly, it might be suspected that filtering a stochastic variable domain violates the principle that these variables are randomly assigned. But stochastic variables are assigned values from their *unfiltered* domains. Secondly, the value assigned to a decision variable must depend only upon the values assigned to stochastic variables occurring *earlier* in the stage structure. Does filtering

Constraints:
$c_1 : \Pr\{x = s \oplus t\} = 1$
Decision variables:
$x \in \{0, 1\}$
Stochastic variables:
$s, t \in \{0, 1\}$
Stage structure:
$V_1 = \emptyset \quad S_1 = \{s, t\}$
$V_2 = \{x\} \quad S_2 = \emptyset$
$L = [\langle V_1, S_1 \rangle, \langle V_2, S_2 \rangle]$

Fig. 1. SCSP used in Proposition 1

the domains of stochastic variables that occur *later* violate this principle? No: constraint filtering makes no assumptions on the values of unassigned variables, it only tells us that assigning a value to a decision variable will inevitably lead to a hard constraint violation. Thirdly, we consider all constraints to be violated if either domain wipe-out occurs, or if the selected value for a stochastic variable has been removed earlier by filtering. This might appear to make the evolutionary fitness function incorrect. But both these cases correspond to hard constraint violations, and considering constraints to be violated in this way is similar to using a penalty function in a genetic or local search algorithm: it only affects the objective function value for non-solutions.

We call the modified method *Filtered Evolved Parameterised Policies* (FEPP) and now state two useful properties.

Proposition 1. *FEPP can learn more policies than EPP with a given ANN.*

Proof sketch. We can show that any policy that can be learned by EPP can also be learned by FEPP. Conversely, we show by example that there exists an SCSP that can be solved by FEPP but not by EPP using a given ANN. Suppose that the ANN is a single *perceptron* [2] whose inputs are the s and t values and whose output is used to select a domain value for x, the SCSP is as shown in Figure 1, and FEPP enforces arc consistency. A single perceptron cannot learn the $\oplus$ (exclusive-OR) function [2] so EPP cannot solve the SCSP. But arc consistency removes the incorrect value from dom(x) so FEPP makes the correct assignment irrespective of the ANN. □

Proposition 2. *Increasing the level of consistency increases the set of policies that can be learned by FEPP with a given ANN.*

Proof sketch. We can show that any policy that can be learned by FEPP with a given ANN and filtering algorithm $\mathcal{A}$ can also be learned with a stronger filtering algorithm $\mathcal{B}$. Conversely, we show by example that there exists an SCSP, an ANN, and filtering algorithms $\mathcal{A}$ and $\mathcal{B}$, such that the SCSP can be solved by FEPP with $\mathcal{B}$ but not $\mathcal{A}$. Let the SCSP be as shown in Figure 2, $\mathcal{A}$ enforce pairwise arc consistency on the disequality constraints comprising c_2, $\mathcal{B}$ enforce hyper-arc consistency on c_2 using the algorithm of [5], and both $\mathcal{A}$ and $\mathcal{B}$ enforce arc consistency on c_1. In any satisfying policy $x = s \oplus t$. The proof rests on the fact that $\mathcal{B}$ reduces dom(x) to $\{0, 1\}$ before search begins so $\oplus$

Constraints:
$c_1 : \Pr\{x < 2 \rightarrow x = s \oplus t\} = 1$
$c_2 : \Pr\{\text{alldifferent}(x, y, u)\} = 1$
Decision variables:
$x \in \{0, 1, 2, 3\}$
$y \in \{2, 3\}$
Stochastic variables:
$s, t \in \{0, 1\}$
$u \in \{2, 3\}$
Stage structure:
$V_1 = \emptyset \quad S_1 = \{s, t\}$
$V_2 = \{x\} \quad S_2 = \{u\}$
$V_3 = \{y\} \quad S_3 = \emptyset$
$L = [\langle V_1, S_1\rangle, \langle V_2, S_2\rangle, \langle V_3, S_3\rangle]$

Fig. 2. SCSP used in Proposition 2

can immediately be enforced. FEPP under $\mathcal{A}$ cannot do this so it is forced to learn $\oplus$, which is impossible for a perceptron. □

Thus FEPP can potentially exploit advanced CP techniques such as global constraints. We state without proof two further propositions.

Proposition 3. *The optimisation problem representing an SCSP has more solutions under FEPP than under EPP, with a given ANN.*

A *solution* here is a set of parameter values for the ANN that represents a satisfying policy tree for the SCSP.

Proposition 4. *The optimisation problem representing an SCSP has more solutions under FEPP if the level of consistency is increased, with a given ANN.*

So even where FEPP has the same learning ability as EPP, it may be more efficient because it solves an optimisation problem with more solutions. Increasing the number of solutions is not guaranteed to make the problem easier to solve, especially as filtering incurs a runtime overhead, but it may do so.

3 Experiments

We now test two hypotheses: does filtering enable an ANN to learn more complex policies in practice as well as in theory (proposition 1)? And where a policy can be learned without filtering, does filtering speed up learning (as we hope is implied by proposition 3)? For our experiments we use Quantified Boolean Formula (QBF) instances. QBF and SCSP are closely related as there is a simple mapping from QBF to Stochastic Boolean Satisfiability, which is a special case of SCSP [1]. QBF-as-SCSP is an interesting test for FEPP because *all* its constraints are hard.

We have implemented a prototype FEPP using a weak form of constraint filtering called *backchecking*. We use the same ANN as in [3]: a *periodic perceptron* [4], which

Table 1. Results on QBF instances transformed to SCSPs

instance	EPP	FEPP
cnt01	0.9	0.03
impl02	2.9	0.02
impl04	—	8.8
TOILET2.1.iv.4	—	31
toilet_a_02_01.4	—	9.5
tree-exa10-10	—	4.0

has been shown to learn faster and require fewer weights than a standard perceptron. Results for EPP and FEPP are shown in Table 1, both tuned roughly optimally to each instance. All times were obtained on a 2.8 GHz Pentium (R) 4 with 512 MB RAM and are medians of 30 runs. "—" indicates that the problem was never solved despite multiple runs with different EPP parameter settings. These preliminary results support both our hypotheses: there are problems that can be solved by FEPP but not (as far as we can tell) by EPP; and where both can solve a problem FEPP is faster. So far we have found no QBF instance on which EPP beats FEPP.

4 Conclusion

FEPP is a true hybrid of neuroevolution and constraint programming, able to benefit from improvements to its evolutionary algorithm, its neural network and its filtering algorithms. In future work we will work on all three of these aspects and test FEPP on real-world optimisation problems involving uncertainty.

References

1. Majercik, S.M.: Stochastic Boolean Satisfiability. In: Handbook of Satisfiability, ch. 27, pp. 887–925. IOS Press, Amsterdam (2009)
2. Minsky, M., Papert, S.: Perceptrons: An Introduction to Computational Geometry. The MIT Press, Cambridge (1972)
3. Prestwich, S.D., Tarim, S.A., Rossi, R., Hnich, B.: Evolving Parameterised Policies for Stochastic Constraint Programming. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 684–691. Springer, Heidelberg (2009)
4. Racca, R.: Can Periodic Perceptrons Replace Multi-Layer Perceptrons? Pattern Recognition Letters 21, 1019–1025 (2000)
5. Régin, J.-C.: A Filtering Algorithm for Constraints of Difference in CSPs. In: 12th National Conference on Artificial Intelligence, pp. 362–367. AAAI Press, Menlo Park (1994)
6. Stanley, K.O., Miikkulainen, R.: A Taxonomy for Artificial Embryogeny. Artificial Life 9(2), 93–130 (2003)
7. Walsh, T.: Stochastic Constraint Programming. In: 15th European Conference on Artificial Intelligence (2002)

The Weighted Spanning Tree Constraint Revisited[⋆]

Jean-Charles Régin[1], Louis-Martin Rousseau[2], Michel Rueher[1], and Willem-Jan van Hoeve[3]

[1] I3S, CNRS, University of Nice-Sophia Antipolis
[2] CIRRELT, University of Montreal
[3] Tepper School of Business, Carnegie Mellon University

1 Introduction

The *weighted spanning tree constraint*, or *wst*-constraint, is defined on an edge-weighted graph G and a value K. It states that G admits a spanning tree with weight at most K [3, 4]. It can be applied to network design problems as well as routing problems, in which it serves as a relaxation. In this work, we assume that we can represent the mandatory and possible edges that can belong to a solution to the *wst*-constraint, e.g., using a subset-bound set variable as in [3].

Dooms and Katriel [3] consider a version of the *wst*-constraint in which the weights of the edges are also variable. They propose several filtering algorithms, including one for the version of the *wst*-constraint that we consider in this paper. Subsequently, a more practical and incremental filtering algorithm for this constraint was proposed by Régin [4].

In this work, we extend the algorithm of Régin [4] in several ways. First, we revisit the computation of the 'replacement cost' of tree edges, and present an algorithm with an almost linear time complexity. Second, we take mandatory edges into account; that is, edges that belong to every spanning tree having a weight at most K or that are imposed by the user. Third, we discuss the incremental behavior of the algorithms when mandatory edges are introduced.

2 Existing Approaches

The task of propagating the *wst*-constraint consists of a check for consistency, the removal (filtering) of inconsistent edges from the domain of possible edges, and potentially fixing edges that must belong to every solution. An important practical aspect is the incrementality of the algorithms, i.e., efficiently re-using data structures and solutions from one propagation event to the next. The consistency of the *wst*-constraint can easily by verified by finding a minimum-spanning tree in the graph, using a classical method such as Prim's algorithm or Kruskal's

[⋆] This work was partially supported by the European Community's 7th Framework Programme (FP7/2007-2013). It was started when L.-M. Rousseau and W.-J. van Hoeve were visiting the University of Nice-Sophia Antipolis (June/July 2009).

algorithm. Assuming that the edges are sorted by non-decreasing weight, this can be done in almost linear time [1]. Identifying inconsistent edges (that cannot participate in a spanning tree of weight at most K) is more involved, however. Dooms and Katriel [3] observed that inconsistent edges can be detected as follows [5]. Let T be a minimum spanning tree, and let (i, j) be a non-tree edge that we wish to evaluate. We now find the maximum-weight edge on the unique i-j path in T. If replacing that maximum-weight edge with (i, j) yields a tree of weight more than K, (i, j) is inconsistent. Similar reasoning can be applied to determine whether a tree edge is mandatory, i.e., when replacing it would yield always a tree of weight more than K [3]. Therefore, the detection of inconsistent and mandatory edges amounts to computing the 'replacement cost' of the edges. Régin [4] also applies the replacement cost for non-tree edges to detect inconsistent edges, but tree edges (and mandatory edges) were not considered.

Several algorithms have been proposed to compute the replacement cost of the edges, for example by Tarjan [5] and Dixon, Rauch, and Tarjan [2]. These algorithms allow to compute all replacement costs in time $O(m\alpha(m, n))$ on a graph with n nodes and m edges, where $\alpha(m, n)$ is the inverse Ackermann function stemming from the complexity of the 'union-find' algorithm [6]. Other approaches, such as those referenced by [3] are based on (or resemble) the algorithms of [5] or [2]. Even though these algorithms allow to find the replacement costs in almost linear time theoretically, the added complexity may not offset the potential savings in practice, as argued by Tarjan [5]. Moreover, it is not obvious how to apply the algorithms incrementally. Therefore, Régin proposed a different algorithm running in $O(n + m + n \log n)$ time [4]. We next briefly describe the main components of this algorithm for later use.

Régin [4] applies Kruskal's algorithm to find a minimum spanning tree. That is, we start from a forest consisting of all nodes in the graph. We then successively add edges, whereby each added edge joins two separate trees. We ensure that the next selected edge has minimum weight among all edges whose extremities are not in the same tree. We use a so-called *ccTree* ('connected component tree') to represent these merges. The leaves of the ccTree are the original graph nodes, while the internal nodes of the ccTree represent the merging of two trees (or connected components), defined in the order in which the edges were added to the tree. An internal node thus represents the edge with which two components have been merged; see Figure 1a and 1b for an example. Therefore, the ccTree contains $n - 1$ internal nodes, where n is the number of nodes in the graph. The computation of the replacement cost of a non-tree edge (i, j) can now be done by finding the lowest common ancestor (LCA) of nodes i and j in the ccTree: the weight of (i, j) minus the weight of the edge corresponding to the LCA is exactly the replacement cost of (i, j). We refer to [4] for further details.

3 Computing the Replacement Cost of Tree Edges

We next present an algorithm that computes the replacement costs of tree edges in time $O(m\alpha(m, n))$. This is the same time complexity as the algorithm proposed by Tarjan [5]. We note that the latter algorithm follows as a corollary from a generic (and relatively complex) algorithm presented in [5]. Our contribution is a description of a more practical algorithm, specific to the problem of computing replacement costs, having the same time complexity. We will apply the algorithm to detect mandatory edges.

Let $G = (V, E)$ be the graph under consideration, with a 'weight' function $w : E \to \mathbb{R}$, and let T be a minimum spanning tree of G. For a subset of edges $S \subseteq E$, we let $w(S)$ denote $\sum_{e \in S} w(e)$. The *replacement cost* of an edge e in T is defined as $w(T_{\not e}) - w(T)$, where $T_{\not e}$ is a minimum spanning tree of $G \setminus e$. It represents the marginal increase of the weight of the minimum spanning tree if e is not used. It can be shown that the new minimum spanning tree can be obtained by replacing e with exactly one other edge, which is called the *replacement edge*. In fact, the replacement cost of e is the weight of its replacement edge minus the weight of e itself.

Let us first describe a basic algorithm for computing the replacement costs for tree edges. We start by computing a minimum spanning tree T, and we label all tree edges as 'unmarked'. We then consider the edges of the graph, ordered by non-decreasing weight. If we encounter a non-tree edge (i, j), we do the following. First, observe that there is a unique i-j path in T, and (i, j) serves as replacement edge for all unmarked edges on this path. Therefore, we will mark a tree edge as soon as we have identified its first replacement edge. For example, in Figure 1, the first non-tree edge that we consider is $(3, 4)$. We thus label the tree edges $(1, 3)$ and $(1, 4)$ as marked, with associated replacement cost 1 and 2, respectively. The next non-tree edge is $(1, 2)$, which is used to mark tree edge $(2, 4)$ with associated replacement cost 2 (edge $(1, 4)$ is already marked).

It can be shown that this basic algorithm computes the replacement costs of all tree edges. Unfortunately, its time complexity is rather high: we may need up to n steps to identify the unmarked edges, which gives an overall time complexity of $O(mn)$. Fortunately, we can efficiently reduce this complexity by *contracting* the marked edges of the tree, i.e., we merge the extremities of marked tree edges. This contraction will be performed by using a 'union-find' data structure [6, 1].

First, we root the minimum spanning tree, i.e., we designate an arbitrary root node, and we organize the nodes in a directed tree with parent information. In addition, each node is associated with a pointer p to its parent in the union-find data structure. Initially the pointer p of every node points to the node itself. When an unmarked edge is discovered, we 'contract' the edge by letting the pointer p now point to its father. We then apply the classical 'find' function, associated with its classical updates. That is, the pointers of the union-find data structure are used to traverse the path between the two extremities of a non-tree edge. Note that we move up in parallel in the tree from the two extremities. We stop when the same node is reached by the two traversals (one from each extremity). For example in Figure 1, suppose we let node 1 be the root of the tree. After processing the first non-tree edge $(3, 4)$, the updated pointers are $p(3) = 1$ and $p(4) = 1$. For the next non-tree edge $(1, 2)$, the algorithm directly proceeds from the parent of 2 (node 4) to $p(4)$, which is node 1.

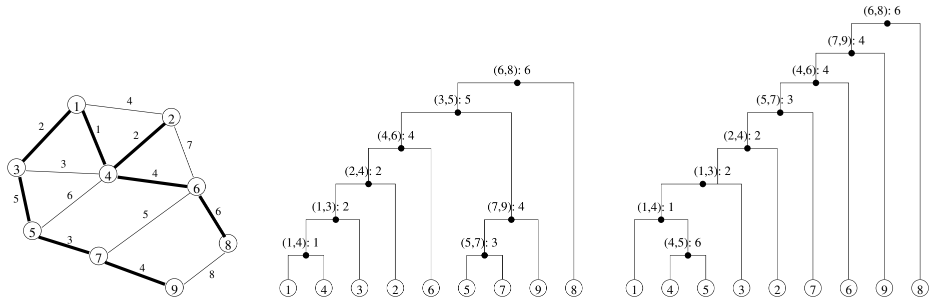
The advantage of this method is that it is easy to implement. Moreover, we will have at most $n - 1$ contractions because the tree contains $n - 1$ edges. In addition we will have at most m requests, thus we obtain the classical union-find complexity of $O(m\alpha(m,n))$. We note that the replacement cost for tree edges can be used to identify mandatory edges: an edge is mandatory if its replacement cost is higher than $K - w(T)$, see also [3].

4 Mandatory Edges and Incrementality

We next consider the implications of introducing mandatory edges on the maintenance of the minimum spanning tree, in the context of the propagation algorithms of Régin [4]. First, observe that we need to update our minimum spanning tree, and other necessary data structures when both tree edges and non-tree edges become mandatory. If a non-tree edge becomes mandatory (for example as a result from inference by other constraints), we clearly need to find a new minimum spanning tree that includes this edge. If a tree edge becomes necessary, it can remain in the tree, but we do need to update the data structures to forbid this edge from being used as a replacement edge. Recall that the main data structure used in [4] is the ccTree. We propose two different methods to update the minimum spanning tree and the ccTree upon the addition of mandatory edges. The first method is based on recomputation. The second method is based on 'repairing' the current minimum spanning tree and ccTree.

The first method can be implemented in a straightforward manner by using the existing algorithms. Namely, we can associate an appropriate low weight value to the mandatory edges, which will then be added first to the minimum spanning tree (assuming that we use Kruskal's algorithm, that adds the edges ordered by non-decreasing weight). After all mandatory edges have been added, the algorithm will proceed with the other edges. Then we can rebuild the ccTree by considering the tree edges in the order of addition, which takes $O(n)$ steps. The advantage of this approach is that the mandatory edges will never appear as an LCA to compute the replacement cost of a non-tree edge, except for the special case when the edge under consideration forms a cycle with mandatory edges only. In fact, we can avoid such special cases by removing all non-tree edges between the nodes in each component formed by the mandatory edges. This first method works well when several edges have become mandatory during one propagation event. When only a few edges become mandatory, our second method will be more efficient.

The second method rebuilds a new ccTree from the existing one. Consider a mandatory (non-tree) edge (i, j). When this edge enters the minimum spanning tree, it will replace the LCA of i and j in the ccTree, i.e., the LCA disappears. As a result, we need to re-build the ccTree up to the point of the previous LCA. That is, we need to revisit the order of the nodes along the paths from i and j to the LCA. Without loss of generality, we assume that i has been added to the ccTree before j. We start by merging i and j (this is the mandatory edge). Then, we proceed by going up the i-LCA path and j-LCA path, starting from i and j, respectively. Let c_i and c_j be the current node on the i-LCA path and j-LCA path, respectively. As long as the weight of the parent of c_i is at least the



a. MST b. The ccTree c. Updated ccTree

Fig. 1. The minimum spanning tree (MST, in bold) for a small example (a.), its ccTree (b.), and the updated ccTree after the addition of the mandatory edge $(4, 5)$ (c.)

weight of the parent of c_j, we let c_i be its parent and continue. If the weight of the parent of c_j is less than the weight of the parent of c_i, we insert c_j between c_i and its parent. In other words, c_j has as 'left' child c_i, and as 'right' child its subtree in the path from j to the LCA (which is always a single node). We then update c_j to be its original parent in the j-LCA path, and repeat the process until the two paths are fully combined (i.e., we reach the position of the previous LCA). Figure 1 provides an example of our second method. To the example presented in Figure 1.a, we introduce the mandatory edge $(4, 5)$. From the ccTree in Figure 1.b, we determine that the LCA for nodes 4 and 5 is the internal node marked with edge $(3, 5)$ with weight 5, which will disappear from the ccTree. Execution of our second method yields the repaired ccTree, depicted in Figure 1.c. The main benefit of this second method is that it needs to update the minimum spanning tree (and the ccTree) only *locally*. In the worst case, its time complexity may be $O(n)$, but the expected time complexity is much lower.

References

[1] Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)

[2] Dixon, B., Rauch, M., Tarjan, R.: Verification and sensitivity analysis of minimum spanning trees in linear time. SIAM J. Comput. 21(6), 1184–1192 (1992)

[3] Dooms, G., Katriel, I.: The "not-too-heavy spanning tree" constraint. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 59–70. Springer, Heidelberg (2007)

[4] Régin, J.-C.: Simpler and Incremental Consistency Checking and Arc Consistency Filtering Algorithms for the Weighted Spanning Tree Constraint. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 233–247. Springer, Heidelberg (2008)

[5] Tarjan, R.E.: Applications of path compression on balanced trees. Journal of the ACM 26(4), 690–715 (1979)

[6] Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM 22, 215–225 (1975)

Constraint Reasoning with Uncertain Data Using CDF-Intervals

Aya Saad, Carmen Gervet, and Slim Abdennadher

German University in Cairo,
New Cairo City, Egypt
{aya.saad,carmen.gervet,slim.abdennadher}@guc.edu.eg

Abstract. Interval coefficients have been introduced in OR and CP to specify uncertain data in order to provide reliable solutions to convex models. The output is generally a solution set, guaranteed to contain all solutions possible under any realization of the data. This set can be too large to be meaningful. Furthermore, each solution has equal uncertainty weight, thus does not reflect any possible degree of knowledge about the data. To overcome these problems we propose to extend the notion of interval coefficient by introducing a second dimension to each interval bound. Each bound is now specified by its data value and its degree of knowledge. This is formalized using the cumulative distribution function of the data set. We define the formal framework of constraint reasoning over this cdf-intervals. The main contribution of this paper concerns the formal definition of a new interval arithmetic and its implementation. Promising results on problem instances demonstrate the approach.

1 Introduction

Interval coefficients have been introduced in Operations Research and Constraint Programming to specify uncertain data in order to provide reliable solutions to convex models. They are at the heart of paradigms such as robust optimization [3, 12] in Operations Research as well as mixed CSP [8], reliable constraint reasoning [15, 16], and quantified CSP [17] in Constraint Programming. These paradigms specify erroneous and incomplete data using uncertainty sets that denote a deterministic and bounded formulation of an ill-defined data. To remain computationally tractable, the uncertainty sets are approximated by convex structures such as intervals (extreme values within the uncertainty set) and interval reasoning can be applied ensuring effective computations.

The concept of convex modeling was coined to formalize the idea of enclosing uncertainty sets and yield reliable solutions; i.e guaranteed to contain any solution produced by any possible realization of the data [5, 2, 16]. As a result, the outcome of such systems is a solution set that can be refined when more knowledge is acquired about the data, and does not exclude any potential solution. The benefits of these approaches are that they deal with real data measurements, produce robust/reliable solutions, and do so in a computationally tractable manner. However, the solution set can sometimes be too large to be meaningful since it encloses all solutions that can be constructed using the data intervals. Furthermore each solution derived has equal uncertainty weight, thus does not reflect any possible degree of knowledge about the data. For instance, consider a collected set of data measurements in traffic flow analysis [9], or in image recognition [7], the data is generally ill-defined but some data values can occur more than others or have a darker shade of grey (hence greater degree of knowledge or certainty). This quantitative information is available during data collection, but lost during the reasoning because not accounted for in the representation of the uncertain data. As a consequence it is not available in the solution set produced. Mainly reliable models offer reliability and robustness, tractable models, but do not account for quantitative information about the data.

This paper addresses this problem. Basically we extend the interval data models with a second dimension: a quantitative dimension added to the measured input data. The main idea introduced in this paper is to show that we can preserve the tractability of convex modeling while enriching the uncertain data sets with a representation of the degree of knowledge available. Our methodology consists of building data intervals employing two dimensional points as extreme values. We assume that with each uncertain data value comes its frequency of occurrence or density function. We then compute the cumulative distribution function (cdf) over this function. The cdf is an aggregated quantitative measurement indicating for a given uncertain value, the probability that the actual data value lies before it. It has been used in different models under uncertainty to analyze the distribution of data sets (e.g. [14] and [10]). It enjoys three main properties that fit an interval representation of data uncertainty: i) the cdf is a monotone, non decreasing function like arithmetic ordering, suitable for interval computations and pruning, ii) it *directly* represents the aggregated probability that a quantity lies within bounds, thus showing the confidence interval of this uncertain data, iii) it brings flexibility to the problem modeling assumptions (e.g. by choosing the data value bounds based on the cdf values, or its sought confidence interval). We introduce the concept of cdf-intervals to represent such convex sets, following the concept of interval coefficients. This requires the decision variables to range over cdf-intervals as well. Basically, in our framework, the elements of a variable's domain are points in a 2D-space, the first dimension for its data value, the second for its aggregated cdf value. It is defined as a cdf-interval specified by its lower and upper bounds. A new domain ordering is defined within the 2D space. This raises the question of performing arithmetic computations over such variables to infer bound consistency. We define the constraint domain over which the calculus in this new domain structure can be performed, including the inference rules.

This paper contains the following contributions: (1) a new representation of uncertain data, (2) a formal framework for solving systems of arithmetic constraints over cdf-intervals, (3) a practical framework including the inference rules within the usual fixed point semantics, (4) an application to interval linear systems. The paper is structured along the contributions listed hereabove.

2 Basic Concepts

This section recalls basic concepts we use to characterize the degree of knowledge, and introduces our notations. These definitions can be found in [10].

2.1 Cumulative Distribution Function

Throughout this paper we assume independence of data.

Definition 1 (density function). *Given a data set $\{N_1, ..., N_n\}$, with available information regarding the occurrence of each data item. The density function $f(x)$ of a data item x, is defined over the population size, m, by:*

$$f(x) = \frac{number\ of\ occurrences\ of\ value N_x}{m} \tag{1}$$

The cumulative distribution function *cdf* of a given value, defines its accumulated density so far. It does not assume any specific distribution function, but follows that of the density function.

Definition 2 (cumulative distribution function). *Given an item value x, with density function $f(x)$, and an unknown variable (commonly referred to as the real-valued random variable) X, the* cdf *of x is the function:*

$$F_X(x) = P(X \leq x) = \sum_{X \leq x} f(x) \tag{2}$$

The cdf *ranges over the interval [0,1].*

Property 1. Every *cdf* is monotone, and right continuous.

In other words, the *cdf* value of a point is the density value of that point in addition to the sum of frequencies of all preceding points, with smaller item value. As illustrated in fig. 1 the *cdf* associated with a density function is always increasing until it reaches the point of stability '1' where the curve remains constant. All the data population resides between two points having *cdf* values '0' and '1' with an average step height equivalent to (m/n), where n is the number of distinct values in the data set and m is the size of its population. The cdf expresses the distribution of the data in an aggregated manner.

2.2 Joint Cumulative Distribution Function

Operations can be performed on *cdf*s but carry a different interpretation than operations over standard arithmetic calculus since they relate to probabilities. The joint operation is essential to our solver and is recalled below. The *joint cdf* is the *cdf* that results from superimposing two variables with a relation, each exerting a *cdf* on its own.

Definition 3 (Joint CDF). *Given two random variables X and Y*[1]

$$F_{XY}(x, y) = P(X \leq x, Y \leq y) \tag{3}$$

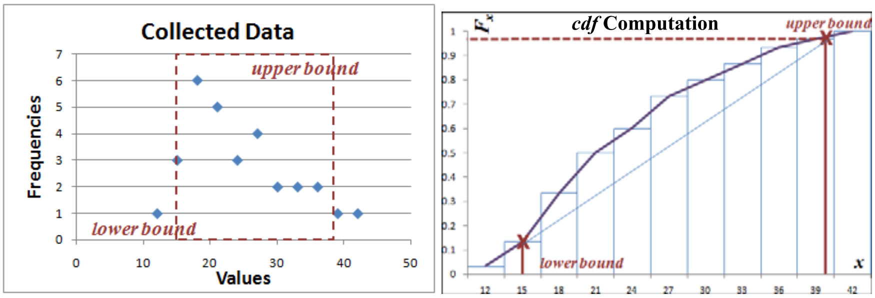
For independent variables $P(X \leq x, Y \leq y) = F_X(x) \times F_Y(y)$.

Definition 4 (Joint CDF over bounded intervals). *When X and Y are bounded intervals; i.e. $X \in [a, b]$ and $Y \in [c, d]$, the joint CDF is defined by:*

$$P(a \leq X \leq b, c \leq Y \leq d) = F_{XY}(b, d) - F_{XY}(a, d) - F_{XY}(b, c) + F_{XY}(a, c) \tag{4}$$

The joint *cdf* F_{XY} of two variables X and Y is used extensively in the computation of the *cdf* resulting from any operation between the variables.

[1] Recall that a random variable maps an event to values, e.g. event that $X \leq x$.



(a) (b)

Fig. 1. Constructing the data interval bounds

2.3 Notations

We assume that a data takes its value in the set of real numbers $\mathbb{R}$, denoted by a,b,c. A cdf value $F_X(x)$ is associated to the uncertainty curve of a given point p. For simplicity, $F_X(x)$ is noted F_x^p, i.e. the cdf value of an uncertain data p at value x. We have $p_x \in \mathbb{R} \times [0, 1]$ with coordinates (x, F_x^p). Data points are denoted by p, q, r possibly subscripted by a data value. Variables are denoted by X, Y, Z and take their value in $\mathcal{U} = \mathbb{R} \times [0, 1]$. Intervals of elements from $\mathcal{U}$ are denoted I, J, K. We denote F^I the approximated linear curve relative to the *cdf* curve inbetween the bounds of I, and F_a^I the cdf value of the data value a plotted on the 2D-interval I.

3 Uncertain Data Representation

Given a measured (also possibly randomly generated) data set denoting the population of an uncertain data, we construct our cdf-intervals as detailed in algorithm 1.

The algorithm runs in $O(n)$ where n is the number of distinct values in the data set. It receives three parameters: the size of the data population, m, a sorted list (ascending order) of the distinct measured data, and a list of their corresponding frequencies. Both lists are of the same size n. The algorithm first computes the *cdf* in a cumulative manner.

```
procedure ConstructIntervalBounds($m$, $Arr[n]$, Freq[$n$])
1. cdf[1] ← Freq[1]/$m$
2. for $i$ = 2 to n do
3.     cdf[$i$] ← (Freq[$i$]/$m$) + cdf[$i - 1$]
4. $i$ = 1,
5. while (Freq[$i$] ≤ $m/n$) do { $m/n$ is the average step value}
6.     $i \leftarrow i + 1$
7. lowerbound ← ($Arr[i]$, cdf[$i$])
8. upperbound ← (cdf$^{-1}$[0.98], 0.98)
```

Algorithm 1. data interval bounds construction

The turning points are then extracted by recording the data value having a density less than the average step value (m/n) and the value with cdf equal to 98%.

Fig. 1 illustrates an example of an interval data construction. For a data set size $n = 11$, and a population size $m = 30$, $Arr[n] = [12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42]$, and its corresponding frequencies $Freq[n] = [1, 3, 6, 5, 3, 4, 2, 2, 2, 1, 1]$, the computed cdf-interval has the following bounds $[(15, 0.13), (39.6, 0.98)]$.

3.1 Interpretation of the Confidence Interval $[p_a, p_b]$

Consider the practical meaning of the interval $[p_a, p_b]$ that we have sought to obtain. This interval is built according to two main sources of information: 1) the monotony and non-decreasing properties of the cdf curve to account for degree of knowledge, 2) the extreme turning points over such a curve. Recall that the cdf curve indicates the aggregated distribution function of a data set. Plotting a point on this curve tells us what are the chances that the actual data value lies on or before this point. The extreme turning points we have considered are such that the lower bound indicates when the slope (thus frequency of occurrence over the population) increases more than the average; and the upper bound that of the cdf reaching a plateau. The measure of this upper bound has been associated with the cdf value of 98%. This corresponds to the distance of $avr + 3\sigma$ when the distribution follows a normal distribution. Such interpretation is a conservative view that can be revised by the decision maker.

It is also important to note the effectiveness of using the *cdf* as an indicator of degree of knowledge. Given a measurement of the data p such that (x, F^p_x) is any point, we have the following due to F^p monotone non-decreasing property:

$$a \leq x \leq b, \quad F^p_a \leq F^p_x \leq F^p_b$$

This implies that we can order (partially) points in this 2D-space $\mathcal{U} = \mathbb{R} \times [0, 1]$. Thus we can construct an algebra over variables taking their value in this space. In particular, we can approximate the cdf curve associated with a data population by the linear (increasing or constant) slope between the two turning points.

3.2 Linear Approximation within $\mathbf{I} = [p_a, p_b]$

Definition 5. *For a given interval* $\mathbf{I} = [p_a, p_b]$, F^I_x *is the projected approximated cdf value of* p_x *onto* F^I *(the cdf associated to the interval), we will denote* $p_x \in \mathbf{I}$ *as* $p_x = (x, F^I_x)$ *for any point lying within the* $\mathbf{I}$ *interval bounds such that:*

$$a < x < b, \quad F^I_x = \frac{F^I_b - F^I_a}{b - a}.(x - a) + F^I_a \tag{5}$$

Property 2. $F^p_a = F^I_a$ *and* $F^q_b = F^I_b$

Example 1. Fig. 2 illustrates the computation of F^I_x. We have $\mathbf{I} = [(18.6, 0.44), (27.8, 0.78)]$. Given a data value $x = 24$ we compute its cdf $F^I_x = 0.64$, and obtain the point $p_x = (24, 0.64)$.

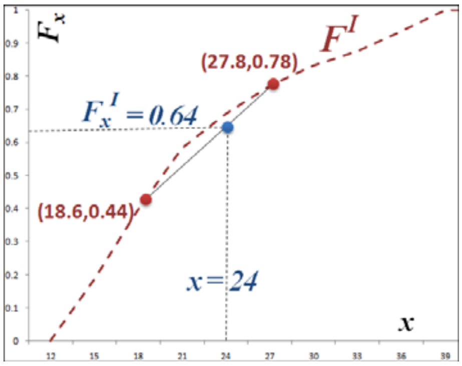


Fig. 2. F_x^I linear projection

4 cdf-Intervals

Our approach follows real interval arithmetic. It adds a second dimension to each uncertain value, requiring us to define a new ordering among points in a two dimensional space, together with new inference rules.

Consider a data population with its *cdf* curve. Create the set of points such that each points p_x is specified by $(x, F_x) \in \mathbb{R} \times [0,1]$. The set $\mathcal{U} = \mathbb{R} \times [0,1]$ is a set of tuples partially ordered, and constitutes a poset with a unique greatest lower bound and least upper bound. Similarly to reliable computing methods the constraint system will produce a solution set as opposed to a solution point. The variables thus denote intervals within the cdf-interval structure and constraint processing needs to be extended to perform arithmetic operations within this algebraic structure.

4.1 cdf-IntervalOrdering $\preccurlyeq$

Definition 6 (Ordering over $\mathcal{U}, \preccurlyeq$). *Let $p_x = (x, F_x^p)$, $q_y = (y, F_y^q) \in \mathcal{U}$, the ordering $\preccurlyeq$ is a partial order defined by:*

$$p_x \preccurlyeq q_y \Leftrightarrow x \leq y \text{ and } F_x^p \leq F_y^q$$

Example 2. Consider the three points $p_x = (1, 0.3), q_y = (2, 0.5)$ and $r_z = (2, 0.1)$. We have $p_x \preccurlyeq q_y$ as well as $r_z \preccurlyeq q_y$, but p_x and r_z are not comparable.

A *cdf*-interval delimited by two points p_x and q_y is specified by the syntax $[p_x, q_y]$ such that $p_x \preccurlyeq q_y$. One important task in interval reasoning is the computation of a new interval from arbitrary points or previous intervals, such that it describes the smallest interval containing a collection of elements. This is based on the meet and joint operators.

Definition 7 (meet and join). *Given the arithmetic ordering and meet and join operations over the reals $(\mathbb{R}, \leq, min, max)$ and the ordering of cdf values within $([0,1], \leq, min, max)$, the meet lub and join glb operators of two points p_x and q_y in $\mathcal{U}$ are defined by:*

$$glb(p_x, q_y) = (min(x, y),\ min(F^p_{lb}, F^q_{lb}))$$
$$lub(p_x, q_y) = (max(x, y),\ max(F^p_{ub}, F^q_{ub})) \quad (6)$$

where $lb = min(x, y)$ *and* $ub = max(x, y)$

The following property establishes the link between the partial ordering $\preccurlyeq$ and the pair (glb, lub) as actual meet and join.

Property 3 (Consistency property).

$$p_x \preccurlyeq q_y \Leftrightarrow p_x = \text{glb}(p_x, q_y)$$
$$p_x \preccurlyeq q_y \Leftrightarrow q_y = \text{lub}(p_x, q_y) \quad (7)$$

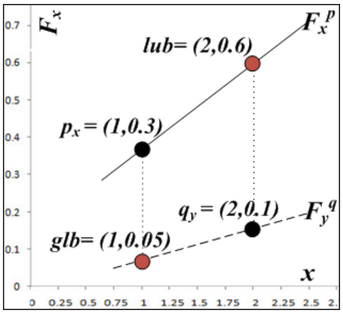


Fig. 3. *glb* and *lub* computation

Fig. 3 illustrates an example which computes *glb* and *lub* of two points $p_x = (1, 0.3)$ and $q_y = (2, 0.1)$. $glb(p_x, q_y) = (1, 0.05)$ and $lub(p_x, q_y) = (2, 0.6)$.

4.2 cdf Domains

The key element to a cdf-Interval domain is the approximated cdf curve it lies on. However, to remain computationally tractable we do not maintain a full domain representation of the points defining the curve. Instead, we approximate the curve by a linear curve whose extreme points are the bounds of the interval. Elements of the interval domain lie on the linear curve. This leads to the following concept of cdf-Interval domain.

Definition 8 (cdf-Domain). *A cdf-Domain is a pair* $[p_a, p_b]$ *satisfying* $p_a \preccurlyeq p_b$ *and denoting the set:*

$$\{p_x = (x, F^p_x) \mid p_a \preccurlyeq p_x \preccurlyeq p_b,\ and\ F^p_x = \frac{F^p_b - F^p_a}{b - a}.(x - a) + F^p_a\} \quad (8)$$

Example 3. Consider a cdf-variable X with domain $I = [(18.6, 0.44), (27.8, 0.78)]$ as illustrated in fig. 2. X can take any point value (x, F^I_x) such that $18.6 \leq x \leq 27.8$ and $F^I_x = \frac{0.78-0.44}{27.8-18.6}.(x - 18.6) + 0.44$.

5 Core Operations on cdf-Intervals

Clearly this work follows the real interval arithmetic introduced in [4]. In particular when the degree of knowledge provides equal weight to each data value the computed intervals are identical. Thus the novelty here lies in the calculus presented with respect to the second dimension, i.e. the degree of knowledge based on the *cdf* values. We consider the standard arithmetic operations interpreted over the set of cdf-intervals. For $\odot \in \{+_{\mathcal{U}}, -_{\mathcal{U}}, \times_{\mathcal{U}}, \div_{\mathcal{U}}\}$ a binary interval arithmetic over two dimensions we seek:

$$[p_a, p_b] \odot [q_c, q_d] = \{p_X \odot q_Y \mid p_X \in [p_a, p_b], q_Y \in [q_c, q_d]\} \qquad (9)$$

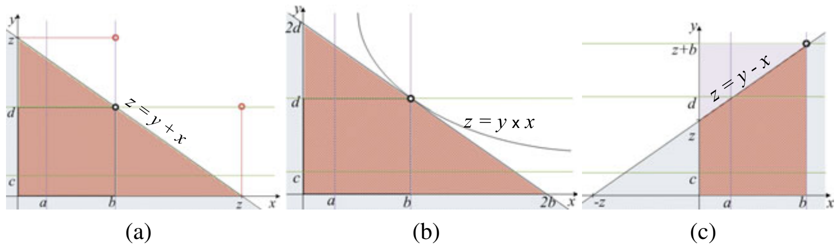


Fig. 4. *cdf* distribution resulting from superimposing two intervals for x and y with a relation: (a) addition (b) multiplication (c) subtraction. $x \in [a, b]$ with a cdf F^I and $y \in [c, d]$ with a cdf F^J.

Any two intervals, each shaping a different distribution *cdf*, can be involved in a relation given by a function. This relation in turn shapes a *cdf* that is based on a double integration of their joint *cdf* over the set of values per interval under the curve of the function [10]. From this generic methodology we derive *cdf* lower and upper bound equations for each binary arithmetic operation. Derived equations are shown by the dark shaded area under the curve of the relation depicted in fig. 4. Proofs are omitted for space reason.

5.1 Addition '$+_{\mathcal{U}}$'

Consider two arbitrary intervals $\mathbf{I} = [p_a, p_b]$ and $\mathbf{J} = [q_c, q_d]$, their arithmetic addition is a result of adding every two points p_x and q_y from both intervals. The resulting *cdf*, F^{I+J} is obtained by superimposing both *cdf* distributions F^I_x and F^J_y as depicted by the dark shaded regions in fig.4(a): it represents the area under the line that describes the relation between x and y, first arguments of points p_x and q_y respectively. The computation of F^{I+J} is based on the joint *cdf* previously discussed in section 2.2; we assume in our calculations that probabilities outside the interval are negligible '= 0' since we are working on the majority of the population inside an interval. The resulting interval is specified by

$$[(lb_+, F^{I+J}_{lb_+}), (ub_+, F^{I+J}_{ub_+})]$$

such that the real interval arithmetic addition is applied to compute the 1^{st}-dimension lower and upper bounds respectively denoted lb_+ and ub_+:

$$lb_+ = min(a + c, a + d, b + c, c + d) \text{ and } ub_+ = max(a + c, a + d, b + c, c + d)$$

$$F^{I+J}_{lb_+} = 0.5[F^J_{lb_+} F^J_c + F^I_a F^I_{lb_+}]$$
$$F^{I+J}_{ub_+} = 0.5[F^I_b + F^J_d]$$

with the cdf value of lb_+on **J** and **I** respectively defined as:

$$F^J_{lb_+} = \begin{cases} F^J_c + a\left[\frac{F^J_d - F^J_c)}{d-c}\right] & if\ d < a+c \\ 1 & otherwise \end{cases}$$
$$F^I_{lb_+} = \begin{cases} F^I_a + c\left[\frac{F^I_b - F^I_a}{b-a}\right] & if\ b < a+c \\ 1 & otherwise \end{cases} \tag{10}$$

Example 4. Given two data populations with associated cdf curves, approximated by the cdf-intervals $\mathbf{I} = [(1, 0.3), (7, 0.65)]$ and $\mathbf{J} = [(2, 0.46), (9, 0.6)]$; the addition of two uncertain data from **I** and **J** respectively is specified by the cdf-interval $[r_{a+c}, r_{b+d}] = (3, 0.38), (16, 0.625)]$ as shown in fig.5. Note that in the absence of second dimension we obtain a regular interval arithmetic addition.

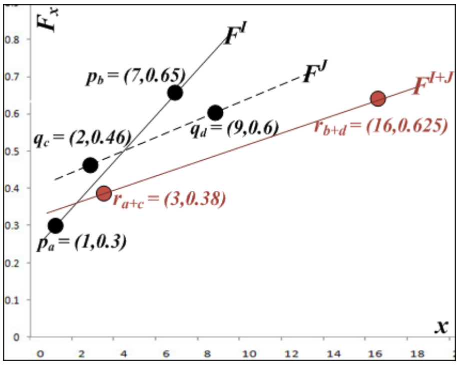


Fig. 5. Example: cdf-Interval addition

5.2 Multiplication '$*_\mathcal{U}$'

Multiplying each pair of points lying in two intervals results in computing the *cdf* $F^{I\times J}$ distribution illustrated by the shaded dark region under the curve in fig.4(b). $\mathbf{I} = [p_a, p_b] \times \mathbf{J} = [q_c, q_d]$ produces the cdf-interval

$$[(lb_\times, F^{I\times J}_{lb_\times}), (ub_\times, F^{I\times J}_{ub_\times})]$$

such that the first dimensions, follows the conventional real interval arithmetic multiplication. The lower and upper bounds are defined by $lb_\times$ and $ub_\times$. Recall that data values can be negative:

$$lb_\times = min(a \times c, a \times d, b \times c, c \times d) \text{ and } ub_\times = max(a \times c, a \times d, b \times c, c \times d)$$

The 2^{nd}-dimension *cdf* for the resulting interval bounds, are computed as follows:

$$F^{I\times J}_{lb_\times} = 0.5(F^I_b F^J_c + F^I_a F^J_d)$$
$$F^{I\times J}_{ub_\times} = max(F^J_d, F^I_b) \tag{11}$$

Example 5. Given two variables ranging over the respective cdf-intervals: $\mathbf{I} = [(1, 0.3), (4, 0.65)]$ and $\mathbf{J} = [(.5, 0.46), (5, 0.6)]$ as illustrated in fig.6. The result of the multiplication is a cdf-interval specified by $[r_{a\times c}, r_{b\times d}] = (1.5, 0.24), (20, 0.65)]$.

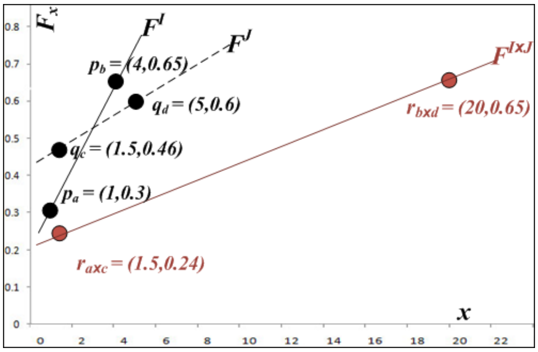


Fig. 6. Example: cdf-Interval multiplication

5.3 Subtraction '$-_{\mathcal{U}}$'

Given two cdf-intervals $\mathbf{I} = [p_a, p_b]$ and $\mathbf{J} = [q_c, q_d]$, the subtraction derives the cdf-interval

$$[(lb_-, F^{J-I}_{lb_-}), (ub_-, F^{J-I}_{ub_-})]$$

such that

$$lb_- = min(c-a, d-a, c-b, d-b) \text{ and } ub_- = max(c-a, d-a, c-b, d-b)$$

and F^{J-I} lower and upper bounds are:

$$F^{J-I}_{lb_-} = 0.5(F^I_b F^J_c + F^I_a F^J_d)$$
$$F^{J-I}_{ub_-} = 0.5 F^I_b [1 + F^J_{ub_-}] \tag{12}$$

5.4 Division '$\%_{\mathcal{U}}$'

Given two cdf-intervals $\mathbf{I} = [p_a, p_b]$ and $\mathbf{J} = [q_c, q_d]$, the division operation derives the resulting cdf-interval

$$[(lb_\div, F^{J\div I}_{lb_\div}),\ (ub_\div, F^{J\div I}_{ub_\div})]$$

such that:

$$lb_\div = min(c \div a, d \div a, c \div b, d \div b) \text{ and } ub_\div = max(c \div a, d \div a, c \div b, d \div b)$$

$F^{J \div I}$ lower and upper bounds are:

$$\begin{aligned} F^{J\div I}_{lb_\div} &= 0.5(F^I_b F^J_c) \\ F^{J\div I}_{ub_\div} &= F^J_d[F^I_b - 0.5F^I_a] \end{aligned} \tag{13}$$

6 Implementation

The constraint system behaves like a solver over real intervals, based on the relational arithmetic of real intervals where arithmetic expressions as interpreted as relations [6]. The relations are handled using the following transformation rules that extend the ones over real intervals with inferences over the cdf values. The handling of these rules is done by a relaxation algorithm which resembles the arc consistency algorithm AC-3 [13]. The solver converges to a fixed point or infers failure. we ensure termination of the generic constraint propagation algorithm because the cdf-domain ordering is reflexive, antisymmetric and transitive. Hereafter we present the main transformation rules for the basic arithmetic operations. For space reasons, when a domain remains unchanged we will use the following notation: $\mathbf{I} = [p_a, p_b]$, $\mathbf{J} = [q_c, q_d]$ and $\mathbf{K} = [r_e, r_f]$. The cdf-variables are denoted by X, Y and Z. Failure is detected if some domain bounds do not preserve the ordering $\preccurlyeq$.

Ordering constraint $X \preccurlyeq Y$

$$\frac{p_b{}' = glb(p_b, q_d), q_c{}' = lub(p_a, q_c)}{\{X \in \mathbf{I}, Y \in \mathbf{J}, X \preccurlyeq Y\} \longmapsto \{X \in [p_a, p_b{}'], Y \in [q_c{}', q_d], X \preccurlyeq Y\}}$$

Equality constraint $X = Y$

$$\frac{p_b{}' = glb(p_b, q_d), p_a{}' = lub(p_a, q_c)}{\{X \in \mathbf{I}, Y \in \mathbf{J}, X = Y\} \longmapsto \{X \in [p_a{}', p_b{}'], Y \in [p_a{}', p_b{}'], X = Y\}}$$

Ternary addition constraints $X +_{\mathbb{U}} Y = Z$

$$\frac{r_f{}' = (ub_+, F^{I+J}_{ub_+}), r_e{}' = (lb_+, F^{I+J}_{lb_+})}{\{X \in \mathbf{I}, Y \in \mathbf{J}, Z \in \mathbf{K}, Z = X +_{\mathbb{U}} Y\} \longmapsto \{X \in \mathbf{I}, Y \in \mathbf{J}, Z \in [r_e{}', r_f{}'], Z = X +_{\mathbb{U}} Y\}}$$

The projection onto Y's domain is symmetrical.

$$\frac{p_b{}' = (ub_-, F^{K-J}_{ub_-}), p_a{}' = (lb_-, F^{K-J}_{lb_-})}{\{X \in \mathbf{I}, Y \in \mathbf{J}, Z \in \mathbf{K}, X = Z -_{\mathbb{U}} Y\} \longmapsto \{X \in [p_a{}', p_b{}'], Y \in \mathbf{J}, Z \in \mathbf{K}, X = Z -_{\mathbb{U}} Y\}}$$

Ternary multiplication constraint $X \times_{\mathbb{U}} Y = Z$

$$\frac{r_f{}' = (ub_\times, F^{I\times J}_{ub_\times}), r_e{}' = (lb_\times, F^{I\times J}_{lb_\times})}{\{X \in \mathbf{I}, Y \in \mathbf{J}, Z \in \mathbf{K}, Z = X \times_{\mathbb{U}} Y\} \longmapsto \{X \in \mathbf{I}, Y \in \mathbf{J}, Z \in [r_e{}', r_f{}'], Z = X \times_{\mathbb{U}} Y\}}$$

The projection onto Y's domain is symmetrical.

$$\frac{p_b{}' = (ub_\div, F^{K\div J}_{ub_\div}), p_a{}' = (lb_\div, F^{K\div J}_{lb_\div})}{\{X \in \mathbf{I}, Y \in \mathbf{J}, Z \in \mathbf{K}, X = \mathbf{K} \div_{\mathbb{U}} \mathbf{J}\} \longmapsto \{X \in [p_a{}', p_b{}'], Y \in \mathbf{J}, Z \in \mathbf{K}, X = \mathbf{K} \div_{\mathbb{U}} \mathbf{J}\}}$$

6.1 Empirical Evaluation

A prototype implementation of the constraint system was done as a module on top of the CHR library in Sicstus Prolog. To evaluate the added value of the new constraint domain, we consider an example provided in [16] , and attached a degree of knowledge: the cdf values to the interval bounds. Example 6 aims at solving a system of linear equations which has cdf-interval coefficients and unknown variables having no certainty degree defined, i.e. the lower bound points are '(0, 0)' whereas upper-bound of the variable interval is '$(\infty, 1)$' with a *cdf* value '1'. Shown below are pruned domains of the variables at fixed point using our inferences. The cdf-intervals attached to the data (here coefficients) were propagated onto the cdf-variables, X_1 and X_2. We can see that inferences on the 1^{st}-dimension yield similar pruning on the resulting variable domains and the additional information coming from the *cdf* component demonstrate the information gained on the density of occurrence for the resulting points within the cdf-domains.

Example 6. Consider the system of linear equations $(\mathbf{A}, \mathfrak{R}, \mathbf{b})$ shown below:

$$\mathbf{A} = \begin{bmatrix} [(-2, 0.32),\ (2, 0.83)] & [(1, 0.41),\ (2, 0.95)] \\ [(-2, 0.35),\ (-1, 0.87)] & [(-1.001, 0.31),\ (-1, 0.75)] \\ [(5.999, 0.28),\ (6, 0.86)] & [(1.5, 0.34),\ (3, 0.96)] \end{bmatrix},$$

$$\mathfrak{R} = \begin{bmatrix} \leq \\ = \\ = \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} [(3, 0.4),\ (4, 0.88)] \\ [(-5, 0.3),\ (5, 0.78)] \\ [(4, 0.29),\ (15, 0.85)] \end{bmatrix}$$

The output domains for the variables X_1 and X_2 after applying the propagation cdf-interval techniques are $[(0.00, 0.05),\ (2.50, 0.69)]$ and $[(0.00,\ 0.06),\ (5.00, 0.59)]$ respectively.

Remark. If we remove any density value for the uncertain coefficients by assigning values '0' and '1' to the lower and upper bounds of the *cdf* dimension, this amounts to considering that the uncertain data is equally distributed along the given interval. The output in this case is $X_1 \in [(0.00, 0.00),\ (2.50, 1.00)]$ and $X_2 \in [(0.00, 0.00),\ (5.00, 1.00)]$. In this sense the cdf-interval constraint model generalizes the real interval arithmetic constraint reasoning.

6.2 cdf-Interval Linear Programming

In this section we also show how the cdf-interval constraint model does also generalize Interval Linear Systems [11]. Thus linear systems with cdf-interval coefficients and variables can also be solved by a polynomial transformation into a linear model that can then be sent to the Simplex method. The approach is very similar to the one used for ILS with positive coefficients (also called POLI systems) [1].

Definition 9. *Let $\mathcal{V}$ be a set of n cdf-variables, and C a set of m linear constraints over $\mathcal{V}$ with cdf-interval coefficients. An cdf-interval Linear System UILS induced by C over $\mathcal{V}$ is a tuple $\langle \mathbf{A}, \mathfrak{R}, \mathbf{b} \rangle$, where $\mathbf{A} \in (\mathbb{R} \times [0..1])^{m \times n}$ is the matrix of the LHS intervals $[glb_A, lub_A]$, $\mathbf{b} \in (\mathbb{R} \times [0..1])^m$ is the vector of the RHS intervals $[glb_b, lub_b]$ and $\mathfrak{R}_i \in \{<, \leqslant, =, \geqslant, >\}\ \forall i = 1, .., m$ is a list of m relations.*

Generated Linear Inequalities. For space reasons we demonstrate the approach of transformation of a positive cfd-interval linear system with positive values into a linear system and omit the formal proof of equivalence of models.

Consider the following minimization problem over cdf-coefficients and variables:

$$\text{Minimize } Z = \sum_{j=1}^{n} [p_{e_j}, p_{f_j}] p_{x_j}$$

$$\text{subject to } \sum_{j=1}^{n} [p_{a_{ij}}, p_{b_{ij}}] p_{x_j} \geq [p_{c_i}, p_{d_i}] \quad \forall i = 1, 2, .., m$$

$$\forall j, \quad p_{x_j} \in \mathbb{R}^+ \times [0..1] \tag{14}$$

The transformation of the above model yields ($m \times 2^{n+1}$) inequalities per dimension. The produced solution set is $S_{di} = \{S_{di}^k | k = 1, 2, ..., 2^{n+1}\}$ where the upper-bound value range is $S_{di} = lub_{k=1}^{2^{n+1}} S_{di}^k$ and the lower-bound value range is $S_{di} = glb_{k=1}^{2^{n+1}} S_{di}^k$; d is the dimension order: 1 or 2.

The transformation is performed in two steps: first, each equality constraint is transformed into two cdf-interval inequalities; then the cdf-constraints are transformed into linear constraints. The output of the transformation applied to example 6 will be:

$$\mathbf{A} = \begin{bmatrix} (-2, 0.32) & (1, 0.41) \\ (-2, 0.35) & (-1.001, 0.31) \\ (1, 0.13) & (1, 0.25) \\ (5.999, 0.28) & (1.5, 0.34) \\ (-6, 0.14) & (-3, 0.04) \end{bmatrix}, \mathfrak{R} = \begin{bmatrix} \leq \\ \leq \\ \leq \\ \leq \\ \leq \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} (4, 0.88) \\ (5, 0.78) \\ (5, 0.7) \\ (15, 0.85) \\ (-4, 0.71) \end{bmatrix}$$

Accordingly, the 1st constraint row is the inequality constraint: $(-2, 0.32) p_{x_1} +_{\mathbb{U}} (1, 0.41)$ $p_{x_2} \preccurlyeq (4, 0.88)$. This constraint in turn will be transformed into two components: $-2x_1 + x_2 \leq 4$ and $0.25[0.41F_{ub}^2 + 0.95F_{lb}^2 + 0.32F_{ub}^1 + 0.83F_{lb}^1] \leq 0.88$. The linear equations resulting from the transformation of the above ouput are presented here, where equations 15 and 16 are in the 1st and 2nd dimensions respectively:

$$\begin{aligned} -2x_1 + 2x_2 &\leq 4 \\ -2x_1 - 1.001x_2 &\leq 5 \\ x_1 + x_2 &\leq 5 \\ 5.999x_1 + 1.5x_2 &\leq 15 \\ -6x_1 - 3x_2 &\leq -4 \end{aligned} \tag{15}$$

$$\begin{aligned} 0.25[0.41F_{ub}^2 + 0.95F_{lb}^2 + 0.32F_{ub}^1 + 0.83F_{lb}^1] &\leq 0.88 \\ 0.25[0.31F_{ub}^2 + 0.75F_{lb}^2 + 0.35F_{ub}^1 + 0.87F_{lb}^1] &\leq 0.78 \\ 0.5[F_{ub}^1 + F_{ub}^2] &\leq 0.7 \\ 0.25[0.34F_{ub}^2 + 0.96F_{lb}^2 + 0.28F_{ub}^1 + 0.86F_{lb}^1] &\leq 0.85 \\ 0.5[F_{ub}^1 + F_{ub}^2] &\leq 0.71 \\ F_{lb}^1 &\leq F_{ub}^1 \\ F_{lb}^2 &\leq F_{ub}^2 \end{aligned} \tag{16}$$

for a point p_{x_i}:x_i is the point value in the 1st-dimension and both F^i_{lb} and F^i_{ub} are the 2nd-dimension lower bound and upper bound values that the point can take.

An additional constraint has been added to the linear equations in the 2nd-dimension; this ensures that the lower bound value is less that its upper bound value. It is clear that resulting equations in both dimensions are linear and can be solved using linear programming techniques.

7 Conclusion

The framework of reliable computing offers robust and tractable approaches to reason with uncertain data by means of convex models of uncertainty sets (e.g using interval coefficients). It does not account however for any degree of knowledge about the data such as the density function. Thus all solutions in the solution set have equal uncertainty weight. This paper addressed this issue and showed how to embed a degree of knowledge in the form of the cumulative distribution function. The paper proposed the novel concept of cdf-interval, whereby an uncertainty set is specified by an interval of points with first coordinate the data uncertainty value, and second coordinate its cdf value. Since the uncertain data and consequently the decision variables are specified by their confidence interval, so is the solution set. We also presented the constraint system over this new domain by extending the real interval arithmetic to cdf-interval arithmetic using the monotone non-increasing property of the cdf function. Finally the application of cdf-intervals to extend the approach over Interval Linear Systems was showcased. We are currently applying this new approach to larger systems of constraints with application to vehicle routing, networking but also finance and image recognition, where the uncertain data is enriched with a degree of knowledge (i.e..the density function drawn from historic data trends or randomly generated).

References

1. Beaumont, O.: Solving interval linear systems with linear programming techniques. Linear Algebra and its Applications 281(1-3), 293–309 (1998)
2. Ben-Haim, Y., Elishakoff, I.: Discussion on: A non-probabilistic concept of reliability. Structural Safety 17(3), 195–199 (1995)
3. Ben-Tal, A., Nemirovski, A.: Robust solutions of uncertain linear programs. Operations Research Letters 25(1), 1–14 (1999)
4. Benhamou, F., de Vinci, R.: Interval constraint logic programming. In: Constraint programming: basics and trends: Châtillon Spring School, France (1994)
5. Chinneck, J., Ramadan, K.: Linear programming with interval coefficients. Journal of the Operational Research Society, 209–220 (2000)
6. Cleary, J.: Logical arithmetic. Future Computing Systems 2(2), 125–149 (1987)
7. Deruyver, A., Hodé, Y.: Qualitative spatial relationships for image interpretation by using a conceptual graph. Image and Vision Computing 27(7), 876–886 (2009)
8. Fargier, H., Lang, J., Schiex, T.: Mixed constraint satisfaction: A framework for decision problems under incomplete knowledge. In: Proceedings of the National Conference on Artificial Intelligence, Citeseer, pp. 175–180 (1996)
9. Grossglauser, M., Rexford, J.: Passive Traffic Measurement for Internet Protocol Operations. In: The Internet as a Large-Scale Complex System, p. 91 (2005)

10. Gubner, J.: Probability and Random processes for electrical and computer Engineers. Cambridge Univ. Pr., Cambridge (2006)
11. Hansen, E.: Global optimization using interval analysis: the one-dimensional case. Journal of Optimization Theory and Applications 29(3), 331–344 (1979)
12. Hoffman, K.: Combinatorial Optimization: Current successes and directions for the future. Journal of Computational and Applied Mathematics 124(1-2), 341–360 (2000)
13. Mackworth, A.: Consistency in networks of relations. Artificial Intelligence 8(1), 99–118 (1977)
14. Tversky, A., Kahneman, D.: Advances in prospect theory: Cumulative representation of uncertainty. Journal of Risk and Uncertainty 5(4), 297–323 (1992)
15. Yorke-Smith, N.: Reliable constraint reasoning with uncertain data. PhD thesis, IC-Parc, Imperial College London (2004)
16. Yorke-Smith, N., Gervet, C.: Certainty closure: Reliable constraint reasoning with incomplete or erroneous data. ACM Transactions on Computational Logic (TOCL) 10(1), 3 (2009)
17. Zhou, K., Doyle, J.C., Glover, K.: Robust and optimal control. Prentice-Hall, Inc., Upper Saddle River (1996)

Revisiting the Soft Global Cardinality Constraint

Pierre Schaus[1], Pascal Van Hentenryck[1,2], and Alessandro Zanarini[1]

[1] Dynadec
[2] Brown University
{pschaus,pvh,alessandro.zanarini}@dynadec.com

Abstract. The Soft Global Cardinality Constraint (SOFTGGC) relaxes the Global Cardinality Constraint (gcc) by introducing a violation variable representing unmet requirements on the number of value occurrences. A first domain consistent filtering algorithm was introduced by Van Hoeve et al. in 2004 using a minimum cost flow algorithm. A simpler and more efficient filtering algorithm was introduced in 2006 by Zanarini et al. using matchings in bipartite graphs. While the consistency check introduced in the second algorithm is correct, we show that the algorithm may not achieve domain consistency when cardinality requirements contain zeroes. We give new domain consistent conditions and show how to achieve domain consistency within the same time bounds. The SOFTGGC constraint was implemented in COMET.

1 Introduction

Régin et al. [3] suggested to soften global constraints by introducing a cost variable measuring the violation of the constraints. This has the advantage that over-constrained satisfaction problems can be turned into a constrained optimization problem solvable by traditional CP solvers; furthermore specialized filtering algorithms can be employed to filter the variables involved in soft constraints. One of the most used global constraints to solve practical problems is the Global Cardinality Constraint (gcc) introduced in [2]:

Definition 1

$$gcc(X,l,u) = \{(d_1,\dots,d_n) \mid d_i \in D_i, l_d \leq |\{d_i \mid d_i = d\}| \leq u_d \ , \forall d \ \in D_X\}$$

*A soft version of this constraint (*SOFTGGC*) was introduced in [4]:*

$$softggc(X,l,u,Z) = \{(d_1,\dots,d_n) \mid d_i \in D_i, d_z \in D_Z, viol(d_1,\dots,d_n) \leq d_z\}$$

where

$$viol(d_1,\dots,d_n) = \sum_{d \in D_X} \max(0, |\{d_i \mid d_i = d\}| - u_d, l_d - |\{d_i \mid d_i = d\}|).$$

The violation represents the sum of excess or shortage for each value. For space reasons, we only consider the *value-based violation* version of the constraint in this paper as the extension to *variable violation* follows directly (as in [5]).

The domain filtering algorithm for SOFTGGC introduced in [5] exploits matching theory and we use the same notation for consistency and clarity. The consistency check and the filtering of the violation variable Z are briefly summarized in Section 2. The main contribution of the paper is in Section 3 where the corrected filtering algorithm for the X variables is presented. Please refer to [5] for some basic notions about matching theory.

2 Consistency Checking and Filtering of $Z^{\min}$ [5]

Let $G(X \cup D, E)$ be an undirected bipartite graph (also known as the value graph) such that one partition represents the variable set and the other one the value set. There is an edge $\{x_i, d\} \in E$ if and only if $d \in D_i$. Two specialized versions of G were introduced in [5]: They differ only by the capacity of value vertices, where vertex capacity is the maximum number of edges belonging to the matching that share the vertex.

Definition 2. *Let G_o (the overflow graph) be a value graph such that the capacities of value-vertices are set to $c(d) = u_d$. Analogously let G_u (the underflow graph) be a value graph such that the capacities of value vertices are set to $c(d) = l_d$. In both G_o and G_u, variable vertices have unit capacities.*

The violation is expressed in terms of overflows and underflows. They are are characterized by Theorem 1 which specifies how to find

– a valid assignment $(d_1, \ldots, d_n)$ that minimizes the total *overflow* :

$$\sum_{d \in D_X} \max(0, |\{d_i \mid d_i = d\}| - u_d),$$

– a valid assignment $(d_1, \ldots, d_n)$ that minimizes the total *underflow*:

$$\sum_{d \in D_X} \max(0, l_d - |\{d_i \mid d_i = d\}|).$$

Theorem 1 ([5]). *Given a maximum matching M_o in the graph G_o, it is not possible to find an assignment with a total overflow less than $BOF = |X| - |M_o|$ (best overflow). Given a maximum matching M_u in the graph G_u, it is not possible to find an assignment with a total underflow less than $BUF = \sum_{d \in D} l_d - |M_u|$ (best underflow).*

Theorem 2 ([5]). *Given a SOFTGGC constraint and two maximum matchings M_o and M_u , respectively in G_o and G_u , it is possible to build a class of assignments with overflow equal to $BOF = |X| - |M_o|$ (best overflow) and $BUF = \sum_{d \in D} l_d - |M_u|$ (best underflow).*

In other words, Theorem 2 tells us that it is possible to find an assignment with a violation equal to $BOF + BUF$. The violation variable can be filtered as $Z^{\min} \leftarrow \max(Z^{\min}, BOF + BUF)$. If $BOF + BUF > Z^{\max}$, then the constraint is inconsistent. The algorithm to find an assignment with a violation equal to $BOF + BUF$ starts from a matching M_u in G_u (having by definition a underflow of BUF and an overflow of 0 but not representing a complete assignment). This matching is then increased in G_o using a classical augmenting-path algorithm. Since the augmenting-path algorithm never decreases the degree of a vertex, the underflow cannot increase (it remains constant). At the end, the overflow is equal to BOF. The final assignment has a violation equal to $BOF + BUF$. See [5] for further details.

3 Filtering of X

While the consistency check is correct, the original paper [5] overlooked the case in which the lower or upper bounds of the value occurrences are zeroes and it does not characterize the conditions to achieve domain consistency in such cases (see Example 1 below). In this section, we review and correct the theorems on which the filtering algorithm is built upon. Theorem 3 shows the properties for which a vertex x is matched in every maximum matching.

Theorem 3. *A variable vertex x is matched in every maximum matching of G_u (G_o) iff it is matched in a maximum matching M_u (M_o) and there does not exist an M-alternating path starting from a free variable vertex and finishing in x.*

Proof
$\Rightarrow$ Suppose there exists an even M-alternating path P starting from a free variable vertex x' such that $P = \{x', \ldots, x\}$; the alternating path is even as x and x' belong to the same vertex partition furthermore x must be matched as P is an alternating path and x' is free. Then $M' = M \oplus P'$ is still a maximum matching in which x is free.
$\Leftarrow$ Suppose there exists a maximum matching M' in which x is free. Any of the adjacent vertices of x are matched otherwise M' is not maximum. We can build an even alternating path starting from x by choosing one of the adjacent vertices of x and then by following the edge belonging to M'. By using such an even alternating path, it is possible to build a new maximum matching in which x is matched and there exists an M-alternating path starting from a free variable-vertex.

Theorem 4 gives the conditions under which forcing an assignment $x \leftarrow v$ leads to a unit increase in the best underflow.

Theorem 4. *Forcing the assignment $x \leftarrow v$ leads to decrease the size of the maximum matching in G_u by one and thus increasing the underflow by one if and only if $l_v > 0$ and $e = \{x, v\}$ does not belong to a maximum matching in G_u, or $l_v = 0$ and the vertex x is matched in every maximum matching in G_u.*

Proof. If $l_v > 0$, then there exists a matching (maybe not maximum) using the edge $x \leftarrow v$. Consequently, if $e = \{x, v\}$ does not belong to any maximum matching of G_u, the size of a maximum matching would decrease by one forcing the assignment $x \leftarrow v$. Now if $l_v = 0$, then the edge $x \leftarrow v$ belongs to no matching of G_u so the size of the maximum matching decreases only if the vertex x is matched in every maximum matching. □

Example 1. Assume $D_1 = \{1, 2\}$, $D_2 = \{1, 3\}$, $D_3 = \{1, 3\}$ with $l_1 = 0, l_2 = 1, l_3 = 1$ (upper bounds are all equal to $|X|$). A maximum underflow matching is $M = \{\{X_1, 2\}, \{X_2, 3\}\}$ that does not incur in any violation. The wrong assumption made in [5] was to consider that values with capacity equal to zero would not cause any sort of underflow increase. However, in some cases, this assumption is incorrect as we now show.

Forcing the assignment $X_2 = 1$ (or equivalently $X_3 = 1$) does not cause an increment of violation; X_2 belongs to an M-alternating path starting from a free variable vertex, i.e., $P = (X_3, 3, X_2)$ therefore there exists a maximum matching M' in which X_2 is free ($M' = \{\{X_1, 2\}, \{X_3, 3\}\}$). Then, X_2 can take the value 1 without increasing the underflow. The assignment $X = (2, 1, 3)$ has total violation equal to zero. However, forcing the assignment $X_1 = 1$ causes an increment of violation; there is no M-alternating path starting from a free variable vertex and ending in X_1 thus X_1 is matched in every maximum matching and it is not free to take the value 1 without increasing the underflow. The best assignment would then have a unit underflow (e.g., $X = (1, 3, 3)$) as only X_1 can take the value 2.

Similarly, Theorem 5 gives the conditions under which forcing an assignment $x \leftarrow v$ leads to a unit increase to the best overflow.

Theorem 5. *Forcing the assignment $x \leftarrow v$ leads to decrease the size of the maximum matching in G_o by one and thus increasing the overflow by one if and only if $u_v > 0$ and $e = \{x, v\}$ does not belong to a maximum matching in G_o, or $u_v = 0$ and the vertex x is matched in every maximum matching in G_o.*

Proof. Similar to proof of theorem 4

Example 2. Assume $D_1 = \{1, 4\}$, $D_2 = \{1, 2, 3\}$, $D_3 = \{1, 3\}$, $D_4 = \{3\}$, $D_5 = \{1, 2, 4\}$ with $u_1 = 1, u_2 = 2, u_3 = 1, u_4 = 0$ (all lower bounds are null). A maximum overflow assignment is $X = (1, 2, -, 3, 2)$ (unit violation). Variable X_1 belongs to an M-alternating path starting from a free variable-vertex $P = (X_3, 1, X_1)$, therefore there exists a maximum matching in which X_1 is free to take any value without increasing the best overflow (e.g., the full assignment $X = (4, 2, 1, 3, 2)$ has still an overflow equal to 1). For the arc $(X_5, 4)$, the situation is different: in fact X_5 is matched in every maximum matching. Therefore, forcing this assignment would increase the overflow to 2; the best assignment we could get is for instance $X = (1, 2, 1, 3, 4)$.

Theorem 6 gives the conditions to reach domain consistency.

Theorem 6. *Let G_o and G_u be the value graphs with respectively upper and lower bound capacities and let M_o and M_u be maximum matching respectively in G_o and G_u ; let BOF and BUF be respectively $BOF = |X| - |M_o|$ and $BUF = \sum_{d \in D} l_d - |M_u|$. The constraint $softggc(X, l, u, Z)$ is domain consistent on X if and only if $\min D_Z \leq BOF + BUF$ and either:*

1. *$BOF + BUF < (\max D_Z - 1)$ or*
2. *if $BOF + BUF = (\max D_Z - 1)$ and for each edge $e = \{x, v\}$ either*
   - *it belongs to a maximum matching in G_u, or $l_v = 0$ and the vertex x is not matched in every maximum matching in G_u **or***
   - *it belongs to a maximum matching in G_o, or $u_v = 0$ and the vertex x is not matched in every maximum matching in G_o or*
3. *if $BOF + BUF = \max D_Z$ and for each edge $e = \{x, v\}$ we have that*
   - *it belongs to a maximum matching in G_u, or $l_v = 0$ and the vertex x is not matched in every maximum matching in G_u **and***
   - *it belongs to a maximum matching in G_o, or $u_v = 0$ and the vertex x is not matched in every maximum matching in G_o .*

Proof. Common to the proof of all the cases is the fact that there exists an assignment with violation $BOF + BUF$. If $BOF + BUF > \max D_Z$ then the constraint is inconsistent since $BOF + BUF$ is a lower bound on the violation.

1. Forcing the assignment of one variable cannot increase BUF by more than one and cannot increase BOF by more than one. So in the worst case the assignment of a variable to a value results in $BUF' = BUF + 1$ and $BOF' = BOF + 1$. By Theorem 2 it is possible to build an assignment for all the variables with violation $BUF' + BOF' = BUF + BOF + 2 \leq \max D_Z$ which is thus consistent.
2. Since $BOF + BUF = (\max D_Z - 1)$ and because of Theorem 2, at most one of BUF or BOF can increase by one by forcing the assignment $x \leftarrow v$. Theorems 4 and 5 tell us the conditions say $\mathcal{C}_u$ and $\mathcal{C}_o$ under which the assignment leads to increase respectively BUF and BOF by one. At most one of these conditions can be satisfied. Hence the condition expressed is nothing else than $\neg(\mathcal{C}_u \wedge \mathcal{C}_o) \equiv \neg\mathcal{C}_u \vee \neg\mathcal{C}_o$.
3. This is similar to previous point except that neither BUF nor BOF can increase by one. Hence the condition expressed is $\neg\mathcal{C}_u \wedge \neg\mathcal{C}_o$. □

Note that once we compute the alternating paths to detect edges belonging to a maximum matching, we get for free also the variable vertices that are matched in every maximum matching (Theorem 3). Therefore the complexity of the filtering algorithm remains unchanged w.r.t. [5].

The algorithm described in this paper is implemented in the constraints `softAtLeast`, `softAtMost` and `softCardinality` of COMET [1].

References

1. Comet 2.0, http://www.dynadec.com
2. Régin, J.-C.: Generalized arc consistency for global cardinality constraint. In: AAAI 1996, pp. 209–215 (1996)

3. Régin, J.C., Petit, T., Bessière, C., Puget, J.-F.: An original constraint based approach for solving over constrained problems. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, p. 543. Springer, Heidelberg (2000)
4. van Hoeve, W.J., Pesant, G., Rousseau, L.-M.: On global warming: Flow-based soft global constraints. J. Heuristics 12(4-5), 347–373 (2006)
5. Zanarini, A., Milano, M., Pesant, G.: Improved algorithm for the soft global cardinality constraint. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, pp. 288–299. Springer, Heidelberg (2006)

A Constraint Integer Programming Approach for Resource-Constrained Project Scheduling

Timo Berthold[1,⋆], Stefan Heinz[1,⋆], Marco E. Lübbecke[2], Rolf H. Möhring[3], and Jens Schulz[3]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
{berthold,heinz}@zib.de
[2] Technische Universität Darmstadt, Fachbereich Mathematik, Dolivostr. 15, 64293 Darmstadt, Germany
luebbecke@mathematik.tu-darmstadt.de
[3] Technische Universität Berlin, Institut für Mathematik, Straße des 17. Juni 136, 10623 Berlin, Germany
{moehring,jschulz}@math.tu-berlin.de

Abstract. We propose a hybrid approach for solving the resource-constrained project scheduling problem which is an extremely hard to solve combinatorial optimization problem of practical relevance. Jobs have to be scheduled on (renewable) resources subject to precedence constraints such that the resource capacities are never exceeded and the latest completion time of all jobs is minimized.

The problem has challenged researchers from different communities, such as integer programming (IP), constraint programming (CP), and satisfiability testing (SAT). Still, there are instances with 60 jobs which have not been solved for many years. The currently best known approach, LAZYFD, is a hybrid between CP and SAT techniques.

In this paper we propose an even stronger hybridization by integrating all the three areas, IP, CP, and SAT, into a single branch-and-bound scheme. We show that lower bounds from the linear relaxation of the IP formulation and conflict analysis are key ingredients for pruning the search tree. First computational experiments show very promising results. For five instances of the well-known PSPLIB we report an improvement of lower bounds. Our implementation is generic, thus it can be potentially applied to similar problems as well.

1 Introduction

The resource-constrained project scheduling problem (RCPSP) is not only theoretically hard [5] but consistently resists computational attempts to obtain solutions of proven high quality even for instances of moderate size. As the problem is of high practical relevance, it is an ideal playground for different optimization communities, such as integer programming (IP), constraint programming (CP), and satisfiability testing (SAT), which lead to a variety of publications, see [6].

⋆ Supported by the DFG Research Center MATHEON *Mathematics for key technologies* in Berlin.

The three areas each come with their own strengths to reduce the size of the search space. Integer programming solvers build on lower bounds obtained from linear relaxations. Relaxation can often be considerably strengthened by additional valid inequalities (*cuts*), which spawned the rich theory of polyhedral combinatorics. Constraint programming techniques cleverly learn about logical implications (between variable settings) which are used to strengthen the bounds on variables (*domain propagation*). Moreover, the constraints in a CP model are usually much more expressive than in the IP world. Satisfiability testing, or SAT for short, actually draws from *unsatisfiable* or conflicting structures which helps to quickly finding reasons for and excluding infeasible parts of the search space. The RCPSP offers footholds to attacks from all three fields but no single one alone has been able to crack the problem. So it is not surprising that the currently best known approach [11] is a hybrid between two areas, CP and SAT. Conceptually, it is the logical next step to integrate the IP world as well. It is the purpose of this study to evaluate the potential of such a hybrid and to give a proof-of-concept.

Our Contribution. Following the constraint integer programming (CIP) paradigm as realized in SCIP [1,12], we integrate the three techniques into a single branch-and-bound tree. We present a CP approach, enhanced by lower bounds from the linear programming (LP) relaxation, supported by the SCIP intern conflict analysis and a problem specific heuristic. We evaluate the usefulness of LP relaxation and conflict analysis in order to solve scheduling problems from the well known PSPLIB [10]. CP's global `cumulative` constraint is an essential part of our model, and one contribution of our work is to make this constraint generically available within the CIP solver SCIP.

In our preliminary computational experiments it turns out that already a basic implementation is competitive with the state-of-the-art. It is remarkable that this holds for both, upper *and* lower bounds, the respective best known of which were not obtained with a single approach. In fact, besides meeting upper bounds which were found only very recently [11], independently of our work, we improve on several best known lower bounds of instances of the PSPLIB.

Related Work. For an overview on models and techniques for solving the RCPSP we refer to the recent survey of [6]. Several works on scheduling problems already combine solving techniques in hybrid approaches. For the best current results on instances of PSPLIB, we refer to [11], where a constraint programming approach is supported by lazily creating a SAT model during the branch-and-bound process by which new constraints, so called *no-goods*, are generated.

2 Problem Description

In the RCPSP we are given a set $\mathcal{J}$ of non-preemptable jobs and a set $\mathcal{R}$ of renewable resources. Each resource $k \in \mathcal{R}$ has bounded capacity $R_k \in \mathbb{N}$. Every job j has a processing time $p_j \in \mathbb{N}$ and resource demands $r_{jk} \in \mathbb{N}$ of each resource $k \in \mathcal{R}$. The starting time S_j of a job is constrained by its predecessors that are given by a precedence graph $D = (V, A)$ with $V \subseteq \mathcal{J}$. An arc $(i, j) \in A$ represents a precedence relationship, i.e., job i must be finished before job j starts. The goal is to schedule all jobs with respect to resource and precedence constraints, such that the latest completion time of all jobs is minimized.

The RCPSP can be modeled easily as a constraint program using the global `cumulative` constraint [3] which enforces that at each point in time, the cumulated demand of the set of jobs running at that point, does not exceed the given capacities. Given a vector $\boldsymbol{S}$ of start time variables S_j for each job j, the RCPSP can be modeled as follows:

$$\begin{array}{lll} \min & \max\limits_{j \in \mathcal{J}} S_j + p_j & \\ \text{subject to} & S_i + p_i \leq S_j & \forall\,(i,j) \in A \\ & \texttt{cumulative}(\boldsymbol{S}, \boldsymbol{p}, \boldsymbol{r}_{.k}, R_k) & \forall\, k \in \mathcal{R} \end{array} \qquad (1)$$

3 Linear Programming Relaxation and Conflict Analysis

For the implementation we use the CIP solver SCIP which performs a complete search in a branch-and-bound manner. The question to answer is how strongly conflict analysis and LP techniques are involved in the solving process by pruning the search tree. Therefore a first version of separation and conflict analysis methods are implemented for the cumulative constraint.

As IP model we use the formulation of [9] with binary start time variables. In the `cumulative` constraint we generate knapsack constraints [1] from the capacity cuts. Propagation of variable bounds and repropagations of bound changes are left to the solver SCIP. For the `cumulative` constraint bounds are updated according to the concept of *core-times* [7]. The core-time of a job is defined by the interval $[ub_j, \ell b_j + p_j]$. A jobs lower bound can be updated from ℓb_j to $\ell b_j^\star$ if its demand plus the demands of the cores exceed the resource capacity in certain time intervals. An explanation of this bound change is given by the set of jobs that have a core during this interval. More formally, let $\mathcal{C} \subset \mathcal{J}$ be the set of jobs whose core is non-empty, i.e., $ub_j < \ell b_j + p_j$ holds for $j \in \mathcal{C}$. The delivered explanation is the local lower bound of job j itself and the local lower and upper bounds of all jobs $k \in \left\{i \in \mathcal{C} : ub_i < \ell b_j^\star \text{ and } \ell b_i + p_i > \ell b_j\right\}$.

This poses the interesting still open question whether it is NP-hard to find a minimum set of jobs from which the bound change can be derived.

To speed up the propagation process, we filter from the `cumulative` constraints, all pairs of jobs that cannot be executed in parallel and propagate them in a global `disjunctive` bounds constraint. This one propagates and checks the constraints in a more efficient manner and can separate further cuts based on forbidden sets. To get tight primal bounds, we apply a primal heuristic that is based on a fast list scheduling algorithm [8]. If an LP solution is available the list of jobs is sorted according to the start times of the jobs, otherwise by weighted local bounds, and α-points [8]. Furthermore, we apply a *justification* improvement heuristic as

Table 1. Summary of the computational results. Detailed results are given in [4].

	480 instances with 30 jobs							480 instances with 60 jobs						
				Nodes		Time in [s]					Nodes		Time in [s]	
Setting	opt	best	wor.	total(k)	geom	total(k)	geom	opt	best	wor.	total(k)	geom	total(k)	geom
default	460	476	4	3 513	173.2	93.0	7.8	385	395	85	34 104	364.3	350.9	27.3
noconflict	436	467	13	8 665	246.6	175.0	11.6	381	390	90	38 099	381.8	362.9	28.3
norelax	454	467	13	7 444	194.0	106.8	6.5	384	390	90	127 684	591.2	355.8	26.1
none	446	465	15	9 356	217.5	135.5	7.7	382	389	91	126 714	599.3	364.8	26.9
bestset	460	476	4	–	–	–	–	391	401	79	–	–	–	–
LAZYFD	480	480	0	–	–	–	–	429	429	51	–	–	–	–

described in [13] whenever a better solution was found. We use hybrid branching [2] only on integer variables.

4 Computational Results

In this section, we analyze the impact of the two features LP relaxation and conflict analysis for the RCPSP using the test sets of the PSPLIB [10]. Due to the lack of space we restrict ourselves mainly to the test sets containing 30 and 60 jobs. For instances with 120 jobs we report improved lower bounds.

All computations were obtained on Intel Xeon Core 2.66 GHz computers (in 64 bit mode) with 4 MB cache, running Linux, and 8 GB of main memory. We used SCIP [12] version 1.2.0.6 and integrated CPLEX release version 12.10 as underlying LP solver. A time limit of one hour was enforced for each instance.

Table 1 presents the results for different settings which differ by disabled features. The setting "norelax" does not take advantage of the LP relaxation, "noconflict" avoids conflict analysis, "none" stands for disabling both these features whereas "default" enables both. The settings "bestset" is the best of the previous four settings for each instance and the last line reports the results for the solver LAZYFD. We compare for how many instances optimality ("opt") was proven, the best known primal solution ("best") was found, and the primal solution was worse ("wor.") than the best known. Besides that we state total time and number of branch-and-bound nodes over all instances in the test set and the shifted geometric means[1] ("geom") over these two performance measures.

First of all the results show that our approach is competitive to the current best known method [11]. We observe further, that using both features leads to a tremendous reduction of the search space. This does not directly transfer to the running time. From that point of view the relaxation seems to be more expensive as the conflict analysis. On the other hand, the relaxation prunes a greater portion of the search space compared to the reduction achieved by the conflict analysis. Using both features, however, leads to the best performance and indicates the potential of this highly integrated approach.

[1] The shifted geometric mean of values $t_1, \ldots, t_n$ is defined as $\left(\prod(t_i + s)\right)^{1/n} - s$ with shift s. We use a shift $s = 10$ for time and $s = 100$ for nodes in order to decrease the strong influence of the very easy instances in the mean values.

Already with this basic implementation we can improve the lower bounds of five large instances with 120 jobs. These are j12018_3 (and j12018_9) where the new lower bound is 100 (and 88). For j12019_6 (and j12019_9) we obtain lower bounds of 89 (and 87). Finally, we prove a lower bound of 75 for j12020_3.

5 Conclusions

We have shown the power of integrating CP, IP, and SAT techniques into a single approach to solve the RCPSP. Already with our basic implementation we are competitive with both, the best known upper *and* lower bounds, and even improve on a few. There is ample room for improvement, like strengthening the LP relaxation by cutting planes or a dynamic edge-finding which can be exploited using SCIP's re-propagation capabilities. This is subject to current research.

References

1. Achterberg, T.: SCIP: Solving Constraint Integer Programs. Math. Programming Computation 1, 1–41 (2009)
2. Achterberg, T., Berthold, T.: Hybrid branching. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 309–311. Springer, Heidelberg (2009)
3. Baptiste, P., Pape, C.L.: Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. Constraints 5, 119–139 (2000)
4. Berthold, T., Heinz, S., Lübbecke, M.E., Möhring, R.H., Schulz, J.: A constraint integer programming approach for resource-constrained project scheduling, ZIB-Report 10-03, Zuse Institute Berlin (2010)
5. Błażewicz, J., Lenstra, J.K., Kan, A.H.G.R.: Scheduling subject to resource constraints: classification and complexity. Discrete Appl. Math. 5, 11–24 (1983)
6. Hartmann, S., Briskorn, D.: A survey of variants and extensions of the resource-constrained project scheduling problem. Eur. J. Oper. Res. (2009) (in press) (Corrected Proof)
7. Klein, R., Scholl, A.: Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling. Eur. J. Oper. Res. 112, 322–346 (1999)
8. Möhring, R.H., Schulz, A.S., Stork, F., Uetz, M.: Solving project scheduling problems by minimum cut computations. Manage. Sci. 49, 330–350 (2003)
9. Pritsker, A.A.B., Watters, L.J., Wolfe, P.M.: Multi project scheduling with limited resources: A zero-one programming approach. Manage. Sci. 16, 93–108 (1969)
10. PSPLib, *Project Scheduling Problem LIBrary*, http://129.187.106.231/psplib/ (last accessed 2010/February/01)
11. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Why cumulative decomposition is not as bad as it sounds. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 746–761. Springer, Heidelberg (2009)
12. SCIP, Solving Constraint Integer Programs, http://scip.zib.de/
13. Valls, V., Ballestín, F., Quintanilla, S.: Justification and RCPSP: A technique that pays. Eur. J. Oper. Res. 165, 375–386 (2005)

Strategic Planning for Disaster Recovery with Stochastic Last Mile Distribution

Pascal Van Hentenryck[1], Russell Bent[2], and Carleton Coffrin[1]

[1] Brown University, Providence RI 02912, USA
[2] Los Alamos National Laboratory, Los Alamos NM 87545, USA

Abstract. This paper considers the single commodity allocation problem (SCAP) for disaster recovery, a fundamental problem faced by all populated areas. SCAPs are complex stochastic optimization problems that combine resource allocation, warehouse routing, and parallel fleet routing. Moreover, these problems must be solved under tight runtime constraints to be practical in real-world disaster situations. This paper formalizes the specification of SCAPs and introduces a novel multi-stage hybrid-optimization algorithm that utilizes the strengths of mixed integer programming, constraint programming, and large neighborhood search. The algorithm was validated on hurricane disaster scenarios generated by Los Alamos National Laboratory using state-of-the-art disaster simulation tools and is deployed to aid federal organizations in the US.

1 Background and Motivation

Every year seasonal hurricanes threaten coastal areas. The severity of hurricane damage varies from year to year, but considerable human and monetary resources are always spent to prepare for and recover from these disasters. It is policy makers who make the critical decisions relating to how money and resources are allocated for preparation and recovery. At this time, preparation and recovery plans developed by policy makers are often ad hoc and rely on available subject matter expertise. Furthermore, the National Hurricane Center (NHC) of the National Weather Service in the United States (among others) is highly skilled at generating ensembles of possible hurricane tracks but current preparation methods often ignore this information.

This paper aims at solving this problem more rigorously by combining optimization techniques and disaster-specific information given by NHC predictions. The problem is not only hard from a combinatorial optimization standpoint, but it is also inherently stochastic because the exact outcome of the disaster is unknown. Although humans have difficulty reasoning over uncertain data, recent work in the optimization community [13,5] has shown that stochastic optimization techniques can find robust solutions in problems with uncertainty to overcome this difficulty.

The paper considers the following abstract disaster recovery problem: How to store a single commodity throughout a populated area to minimize its delivery time after a disaster has occurred. It makes the following technical contributions:

1. It formalizes the single commodity allocation problem (SCAP).
2. It proposes a multi-stage hybrid-optimization decomposition for SCAPs, combining a MIP model for stochastic commodity storage, a hybrid CP/MIP model for multi-trip vehicle routing, and a large neighborhood search model for minimizing the latest delivery time in multiple vehicle routing.
3. It validates the approach on the delivery of potable water for hurricane recovery.

Section 2 of this paper reviews similar work on disaster preparation and recovery problems. Section 3 presents a mathematical formulation of the disaster recovery problem and sets up the notations for the rest of paper. Section 4 presents the overall approach using (hopefully) intuitive models. Section 5 presents a number of modeling and algorithmic improvements that refines each of the initial models; it also presents the final version of the optimization algorithm for SCAPs. Section 6 reports experimental results of our complete algorithm on some benchmark instances to validate the approach and Section 7 concludes the paper.

2 Previous Work

The operations research community has been investigating the field of humanitarian logistics since the 1990s but recent disasters have brought increased attention to these kinds of logistical problems [18,4,10,9]. Humanitarian logistics is filled with a wide variety of optimization problems that combine aspects from classic problems in inventory routing, supply chain management, warehouse location, and vehicle routing. The problems posed by humanitarian logistics add significant complexity to their classical variants. The operations research community recognizes that novel research in this area is required to solve these kinds of problems [18,4]. Some of the key features that characterize these problems are as follows:

1. **Multi-Objective Functions** - High-stake disaster situations often have to balance conflicting objective goals (e.g. operational costs, speed of service, and unserved customers) [3,8,2,12].
2. **Non-Standard Objective Functions** - A makespan time objective in VRPs [3,6] or equitability objectives [2].
3. **Arbitrary Side Constraints** - Limited resources, a fixed vehicle fleet [2], fixed latest delivery time [3,2], or a insufficient preparation budget [8,11].
4. **Stochastic Aspects** - Disasters are inherently unpredictable. Preparations and recovery plans must be robust with respect to many scenarios [8,12].

Humanitarian logistics also studies these problems at a variety of scales in both space and time. Some problems consider a global scale with time measured in days and weeks [8], while others focus on the minute-by-minute details of delivering supplies from local warehouses directly to the survivors [3,2]. This paper considers a scale which is often called the “last mile” of distribution. This involves warehouse selection and customer delivery at the city and state scale.

The operations research community has mainly formulated these problems using MIP models. Many of the humanitarian logistics problems are complex and MIP formulations do not always scale to real world instances [2,3]. Additionally, it was shown that MIP solvers can have difficulty with some of the unique features of these kinds of problems even when problem sizes are small (e.g., with minimizing the latest delivery time in VRPs [6]). Local search techniques are often used to scale the problems to real world instances [3,6]. This paper demonstrates how hybrid optimization methods and recent advances in the optimization community can yield high-quality solutions to such challenges. To the best of our knowledge, SCAPs are the first humanitarian logistic problem to investigate the "last mile" vehicle routing problem and stochastic disaster information simultaneously.

3 The Single Commodity Allocation Problem (SCAP)

In formalizing SCAPs, a populated area is represented as a graph $G = \langle V, E \rangle$ where V represents those sites of interest to the allocation problem, i.e., sites requiring the commodity after the disaster (e.g., hospitals, shelters, and public buildings) and vehicle storage depots. The required commodity can be stored at any node of the graph subject to some side constraints. For simplicity, we assume the graph is complete and the edges have weights representing travel times. The weights on the edges form a metric space but it is not Euclidean due to the transportation infrastructure. Moreover, the travel times can vary in different disaster scenarios due to road damage. The primary outputs of a SCAP are (1) the amount of commodity to be stored at each node; (2) for each scenario and each vehicle, the best plan to deliver the commodities. Figure 1 summarizes the entire problem, which we now describe in detail.

Objectives. The objective function aims at minimizing three factors: (1) The amount of unsatisfied demands; (2) the time it takes to meet those demands; (3) the cost of storing the commodity. Since these values are not expressed in the same units, it is not always clear how to combine them into a single objective function. Furthermore, their relative importance is typically decided by policy makers on a case-by-case basis. For these reasons, this paper uses weights W_x, W_y, and W_z to balance the objectives and to give control to policy makers.

Side Constraints. The first set of side constraints concerns the nodes of the graph which represent the repositories in the populated area. Each repository $R_{i \in 1..n}$ has a maximum capacity RC_i to store the commodity. It also has a one-time initial cost RI_i (the investment cost) and an incremental cost RM_i for each unit of commodity to be stored. As policy makers often work within budget constraints, the sum of all costs in the system must be less than a budget B.

The second set of side constraints concerns the deliveries. We are given a fleet of m vehicles $V_{i \in 1..m}$ which are homogeneous in terms of their capacity VC. Each vehicle has a unique starting depot D_i^+ and ending depot D_i^-. Unlike

Given:

Repositories: $R_{i \in 1..n}$

Capacity: RC_i

Investment Cost: RI_i

Maintenance Cost: RM_i

Vehicles: $V_{i \in 1..m}$

Capacity: VC

Start Depot: D_i^+

End Depot: D_i^-

Scenario Data: $S_{i \in 1..a}$

Scenario Probability: P_i

Available Sites: $AR_i \subset \{1..n\}$

Site Demand: $C_{i,1..n}$

Travel Time Matrix: $T_{i,1..l,1..l}$

Weights: W_x, W_y, W_z

Budget: B

Output:

The amount stored at each warehouse

Delivery schedules for each vehicle

Minimize:

$W_x *$ Unserved Demands $+$

$W_y * MAX^i_{1..m}$Tour Time$_i +$

$W_z *$ Investment Cost $+$

$W_z *$ Maintenance Cost

Subject To:

Vehicle and site capacities

Vehicles start and end locations

Costs $\leq B$

Notes:

Every warehouse that stores a unit must be visited at least once

Fig. 1. Single Commodity Allocation Problem Specification

classic vehicle routing problems [17], customer demands in SCAPs often exceed the vehicle capacity and hence multiple deliveries are often required to serve a single customer.

Stochasticity. SCAPs are specified by a set of a different disaster scenarios $S_{i \in 1..a}$, each with an associated probability P_i. After a disaster, some sites are damaged and each scenario has a set AR_i of available sites where the stored commodities remain intact. Moreover, each scenario specifies, for each site R_i, the demand C_i. Note that a site may have a demand even if a site is not available. Finally, site-to-site travel times $T_{i,1..l,1..l}$ (where $l = |V|$) are given for each scenario and capture infrastructure damages.

Unique Features. Although different aspects of this problem were studied before in the context of vehicle routing, location routing, inventory management, and humanitarian logistics, SCAPs present unique features. Earlier work in location-routing problems (LRP) assumes that (1) customers and warehouses (storage locations) are disjoint sets; (2) the number of warehouses is $\approx 3..10$; (3) customer demands are less than the vehicle capacity; (4) customer demands are atomic.

None of these assumptions hold in the SCAP context. In a SCAP, it may not only be necessary to serve a customer with multiple trips but, due to the storage capacity constraints, those trips may need to come from different warehouses. The key features of SCAP are: (1) each site can be a warehouse and/or customer; (2) one warehouse may have to make many trips to a single customer; (3) one customer may be served by many warehouses; (4) the number of available vehicles is fixed; (5) vehicles may start and end in different depots; (6) the objective is to minimize the time of the last delivery. Minimizing the time of the last delivery is one of the most difficult aspects of this problem as in demonstrated in [6].

4 The Basic Approach

This section presents the basic approach to the SCAP problem for simplifying the reading of the paper. Modeling and algorithmic improvements are presented in Section 5. Previous work on location routing [7,1,15] has shown that reasoning over both the storage problem and the routing problem simultaneously is extremely hard computationally. To address this difficulty, we present a three-stage algorithm that decomposes the storage, customer allocation, and routing decisions. The three stages and the key decisions of each stage are as follows:

1. **Storage & Customer Allocation:** Which repositories store the commodity and how is the commodity allocated to each customer?
2. **Repository Routing:** For each repository, what is the best customer distribution plan?
3. **Fleet Routing:** How to visit the repositories to minimize the time of the last delivery?

The decisions of each stage are independent and can use the optimization technique most appropriate to their nature. The first stage is formulated as a MIP, the second stage is solved optimally using constraint programming, and the third stage uses large neighborhood search (LNS).

Storage & Customer Allocation. The first stage captures the cost and demand objectives precisely but approximates the routing aspects. In particular, the model only considers the time to move the commodity from the repository to a customer, not the maximum delivery times. Let D be a set of delivery triples of the form $\langle source, destination, quantity \rangle$. The delivery-time component of the objective is replaced by

$$W_y * \sum_{\langle s,d,q\rangle \in D} T_{s,d} * q/VC$$

Figure 2 presents the stochastic MIP model, which scales well with the number of disaster scenarios because the number of integer variables only depends on the number of sites n. The meaning of the decision variables is explained in the figure. Once the storage and customer allocation are computed, the uncertainty is revealed and the second stage reduces to a deterministic multi-depot, multiple-vehicle capacitated routing problem whose objective consists in minimizing the latest delivery. To our knowledge, this problem has not been studied before. One of its difficulties in this setting is that the customer demand is typically much larger than the vehicle capacity. As a result, we tackle it in two steps. We first consider each repository independently and determine a number of vehicle trips to serve the repository customers (Repository Routing). A trip is a tour that starts at the depot, visits customers, returns to the depot, and satisfies the vehicle capacity constraints. We then determine how to route the vehicles to perform all the trips and minimize the latest delivery time (Fleet Routing).

Variables:

$Stored_{i \in 1..n} \in [0, RC_i]$ - Units stored
$Open_{i \in 1..n} \in \{0, 1\}$ - More than zero units stored flag

$StoredSaved_{s \in 1..a, i \in 1..n} \in [0, C_{s,i}]$ - Units used at the storage location
$StoredSent_{s \in 1..a, i \in 1..n} \in [0, RC_i]$ - Total units shipped to other locations
$Incoming_{s \in 1..a, i \in 1..n} \in [0, C_{s,i}]$ - Total units coming from other locations
$Unsatisfied_{s \in 1..a, i \in 1..n} \in [0, C_{s,i}]$ - Demand not satisfied
$Sent_{s \in 1..a, i \in 1..n, j \in 1..n} \in [0, RC_i/VC]$ - Trips needed from i to j

Minimize:

$$W_x * \sum_{s \in 1..a} P_s * \sum_{i \in 1..n} Unsatisfied_{s,i} +$$
$$W_y * \sum_{s \in 1..a} P_s * \sum_{i \in 1..n} \sum_{j \in 1..n} T_{s,i,j} * Sent_{s,i,j} +$$
$$W_z * \sum_{i \in 1..n} (RI_i * Open_i + RM_i * Stored_i)$$

Subject To:

$$\sum_{i \in 1..n} (RI_i * Open_i + RM_i * Stored_i) \leq B$$
$$RC_i * Open_i \geq Stored_i \quad \forall i$$
$$StoredSaved_{s,i} + Incoming_{s,i} + Unsatisfied_{s,i} = C_{s,i} \quad \forall s, i$$
$$StoredSaved_{s,i} + StoredSent_{s,i} \leq Stored_i \quad \forall s, i$$
$$\sum_{j \in 1..n} VC * Sent_{s,i,j} = StoredSent_{s,i} \quad \forall s, i$$
$$\sum_{j \in 1..n} VC * Sent_{s,j,i} = Incoming_{s,i} \quad \forall s, i$$
$$StoredSaved_{s,i} + StoredSent_{s,i} = 0 \quad \forall s, i \text{ where } i \text{ not in } AR_s$$

Fig. 2. Storage & Customer Selection: The MIP Model

Repository Routing. Figure 3 shows how to create the inputs for repository routing from the outputs of the MIP model. For a given scenario s, the idea is to compute the customers of each repository w, the number of full-capacity trips $FullTrips_{s,w,c}$ and the remaining demand $Demand_{s,w,c}$ needed to serve each such customer c. The full trips are only considered in the fleet routing since they must be performed by a round-trip. The minimum number of trips required to serve the remaining customers is also computed using a bin-packing algorithm. The repository routing then finds a set of trips serving these customers with minimal travel time. The repository routing is solved using a simple CP model depicted in Figure 4. The model uses two depots for each possible trip (a starting and an ending depot localized at the repository) and considers nodes consisting of the depots and the customers. Its decision variables are the successor variables specifying which node to visit next and the trip variables associating a trip with each customer. The circuit constraint expresses that the successor variables constitute a circuit, the vehicle capacity constraint is enforced with a multi-knapsack constraint, and the remaining constraints associate a trip number with every node. This model is then solved to optimality.

```
Given scenario $s$ and for each repository $w \in 1..n$
    $Customers_{s,w} = \{i \in 1..n : Sent_{s,w,i} > 0\}$
    For $c \in Customers_{s,w}$
        $FullTrips_{s,w,c} = \lfloor Sent_{s,w,c} \rfloor$
        $Demand_{s,w,c} = VC * (Sent_{s,w,c} - \lfloor Sent_{s,w,c} \rfloor)$
    $MinTrips_{s,w} = MinBinPacking(\{Demand_{s,w,c} : c \in Customers_{s,w}\}, VC)$
```

Fig. 3. The Inputs for the Repository Routing

```
Let:
    $Depots^{+}_{s,w} = \{d^{+}_{1}, d^{+}_{2}, ..., d^{+}_{MinTrips_{s,w}}\}$
    $Depots^{-}_{s,w} = \{d^{-}_{1}, d^{-}_{2}, ..., d^{-}_{MinTrips_{s,w}}\}$
    $Nodes_{s,w} = Depots^{+}_{s,w} \cup Depots^{-}_{s,w} \cup Customers_{s,w}$
    $Trips_{s,w} = \{1, 2, ..., MinTrips_{s,w}\}$

Variables:
    $Successor[Nodes_{s,w}] \in Nodes_{s,w}$ - Node traversal order
    $Trip[Nodes_{s,w}] \in Trips_{s,w}$ - Node trip assignment

Minimize:
    $\sum_{n \in Nodes_{s,w}} T_{s,n,Successor[n]}$

Subject To:
    $circuit(Successor)$
    $multiknapsack(Trip, \{Demand_{s,w,c} : c \in Customers_{s,w}\}, VC)$
    for $w^{+}_{i} \in Depots^{+}_{s,w}$: $Trip[w^{+}_{i}] = i$
    for $w^{-}_{i} \in Depots^{-}_{s,w}$: $Trip[w^{-}_{i}] = i$
    for $n \in Customers_{s,w} \cup Depots^{+}_{s,w}$: $Trip[n] = Trip[Successor[n]]$
```

Fig. 4. The CP Model for Repository Routing

```
Given scenario $s$ and for each repository $w \in 1..n$
    $RoundTrips_{s,w} = \{FullTrips_{s,w,c} : c \in Customers_{s,w}\}$
    $Tasks_{s,w} = \{t_1, t_2, ..., t_{Trips_{s,w}}\} \cup RoundTrips_{s,w}$
    For each $t \in RoundTrips_{s,w}$
        $TripTime_t = 2\ T_{s,w,c}$
    For $t \in Tasks_{s,w} \setminus RoundTrips_{s,w}$
        $TaskNodes_t = \{n \in Nodes_{s,w} \setminus Depots^{-}_{s,w} : Trip[n] = t\}$
        $TripTime_t = \sum_{n \in TaskNodes_t} T_{s,n,Successor[n]}$
```

Fig. 5. The Inputs for the Fleet Routing

Fleet Routing. It then remains to decide how to schedule the trips for the fleet to perform and to minimize the latest delivery time. The capacity constraints can be ignored now since each trip satisfies them. Each trip is abstracted into a task at the warehouse location and a service time capturing the time to perform

```
Let:
    $Vehicles_s = \{1, 2, ..., m\}$
    $StartNodes_s = \{D_1^+, \dots, D_m^+\}$
    $EndNodes_s = \{D_1^-, \dots, D_m^-\}$
    $Nodes_s = StartNodes_s \cup EndNodes_s \cup \bigcup_{w \in 1..n} Tasks_{s,w}$
Variables:
    $Successor[Nodes_s] \in Nodes_s$ - Node traversal order
    $Vehicle[Nodes_s] \in Vehicles_s$ - Node vehicle assignment
    $DelTime[Nodes_s] \in \{0, \dots, \infty\}$ - Delivery time

Minimize:
    $MAX_{n \in Nodes_s} DelTime(n)$

Subject To:
    $circuit(Successor)$
    for $n \in StartNodes_s$ such that $n = D_i^+$
        $Vehicle[n] = i$
        $DelTime[n] = Time_{s,n}$
        $DelTime[Successor[n]] = DelTime[n] + TripTime_n + T_{s,n,Successor[n]}$
    for each $n \in EndNodes_s$ such that $n = D_i^-$
        $Vehicle[n] = i$
    for $n \in Nodes_s \setminus StartNodes_s \setminus EndNode_s$
        $Vehicle[n] = Vehicle[Successor[n]]$
        $DelTime[Successor[n]] = DelTime[n] + TripTime_n + T_{s,n,Successor[n]}$
```

Fig. 6. The CP Model for Fleet Routing

the trip. The fleet routing problem then consists of using the vehicles to perform all these tasks while minimizing the latest delivery.

Figure 5 depicts how to compute the inputs for fleet routing given the results of the earlier steps, which consists of computing the proper service times $TripTime_t$ for each trip t. The model for the fleet routing is depicted in Figure 6 and is a standard CP formulation for multiple vehicle routing adapted to minimize the latest delivery time. For each node, the decision variables are its successor, its vehicle, and its delivery time. The objective minimizes the maximum delivery time and the rest of the model expresses the subtour elimination constraints, the vehicle constraints, and the delivery time computation.

The fleet routing problem is solved using LNS [16] to obtain high-quality solutions quickly given the significant number of nodes arising in large instances. At each optimization step, the LNS algorithm selects 15% of the trips to relax, keeping the rest of the routing fixed. The neighborhood is explored using constraint programming allowing up to $(0.15|Nodes_s|)^3$ backtracks.

5 Modeling and Algorithmic Enhancements

We now turn to some modeling and algorithmic improvements to the basic approach which bring significant benefits on real-life applications.

Customer Allocation. The assignment of customers to repositories is a very important step in this algorithm because it directly determines the quality of the trips computed by the repository routing and there is no opportunity for correction. Recall that Section 4 uses

$$\sum_{i \in D}^{i=(s,d,q)} T_{s,d} * q/VC$$

as an approximation of travel distance. Our experimental results indicate that this approximation yields poor customer-to-warehouse allocation when there is an abundance of commodities. To resolve this limitation, we try to solve a slightly stronger relaxation, i.e.,

$$\sum_{i \in D}^{i=(s,d,q)} T_{s,d} * \lceil q/VC \rceil$$

but this ceiling function is too difficult for the stochastic MIP model. Instead, we decompose the problem further and separate the storage and allocation decisions. The stochastic MIP now decides which repository to open and how much of the commodity to store at each of them. Once these decisions are taken and once the uncertainty is revealed (i.e., the scenario s becomes known), we solve a customer allocation problem, modeled as a MIP (see Figure 7). This problem must be solved quickly since it is now considered after the uncertainty is revealed. Unfortunately, even this simplified problem can be time consuming to solve optimally. However, a time limit of between 30 and 90 seconds results in solutions within 1% (on average) of the best solution found in one hour. Our results indicate that even suboptimal solutions to this problem yield better customer allocation than those produced by the stochastic MIP.

Path-Based Routing. The delivery plans produced by the basic approach exhibit an obvious limitation. By definition of a trip, the vehicle returns to the repository at the end of trip. In the case where the vehicle moves to another repository next, it is more efficient to go directly from its last delivery to the next repository (assuming a metric space which is the case in practice). To illustrate this point, consider Figure 8 which depicts a situation where a customer (white node) receives deliveries from multiple repositories (shaded nodes). The figure shows the savings when moving from a tour-based (middle picture) to a path-based solution (right picture). It is not difficult to adapt the algorithm from a tour-based to a path-based routing. In the repository routing, it suffices to ignore the last edge of a trip and to remember where the path ends. In the fleet routing, only the time matrix needs to be modified to account for the location of the last delivery.

Set-Based Repository Routing. The SCAP problems generated by hurricane simulators have some unique properties that are not common in traditional VRPs. One of these features appears during repository routing: The first stage solution generates customer demands that are distributed roughly uniformly through the range $0..VC$. This property allows for a repository-routing formulation that

Variables:

$Sent_{i\in 1..n, j\in 1..n} \in [0, Stored_i]$ - Units moved from i to j

$VehicleTrips_{i\in 1..n, j\in 1..n} \in [0..\lceil Stored_i/VC \rceil]$ - Trips needed from i to j

Minimize:

$$W_x * \sum_{i\in 1..n} (C_{s,i} - \sum_{j\in 1..n} Sent_{j,i}) +$$
$$W_y * \sum_{i\in 1..n} \sum_{j\in 1..n} T_{s,i,j} * VehicleTrips_{i,j}$$

Subject To:

$$\sum_{j\in 1..n} Sent_{i,j} \leq Stored_i \quad \forall i$$
$$\sum_{j\in 1..n} Sent_{j,i} \leq C_{s,i} \quad \forall i$$
$$Sent_{i,j} = 0 \quad \forall i, j \text{ where } i \text{ not in } AR_s$$
$$VehicleTrips_{i,j} \geq Sent_{i,j}/VC \quad \forall i, j$$

Fig. 7. The MIP Model for Customer Allocation

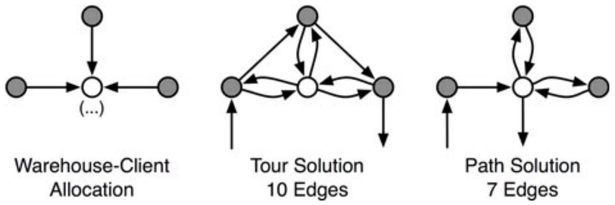


Fig. 8. Illustrating the Improvement of Path-Based Routing

scales much better than the pure CP formulation described earlier. Indeed, if the customer demands $d_1, \ldots, d_c$, are uniformly distributed in the range $0..VC$, the number of sets satisfying the vehicle capacity is smaller than c^3 when c is not too large (e.g., $c \leq 50$). This observation inspires the following formulation:

1. Use CP to enumerate all customer sets satisfying the capacity constraint.
2. Use CP to compute an optimal trip for those customer sets.
3. Use MIP to find a partition of customers with minimal delivery time.

This hybrid model is more complex but each subproblem is small and it scales much better than the pure CP model.

Aggregate Fleet Routing. The most computationally intense phase is the fleet routing and we now investigate how to initialize the LNS search with a high-quality solution. Recall that the fleet routing problem associates a node with every trip. Given a scenario s, a lower bound for the number of trips is,

$$\sum_{i\in 1..n} StoredSent_{s,i}/VC$$

```
MULTI-STAGE-SCAP($\mathcal{G}$)
1  $\mathcal{D} \leftarrow StochasticStorageMIP(\mathcal{G})$
2  for $s \in 1..a$
3  do $\mathcal{C} \leftarrow CustomerAllocationProblem(\mathcal{G}_s, \mathcal{D}_s)$
4      for $w \in 1..n$
5      do $\mathcal{T} \leftarrow RepositoryPathRoutingProblem(\mathcal{G}_s, \mathcal{C}_w)$
6      $\mathcal{I} \leftarrow AggregateFleetRouting(\mathcal{G}_s, \mathcal{T})$
7      $\mathcal{S}_s \leftarrow TripBasedFeetRouting(\mathcal{G}_s, \mathcal{T}, \mathcal{I})$
8  return $\mathcal{S}$
```

Fig. 9. The Final Hybrid Stochastic Optimization Algorithms for SCAPs

Clearly, the size and complexity of this problem grows with the amount of commodities moved. To find high-quality solutions to the fleet routing subtask, the idea is to aggregate the trips to remove this dependence on the amount of commodities delivered. More precisely, we define an aggregate fleet routing model in which all trips at a repository are replaced by an aggregate trip whose service time is the sum of all the trip service times. The number of nodes in the aggregate problem is now proportional to the number of repositories. Finding a good initial solution is not important for smaller problems (e.g., $n \approx 25, m \approx 4$), but it becomes critical for larger instances (e.g., $n \approx 100, m \approx 20$). Since the aggregate problem is much simpler, it often reaches high-quality solution quickly.

The Final Algorithm. The final algorithm for solving a SCAP instance $\mathcal{G}$ is presented in Figure 9.

6 Benchmarks and Results

Benchmarks. The benchmarks were produced by Los Alamos National Laboratory and are based on the infrastructure of the United States. The disaster scenarios were generated by state-of-the-art hurricane simulation tools similar to those used by the National Hurricane Center. Their sizes are presented in Table 1 (The table also depicts the time limit used for fleet routing). Benchmark 3 features one scenario where the hurricane misses the region; this results in the minimum demand being zero. This is important since any algorithm must be robust with respect to empty disaster scenarios which arise in practice when hurricanes turn away from shore or weaken prior to landfall. All of the experimental results have fixed values of W_x, W_y, and W_z satisfying the field constraint $W_x > W_y > W_z$ and we vary the value of the budget B to evaluate the algorithm. The results are consistent across multiple weight configurations, although there are variations in the problem difficulties. It is also important to emphasize that, on these benchmarks, the number of trips is in average between 2 and 5 times the number of repositories and thus produces routing problems of significant sizes.

The Algorithm Implementation and the Baseline Algorithm. The final algorithm was implemented in the COMET system [14] and the experiments were run on

Table 1. SCAP Benchmark Statistics

Benchmark	n	m	a	Min Demand	Max Demand	Timeout
BM1	25	4	3	550	2700	30
BM2	25	5	3	6000	8384	60
BM3	25	5	3	0	11000	60
BM4	30	5	3	3500	11000	90
BM5	100	20	3	8200	22000	600

Table 2. SCAP Benchmark Runtime Statistics (Seconds)

Benchmark	$\mu(T_1)$	$\sigma(T_1)$	$\mu(T_\infty)$	$\sigma(T_\infty)$	$\mu(STO)$	$\sigma(STO)$	$\mu(CA)$	$\mu(RR)$	$\mu(AFR)$	$\mu(FR)$
BM1	196.3	18.40	78.82	9.829	0.9895	0.5023	11.78	0.2328	23.07	30.00
BM2	316.9	59.00	120.2	20.97	0.5780	0.2725	16.83	0.2343	28.33	60.00
BM3	178.4	15.89	102.1	15.02	0.3419	0.1714	7.192	0.1317	11.98	40.00
BM4	439.8	48.16	169.0	22.60	0.9093	0.4262	22.71	0.2480	33.28	90.00
BM5	3179	234.8	1271	114.5	46.71	25.05	91.06	1.0328	351.7	600.0

Intel Xeon CPU 2.80GHz machines running 64-bit Linux Debian. To validate our results, we compare our delivery schedules with those of an agent-based algorithm. The agent-based algorithm uses the storage model but builds a routing solution without any optimization. Each vehicle works independently to deliver as much commodity as possible using the following heuristic:

```
Greedy-Truck-Agent()
1  while ∃ commodity to be picked up ∧ demands to be met
2     do if I have some commodity
3           then drop it off at the nearest demand location
4           else pick up some water from the nearest warehouse
5  goto final destination
```

This agent-based algorithm roughly approximates current relief delivery procedures and is thus a good baseline for comparison.

Efficiency Results. Table 2 depicts the runtime results. In particular, the table reports, on average, the total time in seconds for all scenarios (T_1), the total time when the scenarios are run in parallel (T_∞), the time for the storage model (STO), the client-allocation model (CA), the repository routing (RR), the aggregate fleet routing (AFR), and fleet routing (FR). The first three fields(T_1, T_∞, STO) are averaged over ten identical runs on each of the budget parameters. The last four fields (CA, RR, AFR, FR) are averaged over ten identical runs on each of the budget parameters and each scenario. Since these are averages, the times of the individual components do not sum to the total time. The results show that the approach scales well with the size of the problems and is a practical approach for solving SCAPs.

Table 3. Improvements over the Baseline Algorithm

Benchmark	BM1	BM2	BM3	BM4	BM5
Improvement(%)	57.7	40.6	68.8	51.7	50.6

Table 4. Correlations for the Distances in Customer Allocation and Fleet Routing

Benchmark	BM1	BM2	BM3	BM4	BM5
Correlation	0.9410	0.9996	0.9968	0.9977	0.9499

Table 5. The Difference in Delivery Times Between Vehicles

Benchmark	BM1	BM2	BM3	BM4	BM5
Absolute Difference	6.7	59.4	39.5	49.1	749
Relative Difference(%)	10.7	12.8	6.7	8.7	46.2

Quality of the Results. Table 3 depicts the improvement of our SCAP algorithm over the baseline algorithm. Observe the significant and uniform benefits of our approach which systematically delivers about a 50% reduction in delivery time.

Table 4 describes the correlations between the distances in the customer allocation and fleet routing models. The results show strong correlations, indicating that the distances in the customer allocation model are a good approximation of the actual distances in the fleet routing model. Table 5 also reports results on the absolute and relative differences between vehicles in the solutions. They indicate that the load is nicely balanced between the vehicles. More precisely, the maximum delivery times are often within 10% of each other on average, giving strong evidence of the quality of our solutions. Benchmark 5 is an exception because it models emergency response at a state level, not at a city level. In that benchmark, some vehicles have a significantly reduced load because they would have to travel to the other side of the state to acquire more load, which would take too much time to reduce the maximum delivery objective.

Behavioral Analysis. Figure 10 presents the experimental results on benchmark 5 (other benchmarks are consistent, but omitted for space reasons). The graph on the left shows how the satisfied demand increases with the budget while the graph on the right shows how the last delivery time changes. Given the weight selection, it is expected that the demand and routing time will increase steadily as the budget increases until the total demand is met. At that point, the demand should stay constant and the routing time should decrease. The results confirm this expectation. The experimental results also indicate the significant benefits provided by our approach compared to the baseline algorithm.

Fleet Routing. Figure 11 presents experimental results comparing aggregate (AFR), tour-based (TFR), and path-based (PFR) fleet routing (Only BM1 is presented but other results are consistent). The key insight from these results

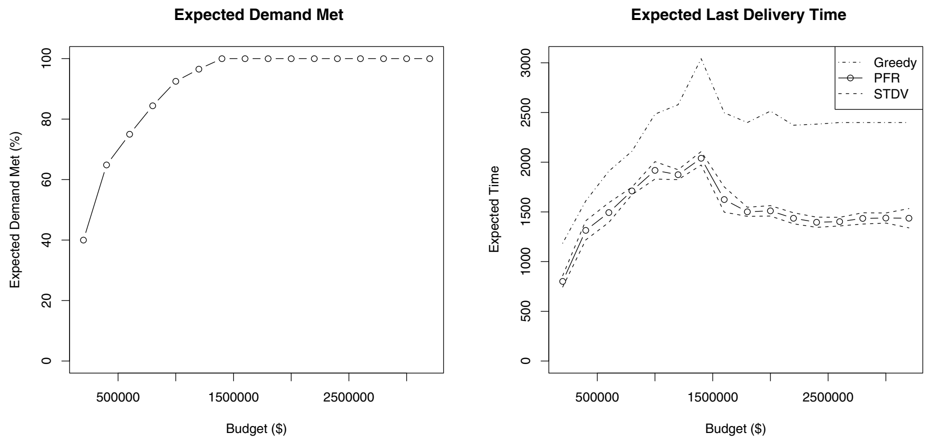


Fig. 10. Varying the Budget on Benchmark 5

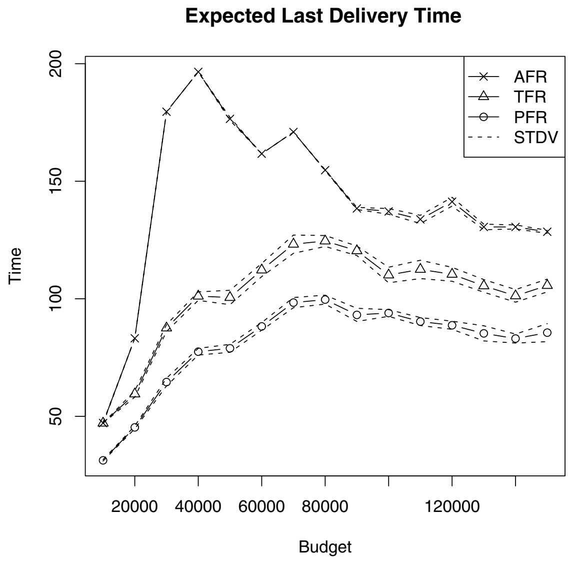


Fig. 11. Comparing Various Algorithms for Fleet Routing

is to show the benefits of allowing the trips of a repository to be performed by multiple vehicles. Note also the significant improvements obtained by considering paths instead of tours.

Customer Allocation. As mentioned earlier, the benefits of separating customer allocation from storage decisions are negligible when the budget is small. However, they become significant when the budget increases and can produce a reduction by up to 16% of the expected maximum delivery time.

7 Conclusion

This paper studied a novel problem in the field of humanitarian logistics, the Single Commodity Allocation Problem (SCAP). The SCAP models the strategic planning process for disaster recovery with stochastic last mile distribution. The paper proposed a multi-stage stochastic hybrid optimization algorithm that yields high quality solutions to real-world benchmarks provided by Los Alamos National Laboratory. The algorithm uses a variety of technologies, including MIP, constraint programming, and large neighborhood search, to exploit the structure of each individual optimization subproblem. The experimental results on water allocation benchmarks indicate that the algorithm is practical from a computational standpoint and produce significant improvements over existing relief delivery procedures. This work is currently deployed at LANL as part of its mission to aid federal organizations in planning and responding to disasters.

References

1. Albareda-Sambola, M., Diaz, J.A., Fernandez, E.: A compact model and tight bounds for a combined location-routing problem. Computer & Operations Research 32, 407–428 (2005)
2. Balcik, B., Beamon, B., Smilowitz, K.: Last mile distribution in humanitarian relief. Journal of Intelligent Transportation Systems 12(2), 51–63 (2008)
3. Barbarosoglu, G., Özdamar, L., Çevik, A.: An interactive approach for hierarchical analysis of helicopter logistics in disaster relief operations. European Journal of Operational Research 140(1), 118–133 (2002)
4. Beamon, B.: Humanitarian relief chains: Issues and challenges. In: 34th International Conference on Computers & Industrial Engineering, pp. 77–82 (2008)
5. Bianchi, L., Dorigo, M., Gambardella, L., Gutjahr, W.: A survey on metaheuristics for stochastic combinatorial optimization. Natural Computing 8(2) (2009)
6. Campbell, A.M., Vandenbussche, D., Hermann, W.: Routing for relief efforts. Transportation Science 42(2), 127–145 (2008)
7. Burke, L.I., Tuzun, D.: A two-phase tabu search approach to the location routing problem. European Journal of Operational Research 116, 87–99 (1999)
8. Duran, S., Gutierrez, M., Keskinocak, P.: Pre-positioning of emergency items worldwide for care international. Interfaces (2008) (submitted)
9. Fritz institute (2008), http://www.fritzinstitute.org
10. United States Government. The federal response to hurricane katrina: Lessons learned (2006)
11. Griffin, P., Scherrer, C., Swann, J.: Optimization of community health center locations and service offerings with statistical need estimation. IIE Transactions (2008)
12. Gunnec, D., Salman, F.: A two-stage multi-criteria stochastic programming model for location of emergency response and distribution centers. In: INOC (2007)
13. Kall, P., Wallace, S.W.: Stochastic Programming. Wiley Interscience Series in Systems and Optimization. John Wiley & Sons, Chichester (1995)
14. Comet 2.1 User Manual. Dynadec website, http://dynadec.com/

15. Nagy, G., Salhi, S.: Nested heuristic methods for the location-routing problem. Journal of Operational Research Society 47, 1166–1174 (1996)
16. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M.J., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998)
17. Toth, P., Vigo, D.: The Vehicle Routing Problem. SIAM Monographs on Discrete Mathematics and Applications, Philadelphia, Pennsylvania (2001)
18. Van Wassenhove, L.: Humanitarian aid logistics: supply chain management in high gear. Journal of the Operational Research Society 57(1), 475–489 (2006)

Massively Parallel Constraint Programming for Supercomputers: Challenges and Initial Results

Feng Xie[1] and Andrew Davenport[2]

[1] Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada
xief@mcmaster.ca

[2] IBM T. J. Watson Research Center, Yorktown Heights, NY, USA
davenport@us.ibm.com

1 Introduction

In this paper we present initial results for implementing a constraint programming solver on a massively parallel supercomputer where coordination between processing elements is achieved through message passing. Previous work on message passing based constraint programming has been targeted towards clusters of computers (see [1,2] for some examples). Our target hardware platform is the IBM Blue Gene supercomputer. Blue Gene is designed to use a large number of relatively slow (800MHz) processors in order to achieve lower power consumption, compared to other supercomputing platforms. Blue Gene/P, the second generation of Blue Gene, can run continuously at 1 PFLOPS and can be scaled to 884,736-processors to achieve 3 PFLOPS performance. We present a dynamic scheme for allocating sub-problems to processors in a parallel, limited discrepancy tree search [3]. We evaluate this parallelization scheme on resource constrained project scheduling problems from PSPLIB [4].

2 A Dynamic Parallelization Scheme

Parallelization of search algorithms over a small number of processors or cores can often be achieved by statically decomposing the problem into a number of disjoint sub-problems as a preprocessing step, prior to search. This might be achieved by fixing some variables to different values in each sub-problem (as is explored in [5]). The advantage of such a static decomposition scheme is that each processor can work independently on its assigned part of the search space and communication is only needed to terminate the solve. When scaling this static decomposition scheme to large numbers of processors, this approach may sometimes lead to poor load-balancing and processor idle time.

Dynamic work allocation schemes partition the search space among processors in response to the evolving search tree, for example by reassigning work among processors during problem solving. Work-stealing is an example of a dynamic decomposition scheme that has been used in programming languages such as CILK [6], and in constraint programming [7] on shared memory architectures. We have developed a simple dynamic load balancing scheme for distributed hardware environments based on message passing. The basic idea behind the approach is that (a) the processors are divided into master and worker processes; (b) each worker processor is assigned sub-trees to explore by a master; and (c) the master processors are responsible for coordinating the sub-trees assigned to worker processors. A master process has a global view of the full search tree. It keeps track of which sub-trees have been explored and which are to be explored. Typically a single master processor is coordinating many worker processors. Each worker processor implements a tree-based search. The master processor assigns sub-problems to each worker, where each sub-problem is specified by a set of constraints. These constraints are communicated in a serialized form using message-passing. A worker may receive a new sub-problem from its assigned master processor either at the beginning of problem solving, or during problem solving after exhausting tree search on its previously assigned sub-problem. On receiving a message from its master specifying a sub-problem as a set of constraints, a worker processor will establish an initial state to start tree search by creating and posting constraints to its constraint store based on this message.

2.1 Problem Pool Representation

A problem pool is used by the master to keep track of which parts of the search space have been explored by the worker processors, which parts are being explored and which parts are remaining to be explored. Each master processor maintains a *job tree* to keep track of this information. A job tree is a representation of the tree explored by the tree search algorithm generated by the worker processors. A node in the job tree represents the state of exploration of the node, with respect to the master's worker processors. Each node can be in one of three states: *explored*, where the sub tree has been exhausted; *exploring*, where the subtree itself is assigned to a worker, and no result is received; or *unexplored*, where the subtree has not been assigned to any worker. An edge in a job tree is labelled with a representation of a constraint posted at the corresponding branch in the search tree generated by the tree search algorithm executed by the worker processors. A job tree is dynamic structure that indicates how the whole search tree is partitioned among the workers at a certain time point in problem solving. In order to minimize the memory use and shorten the search time for new jobs, a job tree is expanded and shrunk dynamically in response to communications with the worker processors. When a worker processor become idle (or at their initialization) they request work from their master processor. In response to such a request, a master processor will look up a node in its job tree which is in an unexplored state, and send a message to the worker processor consisting of the sub-problem composed of the serialized set of constraints on the edges from the root node of the job tree to the node.

2.2 Work Generation

Work generation occurs firstly during an initialization phase of the solve, and then dynamically during the solve itself. The initial phase of work generation involves creating the initial job tree for each of the master processors. The master processor creates its initial job tree by exploring some part of the search space of the problem, up to some (small) bound on the number of nodes to explore. If during this initial search a solution is found, the master can terminate. Otherwise, the master initializes its job tree from the search tree explored during this phase. The master processor then enters into a job dispatching loop where it responds to requests for job assignments from the worker processors.

The second phase of work generation occurs as workers themselves explore the search space of their assigned sub-problems and detect that they are exploring a large search tree which requires further parallelization. Job expansion is a mechanism for a worker to release free jobs if it detects that it is working on a large subtree. We use a simple scheme based on a threshold of searched nodes as a rough indicator of the "largeness" of the job subtree. If the number of nodes searched by a worker exceeds this threshold without exhausting the subtree or finding a solution, the worker will send a job expansion request to its master and pick a smaller part of the job to keep working on. Meanwhile, the master updates the job tree using the information offered by the worker, eventually dispatching the remaining parts of the original search tree to other worker processors.

Job expansion has two side effects. First, it introduces communication overhead because the job expansion information needs to be sent from the worker processor to the master processor. Secondly, the size of the job tree may become large, slowing down the search for unexplored nodes in response to worker job requests. The job tree can be pruned when all siblings of some explored node n are explored. In this case, the parent of node n can be rendered as explored and the siblings can be removed from the job tree.

2.3 Job Dispatching

A master process employs a tree search algorithm to look for unexplored nodes in its job tree in response to job requests from the workers. The search algorithm used by the master to dispatch unexplored nodes in the job tree is customizable. It partially determines how the search tree is traversed as a whole. If a worker makes a job request and no unexplored nodes are available, the state of the worker is changed to idle. Once new jobs become available, the idle workers are woken up and dispatched these jobs.

2.4 Multiple Master Processes

The results presented in this section are from execution runs on BlueGene/L on instances of resource constrained project scheduling problems from PSPLIB [4]. The constraint programming solver executed by the worker processes uses the SetTimes branching heuristic and timetable and edge-finding resource constraint propagation [8]. The job expansion threshold is set at 200 nodes. Due to space limitations, we only present limited results in this section. However they are representative of what we see for other problems in PSPLIB.

Figure 1 (left) shows the scaling performance of the parallelization scheme with a single master process, as we vary the number of processors from 64 to

1024 (on the 120 activity RCPSP instance 1-2 from PSPLIB). We manage to achieve good linear scaling up to 256 processors. However the single master process becomes a bottleneck when we have more than 256 worker processors, where we see overall execution time actually slow down as we increase the number of processors beyond 256.

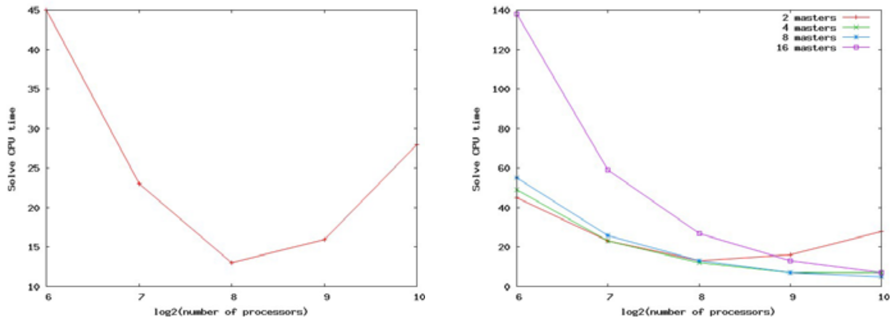


Fig. 1. Scaling with one master process (left) and multiple (right) master processes

The master processor can be a bottleneck as the number of workers assigned to it increases. In this case, multiple masters can be used to improve scalability. In the multi-master mode, the full search tree is divided among the masters at the beginning of problem solving. This is a static decomposition scheme, in that sub-trees are not reallocated between masters dynamically during search. We implemented a simple static decomposition scheme based on measuring the reduction in search space size as we evaluate possible branches from the root node. We then distribute the sub-trees resulting from following these branches among the masters, so that as much as possible each master has a similar sized search space to explore. Figure 1 (right) plots the scaling performance of the scheduler (on the 120 activity RCPSP instance 1-2 from PSPLIB) with multiple master processes, as we vary the number of processors from 64 to 1024. We see here that with multiple master processors, we can achieve good scaling up to 1024 processors. We also present results showing execution time scaling for solving feasible makespan satisfaction problems from PSPLIB in Table 1, for varying numbers of processors p. A single master is used for all the test cases except for 256 and 512 processes, where we used two and four masters respectively.

To summarize, with a single master processor, we are able to achieve good scaling up to 256 processors. With multiple masters, we achieve good scaling up to 512 and sometimes 1024 processors. However the decomposition scheme used to distribute sub-problems over multiple masters can impact scaling. In our experiments we have not managed to achieve good scaling with greater than 1024 processors and multiple masters. We believe that to scale well beyond 1024 processors requires developing techniques to dynamically allocate job trees between multiple masters.

Table 1. Execution time (in seconds) for solving fixed makespan satisfaction PSPLIB resource-constrained project scheduling problems with 60 and 90 activities

Problem (makespan)	Size	CPU time					
		p=16	p=32	p=64	p=128	p=256	p=512
14-4 (65)	60	30	14	7.0	3.1	2.1	2.0
26-3 (76)	60	>600	>600	90	75	24	10
26-6 (74)	60	63	18	8.1	5.0	2.0	1.0
30-10 (86)	60	>600	>600	>600	>600	216	88
42-3 (78)	60	>600	>600	>600	>600	256	81
46-3 (79)	60	148	27	13	6.0	3.1	2.0
46-4 (74)	60	>600	>600	>600	>600	104	77
46-6 (90)	60	>600	>600	477	419	275	122
14-6 (76)	90	>600	371	218	142	48	25
26-2 (85)	90	294	142	86	35	16	9.0
22-3 (83)	90	50	24	12	5	3.0	0.07

References

1. Michel, L., See, A., Van Hentenryck, P.: Transparent parallelization of constraint programming. INFORMS Journal of Computing (2009)
2. Duan, L., Gabrielsson, S., Beck, J.: Solving combinatorial problems with parallel cooperative solvers. In: Ninth International Workshop on Distributed Constraint Reasoning (2007)
3. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: 14th International Joint Conference on Artificial Intelligence (1995)
4. Kolisch, R., Sprecher, A.: PSPLIB - a project scheduling library. European Journal of Operational Research 96, 205–216 (1996)
5. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: Twenty-first International Joint Conference on Artificial Intelligence, IJCAI 2009 (2009)
6. Blumofe, R.D., Joerg, C.F., Bradley, C.K., Leiserson, C.E., Randall, K., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 207–216 (1995)
7. Michel, L., See, A., Van Hentenryck, P.: Parallelizing constraint programs transparently. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 514–528. Springer, Heidelberg (2007)
8. Baptiste, P., Pape, C.L., Nuijten, W.: Constraint-Based Scheduling - Applying Constraint Programming to Scheduling Problems. International Series in Operations Research and Management Science. Springer, Heidelberg (2001)

Boosting Set Constraint Propagation for Network Design

Justin Yip[1], Pascal Van Hentenryck[1], and Carmen Gervet[2]

[1] Brown University, Box 1910, Providence, RI 02912, USA
[2] German University in Cairo, New Cairo, Egypt

Abstract. This paper reconsiders the deployment of synchronous optical networks (SONET), an optimization problem naturally expressed in terms of set variables. Earlier approaches, using either MIP or CP technologies, focused on symmetry breaking, including the use of SBDS, and the design of effective branching strategies. This paper advocates an orthogonal approach and argues that the thrashing behavior experienced in earlier attempts is primarily due to a lack of pruning. It studies how to improve domain filtering by taking a more global view of the application and imposing redundant global constraints. The technical results include novel hardness results, propagation algorithms for global constraints, and inference rules. The paper also evaluates the contributions experimentally by presenting a novel model with static symmetric-breaking constraints and a static variable ordering which is many orders of magnitude faster than existing approaches.

1 Introduction

This paper reconsiders the deployment of synchronous optical networks (SONET), an optimization problem originally studied in the operation research community[1]. The SONET problem is defined in terms of a set of clients and a set of communication demands between pairs of clients who communicate through optical rings. The task is to allocate clients on (possibly multiple) rings, satisfying the bandwidth constraints on the rings and minimizing the equipment cost. This problem has been tackled previously using mixed integer programming (MIP)[1] and constraint programming (CP)[2,3]. Much attention was devoted to variable branching heuristics and breaking ring symmetries (since all rings are identical). It was shown that sophisticated symmetry-breaking techniques dramatically reduce the computational times, both for MIP and CP formulations. The difficulty of finding good branching heuristics, which do not clash with symmetry breaking, was also mentioned.

This paper takes another look at the problem and studies the possibility that the thrashing behavior experienced in earlier attempts is primarily due to lack of pruning. The key observation is that existing models mainly consist of binary constraints and lack a global perspective. Instead of focusing on symmetry breaking and branching heuristics, we study how to strengthen constraint propagation by adding redundant global set-constraints. We propose two classes of

Table 1. Overview of Hardness of Complete Filtering Algorithms

		sb-domain	sbc-domain
NonEmptyIntersection	$\|X \cap Y\| \geq 1$	Polynomial	Polynomial (Thm. 2)
AllNonEmptyIntersection	$\forall i, \|X \cap Y_i\| \geq 1$	Polynomial (Thm. 3)	NP-hard (Thm. 5)
SubsetOfUnion	$\bigcup_i Y_i \supseteq X$	Polynomial (Thm. 6)	?
SubsetOfOpenUnion	$\bigcup_{i \in Y} X_i \supseteq s$	Polynomial (Thm. 8)	NP-hard (Thm. 9)

redundant constraints and we investigate the complexity of these set constraints and the design of filtering algorithms. Like many other global constraints for set variables [4,5], complete filtering algorithms are often intractable but we propose inference rules that can reduce the search space effectively. The considered set constraints, their complexity results, and some of the open questions, are summarized in Table 1. The technical results were evaluated experimentally on the standard SONET benchmarks. They indicate that the enhanced model, with static symmetry-breaking constraints and a static variable ordering, is many orders of magnitude faster than existing approaches.

This paper is organized as follows. Section 2 gives a formal description of the SONET problem and its CP model. Section 3 recalls basic definitions about set domains and fixes the notation used in the paper. Sections 4–8 constitute the core of the paper and study the various constraints used in the model. Section 9 presents the experimental results and Section 10 concludes the paper.

2 The SONET Problem

Problem Description. The SONET problem [1] is a network topology design problem for optical fiber network, the goal is to find a topology that minimizes the cost such that all clients' traffic demands are met. An input instance is a weighted undirected demand graph $G = \langle N, E; d \rangle$, where each node $u \in N$ represents a client and weighted edges $(u, v) \in E$ correspond to traffic demands of a pair of clients. Demand $d(u, v)$ is always integral. Two clients can communicate only if both of them are installed on the same ring, which requires an expensive equipment called an add-drop multiplexer (ADM). A demand can be split into multiple rings. The input also specifies the maximum number of rings r, the maximum number of ADMs allowed on the same ring a, and the bandwidth capacity of each ring c. A solution of the SONET problem is an assignment of rings to nodes and of capacity to demands such that 1) all demands of each client pairs are satisfied; 2) the ring traffic does not exceed the bandwidth capacity; 3) at most r rings are used; 4) at most a ADMs on each ring; and 5) the total number of ADMs used is minimized.

The Basic CP Model. The core CP model [6,2] include three types of variables: Set variable X_i represents the set of nodes assigned to ring i, set variable Y_u represents the set of rings assigned to node u, and integer variable $Z_{i,e}$ represents the amount of bandwidth assigned to demand pair e on ring i. The model is

$$\texttt{minimize} \sum_{i \in R} |X_i| \ \texttt{s.t.}$$

$$|Y_u \cap Y_v| \geq 1 \quad \forall (u,v) \in E \tag{1}$$

$$Z_{i,(u,v)} > 0 \Rightarrow i \in (Y_u \cap Y_v) \quad \forall i \in R, (u,v) \in E \tag{2}$$

$$\sum_{i \in R} Z_{i,e} = d(e) \quad \forall e \in E \tag{3}$$

$$u \in X_i \Leftrightarrow i \in Y_u \quad \forall i \in R, u \in N \tag{4}$$

$$|X_i| \leq a \quad \forall i \in R \tag{5}$$

$$\sum_{e \in E} Z_{i,e} \leq c \quad \forall i \in R \tag{6}$$

$$X_i \preceq X_j \ \ \forall i,j \in R : i < j \tag{7}$$

(1) ensures nodes of every demand pair lie on at least one common ring. (2) ensures that there is a flow for a demand pair on a particular ring i only if both client are on that ring. (3) guarantees that every demand is satisfied. (4) channels between the first two types of variables. (5) makes sure that there are at most a ADMs on each ring. (6) makes sure that the total traffic flow on each ring does not exceed the bandwidth capacity. (7) is a symmetry-breaking constraint that removes symmetric solutions caused by interchangeability of rings.

Extended Model. Smith [2][Section 5] proposed a few implied constraints to detect infeasible assignments early in the search. For space reasons, we only show some of them which will be generalized by our redundant global constraints:

$$|X_i| \neq 1 \ \ \forall i \in R \tag{8}$$

$$|Y_u| \geq \lceil \frac{|\delta_u|}{a-1} \rceil \ \ \forall u \in N \tag{9}$$

$$Y_u = \{i\} \Rightarrow \delta_u \cup \{u\} \subseteq X_i \ \ \forall u \in N, i \in R \tag{10}$$

$$Y_u = \{i,j\} \Rightarrow \delta_u \cup \{u\} \subseteq X_i \cup X_j \ \ \forall u \in N, i,j \in R \tag{11}$$

In those constraints, δ_u denotes the neighbors of node u.

Our Extended Model. We propose two constraints to boost propagation:

$$\bigcup_{i \in \delta_u} Y_i \supseteq Y_u \ \ \forall u \in N \tag{12}$$

$$\bigcup_{i \in Y_u} X_i \supseteq \delta_u \ \ \forall u \in N \tag{13}$$

subsetOfUnion (12) generalizes (8), it forces a node not to lie on rings with no contribution. *subsetOfOpenUnion* (13) generalizes (9), (10), and (11) and ensures that the rings of a node accommodate all its neighbors.

3 The Set Domains

Our algorithms consider both the traditional subset-bound domain and subset-bound with cardinality domain.

Definition 1. *A subset-bound domain (sb-domain) $sb\langle R, P\rangle$ consists of a required set R and a possible set P, and represents the set of sets*

$$sb\langle R, P\rangle \equiv \{s \mid R \subseteq s \subseteq P\} \tag{14}$$

Definition 2. *A subset-bound + cardinality domain (sbc-domain) $sbc\langle R, P, \check{c}, \hat{c}\rangle$ consists of a required set R and a possible set P, a minimum and maximum cardinalities $\check{c}$ and $\hat{c}$, and represents the set of sets*

$$sbc\langle R, P, \check{c}, \hat{c}\rangle \equiv \{s \mid R \subseteq s \subseteq P \wedge \check{c} \leq |s| \leq \hat{c}\} \tag{15}$$

We now give the definition of bound consistency for these set domains.

Definition 3 (sbc-bound consistency). *A set constraint $\mathcal{C}(X_1, ..., X_m)$ (X_i are set variables using the sbc-domain) is said to be sbc-bound consistent if and only if $\forall 1 \leq i \leq m$,*

$$\exists x_1 \in d(X_1), ..., x_m \in d(X_m) \ s.t. \ C(x_1, ..., x_m) \tag{16}$$

$$\wedge \ R_{X_i} = \bigcap_{\forall 1 \leq j \leq m, x_j \in d(X_j): C(x_1, ..., x_m)} x_i \tag{17}$$

$$\wedge \ P_{X_i} = \bigcup_{\forall 1 \leq j \leq m, x_j \in d(X_j): C(x_1, ..., x_m)} x_i \tag{18}$$

$$\wedge \ \exists x_1 \in d(X_1), ..., x_m \in d(X_m) \ s.t. \ |x_i| = \check{c_{X_i}} \wedge C(x_1, ..., x_m) \tag{19}$$

$$\wedge \ \exists x_1 \in d(X_1), ..., x_m \in d(X_m) \ s.t. \ |x_i| = \hat{c_{X_i}} \wedge C(x_1, ..., x_m) \tag{20}$$

where $d(X_i) = sbc\langle R_{X_i}, P_{X_i}, \check{c_{X_i}}, \hat{c_{X_i}}\rangle$ denotes the domain of X_i.

The definition is similar for the subset-bound domain but it omits the cardinality rules. In the following, we use $bc_\theta\langle\mathcal{C}\rangle$ to denote a bound consistency propagator (or complete filtering algorithm) for constraint $\mathcal{C}$ on a θ-domain. We call *free elements* the elements in the possible set that are not in required set and *empty spots* the maximum number of free elements that the set can include.

Example 1. Consider domain $sbc\langle\{1, 2\}, \{1, .., 6\}, 3, 5\rangle$. $\{3, 4, 5, 6\}$ are free elements, and the domain has 3 empty spots since it can take at most 5 elements while 2 of them are already fixed by required set.

4 Non-empty Intersection Constraint

Reference [2] does not specify how the constraint propagator for the non-empty intersection constraint($|X \cap Y| \geq 1$) is implemented. This section presents a sound and complete propagator for the sbc-domain. First note that the sbc-domain gives stronger propagation than the sb-domain.

Theorem 1. *Enforcing bound consistency on the conjunction of constraints*

$$|X \cap Y| \geq 1 \wedge \check{c_X} \leq |X| \leq \hat{c_X} \wedge \check{c_Y} \leq |Y| \leq \hat{c_Y}$$

is strictly stronger for the sbc-domain than for the sb-domain.

Proof. Consider $X \in sb\langle\{1\},\{1..5\}\rangle$, $Y \in sb\langle\{6\},\{2,..,6\}\rangle$, $\check{c_X} = \hat{c_X} = \check{c_Y} = \hat{c_Y} = 2$. For the sbc-domain, after enforcing bound consistency on each constraint, $X \in sbc\langle\{1\},\{1,..,4\},2,2\rangle$ and $Y \in sbc\langle\{6\},\{3,..,6\},2,2\rangle$. X and Y can each take two elements, one of which is fixed, and elements 2 in X and 5 in Y are removed. All 3 constraints are bound-consistent for the sb-domain. □

Algorithm 1 presents the filtering algorithm for the sbc-domain which relies on insights from the length-lex domain [7] and the *atmost* algorithm studied in [8]. For simplicity, it assumes the cardinality of both input variables are bounded, but it can easily be generalized to unbounded case. It divides all elements in the universe into 9 different regions, according to how they belong in the domains. The algorithm mostly performs a case analysis of the number of empty spots in both domains. It essentially detects if the overlap region is too small (that contains only one element), in which case that element is inserted into the required set of both variables. On the other hand, if there are too few empty spots left and the variables have no fixed overlapping element, the variables cannot include elements not in the overlapping area.

Example 2. Let $X \in sbc\langle\{1\},\{1,2,3,4\},2,2\rangle$ and $Y \in sbc\langle\{3,5\},\{3,4,5,6\},3,3\rangle$. There is a solution since $P_XP_Y = \{4\}$ and $P_XR_Y = \{3\}$ are both non-empty (lines 6–9). The only empty spot of X has to be used to accommodate the common element since the required element $\{1\}$ is not in the common region. As a consequence, it must require either 3 or 4 and element 2 which is not in the common region can be removed (lines 11–13).

Example 3. Let $X = \{1,2\}$ and $Y \in sbc\langle\{3\},\{2,3,4\},2,2\rangle$. There is a solution since the overlapping is non-empty. Since there are only one choice in the common region, Y_{sbc} must take element 2 (lines 8–9).

Theorem 2. *Algorithm 1 is sound and complete, and takes $O(n)$ time.*

The correctness proof is based on a case analysis with four different cases and is omitted for space reasons.

5 All Non-empty Intersection Constraint

In SONET, a node u must share rings with all its neighbors. It naturally raises a question whether or not there exists a global constraint achieving more pruning. We define a new global constraint

$$allNonEmptyIntersect(X, \{Y_1,..,Y_n\}) \equiv (\forall 1 \leq i \leq n, |X \cap Y_i| \geq 1) \quad (21)$$

which allows us to rewrite (1) into

$$allNonEmptyIntersect(Y_u, \{Y_v | v \in \delta_u\}) \;\; \forall u \in N. \quad (22)$$

Algorithm 1. $bc_{sbc}\langle nonEmptyIntersection\rangle(X_{sbc} = sbc\langle R_X, P_X, c_X, c_X\rangle, Y_{sbc})$

```
Require: $X_{sbc}, Y_{sbc}$ are both bound consistent
 1: $P_XE_Y, E_XP_Y \leftarrow P_X \setminus (R_X \cup P_Y), P_Y \setminus (R_Y \cup P_X)$
 2: $P_XP_Y, R_XR_Y \leftarrow (P_X \cap P_Y) \setminus (R_X \cup R_Y), R_X \cap R_Y$
 3: $R_XP_Y, P_XR_Y \leftarrow R_X \cap (P_Y \setminus R_Y), (P_X \setminus R_X) \cap R_Y$
 4: if $|R_XR_Y| > 0$ then
 5:    return $true$
 6: if $|P_XP_Y| + |R_XP_Y| + |P_XR_Y| = 0$ then
 7:    return $\perp$
 8: else if $|P_XP_Y| + |R_XP_Y| + |P_XR_Y| = 1$ then
 9:    insert $e$ into $X_{sbc}$, $Y_{sbc}$ (where $\{e\} = P_X \cup P_Y$)
10: else
11:    $c'_X, c'_Y \leftarrow c_X - |R_X|, c_Y - |R_Y|$
12:    if $c'_X = 1 \wedge R_XP_Y = \emptyset$ then
13:       exclude $P_XE_Y$ from $X_{sbc}$
14:    if $c'_Y = 1 \wedge P_XR_Y = \emptyset$ then
15:       exclude $E_XP_Y$ from $Y_{sbc}$
16: return $true$
```

Theorem 3. *$bc_{sb}\langle allNonEmptyIntersect(X, \{Y_1, .., Y_n\})\rangle$ is decomposable.*

Proof. (sketch) From reference [5], $bc_{sb}(\forall i < j, |Y_i \cap Y_j| \geq 1)$ is decomposable. Our constraint is a special case of it which can be transformed to the general case by amending a dummy element to the possible set of each Y_i. □

Unfortunately, the result does not hold for the sbc-domain.

Theorem 4. *$bc_{sbc}\langle allNonEmptyIntersect(X, \{Y_1, .., Y_n\})\rangle$ is strictly stronger than enforcing BC on its decomposition (i.e. $\forall 1 \leq i \leq n$, $bc_{sbc}\langle |X \cap Y_i| \geq 1\rangle$).*

Proof. Consider $allNonEmptyIntersect(X, \{Y_1, Y_2, Y_3\})$. $X \in sbc\langle\emptyset, \{1..6\}, 2, 2\rangle$, $Y_1 \in sbc\langle\emptyset, \{1, 2\}, 1, 1\rangle$, $Y_2 \in sbc\langle\emptyset, \{3, 4\}, 1, 1\rangle$, and $Y_3 \in sbc\langle\emptyset, \{5, 6\}, 1, 1\rangle$. It is bound consistency on each constraint in the decomposition. However, there is no solution since X can only takes two elements and the possible sets of Y_1, Y_2 and Y_3 are disjoint. □

Theorem 5. *$bc_{sbc}\langle allNonEmptyIntersect(X, \{Y_1, .., Y_n\})\rangle$ is NP-hard.*

Proof. Reduction from 3-SAT. Instance: Set of n literals and m clauses over the literals such that each clause contains exactly 3 literals. Question: Is there a satisfying truth assignment for all clauses?

We construct a set-CSP with three types of variables. The first type corresponds to literals: for each literal, we construct a set variable X_i with domain $sbc\langle\emptyset, \{i, \neg i\}, 1, 1\rangle$, values in the possible set corresponds to true and false. The second type corresponds to clauses: for every clause j $(x_p \vee \neg x_q \vee x_r)$, we introduce one set variable Y_j with domain $sbc\langle\emptyset, \{p, -q, r\}, 1, 3\rangle$. The third type contains just one set variable Z correspond to the assignment, its domain is $sbc\langle\emptyset, \{1, -1, .., n, -n\}, n, n\rangle$. The constraint is in the form,

$$allNonEmptyIntersect(Z, \{X_1, .., X_n, Y_1, .., Y_m\})$$

Set variables X_i guarantees that Z is valid assignment (i.e., for every i, it can only pick either i or $-i$, but not both). Y_j and Z overlap if and only if at least one of the literals is satisfied. The constraint has a solution if and only if the 3-SAT instance is satisfiable. Therefore, enforcing bound consistency is NP-hard. □

6 Subset of Union

This section considers constraint (12) which is an instance of

$$subsetOfUnion(X, \{Y_1, .., Y_m\}) \equiv \bigcup_{1 \leq i \leq m} Y_i \supseteq X \tag{23}$$

Constraint (12) is justified by the following reasoning for a node u and a ring i it belongs to: If i is not used by any of u's neighbors, u does not need to use i. As a result, the rings of node u must be a subset of the rings of its neighbors. We first propose two simple inference rules to perform deductions on this constraint.

Rule 1 (SubsetOfUnion : Element Not in Union)

$$\frac{i \in P_X \wedge \forall 1 \leq j \leq m, i \notin P_{Y_j}}{subsetOfUnion(X, \{Y_1, .., Y_m\}) \longmapsto i \notin X \wedge subsetOfUnion(X, \{Y_1, .., Y_m\})}$$

Rule 2 (SubsetOfUnion : Element Must Be in Union)

$$\frac{i \in R_X \wedge i \in P_{Y_k} \wedge |\{i \in P_{Y_j} \mid 1 \leq j \leq m\}| = 1}{subsetOfUnion(X, \{Y_1, .., Y_m\}) \longmapsto i \in Y_k \wedge subsetOfUnion(X, \{Y_1, .., Y_m\})}$$

Two above rules are sufficient to enforce bound consistency on the sb-domain but not on the sbc-domain. It is an open issue to determine if bound consistency can be enforced in polynomial time on the sbc-domain.

Theorem 6. *$bc_{sb}\langle subsetOfUnion(X, \{Y_1, .., Y_m\})\rangle$ is equivalent to enforcing rule 1 and rule 2 until they reach the fix-point.*

Proof. Consider an element $e \in P_X$. It has a support or otherwise it would be removed by rule 1. It does not belong to all solutions since, given any feasible assignment to the constraint that contains e, removing e from X still leaves us with a feasible solution. Hence e does not belong to the required set. An element $e \in P_{Y_i}$ always has a support since adding e to any feasible assignment would not make it invalid. An element $e \in P_{Y_i}$ belongs to all solutions if it must be in the union and Y_i is the only variable that contains e (rule 2). □

Theorem 7. *$bc_{sbc}\langle subsetOfUnion(X, \{Y_1, .., Y_m\})\rangle$ is strictly stronger than enforcing rule 1 and rule 2 until they reach the fix-point.*

Proof. Consider the domains $X \in sbc\langle\emptyset, \{1, .., 6\}, 0, 2\rangle$, $Y_1 \in sbc\langle\emptyset, \{1, 2\}, 1, 1\rangle$, $Y_2 \in sbc\langle\emptyset, \{3, 4\}, 1, 1\rangle$ and $Y_3 \in sbc\langle\emptyset, \{1, .., 5\}, 2, 2\rangle$. Applying the domain reduction rules, the domain of X becomes $sbc\langle\emptyset, \{1, .., 5\}, 2, 2\rangle$. $5 \in P_{Y_3}$ has no solution since X has only two empty spots, one for $\{1, 2\}$ and the other for $\{3, 4\}$ as Y_1 and Y_2 are disjoint. The constraint is thus not bound consistent. □

7 Subset of Open Union

The SONET model contains a dual set of variables. Variable Y_u represents the set of rings node u lies on and ring variable X_i represents the set of nodes that ring i contains. Variable Y_u indirectly specifies the set of nodes that u can communicate with. Such set should be a superset of δ_u. We propose a global constraint that enforce this relation:

$$subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \equiv \bigcup_{i \in Y} X_i \supseteq s \tag{24}$$

which is used in constraint (13) of the model.

Example 4. Suppose node 1 has 5 neighbors (i.e., $\delta_1 = \{2, .., 6\}$), each pair has a demand of one unit. There are 2 rings, each ring can accommodate at most 2 ADMs. There is no solution since 2 rings can at most accommodate 4 neighbors. Using 5 *nonEmptyIntersection* constraints cannot detect such failure.

subsetOfOpenUnion is sometimes called an open constraint[9], since the scope of the constraint is defined by Y. Complete filtering is polynomial for the sb-domain but intractable for the sbc-domain.

Rule 3 (SubsetOfOpenUnion : Failure)

$$\frac{\bigcup_{i \in P_Y} P_{X_i} \not\supseteq s}{subsetOfOpenUnion(s, X, \{Y_1, .., Y_m\}) \longmapsto \bot}$$

Rule 4 (SubsetOfOpenUnion: Required Elements)

$$\frac{i \in P_Y \wedge e \in P_{X_i} \wedge e \in s \wedge |\{e \in P_{X_j} \mid j \in P_Y\}| = 1}{\begin{array}{l} subsetOfOpenUnion(s, X, \{Y_1, .., Y_m\}) \\ \longmapsto i \in Y \wedge e \in X_i \wedge subsetOfOpenUnion(s, X, \{Y_1, .., Y_m\}) \end{array}}$$

Theorem 8. *$bc_{sb}\langle subsetOfOpenUnion(s, Y, \{X_1, .., X_m\})\rangle$ is equivalent to enforcing rule 3 and rule 4 until they reach a fix-point.*

Proof. There is no feasible assignment if the union of all possible X_i is not a superset of s (rule 3). Suppose there is a feasible solution. Consider an element $e \in P_Y$ or $e \in P_{X_i}$: It must have a support since any feasible assignment would remain feasible after adding e to it. An element $e \in P_{X_i}$ which is also in s belongs to all solutions if it belongs to exactly one variable X_i. In such case, we include e in X_i and i in Y since X_i must be in the scope (rule 4). □

Theorem 9. *$bc_{sbc}\langle subsetOfOpenUnion(s, Y, \{X_1, .., X_m\})\rangle$ is NP-hard.*

Proof. Reduction from Dominating Set. The problem of dominating set is defined as follows. Input instance: A graph $G = \langle V, E\rangle$ and an integer $k \leq |V|$. Question: Does there exist a subset V' of V such that $|V'| \leq k$ and every node in $V \setminus V'$ is a neighbor of some nodes in V'?

Given an instance with a graph G and a constant k, we construct an instance of CSP that $s = V$, $Y \in \langle \emptyset, V, 0, k \rangle$ and, for every $i \in V$, $X_i = \delta_i^G \cup \{i\}$ (where δ_i^G denotes the neighborhood of node i in graph G). Intuitively, Y corresponds to a dominating set with size at most k, X_i is a vertex that can "dominate" at most all elements in its domain (which is also the neighbors in the originally graph). The constraint is consistent if and only if there exists a dominating set of size not more than k.

$\Rightarrow$ Given a dominating set V' in the original graph G, the constraint is consistent since we can construct a solution by setting $Y = V'$, every element in Y actually corresponds to a node in the dominating set. Since every node in $V \setminus V'$ is the neighbor or at least on node in V', every element in δ_u also belongs to the domain of some X_i $(i \in Y)$.

$\Leftarrow$ Given a consistent assignment of Y and X_i for all $i \in Y$, all elements in δ_u are covered by some X_i and hence Y is the dominating set. □

Since the constraint is intractable, we present a number of inference rules particularly useful in practice. The first inference rule reasons about the cardinality of Y. The union of X_i must be a superset of s. Since Y determines the number of X_i in the union, we can get an upper bound on the union cardinality by reasoning on the maximal cardinalities of the X_i. If the upper bound is less than $|s|$, there is no solution. Otherwise, we obtain a lower bound of cardinality of Y.

Example 5. Suppose $X_1 = X_2 = X_3 \in sbc\langle \emptyset, \{1, .., 8\}, 0, 3\rangle$, $Y \in sbc\langle \emptyset, \{1, 2, 3\}, 2, 3\rangle$ and $s = \{1, .., 8\}$. Each of X_i has 3 empty spots. We need at least $\lceil 8/3 \rceil = 3$ X_i to accommodate every element in s. It implies $|Y| > 2$.

Rule 5 (SubsetOfOpenUnion : Lower Bound of $|Y|$)

$$\frac{\max_{t \in d(Y):|t|=\check{c_Y}} \sum_{i \in t} (\hat{c_{X_i}} - |R_{X_i} \setminus s|) < |s|}{\begin{array}{l} subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \\ \longmapsto |Y| > \check{c_Y} \wedge subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \end{array}}$$

Theorem 10. *Rule 5 is sound.*

Proof. Any feasible assignment to the constraint satisfies $\bigcup_{i \in y}(x_i \cap s) \supseteq s$. Consider the set $x_i \cap s$. x_i is in $d(X_i) = sbc\langle R_{X_i}, P_{X_i}, \check{c_{X_i}}, \hat{c_{X_i}}\rangle$. We divide it into two parts: First, the elements in $R_{X_i} \cap s$ are fixed. Second, x_i can choose $\hat{c_{X_i}} - |R_{X_i}|$ elements freely from the set $P_{X_i} \setminus R_{X_i}$. The cardinality of the set $x_i \cap s$ is the sum of two parts and can be bounded from above

$$(\hat{c_{X_i}} - |R_{X_i}|) + |R_{X_i} \cap s| = \hat{c_{X_i}} - |R_{X_i} \setminus s| \geq |x_i \cap s|$$

Therefore we obtain the following inequality,

$$\sum_{i \in y} (\hat{c_{X_i}} - |R_{X_i} \setminus s|) \geq \sum_{i \in y} |x_i \cap s| \geq |\bigcup_{i \in y} (x_i \cap s)| \geq |s| \tag{25}$$

Cardinalities of y that do not meet this condition belong to no solution. □

A similar reasoning on the cardinalities of Y can remove elements of Y that corresponds to small X_i.

Example 6. Suppose $X_1 = X_2 \in sbc\langle\emptyset, \{1, .., 6\}, 0, 3\rangle$, $X_3 \in sbc\langle\emptyset, \{1, .., 6\}, 0, 2\rangle$, $Y \in sbc\langle\emptyset, \{1, 2, 3\}, 2, 2\rangle$ and $s = \{1, .., 6\}$. We need to choose two sets among X_1, X_2 and X_3. If X_3 is chosen, it provides 2 empty spots and we need 4 more spots. However, neither X_1 nor X_2 is big enough to provide 4 empty spots. It implies that Y cannot take X_3.

Rule 6 (SubsetOfOpenUnion : Pruning Elements of Y)

$$\frac{\max_{t \in d(Y): i \in t} \sum_{j \in t} (\hat{c_{X_j}} - |R_{X_j} \setminus s|) < |s| \wedge i \in P_Y}{\begin{array}{l} subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \\ \longmapsto i \notin Y \wedge subsetOfOpenUnion(s, Y, \{X_1, .., X_m\}) \end{array}}$$

Theorem 11. *Rule 6 is sound.*

Proof. Expression (25) gives a upper bound of empty spots that X_i can provide. If all possible values of Y containing element i do not provide enough empty spots to accommodate all elements in s, X_i is too small and $i \notin Y$. $\square$

8 Combination of *subsetOfOpenUnion* and *Channeling*

This section explores the combination of the *subsetOfOpenUnion* and channeling constraints. Indeed, in the SONET model, the X_i and Y_u are primal and dual variables channeled using the constraint: $i \in Y_u \Leftrightarrow u \in X_i$. In other words, when Y_u takes element i, one spot in X_i is used to accommodate u. Exploiting this information enables us to derive stronger inference rules.

The first inference rule assumes that Y is bound and reduces the open constraints to a global cardinality constraint. It generalizes the last two constraints (10) and (11) in Smith's extended model which apply when $1 \leq |Y_u| \leq 2$.

Definition 4 (Global lower-bounded cardinality constraint). *We define a specialized global cardinality constraint, where only the lower bound is specified.* $GCC_{lb}(\{X_1, .., X_m\}, [l_1, .., l_n]) \equiv \forall 1 \leq j \leq n, |\{X_i \ni j | 1 \leq i \leq m\}| \geq l_j$

Example 7. Suppose node 1 has 3 neighbors, $Y_1 = \{1, 2\}$. X_1 and X_2 must contain $\{1\}$ and each element in {2,3,4} has to be taken at least once. It is equivalent to $GCC_{lb}(\{X_1, X_2\}, [2, 1, 1, 1])$. By a simple counting argument, there is no solution.

Rule 7 (SubsetOfOpenUnion and Channeling : Global Cardinality)

$$\frac{Y_u = y \ (Y_u \ \textit{is bounded})}{\begin{array}{l} subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \\ \longmapsto GCC_{lb}(\{X_i | i \in Y_u\}, [l_1, .., l_n]) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \end{array}}$$
$$\textit{where } l_u = |Y_u|, \ l_i = 1 \ \textit{if } i \in s \textit{ and otherwise } l_i = 0$$

Theorem 12. *Rule 7 is sound.*

Proof. When Y_u is bounded, the scope for the union is fixed. The union constraint requires that the union of set has to be a superset of s and hence each element of s has to be taken at least once. The channeling constraint requires each variable X_i contains element u and, as Y_u defines the scope, element u has to be taken exactly $|Y_u|$ times. It reduces to a GCC_{lb}. □

Moreover, it is possible to strengthen the earlier cardinality-based inference rules to include the channeling information. We omit the proofs which are essentially similar to the earlier ones.

Rule 8 (SubsetOfOpenUnion and Channeling : Lower Bound of $|Y|$)

$$\frac{u \notin s \wedge \max_{t \in d(Y):|t|=\check{c_Y}} \sum_{i \in t}(\hat{c_{X_i}} - |R_{X_i} \setminus s| - (R_{X_i} \not\ni u)) < |s|}{\begin{array}{l} subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \\ \longmapsto |Y| > \check{c_Y} \wedge subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \end{array}}$$

Rule 9 (SubsetOfOpenUnion and Channeling : Pruning Y)

$$\frac{\max_{t \in d(Y_u):i \in t} \sum_{j \in t}(\hat{c_{X_j}} - |R_{X_j} \setminus s| - (R_{X_j} \not\ni u) < |s| \wedge i \in P_{Y_u}}{\begin{array}{l} subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \\ \longmapsto i \notin Y_u \wedge subsetOfOpenUnion(s, Y_u, \{X_1, .., X_m\}) \wedge \bigwedge_i u \in X_i \Leftrightarrow i \in Y_u \end{array}}$$

9 Experimental Evaluation

We now describe the experimental evaluation of our approach. We start by describing earlier results on MIP and CP models. We then present our search procedure and presents the computational results. We then describe the impact of various factors, including the branching heuristics, symmetry-breaking constraints, and the proposed global constraints.

The MIP Formulation. The problem was first solved using an MIP formulation[1]. The input was preprocessed before the search and some variables were preassigned. Valid inequalities were added during the search in order to tighten the model representation. Several variable-ordering heuristics, mainly based on the neighborhood and demand of nodes, were devised and tested. Several symmetry-breaking constraints were evaluated too, Table 1 in [10] indicates minuscule differences in performance among different symmetry-breaking constraints.

CP Formulations. Smith [2] introduced a four-stage search procedure in her CP program: First decide the objective value, then decide how many rings each node lies on (label the cardinality of Y_u), then decide which rings each node lies on (label the element of Y_u), and finally decide how much bandwidth assigned to demand pairs on each ring. A few variable-branching heuristics were examined with a dynamic ordering giving the best results. Symmetry-breaking techniques were also investigated. To avoid clashing with variable ordering, SBDS (symmetry breaking during search) was used. SBDS was very effective on the SONET problems, although it generated a huge number of no-good constraints, inducing a significant overhead to the system. Recall also that Smith's model included a few simple redundant constraints reasoning on the cardinality of node variables (Y_u). Please refer to Section 5 in [2] for a detailed discussion.

Another CP model was proposed in [3] and it broke symmetries by adding a lexicographic bound to set variable domain. With the additional lexicographic component, the solver obtained a tighter approximation of the set-variable domain. The lexicographical information was used not only for breaking symmetries, but also for cardinality reasoning. This method provided a much simpler mechanism to remove symmetries. However, as mentioned by the authors, different components of the set domain (the membership component, the cardinality restriction, and the lexicographical bound) did not interact effectively.

Our Search Procedure. Our CP algorithm *Boosting* implements all the constraints presented in this paper and uses a static four-stage search inspired by Smith's heuristics [2]. The algorithm first branches on the objective value, starting from the minimum value and increasing the value by one at a time from the infeasible region. The first feasible solution is thus optimal. Then it decides the cardinality of Y_u. Third, it decides the value of Y_u. Last, the algorithm decides the flow assigned to each pair of nodes on a ring. Proposition 2 in [1] shows that there is an integral solution as long as all the demands are integral and the algorithm only needs to branch on integers. In each stage, variables are labeled in the order given by the instance.

Benchmarks and Implementations. The benchmarks include all the large capacitated instances from [1]. Small and medium instances take negligible time and are omitted. Our algorithm was evaluated on an Intel Core 2 Duo 2.4GHz laptop with 4Gb of memory. The MIP model [1] used CPLEX on a Sun Ultra 10 Workstation. Smith's algorithm [2] used ILOG Solver on one 1.7GHz processor. Hybrid[3] was run using the Eclipse constraint solver on a Pentium 4 2GHz processor, with a timeout of 3000 seconds.

Comparison of the Approaches. Table 2 reports the CPU time and number of backtracks (bt) required for each approach to prove the optimality of each instance. Our *Boosting* algorithm is, on average, more than 3700 times faster than the MIP and Hybrid approaches and visits several orders on magnitude less nodes than them. *Boosting* is more than 16 times faster than the SBDS approach when the machines are scaled and produces significantly higher speedups on the most difficult instances (e.g., instance 9). The SBDS method performs fewer backtracks in 6 out of 15 instances, because it eliminates symmetric subtrees earlier than our static symmetry-breaking constraint. However, even when the CPU speed is scaled, none of 15 instances are solved by SBDS faster than *Boosting*. This is explained by the huge number of symmetry-breaking constraints added during search. The empirical results confirm the strength of the light-weight

Table 2. Experimental Results on Large Capacitated Instances

		MIP		Hybrid		SBDS		**Boosting**	
		Sun Ultra 10		P4, 2GHz		P(M), 1.7GHz		C2D 2.4GHz	
#	opt	nodes	time	bt	time	bt	time	bt	time
1	22	5844	209.54	532065	2248.68	990	0.95	**444**	**0.09**
2	20	1654	89.23	-	-	451	0.65	**41**	**0.01**
3	22	4696	151.54	65039	227.71	**417**	0.62	573	**0.12**
4	23	50167	1814	476205	1767.82	**1419**	1.52	1727	**0.32**
5	22	36487	1358.83	-	-	**922**	0.7	982	**0.18**
6	22	9001	343.54	-	-	306	0.29	**291**	**0.06**
7	22	13966	568.96	270310	1163.94	**982**	1.15	2504	**0.53**
8	20	441	23.38	11688	54.73	34	0.09	**25**	**0.01**
9	23	25504	701.71	-	-	35359	45.13	**2769**	**0.56**
10	24	8501	375.48	-	-	4620	6.75	**4066**	**0.83**
11	22	5015	316.77	-	-	352	0.54	**263**	**0.06**
12	22	6025	213.4	-	-	**1038**	1.09	1572	**0.31**
13	21	2052	65.06	255590	1300.91	**105**	0.14	397	**0.08**
14	23	61115	2337.29	-	-	1487	1.66	**613**	**0.11**
15	23	100629	4324.19	-	-	13662	19.59	**1240**	**0.25**
	avg	22073.13	859.53	268482.83	1127.30	4142.93	5.39	**1167.13**	**0.23**

and effective propagation algorithms proposed in this paper. While earlier attempts focused on branching heuristics and sophisticated symmetry-breaking techniques, the results demonstrate that effective filtering algorithms are key to obtaining strong performance on this problem. The remaining experimental results give empirical evidence justifying this observation.

The Impact of Branching Heuristics. We now study the impact of the branching heuristics and evaluate various variable orderings for the static labeling procedure of *Boosting*. Various variable orderings were studied in [1,2]. Most of them are based on the node demands and degrees. Our experiments considered four different heuristics: minimum-degree-first, maximum-degree-first, minimum-demand-first, and maximum-demand-first. To avoid a clash between the variable heuristics and the symmetry-breaking constraint, the lexicographic constraint uses the same static order as the branching heuristic. Table 3 (Left) reports the average number of backtracks and time to solve all 15 instances, where row *Given* is the node ordering from the instance data. The results show that, with the exception of the *max-demand* heuristic, all variable orderings produce very similar number of backtracks and runtime performance. Moreover, the *max-demand* heuristic is still orders of magnitude faster than earlier attempts. This indicates that the variable ordering is not particularly significant when stronger filtering algorithms are available.

The Impact of Symmetry-Breaking Constraints. We compare different symmetry-breaking constraints. In particular, we compare breaking symmetries using the 01-lex and the length-lex ordering[11]. The length-lex ordering first ranks set by

Table 3. The Impact of Branching Heuristics (Left), Symmetry Breaking (Right)

	avg bt	avg time
Given	1167.13	0.23
Min-Degree	1158.40	0.23
Max-Degree	1172.93	0.24
Min-Demand	1144.60	0.23
Max-Demand	1683.53	0.36

	avg bt	avg time
01-Lex	1167.13	0.23
Length-Lex	1411.53	0.38

Table 4. The Impact of Redundant Constraints

NonEmptyIntersection $\lvert X \cap Y \rvert \geq 1$	SubsetOfUnion $\bigcup_i Y_i \supseteq X$	SubsetOfOpenUnion $\bigcup_{i \in Y} X_i \supseteq s$	avg bt	avg time
$\star$	$\star$	$\star$	1167.13	0.23
$\star$	$\star$		2018.67	0.33
$\star$		$\star$	1190.13	0.23
	$\star$	$\star$	1556.67	0.36
$\star$			2177.93	0.33
	$\star$		13073.73	2.19
		$\star$	1670.47	0.37
			17770.93	2.82

cardinalities, while the 01-lex orders sets based on their characteristic vectors. Table 3 (Right) reports the results of the *Boosting* algorithm with both types of symmetry-breaking constraints. The difference is still negligible when compared with the benefits of global constraints, although the 01-lexicographic order seems more effective on these benchmarks in average.

The Impact of Redundant Constraints. We conclude the experimental section by analyzing the impact of each redundant constraint. Our study simply enumerated and evaluated all combinations. The results are presented in Table 4, where $\star$ indicates that the corresponding constraint was used in the model. For cases where the sbc-domain complete propagator $nonEmptyIntersection$ is absent, a sb-domain implementation is used instead. The table reports the average number of backtracks and the CPU time. Using all three redundant constraints (first row) gives the best results both in the number of backtracks and in CPU time. The model in which $subsetOfUnion$ constraint is absent (third row) achieves the same solving time as the complete model, with some more backtrackings. It suggests that constraint $subsetOfUnion$ brings the least contribution to the efficiency. Removing $subsetOfOpenUnion$ dampens the search the most, doubling the number of backtracks. Thrashing is caused when both binary intersection constraints and $subsetOfOpenUnion$ are removed (sixth row), the resulting algorithm being almost 10 times slower and visiting 11 times more nodes than the complete model. The worst performance is the last row, which essentially corresponds to Smith's model with a static symmetry-breaking constraint and a static labeling heuristic. Overall, these results suggest that, on the SONET application, the performance of the algorithm is strongly correlated to the strength of constraint propagation. The variable heuristics and the symmetry-breaking technique have marginal impact on the performance.

10 Conclusion

This paper reconsiders the SONET problem. While earlier attempts focused on symmetry breaking and the design of effective search strategies, this paper took an orthogonal view and aimed at boosting constraint propagation by studying a variety of global constraints arising in the SONET application. From a modeling standpoint, the main contribution was to isolate two classes of redundant constraints that provide a global view to the solver. From a technical standpoint, the scientific contributions included novel hardness proofs, propagation algorithms, and filtering rules. The technical contributions were also evaluated on a simple and static model that performs a few orders of magnitude faster than earlier attempts. Experimental results also demonstrated the minor impact of variable orderings and symmetry-breaking techniques, once advanced constraint propagation is used. More generally, these results indicate the significant benefits of constraint programming for this application and the value of developing effective constraint propagation over sets.

References

1. Sherali, H.D., Smith, J.C., Lee, Y.: Enhanced model representations for an intraring synchronous optical network design problem allowing demand splitting. INFORMS Journal on Computing 12(4), 284–298 (2000)
2. Smith, B.M.: Symmetry and search in a network design problem. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 336–350. Springer, Heidelberg (2005)
3. Sadler, A., Gervet, C.: Enhancing set constraint solvers with lexicographic bounds. J. Heuristics 14(1), 23–67 (2008)
4. Sadler, A., Gervet, C.: Global reasoning on sets. In: FORMUL, CP 2001 (2001)
5. Bessière, C., Hebrard, E., Hnich, B., Walsh, T.: Disjoint, partition and intersection constraints for set and multiset variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 138–152. Springer, Heidelberg (2004)
6. Sadler, A., Gervet, C.: Hybrid set domains to strengthen constraint propagation and reduce symmetries. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 604–618. Springer, Heidelberg (2004)
7. Hentenryck, P.V., Yip, J., Gervet, C., Dooms, G.: Bound consistency for binary length-lex set constraints. In: AAAI 2008, pp. 375–380 (2008)
8. van Hoeve, W.J., Sabharwal, A.: Filtering atmost1 on pairs of set variables. In: CPAIOR 2005, pp. 382–386 (2005)
9. van Hoeve, W.J., Régin, J.C.: Open constraints in a closed world. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, pp. 244–257. Springer, Heidelberg (2006)
10. Sherali, H.D., Smith, J.C.: Improving discrete model representations via symmetry considerations. Manage. Sci. 47(10), 1396–1407 (2001)
11. Gervet, C., Hentenryck, P.V.: Length-lex ordering for set csps. In: AAAI 2006 (2006)

More Robust Counting-Based Search Heuristics with Alldifferent Constraints

Alessandro Zanarini[1,2,3] and Gilles Pesant[1,2]

[1] École Polytechnique de Montréal, Montreal, Canada
[2] CIRRELT, Université de Montréal, Montreal, Canada
[3] Dynadec Europe, Belgium
{Alessandro.Zanarini,Gilles.Pesant}@cirrelt.ca

Abstract. Exploiting solution counting information from individual constraints has led to some of the most efficient search heuristics in constraint programming. However, evaluating the number of solutions for the `alldifferent` constraint still presents a challenge: even though previous approaches based on sampling were extremely effective on hard instances, they are not competitive on easy to medium difficulty instances due to their significant computational overhead. In this paper we explore a new approach based on upper bounds, trading counting accuracy for a significant speedup of the procedure. Experimental results show a marked improvement on easy instances and even some improvement on hard instances. We believe that the proposed method is a crucial step to broaden the applicability of solution counting-based search heuristics.

1 Introduction

AI problem solving relies on effective inference and search. This is true in particular for Constraint Programming where, after many years of advances on inference, there has been a more recent focus on search heuristics. The kind of search heuristics considered in this paper rely on counting the solutions to individual substructures of the problem [13]. Given a constraint γ defined on the set of variables $\{x_1, \ldots, x_k\}$ and respective finite domains D_i $_{1 \leq i \leq k}$, let $\#\gamma(x_1, \ldots, x_k)$ denote the number of solutions of constraint γ. Given a variable x_i in the scope of γ, and a value $d \in D_i$, we call

$$\sigma(x_i, d, \gamma) = \frac{\#\gamma(x_1, \ldots, x_{i-1}, d, x_{i+1}, \ldots, x_k)}{\#\gamma(x_1, \ldots, x_k)}$$

the *solution density*[1] of pair (x_i, d) in γ. It measures how often a certain assignment is part of a solution of the constraint γ. One simple — yet very effective — solution counting-based heuristic is *maxSD* which, after collecting the solution densities from the problem constraints, branches on the variable-value pair with the highest solution density [13].

[1] Also referred to as *marginal* in some of the literature.

For some constraints, computing solution densities can be done efficiently and even, in some cases, at (asymptotically) no extra cost given the filtering algorithm already implemented in the constraint. For the `alldifferent` constraint, computing the number of solutions is equivalent to the problem of computing the permanent of the related (0-1) adjacency matrix A that is built such that a_{ij} is equal to 1 iff $j \in D_i$. The permanent of a $n \times n$ matrix A is formally defined as[2]

$$per(A) = \sum_{\sigma \in S_n} \prod_i a_{i,\sigma(i)} \tag{1}$$

where S_n denotes the symmetric group, i.e. the set of $n!$ permutations of $[n]$. Given a specific permutation, the product is equal to 1 if and only if all the elements are equal to 1 i.e. the permutation is a valid assignment for the `alldifferent` constraint. Hence, the sum over all the permutations gives us the total number of `alldifferent` solutions.

The problem of computing the permanent has been studied for the last two centuries and it is still a challenging problem to address. Even though the analytic formulation of the permanent resembles that of the determinant, there have been few advances on its exact computation. In 1979, Valiant [12] proved that the problem is $\#P$-complete, even for 0-1 matrices, that is, under reasonable assumptions, it cannot be computed in polynomial time in the general case. The focus then moved to approximating the permanent. A sampling approach proposed by Rasmussen was improved in [13] by adding propagation. Although providing a very good approximation, it is time consuming and suitable mainly for hard instances where the accuracy of the heuristic can balance the time spent in computing the solution densities.

In this paper we explore a different approach, trading some of the accuracy for a significant speedup in the counting procedure, in order to provide an algorithm that performs well on easy instances while keeping the lead in solving hard ones. A portfolio of heuristics could have been an alternative, first trying a computationally cheap heuristic to take care of easy instances and switching to our counting-based heuristic after a certain time limit. But as we shall see, our proposal not only improves the performance on easy instances but also on hard ones.

In the rest of this paper, Section 2 presents some known upper bounds for the permanent and their integration in solution counting-based heuristics for the `alldifferent` constraint. Section 3 evaluates our proposal on benchmark problems. Final comments are given in Section 4.

2 Bounds for `alldifferent` Solution Counting

2.1 Upper Bounds for the Permanent

In the following, we denote by A the $n \times n$ (0-1) adjacency matrix as defined in the previous section, with r_i the sum of the elements in the i_{th} row (i.e.

[2] We address the fact that the adjacency matrix may not be square in Section 2.1.

$r_i = \sum_{j=1}^{n} a_{ij}$). Note that the permanent is defined on square matrices i.e. the related bipartite graph needs to have $|V_1| = |V_2|$. In order to overcome this limitation, we can augment the graph by adding $|V_2| - |V_1|$ fake vertices to V_1 (without loss of generality $|V_1| \leq |V_2|$) each one connected to all vertices in V_2. The effect on the number of maximum matchings is stated in the following theorem.

Theorem 1. *Let $G(V_1 \cup V_2, E)$ be a bipartite graph with $|V_1| \leq |V_2|$ and the related augmented graph $G'(V_1' \cup V_2, E')$ a graph such that $V_1' = V_1 \cup V_{fake}$ with $|V_{fake}| = |V_2| - |V_1|$ and the edge set $E' = E \cup E_{fake}$ with $E_{fake} = \{(v_i, v_j) \mid v_i \in V_{fake},\quad v_j \in V_2\}$. Let $|M_G|$ and $|M_{G'}|$ be the number of maximum matchings respectively in G and G'. Then $|M_G| = |M_{G'}|/|V_{fake}|!$.*

Proof. Given a maximum matching $m \in M_G$ of size $|V_1|$, since m covers all the vertices in V_1 then there exists exactly $|V_2| - |V_1|$ vertices in V_2 not matched. In the corresponding matching (possibly not maximum) $m' = m$ in G', the vertices in V_2 that are not matched can be matched with any of the vertices in V_{fake}. Since each vertex in V_{fake} is connected to any vertex in V_2 then there exists exactly $|V_{fake}|!$ permutations to obtain a perfect matching in G' starting from a maximum matching m in G. If there is no maximum matching of size $|V_1|$ for G then clearly there isn't any of size $|V_2|$ for G' either.

For simplicity in the rest of the paper we assume $|X| = |D_X|$.

In 1963, Minc [6] conjectured that the permanent can be bounded from above by the following formula:

$$perm(A) \leq \prod_{i=1}^{n} (r_i!)^{1/r_i}. \tag{2}$$

Proved only in 1973 by Brégman [1], it was considered for decades the best upper bound for the permanent. Recently, Liang and Bai [4], inspired by Rasmussen's work, proposed a new upper bound (with $q_i = min\{\lceil \frac{r_i+1}{2} \rceil, \lceil \frac{i}{2} \rceil\}$):

$$perm(A)^2 \leq \prod_{i=1}^{n} q_i(r_i - q_i + 1). \tag{3}$$

None of the two upper bounds strictly dominates the other. In the following we denote by $UB^{BM}(A)$ the Brégman-Minc upper bound and by $UB^{LB}(A)$ the Liang-Bai upper bound. Jurkat and Ryser proposed in [3] another bound:

$$perm(A) \leq \prod_{i=1}^{n} min(r_i, i).$$

However it is considered generally weaker than $UB^{BM}(A)$ (see [11] for a comprehensive literature review). Soules proposed in [10] some general sharpening techniques that can be employed on any existent permanent upper bound in order to improve them. The basic idea is to apply an appropriate combination of functions (such as row or column permutation, matrix transposition, row or column scaling) and to recompute the upper bound on the modified matrix.

2.2 Solution Counting Algorithm for `alldifferent`

Aiming for very fast computations, we opted for a direct exploitation of UB^{BM} and UB^{LB} in order to compute an approximation of solution densities for the `alldifferent` constraint. An initial upper bound on the number of solutions of the $\texttt{alldifferent}(x_1, \ldots, x_n)$ constraint with related adjacency matrix A is simply

$$\#\texttt{alldifferent}(x_1, \ldots, x_n) \leq min\{UB^{BM}(A), UB^{LB}(A)\}$$

Note that in Formula 2 and 3, the r_i are equal to $|D_i|$; since the $|D_i|$ range from 0 to n, the factors can be precomputed and stored: in a vector $BMfactors[r] = (r!)^{1/r}, r = 0, \ldots, n$ for the first bound and similarly for the second one (with factors depending on both $|D_i|$ and i). Assuming that $|D_i|$ is returned in $O(1)$, computing the formulas takes $O(n)$ time.

Recall that matrix element $a_{ij} = 1$ iff $j \in D_i$. Assigning j to variable x_i translates to replacing the i_{th} row by the unit vector $e(j)$ (i.e. setting the i_{th} row of the matrix to 0 except for the element in column j). We write $A_{x_i=j}$ to denote matrix A except that x_i is fixed to j. We call *local probe* the assignment $x_i = j$ performed to compute $A_{x_i=j}$ i.e. a temporary assignment that does not propagate to any other constraint except the one being processed. Solution densities are then approximated as

$$\sigma(x_i, j, \texttt{alldifferent}) \approx \frac{min\{UB^{BM}(A_{x_i=j}), UB^{LB}(A_{x_i=j})\}}{\eta}$$

where η is a normalizing constant.

The local probe $x_i = j$ may trigger some local propagation according to the level of consistency we want to achieve; therefore $A_{x_i=j}$ is subject to the filtering performed on the constraint being processed. Since the two bounds in Formula 2 and 3 depend on $|D_i|$, a stronger form of consistency would likely lead to more changes in the domains and on the bounds, and presumably to more accurate solution densities. We come back to this in Section 2.3.

Once the upper bounds for all variable-value pairs have been computed, it is possible to further refine the solution count as follows:

$$\#\texttt{alldifferent}(x_1, \ldots, x_n) \leq \min_{x_i \in X} \sum_{j \in D_i} min\{UB^{BM}(A_{x_i=j}), UB^{LB}(A_{x_i=j})\}$$

This bound on the solution count depends on the consistency level enforced in the `alldifferent` constraint during the local probes. It is the one we will evaluate in Section 2.3.

If we want to compute $\sigma(x_i, j, \texttt{alldifferent})$ for all $i = 1, \ldots, n$ and for all $j \in D_i$ then a trivial implementation would compute $A_{x_i=j}$ for each variable-value pair; the total time complexity would be $O(mP + mn)$ (where m is the sum of the cardinalities of the variable domains and P the time complexity of the filtering).

Although unable to improve over the worst case complexity, in the following we propose an algorithm that performs definitely better in practice. We first introduce some additional notation: we write as D'_i the variable domains after enforcing θ-consistency[3] on that constraint alone and as $\tilde{I}$ the set of indices of the variables that were subject to a domain change due to a local probe and the ensuing filtering, that is, $i \in \tilde{I}$ iff $|D'_i| \neq |D_i|$. We describe the algorithm for the Brégman-Minc bound — it can be easily adapted for the Liang-Bai bound.

The basic idea is to compute the bound for the matrix A and reuse it to speed up the computation of the bounds for $A_{x_i=j}$ for all $i = 1, \dots, n$ and $j \in D_i$. Let

$$\gamma_k = \begin{cases} \frac{BMfactors[1]}{BMfactors[|D_k|]} & \text{if } k = i \\ \frac{BMfactors[|D'_k|]}{BMfactors[|D_k|]} & \text{if } k \in \tilde{I} \setminus \{i\} \\ 1 & \text{otherwise} \end{cases}$$

$$\begin{aligned} UB^{BM}(A_{x_i=j}) &= \prod_{k=1}^{n} BMfactors[|D'_k|] = \prod_{k=1}^{n} \gamma_k \; BMfactors[|D_k|] \\ &= UB^{BM}(A) \prod_{k=1}^{n} \gamma_k \end{aligned}$$

Note that γ_k with $k = i$ (i.e. we are computing $UB^{BM}(A_{x_i=j})$) does not depend on j; however $\tilde{I}$ does depend on j because of the domain filtering.

Algorithm 1. Solution Densities

```
1  UB = BMbound(A) ;
2  for $i = 1, \dots, n$ do
3      varUB = UB * BMfactors[1] / BMfactors[$|D_i|$] ;
4      total = 0;
5      forall $j \in D_i$ do
6          set $x_i = j$;
7          enforce $\theta$-consistency;
8          VarValUB[i][j] = varUB;
9          forall $k \in \tilde{I} \setminus \{i\}$ do
10             VarValUB[i][j] = VarValUB[i][j] * BMfactors[$|D'_k|$] / BMfactors[$|D_k|$];
11         total = total + VarValUB[i][j];
12         rollback $x_i = j$;
13     forall $j \in D_i$ do
14         SD[i][j] = VarValUB[i][j]/total;
15 return SD;
```

[3] Any form of consistency.

Algorithm 1 shows the pseudo code for computing $UB^{BM}(A_{x_i=j})$ for all $i = 1, \dots, n$ and $j \in D_i$. Initially, it computes the bound for matrix A (line 1); then, for a given i, it computes γ_i and the upper bound is modified accordingly (line 3). Afterwards, for each $j \in D_i$, θ-consistency is enforced (line 7) and it iterates over the set of modified variables (line 9-10) to compute all the γ_k that are different from 1. We store the upper bound for variable i and value j in the structure $VarValUB[i][j]$. Before computing the bound for the other variables-values the assignment $x_i = j$ needs to be undone (line 12). Finally, we normalize the upper bounds in order to correctly return solution densities (line 13-14). The time complexity is $O(mP + m\tilde{I})$.

If the matrix A is dense we expect $|\tilde{I}| \simeq n$, therefore most of the γ_k are different from 1 and need to be computed. As soon as the matrix becomes sparse enough then $|\tilde{I}| \ll n$ and only a small fraction of γ_k needs to be computed, and that is where Algorithm 1 has an edge. In preliminary tests conducted over the benchmark problems presented in the Section 3, Algorithm 1 with arc consistency performed on average 25% better than the trivial implementation.

2.3 Counting Accuracy Analysis

But how accurate is the counting information we compute from these bounds? We compared the algorithm based on upper bounds with the previous approaches: Rasmussen's algorithm, Furer's algorithm and the sampling algorithm proposed in [13]. We generated `alldifferent` instances of size n ranging from 10 to 20 variables; variable domains were partially shrunk with a percentage of removal of values p varying from 20% to 80% in steps of 10%. We computed the exact number of solutions and removed those instances that were infeasible or for which enumeration took more than 2 days (leaving about one thousand instances). As a reference, the average solution count for the `alldifferent` instances with 20% to 60% of values removed is close to one billion solutions (and up to 10 billions), with 70% of removals it decreases to a few millions and with 80% of removals to a few thousands.

Randomized algorithms were run 10 times and we report the average of the results. In order to verify the performance with varying sampling time, we set a timeout of respectively 1, 0.1, 0.01, and 0.001 second. The running time of the counting algorithm based on upper bounds is bounded by the completion of Algorithm 1. The measures used for the analysis are the following:

counting error: relative error on the solution count of the constraint (computed as the absolute difference between the exact solution count and the estimated one and then divided by the exact solution count)

maximum solution density error: maximum absolute error on the solution densities (computed as the maximum of the absolute differences between the exact solution densities and the approximated ones)

average solution density error: average absolute error on the solution densities (computed as the average of the absolute differences between the exact solution densities and the approximated ones)

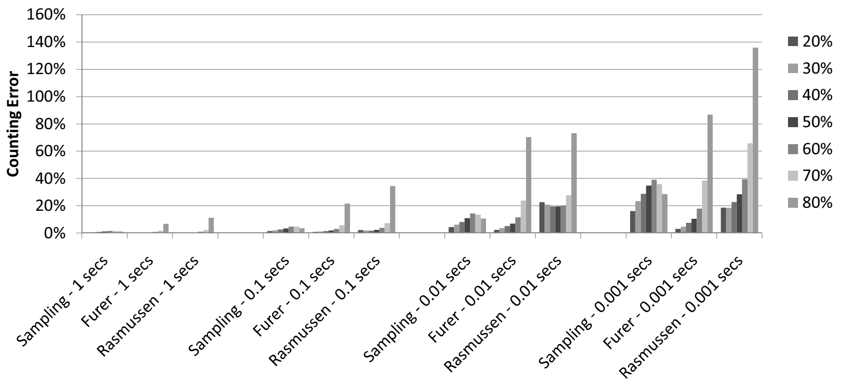


Fig. 1. Counting Error for one thousand alldifferent instances with varying variable domain sizes

Note that we computed absolute errors for the solution densities because counting-based heuristics usually compare the absolute value of the solution densities.

Plot 1 shows the counting error for the sampling algorithm, Rasmussen's and Furer's with varying timeout. Different shades of gray indicate different percentages of removals; series represent different algorithms and they are grouped based on the varying timeouts.

The relative counting error is maintained reasonably low for 1 and 0.1 second of sampling, however it increases considerably if we further decrease the timeout. Note that at 0.001 the sampling algorithm reaches its limit being able to sample only a few dozens solutions (both Rasmussen's and Furer's are in the order of the hundreds of samples). We left out the results of the algorithm based on upper bounds to avoid a scaling problem: the counting error varies from about 40% up to 2300% when enforcing domain consistency in Algorithm 1 (UB-DC) and up to 3600% with arc consistency (UB-AC) or 4800% with forward checking (UB-FC). Despite being tight upper bounds, they are obviously not suitable to approximate the solution count. Note nonetheless their remarkable running times: UB-DC takes about one millisecond whereas UB-AC and UB-FC about a tenth of a millisecond (with UB-FC being slightly faster).

Despite the poor performance in approximating the solution count, they provide a very good tradeoff in approximation accuracy and computation time when deriving solution densities.

Figure 2 and 3 show respectively the maximum and average solution density errors (note that the maximum value in the y-axis is different in the two plots). Again the sampling algorithm shows a better accuracy w.r.t. Rasmussen's and Furer's. Solution density errors are very well contained when using the upper bound approach: they are the best one when compared to the algorithms with an equivalent timeout and on average comparable to the results obtained by the sampling algorithm with a timeout of 0.01 seconds. Therefore, upper bounds offer a good accuracy despite employing just a tenth (UB-DC) or a hundredth

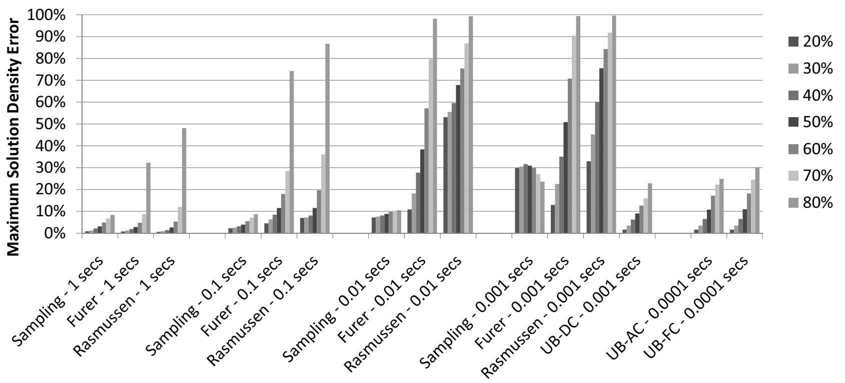


Fig. 2. Maximum Solution Density Error for one thousand alldifferent instances with varying variable domain sizes

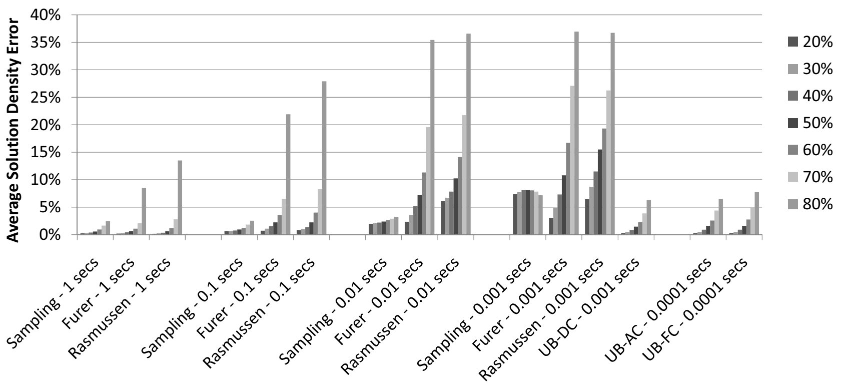


Fig. 3. Average Solution Density Error for one thousand alldifferent instances with varying variable domain sizes

(UB-AC, UB-FC) of the time of the sampling algorithm with comparable accuracy. Furthermore, errors for the upper bound algorithm are quite low when the domains are dense (low removal percentage) and on par with the sampling algorithm with a timeout of 0.1 or even 1 second. Note that in the context of search heuristics dense domains are more likely to happen closer to the root of the search tree hence when it is important to have a good heuristic guidance. Finally, as expected, enforcing a higher level of consistency during the local probes brings more accuracy, however the difference between UB-DC, UB-AC and UB-FC is not striking.

3 Experimental Results

In addition to counting accuracy, we measured the performance of search heuristics using such information to solve combinatorial problems by running experiments on two benchmark problems. We compared the maxSD heuristic [13] to Impact Based Search (IBS) and to the dom/ddeg variable selection heuristic coupled with the minconflicts or the lexicographic value selection heuristic. IBS selects the variable whose instantiation triggers the largest search space reduction (highest impact) that is approximated as the reduction of the Cartesian product of the variables' domains (see [8]). For the maxSD heuristic, we used the counting algorithm proposed in [13] and alternatively the approach based on UB^{BM} and UB^{LB}; for Algorithm 1 we tested three consistency levels (forward checking – UB-FC, arc consistency – UB-AC, and domain consistency – UB-DC). In order to get the best out of the heuristic presented in [13] throughout the instance sets, we decided to modify slightly the counting algorithm. The original counting algorithm first tries with an exact enumeration of the solutions for 0.2 second and in case of timeout proceeds with the sampling for the same amount of time. Here, we decided to do the same (exact+sampl) except that the sampling phase does not have a timeout but is instead bounded by the sample size, that is set dynamically. We got good results by setting the number of samples for each constraint to ten times the maximum domain size of the variables in the scope of the constraint. We also report on sampling used alone (sampl) and on maintaining either arc or domain consistency during sampling. Again for heuristics that have some sort of randomization, we took the average over 10 runs. We used Ilog Solver 6.6 on a AMD Opteron 2.4 GHz with 1GB of RAM. For each instance the timeout was set to 20 minutes.

3.1 Quasigroup with Holes Problem

We first tested our algorithm on the Quasigroup with Holes Problem (QWH). It is defined on a $n \times n$ grid whose squares each contain an integer from 1 to n such that each integer appears exactly once per row and column. The most common model uses a matrix of integer variables and an `alldifferent` constraint for each row and each column. We tested on the 40 hard instances used in [13] that have $n = 30$ and 42% of holes and we generated 60 additional instances outside the phase transition respectively with 45%, 47% and 50% of holes, using [2].

Results are shown in Table 1 (timeout instances are included in the averages). Figures 4 and 5 show the percentage of solved instances within a given time for the instance sets with respectively 42% and 45% of holes (time is not cumulative). Every heuristic solved at least 95% of instances in each set except for the instances with 42% of holes, where dom/ddeg solved 73% of them, IBS solved 85%, maxSD sampl-DC solved 80%, and maxSD sampl-AC solved 85% (see Figure 4). Note that all the heuristics based on maxSD solved every instance with 45% of holes or more. maxSD sampl brings an impressive reduction of the number of backtracks compared to the first two heuristics but without a significant computational advantage. In order to speed it up, we can add exact counting to produce more accurate information. maxSD exact+sampl is the heuristic with the lowest number of backtracks on the hard instances together with a significantly lower runtime; however, as expected, it runs longer on the easy instances: this can be explained by the fact that the easy instances have more loose constraints

Table 1. Average solving time (in seconds), median solving time and average number of backtracks for 100 QWH instances of order 30

heuristic	time	median	bckts	time	median	bckts
	42% holes			45% holes		
dom/ddeg; minconflicts	497.0	243.8	752883	6.6	1.0	11035
IBS	344.9	72.3	914849	94.1	19.9	247556
maxSD sampl-DC	398.8	358.3	15497	20.0	16.2	619
maxSD sampl-AC	339.7	285.4	15139	29.0	14.3	1349
maxSD exact+sampl-DC	132.0	72.2	4289	115.2	110.0	517
maxSD exact+sampl-AC	142.9	67.9	5013	125.7	110.8	1092
maxSD UB-DC	110.5	13.6	31999	1.3	1.0	164
maxSD UB-AC	82.4	3.7	68597	0.7	0.2	582
maxSD UB-FC	105.5	7.8	104496	0.5	0.3	447
heuristic	time	median	bckts	time	median	bckts
	47% holes			50% holes		
dom/ddeg; minconflicts	60.3	0.1	118089	0.1	0.1	36
IBS	5.1	2.2	16126	2.8	2.2	10012
maxSD sampl-DC	22.8	10.0	657	19.4	13.3	355
maxSD sampl-AC	6.3	6.1	34	7.7	7.7	8
maxSD exact+sampl-DC	187.3	187.8	8	269.0	270.0	29
maxSD exact+sampl-AC	191.0	187.0	450	262.0	263.5	2
maxSD UB-DC	1.5	1.5	20	2.4	2.3	3
maxSD UB-AC	0.3	0.3	30	0.3	0.3	2
maxSD UB-FC	0.3	0.3	56	0.3	0.3	6

therefore the initial exact enumeration is more likely to time out. In Figure 4 we can see that the sampling algorithm alone is able to solve some instances within few seconds whereas maxSD exact+sampl-DC does not solve any instance within 40 seconds because of the high overhead due to exact enumeration. Sampling alone struggles more with the hard instances and it ends up solving just 85% of the instances whereas maxSD exact+sampl-DC solves 97% of the instances.

The previous heuristics were significantly outperformed in all the instance sets by the heuristics based on upper bounds. As shown in Figure 6, maxSD UB-DC, maxSD UB-AC, maxSD UB-FC are very quick in solving easy instances and yet they are capable of solving the same number of instances as maxSD exact+sampl-DC. The latter heuristic shows its limit already in the set of instances with 45% of holes where no instance is solved within a hundred seconds, whilst maxSD based on upper bounds almost instantaneously solves all the instances. maxSD UB-AC was overall the best of the set on all the instances with up to a two orders of magnitude advantage over IBS in terms of solving time and up to four orders of magnitude for the number of backtracks. Enforcing a higher level of consistency leads to better approximated solution densities and to a lower number of backtracks, but it is more time consuming than simple arc consistency. A weaker level of consistency like forward checking can pay off on easy instances but it falls short compared to UB-AC on the hard ones. Note also that maxSD UB-DC increases the solving time, despite lowering the backtracks, when the

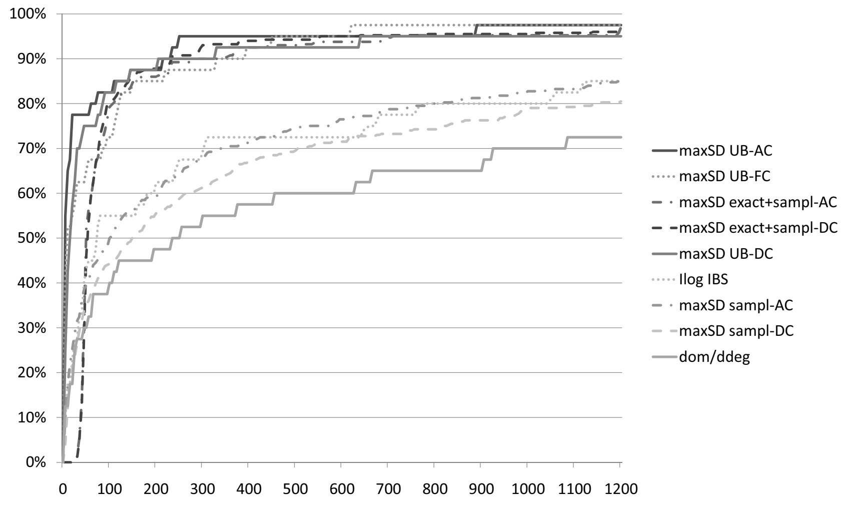


Fig. 4. Percentage of solved instances vs time for QWH instances with 42% of holes

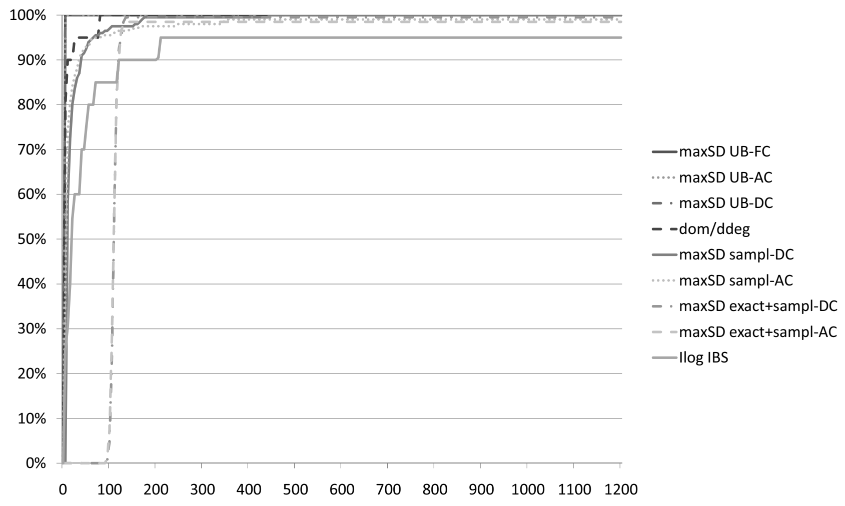


Fig. 5. Percentage of solved instances vs time for QWH instances with 45% of holes

instances have more holes (apart from the 42% holes instances): in those cases m increases and the overhead of propagation becomes important (see Section 2.2). However we could not reuse the maximum matching and the strongly connected components (see [9]) computed for the propagation (there is no access to the underlying propagation code) — a more coupled integration of the counting algorithm with the propagation algorithm could lead to a performance gain. We did not consider attempting exact counting before computing the upper bound (exact+UB) because this would have caused timeouts on the easier instances,

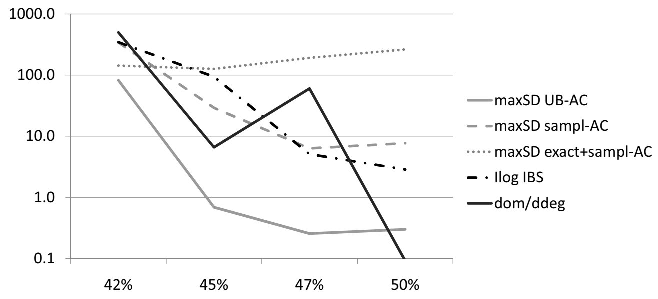


Fig. 6. Total average time vs % holes

as observed for exact+sampl, and unnecessarily increased the runtimes on such instances.

We end this section by mentioning a few unsuccessful trials. We tried computing the permanent bound of Algorithm 1 (line 1) incrementally during search. We also tried to apply one technique inspired by [10]. Soules observed that $per(A) = per(A^T)$ however $UB^{BM}(A)$ is not necessarily equal to $UB^{BM}(A^T)$ (the same for the Liang-Bai bound). We implemented the code to compute the bounds on both the matrix and its transposed and we kept the minimum of such upper bounds. Neither of these two attempts led to any significant improvement.

Adding Randomized Restarts. The QWH problem exhibits heavy-tail behavior in runtime distributions when the instances are generated close to the phase transition. Nonetheless, heavy-tails can be largely avoided by adding randomized restarts to the search procedure. We tried a subset of the heuristics tested above with randomized restart techniques. All the heuristics have been randomized such that one variable-value pair is chosen at random with equal probability between the best two provided by the heuristic. We implemented Walsh's universal strategy to generate the restart cutoff sequence (that is $\alpha r^0, \alpha r^1, \alpha r^2, \ldots$ with $r = 2$ and α equal to 5% the number of variables). The heuristics were not able to gain from the randomized restarts on the easier instances but only on the ones with 42% of holes. maxSD UB-FC improved by 35% the average running time whereas IBS and dom/ddeg; minconflicts degraded their performance by respectively about 45% and 32%.

3.2 Travelling Tournament Problem with Predefined Venues

The Travelling Tournament Problem with Predefined Venues (TTPPV) was introduced in [5] and consists of finding an optimal single round robin schedule for a sport event. Given a set of n teams, each team has to play against each other team. In each game, a team is supposed to play either at home or away, however no team can play more than three consecutive times at home or away. The particularity of this problem resides on the venues of each game, that are predefined, i.e. if team a plays against b we already know whether the game is going to be held at a's home or at b's home. A TTPPV instance is said to be balanced if the number of home and away games differ by at most one for each team; otherwise it is referred to as unbalanced or random.

The TTPPV was originally introduced as an optimization problem where the sum of the travelling distance of each team has to be minimized, however [5] shows that it is particularly difficult to find even a feasible solution using traditional integer linear programming methods (ILP). Balanced instances of size 18 and 20 (the size is the number of teams) were taking from roughly 20 to 60 seconds to find a first feasible solution with ILP; unbalanced instances could take up to 5 minutes (or even time out after 2 hours of computation). Hence, the feasibility version of this problem already represents a challenge. Therefore we attempted to tackle it with Constraint Programming and solution counting heuristics.

We modelled the problem in the following way:

$$\texttt{alldifferent}((x_{ij})_{1\leq j\leq n-1}) \qquad 1 \leq i \leq n \tag{4}$$

$$\texttt{regular}((x_{ij})_{1\leq j\leq n-1}, PV_i) \qquad 1 \leq i \leq n \tag{5}$$

$$\texttt{alldifferent}((x_{ij})_{1\leq i\leq n}) \qquad 1 \leq j \leq n-1 \tag{6}$$

$$x_{ij} = k \iff x_{kj} = i \qquad 1 \leq i \leq n, 1 \leq j \leq n-1 \tag{7}$$

$$x_{ij} \in \{1, \ldots, n\} \qquad 1 \leq i \leq n, 1 \leq j \leq n-1 \tag{8}$$

A variable $x_{ij} = k$ means that team i plays against team k at round j. Constraint (7) enforces that if team a plays against b then b plays against a in the same round; constraint (4) enforces that each team plays against every other team; the home-away pattern associated to the predefined venues of team i (PV_i) is defined through a regular constraint (5). Finally constraint (6) is redundant and used to achieve additional filtering.

We tested 40 balanced and 40 unbalanced instances borrowed from [5] with sizes ranging from 14 to 20. For the `regular` constraint we used the counting algorithm proposed in [13]. Results are reported in Table 2 for balanced and unbalanced instances (timeout instances are included in the averages). Figure 7 shows the percentage of unbalanced instances solved within a given time limit (time is not cumulative).

Table 2. Average solving time (in seconds), number of backtracks, and percentage of instances solved for 80 TTPPV instances

	balanced			unbalanced		
heuristic	time	bckts	%solved	time	bckts	% solved
dom/ddeg; lexico	0.1	27	100%	901.2	2829721	25%
IBS	10.7	8250	100%	631.7	1081565	50%
maxSD sampl-DC	25.9	2	100%	140.7	3577	91%
maxSD exact+sampl-DC	120.4	1	100%	216.9	1210	91%
maxSD UB-DC	6.7	1	100%	36.8	245	98%
maxSD UB-AC	0.6	1	100%	30.6	2733	98%
maxSD UB-FC	0.5	1	100%	30.5	2906	98%

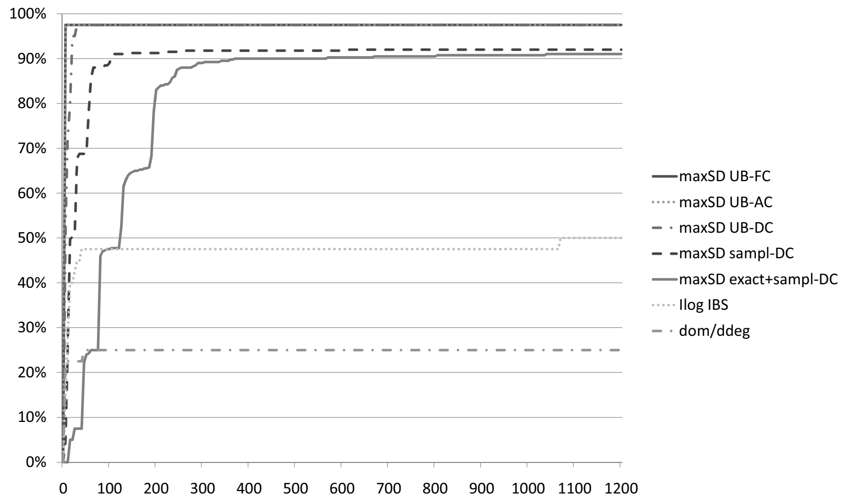


Fig. 7. Percentage of solved instances vs time for non balanced instances of the TTPPV

Balanced instances do not present a challenge for any of the heuristics tested: the lightweight heuristic dom/ddeg; lexico is the one performing better together with maxSD based on upper bounds. The sampling algorithm here shows its main drawbacks i.e. it is not competitive in solving easy instances: the number of backtracks is low indeed but the time spent in sampling is simply a waste of time on easy instances though crucial on difficult ones. Exact enumeration adds another constant overhead to the counting procedure with the results of being three orders of magnitude slower than upper bounds based on arc consistency or forward checking.

Unbalanced instances are harder to solve and none of the heuristics were able to solve all 40 instances within the time limit — note that in this set of instances six out of forty are infeasible. maxSD is significantly faster than any other heuristic: counting based on upper bounds also allowed to cut computing time by almost 80% w.r.t. the sampling algorithm and by 85% w.r.t. exact enumeration and sampling. 90% of the instances are solved in 100 seconds by maxSD sampl-DC whereas maxSD UB-AC and maxSD UB-FC take less than 2 seconds to solve 97.5% of the instances (maxSD UB-FC takes slightly more).

Remarkably, maxSD with upper bounds proved the infeasibility of five of the six instances, and with small search trees. None of the other heuristics tested were able to prove the infeasibility of any of the six instances. Gains are remarkable also in the number of backtracks (three orders of magnitude better than the other heuristics). maxSD with upper bound-based counting turned out to be the most consistent heuristic, performing very well both on hard and easy instances with an average solving time up to 20 times better than IBS.

4 Conclusion and Future Work

Solution counting heuristics are to date among the best generic heuristics to solve CSPs. We believe that the basic idea and some of the algorithms are relevant to general AI, not just within the CSP framework, and to OR as well. As an indication, this line of work recently inspired new branching direction heuristics for mixed integer programs [7].

In this paper, we propose an algorithm to compute solution densities for `alldifferent` constraints that actually broaden the applicability of such heuristics. The new approach is suitable both for easy and hard problems and it proves the competitiveness of solution counting based heuristics w.r.t. other state-of-the-art heuristics. In the future, we would like to try the more sophisticated upper bounds proposed in [10] and [11] to see whether they can bring an actual benefit to our heuristics. An interesting combination is to use upper bounds to identify a small subset of promising variable-value pairs and then apply an algorithm with a better approximation accuracy (such as sampling) on the selected subset.

References

1. Bregman, L.M.: Some Properties of Nonnegative Matrices and their Permanents. Soviet Mathematics Doklady 14(4), 945–949 (1973)
2. Gomes, C., Shmoys, D.: Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem. In: COLOR 2002: Proceedings of Computational Symposium on Graph Coloring and Generalizations, pp. 22–39 (2002)
3. Jurkat, W.B., Ryser, H.J.: Matrix Factorizations of Determinants and Permanents. Journal of Algebra 3, 1–27 (1966)
4. Liang, H., Bai, F.: An Upper Bound for the Permanent of (0,1)-Matrices. Linear Algebra and its Applications 377, 291–295 (2004)
5. Melo, R.A., Urrutia, S., Ribeiro, C.C.: The traveling tournament problem with predefined venues. Journal of Scheduling 12(6), 607–622 (2009)
6. Minc, H.: Upper Bounds for Permanents of (0, 1)-matrices. Bulletin of the American Mathematical Society 69, 789–791 (1963)
7. Pryor, J.: Branching Variable Direction Selection in Mixed Integer Programming. Master's thesis, Carleton University (2009)
8. Refalo, P.: Impact-Based Search Strategies for Constraint Programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)
9. Régin, J.-C.: A Filtering Algorithm for Constraints of Difference in CSPs. In: AAAI 1994: Proceedings of the Twelfth National Conference on Artificial Intelligence, vol. 1, pp. 362–367. American Association for Artificial Intelligence, Menlo Park (1994)
10. Soules, G.W.: New Permanental Upper Bounds for Nonnegative Matrices. Linear and Multilinear Algebra 51(4), 319–337 (2003)
11. Soules, G.W.: Permanental Bounds for Nonnegative Matrices via Decomposition. Linear Algebra and its Applications 394, 73–89 (2005)
12. Valiant, L.: The Complexity of Computing the Permanent. Theoretical Computer Science 8(2), 189–201 (1979)
13. Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics. Constraints 14(3), 392–413 (2009)

Author Index