Tobias Achterberg   J. Christopher Beck (Eds.)

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

8th International Conference, CPAIOR 2011
Berlin, Germany, May 23-27, 2011
Proceedings

Springer

Volume Editors

Tobias Achterberg
Zuse Institut Berlin
Takustr. 7
14195 Berlin, Germany
E-mail: achterberg@zib.de

J. Christopher Beck
University of Toronto
Department of Mechanical
and Industrial Engineering
5 King's College Rd.
Toronto, ON, M5S 3G8, Canada
E-mail: jcb@mie.utoronto.ca

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-21310-6 e-ISBN 978-3-642-21311-3
DOI 10.1007/978-3-642-21311-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: Applied for

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

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The 8th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2011) was held in Berlin, Germany, May 23–27, 2011.

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.

Submissions for this year were 22 long papers and 13 short papers. Each paper received three independent peer reviews which formed the basis for the acceptance of 13 long papers and 7 short papers. These papers are published in full in the proceedings. Many thanks to the members of the Program Committee and the external reviewers, who reviewed all the submissions in detail and discussed conflicting papers deeply. In addition, the Program Chairs solicited late-breaking abstracts for presentation at the conference. The number of selected abstracts was not yet available at press time.

Thanks to the Department of Scientific Information of the Zuse Institute Berlin, video recordings of the presentations of CPAIOR 2011 were made. They can be found on the CPAIOR 2011 webpage at `cpaior2011.zib.de`. We would like to especially thank Wolfgang Dalitz and the Web Technology and Multimedia group for creating this valuable record of the conference.

This volume includes abstracts of the three invited talks of CPAIOR:

– Craig Boutilier, University of Toronto, Canada, on the use of AI and OR techniques for preference elicitation and learning in social choice
– Ian Gent, University of St. Andrews, UK, on constraint propagation in CP and SAT
– Andrea Lodi, University of Bologna, Italy, on bilevel programming and its impact in branching, cutting and complexity

CPAIOR 2011 also included a one-day Master Class and one day of workshops. The Master Class is intended for PhD students, researchers, and practitioners and was held on the theme of search in AI, OR, CP, and SAT. Four speakers addressed each of these topics individually and then participated in a panel discussion to highlight opportunities for cross-fertilization. The speakers were:

– John Chinneck, Carleton University, Canada, on search in mixed-integer programming
– Gilles Pesant, Ecole Polytechnique, Canada, on search in constraint programming

- Nathan Sturtevant, University of Denver, USA, on search in AI
- Marijn Heule, Delft University of Technology, The Netherlands, on search for satisfiability

The one-day workshop program consisted of four workshops:

- *Energy*
  Organized by Armin Fügenschuh, Benjamin Hiller, Jesco Humpola, and Thorsten Koch, all from Zuse Institute Berlin, Germany
- *Hybrid Methods for Nonlinear Combinatorial Optimization Problems*
  Organized by Stefano Gualandi, Universita di Pavia, Italy, and Pietro Belotti, Clemson University, USA
- *Innovative Scheduling and Other Applications Using CP-AI-OR*
  Organized by Armin Wolf, Fraunhofer-Institut für Rechnerarchitektur und Softwaretechnik, FIRST, Germany, and Petra Hofstedt, Brandenburgische Technische Universität Cottbus, Germany
- *Mathematical Optimization of Railway Systems*
  Organized by Ralf Borndörfer, Zuse Institute Berlin/TU Darmstadt, Germany, Holger Flier, ETH Zürich, Switzerland, Martin Fuchsberger, ETH Zürich, Switzerland, and Thomas Schlechte, Zuse Institute Berlin, Germany

The staff at Zuse Institute Berlin did an outstanding job providing administrative support, making sure the money was in the right place at the right time, and in handling the registrations. In particular, we would like to thank Annerose Steinke, Sylke Arencibia, Sybille Mattrisch, and Bettina Kasse.

A special thanks goes to the Conference Chairs, Timo Berthold, Ambros Gleixner, Stefan Heinz, and Thorsten Koch, for the organization and substantial efforts on sponsorship, publicity, logistics, and all the other things that have to happen behind the scenes to make a conference work.

Finally, we would like to thank the sponsors who made it possible to organize this conference:

DFG Research Center Matheon, Zuse Institute Berlin, the Association for Constraint Programming, SAS, IBM, AIMMS, Gurobi Optimization, FICO, the Institute for Computational Sustainability, GAMS, IVU Traffic Technologies AG, MOSEK Optimization, National ICT Australia, Jeppesen, the ABB Group, atesio GmbH, ProCom GmbH, AMPL, and OGE.

May 2011

Tobias Achterberg
J. Christopher Beck

Conference Organization

Program Chairs

Tobias Achterberg, J. Christopher Beck

Conference Chairs

Timo Berthold, Ambros M. Gleixner, Stefan Heinz, Thorsten Koch

Program Committee

Oliver Bastert
Bob Bixby
John Chinneck
Andrew Davenport
Sophie Demassey
Bernard Gendron
Carla Gomes
Youssef Hamadi
Emmanuel Hebrard
Susanne Heipcke
John Hooker
Jeff Linderoth
Andrea Lodi
Michele Lombardi
Luc Mercier
Laurent Michel
Ian Miguel
Michela Milano
Laurent Perron
Gilles Pesant
Marc Pfetsch
Claude-Guy Quimper
Ted Ralphs
Jean-Charles Régin
Andrea Rendl
Louis-Martin Rousseau
Ashish Sabharwal
Meinolf Sellmann
Paul Shaw
Helmut Simonis
Paolo Toth
Michael Trick
Gilles Trombettoni
Klaus Truemper
Pascal Van Hentenryck
Willem-Jan van Hoeve
Mark Wallace
Armin Wolf
Tallys Yunes
Alessandro Zanarini

External Reviewers

Alejandro Arbelaez
David Bergman
Andre Cire
Lars Kotthoff
Robert Mateescu
Eoin O'Mahony
Yves Papegay
Philippe Refalo
Fabrizio Riguzzi
Petr Vilím

Table of Contents

Preference Elicitation and Preference Learning in Social Choice

Craig Boutilier

Department of Computer Science, University of Toronto,
Toronto, ON, Canada
cebly@cs.toronto.edu
http://www.cs.toronto.edu/~cebly

Social choice has been the subject of intense investigation within computer science, AI, and operations research, in part because of the ease with which preference data from user populations can now be elicited, assessed, or estimated in online settings. In many domains, the preferences of a group of individuals must be aggregated to form a single consensus recommendation, placing us squarely in the realm of social choice.

I argue that the application of social choice models and voting schemes to domains like web search, product recommendation and social networks places new emphasis, in the design of preference aggregation schemes, on issues such as: articulating decision criteria suitable to the application at hand; approximate winner determination; incremental preference elicitation; learning methods for models of population preferences; and more nuanced analysis of the potential for manipulation.

In this talk, I'll provide an overview of some of these challenges and outline some of our recent work tackling of them, including methods for: learning probabilistic models of population preferences from choice data; robust optimization (winner determination) in the presense of incomplete user preferences; and incremental vote/preference elicitation for group decision making. Each of these poses interesting modeling and optimization challenges that are best tackled using a combination of techniques from AI, operations research, and statistics.

Parts of this talk describe joint work with Tyler Lu, Department of Computer Science, University of Toronto.

Propagation in Constraints: How One Thing Leads to Another

Ian P. Gent

School of Computer Science, University of St. Andrews,
St. Andrews, Scotland, UK
ipg@cs.st-andrews.ac.uk

Abstract of Invited Talk

At a conference such as CPAIOR, we have experts from many different approaches to searching huge combinatorial spaces. Much of what we all do is common, for example similar search methods, heuristics, and learning techniques. So what is it that is essentially different about Constraint Programming in particular? One answer is the power and diversity of constraint propagation algorithms. By contrast, other search disciplines often rely on just one propagation technique, such as unit propagation in SAT.

So to paraphrase the property expert, for the duration of this talk I'll say that the three most important aspects of Constraint Programming are "Propagation, Propagation, Propagation." I'll talk about all three. First, what Propagation is and the theory of how propagation algorithms work. Second, how Propagation forms the beating heart of a modern constraint solver, typically throwing away a lot of the theory in the process. And third, some aspects of Propagation in constraints being researched right now.

I hope to bring out some of the beauty and surprises in propagation. I also hope to show how ideas from one discipline have propagated to another, for example the introduction of watched literals from SAT into Constraint Programming. I also hope one propagation will lead to another as some of these lovely ideas propagate again.

On Bilevel Programming and Its Impact in Branching, Cutting and Complexity

Andrea Lodi

DEIS, Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
andrea.lodi@unibo.it

Abstract. Bilevel programming is a rich paradigm to express a variety of real-world applications including game theoretic and pricing ones. However, what we are interested in this talk is to discuss the bilevel nature of two of the most crucial ingredients of enumerative methods for solving combinatorial optimization problems, namely *branching* and *cutting*.

Specifically, we discuss a new branching method for 0-1 programs called *interdiction branching* [3] that exploits the intrinsic bilevel nature of the problem of selecting a branching disjunction. The method is designed to overcome the difficulties encountered in solving problems for which branching on variables is inherently weak. Unlike traditional methods, selection of the disjunction in interdiction branching takes into account the best feasible solution found so far.

On the cutting plane side, we examine the nature of the so-called separation problem, which is that of generating a valid inequality violated by a given real vector, usually arising as the solution to a relaxation of the original problem. We show that the problem of generating a maximally violated valid inequality often has a natural interpretation as a bilevel program [2]. In some cases, this bilevel program can be easily reformulated as a single-level mathematical program, yielding a standard mathematical programming formulation for the separation problem. In other cases, no reformulation exists yielding surprisingly interesting examples of problems arising in the complexity hierarchies introduced by Jeroslow [1].

Keywords: Bilevel programming, branching, cutting, complexity.

References

1. Jeroslow, R.: The polynomial hierarchy and a simple model for competitive analysis. Mathematical Programming 32, 146–164 (1985)
2. Lodi, A., Ralphs, T.K., Woeginger, G.: Bilevel Programming and Maximally Violated Valid Inequalities. Technical Report OR/11/3, DEIS - Università di Bologna
3. Lodi, A., Ralphs, T.K., Rossi, F., Smriglio, S.: Interdiction Branching. Technical Report OR/09/10, DEIS - Università di Bologna

Optimization Methods for the Partner Units Problem*

Markus Aschinger[1], Conrad Drescher[1], Gerhard Friedrich[2], Georg Gottlob[1], Peter Jeavons[1], Anna Ryabokon[2], and Evgenij Thorstensen[1]

[1] Computing Laboratory, University of Oxford
[2] Institut für Angewandte Informatik, Alpen-Adria-Universität Klagenfurt

Abstract. In this work we present the Partner Units Problem as a novel challenge for optimization methods. It captures a certain type of configuration problem that frequently occurs in industry. Unfortunately, it can be shown that in the most general case an optimization version of the problem is intractable. We present and evaluate encodings of the problem in the frameworks of answer set programming, propositional satisfiability testing, constraint solving, and integer programming. We also show how to adapt these encodings to a class of problem instances that we have recently shown to be tractable.

1 Introduction

The Partner Units Problem (Pup) has recently been proposed as a new challenge in automated configuration [8]. It captures the essence of a specific type of configuration problem that frequently occurs in industry.

Informally the Pup can be described as follows: Consider a set of sensors that are grouped into zones. A zone may contain many sensors, and a sensor may be attached to more than one zone. The Pup then consists of connecting the sensors and zones to control units, where each control unit can be connected to the same fixed maximum number *UnitCap* of zones and sensors.[1] Moreover, if a sensor is attached to a zone, but the sensor and the zone are assigned to different control units, then the two control units in question have to be directly connected. However, a control unit cannot be connected to more than *InterUnitCap* other control units (the partner units).

For an application scenario consider, for example, a museum where we want to keep track of the number of visitors that populate certain parts (zones) of the building. The doors leading from one zone to another are equipped with sensors. To keep track of the visitors the zones and sensors are attached to control units; the adjacency constraints on the control units ensure that communication between units can be kept simple.

* Work funded by FFG FIT-IT Grant 825071 (Klagenfurt) and EPSRC Grant EP/G055114/1 (Oxford).

[1] For ease of presentation and without loss of generality we assume that *UnitCap* is the same for zones and sensors.

Let us emphasize that the Pup is not limited to this application domain: it occurs whenever sensors that are grouped into zones have to be attached to control units, and communication between units should be kept simple. Typical applications include intelligent traffic management, or surveillance and security applications. The Pup has been introduced as a novel benchmark instance at this year's answer set programming competition [2].

Figure 1 shows a Pup instance and a solution for the case *UnitCap* = *InterUnitCap* = 2. In this example six sensors (left) and six zones (right), which are completely inter-connected, are partitioned into units (shown as squares) respecting the adjacency constraints. Note that for the given parameters this is a maximal solvable instance; it is not possible to connect a new zone or sensor to any of the existing ones.

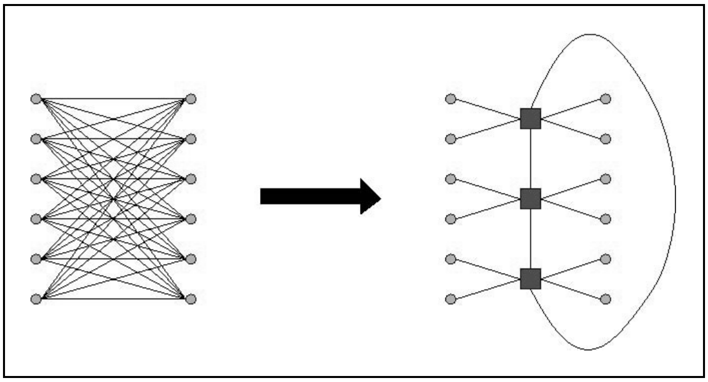


Fig. 1. Solving a $K_{6,6}$ Partner Units Instance — Partitioning Sensors and Zones into Units on a Circular Unit Layout

Very recently, we have shown that the case where *InterUnitCap* = 2 and *UnitCap* = k for some fixed k is tractable, by giving a specialized NLogSpace algorithm that is based on the notion of a path decomposition [1]. While this case is of some importance for our partners in industry, the general case is also interesting: Consider, for example, a grid of rooms, where every room is accessible from each neighboring room, and all the doors are fitted with a sensor. Moreover, assume there are doors to the outside on two sides of the building; the corresponding instance is shown in Figure 2, with rooms as black squares, and doors as circles. It is not hard to see that this instance is unsolvable for *InterUnitCap* = 2 and *UnitCap* = 2. However, it is easily solved for *InterUnitCap* = 4 and *UnitCap* = 2: Every room goes on a distinct unit, together with the sensors to the west and to the north; the connections between units correspond to those between rooms. Clearly this solution is optimal, in the sense of using the least possible number of units.

In this paper we present and evaluate encodings in the optimization frameworks of answer set programming, constraint satisfaction, Sat-solving, and integer programming, that can be used to solve the general version of the Pup.

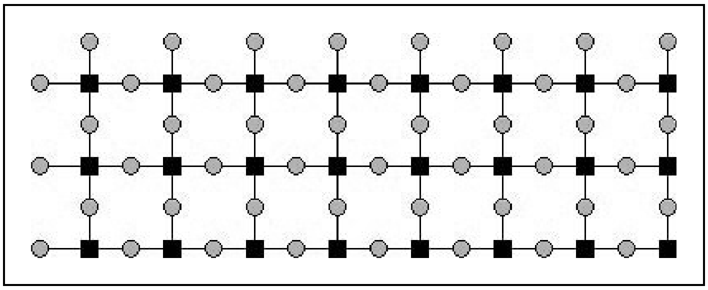


Fig. 2. A Grid-like Pup Instance

We also show how to adapt these encodings to the special case *InterUnitCap* $= 2$, and compare the adapted encodings against our specialized algorithm.

It is worth emphasizing that we do not take our encodings/algorithms to be the final answer on the Pup. Instead we hope that our work will spark interest in the problem across the different optimization research communities, eventually resulting in better encodings and better theoretical understanding of the problem.

The remainder of this paper is organized as follows: In section 2 we give the basic formal definitions, and identify general properties of the Pup. Then, in section 3 we present problem models in the frameworks of answer set programming, propositional satisfiability testing, integer programming, and constraint solving; these problem models can be used for arbitrary fixed values of *InterUnitCap*. In section 4 we briefly recall our recent tractability results for the case *InterUnitCap* $= 2$, and show how the various Pup encodings presented in this paper can be adapted to this special case. Finally, in section 5 we evaluate the performance of our encodings, and in section 6 we list some directions for future research.

2 The Partner Units Problem

2.1 Formal Definition

Formally, the Pup consists of partitioning the vertices of a bipartite graph $G = (V_1, V_2, E)$ into a set U of bags such that each bag

- contains at most *UnitCap* vertices from V_1 and at most *UnitCap* vertices from V_2; and
- has at most *InterUnitCap* adjacent bags, where the bags U_1 and U_2 are adjacent whenever $v_i \in U_1$ and $v_j \in U_2$ and $(v_i, v_j) \in E$.

To every solution of the Pup we can associate a solution graph. For this we associate to every bag $U_i \in \mathcal{U}$ a vertex v_{U_i}. Then the solution graph G^* has the vertex set $V_1 \cup V_2 \cup \{v_{U_i} \mid U_i \in \mathcal{U}\}$ and the set of edges $\{(v, v_{U_i}) \mid v \in U_i \wedge U_i \in \mathcal{U}\}$

$\cup \{(v_{U_i}, v_{U_j}) \mid U_i$ and U_j are adjacent.$\}$. In the following we will refer to the subgraph of the solution graph induced by the v_{U_i} as the *unit graph*.

The following are the two most important reasoning tasks for the PUP: Decide whether there is a solution, and find an optimal solution, that is, one that uses the minimal number of control units. We are especially interested in the latter problem. For this we consider the corresponding decision problem: Is there a solution with a specified number of units? The rationale behind the optimization criterion is that (a) units are expensive, and (b) connections are cheap.

2.2 The Partner Units Problem Is Intractable

By a reduction from BINPACKING, it can be shown that the optimization version of the PUP is intractable when *InterUnitCap* $= 0$, and *UnitCap* is part of the input. Observe that clearly the PUP is in NP (cf. also section 3).

Theorem 1 ([1]). *Deciding whether a* PUP *instance has a solution with a given number of units is* NP*-complete, when InterUnitCap* $= 0$*, and UnitCap is part of the input.*

In practice, however, the value of *UnitCap* will typically be fixed.

2.3 Forbidden Subgraphs of the PUP

In solvable instances sensors cannot belong to arbitrarily many zones (and vice versa) [1]:

Lemma 1 (Forbidden Subgraphs of the PUP). *A* PUP *instance has no solution if it contains* $K_{1,n}$ *or* $K_{n,1}$ *as a subgraph, where* $n = ((\mathit{InterUnitCap} + 1) * \mathit{UnitCap}) + 1$.

2.4 *K*-Regular Graphs

There is an interesting connection between most general solutions to the PUP and k-regular unit graphs, where a graph is k-regular if every vertex has exactly k neighbors: In k-regular unit graphs we are exploiting the *InterUnitCap* capacity for connections between units to the fullest. Hence, k-regular unit graphs are the most general solutions (if they exist).

In the case where $k = 2$, there is exactly one k-regular graph, the cycle; we exploit this fact in section 4. In the case where k is odd, k-regular unit graphs only exist if there is an even number of units ("hand-shaking lemma"). Moreover, for $k > 2$ the number of distinct most general unit graphs grows rapidly: E.g. for $k = 4$ there are six distinct graphs on eight vertices, and 8037418 on sixteen vertices; for twenty vertices not all distinct graphs have been constructed [13]. It can be shown that all these solution topologies can be forced:

Observation 1 (PUP instances and k-regular graphs). *For every k-regular graph G_k there exists a* PUP *instance G with InterUnitCap* $= k$ *such that in every solution of G the unit graph is G_k.*

Proof. Construct the instance G as follows:

1) First connect $2 * \mathit{UnitCap}$ vertices (i.e. sensors and zones) to each node in G_k. Let the set of all sensors (zones) be V_1 (V_2).
2) The instance G contains all edges (v_1, v_2), where $v_1 \in V_1$ and $v_2 \in V_2$ are connected to either the same or adjacent nodes in G_k.

We show that every solution is isomorphic to G_k. We consider two cases:

– $0 \leq k \leq 1$: The result is immediate.
– $k > 1$: Let u_0 be a node in G_k with neighbors $u_j : 1 \leq j \leq k$. Denote by $V_1^{u_i}$ and $V_2^{u_i}$ the sensors and zones created in G for $u_i : 0 \leq i \leq k$. Let G'_k be an optimal solution for G. We need two observations: (1) For each $0 \leq i \leq k$ both $V_1^{u_i}$ and $V_2^{u_i}$ are on the same unit in G'_k. (2) For $0 \leq i < j \leq k$ if $V_1^{u_i} \cup V_2^{u_i}$ and $V_1^{u_j} \cup V_2^{u_j}$ are connected in G then their units are connected in G'_k.

Hence, if $\mathit{InterUnitCap} > 2$, and there are no restrictions on the solution topology in the application domain, then it is practically not feasible to iteratively try all most general solution topologies. The solution topology will have to be inferred instead.

2.5 Bounds on the Number of Units Required

Let us next point out that the number of units used when solving an instance $G = (V_1, V_2, E)$ is bounded from below by $lb = \lceil \frac{\max(|V_1|,|V_2|)}{\mathit{UnitCap}} \rceil$. Clearly it can also be bounded from above by $ub = |V_1| + |V_2|$ — we never need empty units. If $\mathit{InterUnitCap} = 2$ and $\mathit{UnitCap} > 1$ we can show that the stronger $ub = \max(|V_1|, |V_2|)$ holds for connected instances [1]. Now, if there are multiple connected components C_i in the instance with upper bounds ub_i, then we have $ub = \sum ub_i$. We conjecture that this also holds for $\mathit{InterUnitCap} > 2$, but have so far been unable to prove it. These bounds are exploited in the problem encodings below either for keeping the problem model small, or to limit the depth of iterative deepening search. In this approach we first try to find a solution with lb units; if that fails increase lb by one; the first solution found will be optimal. For both approaches better upper bounds are desirable.

2.6 Symmetry Breaking

If we don't use iterative deepening search, then in some problem models we might obtain solutions with empty units. Here we can do symmetry breaking, by demanding that whenever unit j has a sensor or zone assigned to it, then every unit $j' < j$ also has some sensor or zone assigned to it.

We can also rule out a lot of the connections between sensors and units (or alternatively, between zones and units) immediately. Consider sensors and units: Sensor 1 must be somewhere, so it might as well be on unit 1. Sensor 2 can either be on unit 1 or on a new unit, let's say 2, and so on. Unfortunately, we cannot do this on both sensors and zones, since the edges may disallow a zone and a sensor on the same unit.

3 Encodings for the General Case

We are next going to outline encodings of the PUP where *InterUnitCap* is an arbitrary fixed constant. Due to cost considerations we are especially interested in the optimization version of the PUP: We want to minimize the number of expensive units used, but do not consider the cost for the cheap connections between them.

In particular, we show how the problem can be encoded in the frameworks of propositional satisfiability testing (SAT), integer programming (IP), and constraint solving (CSP), all of which can be considered as state-of-the-art for optimization problems [11]. In addition we will also describe an encoding in answer set programming (ASP), a currently very successful knowledge representation formalism.

3.1 Answer Set Programming

First, we show how to encode the PUP in answer set programming [9,12] which has its roots in logic programming and deductive databases. This knowledge representation language is based on a decidable fragment of first-order logic and is extended with language constructs such as aggregation and weight constraints. Already the propositional variant allows the formulation of problems beyond the first level of the polynomial hierarchy. In case standard propositional logic is employed[2], an answer set corresponds to a minimal logical model by definition of [12].

In our encodings a solution (i.e. a configuration) is the restriction of an answer set to those literals that satisfy the defined solution schema.

To encode a PUP instance in ASP we represent the zones and sensors by the unary predicates `zone/1` and `sensor/1`. The edges between zones and sensors are represented by the binary predicate `zone2sensor/2`. The number of available units `lower` $= \left\lceil \frac{\max(|Sensors|,|Zones|)}{2} \right\rceil$, `unitCap` and `interUnitCap` are each specified by a constant. The PUP is then encoded via the following logical sentences employing the syntax described in [3]:

```
(1)  unit(1..lower).
(2)  1 { unit2zone(U,Z) : unit(U) } 1 :- zone(Z).
(3)  1 { unit2sensor(U,S) : unit(U) } 1 :- sensor(S).
(4)  :- unit(U), unitCap+1 { unit2zone(U,Z): zone(Z) }.
(5)  :- unit(U), unitCap+1 { unit2sensor(U,S): sensor(S) }.
(6)  partnerunits(U,P) :- unit2zone(U,Z), zone2sensor(Z,S),
                          unit2sensor(P,S), U!=P.
(7)  partnerunits(U,P) :- partnerunits(P,U), unit(U), unit(P).
(8)  :- unit(U), interUnitCap+1 { partnerunits(U,P): unit(P) }.
```

The first statement generates the required number of units represented as facts: `unit(1). unit(2). ... unit(lower)`. The second and the third clause ensure that

[2] All literals in rules are negation free. $\bot$, $\rightarrow$, $\wedge$, $\vee$ are used to formulate (disjunctive) rules.

each zone and sensor is connected to exactly one unit. The edges between units and zones (rsp. sensors) are expressed by `unit2zone/2` (rsp. `unit2sensor/2`) predicates. We use cardinality constraints [17] of the form $l\ \{L_1, \ldots, L_n\}\ u$ specifying that at least l but at most u literals of $L_1, \ldots, L_n$ must be true. So called *conditions* (expressed by the symbol ":") restrict the instantiation of variables to those values that satisfy the condition. For example, in the second rule, for any instantiation of variable `Z` a collection of ground literals `unit2zone(U,Z)` is generated where the variable `U` is instantiated to all possible values s.t. `unit(U)` is true. In this collection at least one and at most one literal must be true.

The fourth and the fifth rule guarantee that one unit controls at most *UnitCap* zones and *UnitCap* sensors. In these rules the head of the rule is empty which implies a contradiction in case the body of the rule is fulfilled. The last three rules define the connections between units and limit the number of partner units to *InterUnitCap*. Note that rules 4, 5 and 8 can be rephrased by moving the cardinality constraint on the left-hand-side of the rule and adapting the boundaries. We used the depicted encoding because it follows the Guess/Check/Optimize pattern formulated in [12]. Depending on the particular encoding runtimes may vary.

Alternatively, ASP solvers provide built-in support for optimization by restricting the set of answer sets according to an objective function. For example, for minimizing the number of units, the upper bound on the number of units used has to be provided as a constant `upper` $= \max(|Zones|, |Sensors|)$. The unit generation rule of the original program (line 1) then has to be replaced by:

```
(1') unit(1..upper).
(2') unitUsed(U):- unit2zone(U,Z).
(3') unitUsed(U):- unit2sensor(U,S).
(4') lower { unitUsed(X):unit(X) } upper.
(5') unitUsed(X):- unit(X), unit(Y), unitUsed(Y), X<Y.
(6') #minimize[unitUsed(X)].
```

Here, the second and the third rule express the property that a used unit always has to be non-empty. Rule `4'` states that the number of used units must be between `lower` and `upper`. Rule `5'` expresses an ordering on the units: units with smaller numbers should be used first. This statement improves the performance of the solver. The last rule expresses that the optimization criterion is the number of units used in a solution.

3.2 Propositional Satisfiability Testing

We next show how to encode the PUP as a propositional satisfiability problem. We are given sensors $[1, S]$, zones $[1, Z]$, and units $[1, U]$, as well as *UnitCap* and *InterUnitCap*.

Let su_{ij} denote that sensor i is assigned to unit j, and zu_{ij} that zone i is assigned to unit j. First of all, every sensor and zone must belong to a unit, so

$$\forall 1 \leq i \leq S \bigvee_{1 \leq j \leq U} su_{ij} \text{ and } \forall 1 \leq i \leq Z \bigvee_{1 \leq j \leq U} zu_{ij}.$$

Furthermore, every sensor and zone belongs to at most one unit, therefore we have

$$\forall 1 \leq i \leq S. \forall 1 \leq j < j' \leq U. \left(\neg su_{ij} \vee \neg su_{ij'}\right)$$

and the same for zones.

Now we need to count both the number of zones and sensors on a unit, and forbid both numbers to be above *UnitCap*. For this we use a sequential counter, similar to the one presented in [18]. Let sc_{ijk} mean that unit j has k sensors assigned (ignore the i for now). We need to say that every sensor counts as one,

$$\forall 1 \leq i \leq S. \forall 1 \leq j \leq U. \left(su_{ij} \rightarrow sc_{ij1}\right),$$

and also that we increment this number when we see something new:

$$\begin{gathered}\forall 1 \leq i < i' \leq S. \forall 1 \leq j \leq U. \forall 1 \leq k \leq \mathit{UnitCap}. \\ \left(su_{i'j} \wedge sc_{ijk} \rightarrow sc_{i'j(k+1)}\right)\end{gathered}$$

The fact that we keep track of what we have seen (using index i) is to make sure, for example, that sc_{ij5} is only true if there are five distinct sensors on a unit. Finally, we forbid too many sensors on a unit via

$$\forall 1 \leq i \leq S. \forall 1 \leq j \leq U. \neg sc_{ij(\mathit{UnitCap}+1)}.$$

Repeat this trick for zones using zc_{ijk}.

Finally, we need to use the edges. Let sz_{ij} be given, and mean that sensor i has an edge to zone j. Also, let uu_{ij} mean that units i and j are partnered. We need to define this as

$$\begin{gathered}\forall 1 \leq i \leq S. \forall 1 \leq j \leq Z. \forall 1 \leq k < k' \leq U. \\ \left(\left(\left(su_{ik} \wedge zu_{jk'}\right) \vee \left(su_{ik'} \wedge zu_{jk}\right)\right) \wedge sz_{ij} \rightarrow uu_{kk'}\right)\end{gathered}$$

and also, by symmetry,

$$\forall 1 \leq i < j \leq U. \left(uu_{ij} \rightarrow uu_{ji}\right).$$

Now we can count the partnered units like we did before, using pc_{ijk}, and then forbidding $pc_{ij(y+1)}$. Technically, we don't need both uu_{ij} and uu_{ji}, but having both makes the encoding simpler in the definitions above. We may skip uu_{ii} — but we may also leave them in, as the clauses forcing uu_{ij} have $i < j$, and thus uu_{ii} is never forced. Therefore,

$$\forall 1 \leq i \leq j \leq U. \left(uu_{ij} \rightarrow pc_{ij1}\right),$$

and

$$\forall 1 \leq i < i' \leq U. \forall 1 \leq j \leq U. \left(uu_{i'j} \wedge pc_{ijk} \rightarrow pc_{i'j(k+1)}\right).$$

Finally, we forbid too many partners, and we are done:

$$\forall 1 \leq i \leq j \leq U. \neg pc_{ij(\mathit{InterUnitCap}+1)}.$$

3.3 Integer Programming

We next show how the Pup can be encoded into integer programming. If *InterUnitCap* = 2 we set |Units| = max(|Sensors|, |Zones|); otherwise it is |Units| = |Sensors| + |Zones|. Then we make matrices of Boolean variables su_{ij} (and zu_{ij}, respectively) sensor s_i (zone z_i) is assigned to unit u_j. These matrices are constrained to enforce that each sensor/zone is assigned exactly one unit, and that no unit is assigned more than *UnitCap* sensors/zones:

$$\begin{array}{cccc|c}
su_{1,1} & su_{2,1} & su_{3,1} & \dots & \sum \leq \mathit{UnitCap} \\
su_{1,2} & su_{2,2} & \dots & \dots & \sum \leq \mathit{UnitCap} \\
su_{1,3} & \dots & \dots & \dots & \sum \leq \mathit{UnitCap} \\
\dots & \dots & \dots & \dots & \dots \\
\hline
\sum = 1 & \sum = 1 & \dots & \dots &
\end{array}$$

The zone-units matrix looks identical. Next we need a Boolean variable $\mathit{UnitUsed}_i$ that indicates whether u_i is assigned any sensors/zones. This can be achieved by constraints $su_{ji} \leq \mathit{UnitUsed}_i$ and $zu_{ji} \leq \mathit{UnitUsed}_i$, for all j. Observe that in principle even for unused units $\mathit{UnitUsed}_i$ can be set to one — a possibility that will be excluded by the objective function.

For the constraints on the connections between units it is convenient to increase *InterUnitCap* by one, and stipulate that every unit is connected to itself. We then obtain a symmetric matrix of Boolean uu_{ij} variables, which can be used to indicate whether unit i is connected to unit j:

$$\begin{array}{cccc|c}
1 & uu_{1,2} & uu_{1,3} & \dots & \sum \leq \mathit{InterUnitCap} + 1 \\
uu_{2,1} & 1 & \dots & \dots & \sum \leq \mathit{InterUnitCap} + 1 \\
uu_{3,1} & \dots & 1 & \dots & \sum \leq \mathit{InterUnitCap} + 1
\end{array}$$

In addition to enforcing that *InterUnitCap* is not exceeded, the entries in this matrix are subject to the following constraints:

– $uu_{ij} = uu_{ji}$ (symmetry); and
– $uu_{ij} \geq (su_{ki} + zu_{lj}) - 1$, for all connections (s_k, z_l) between sensors and zones — if a sensor s_k and a zone z_l are connected yet assigned different units u_i, u_j then these units are connected.

This model allows more connections between units than are actually needed, in this case mandating a post-processing step for solutions.

As a last constraint we add that the number of units used is bounded from below:

$$\left\lceil \frac{\max(|\text{Sensors}|, |\text{Zones}|)}{2} \right\rceil \leq \sum_j \mathit{UnitUsed}_j .$$

Finally, we add the objective function $\sum_j \mathit{UnitUsed}_j$, subject to minimization. As usual, first a linear relaxation with cost C is solved, and only then is the problem solved over the integers, posting the cost C as a lower bound.

3.4 Constraint Satisfaction Problem

Finally, we model the Pup as a Csp by letting sensors and zones be variables $S = \{s_1, \ldots, s_n\}$ and $Z = \{z_1, \ldots, z_m\}$. For the domains we use (a numbering of) the units $U_1, \ldots, U_n$.

We post a global cardinality constraint $\texttt{gcc(}U_i\texttt{,[}s_1, \ldots, s_n\texttt{],}C\texttt{)}$ on the sensors for every U_i , where C is a variable with domain $\{0, \ldots, \mathit{UnitCap}\}$, and do likewise for the zones. These constraints ensure that each unit occurs at most *UnitCap* times in any assignment to S and Z.

Tracking connections between units via Boolean variables is done using a matrix of Boolean uu_{ij} variables as in the integer programming model, but using cardinality constraints to count the number of ones.

In addition for each connection (s, z) we post implicational constraints that exclude the value j from the domain of sensor s if z is assigned to unit i and $uu_{ij} = 0$ (and vice versa):

$$(s = U_i \wedge uu_{ij} = 0) \rightarrow z \neq U_j \text{ and } (z = U_i \wedge uu_{ij} = 0) \rightarrow s \neq U_j$$

4 A Special Case: *InterUnitCap* = 2

In this section we focus on the case where *InterUnitCap* = 2. We first briefly recall the fundamental ideas of our recent tractability results for this case; for the details the interested reader is referred to [1]. We then show how the fundamental ideas from this work can be incorporated into the Pup encodings presented above.

4.1 A Specialized Algorithm for *InterUnitCap* = 2

The basic observation in the case *InterUnitCap* = 2 is that the unit graph in a solution of a connected Pup instance is always either a path or a cycle. This holds because the number of neighbors of a unit is bounded by two. Based on this observation we have developed a non-deterministic algorithm DecPup that decides the Pup. Basically, DecPup recursively guesses the contents of the units. It turns out that this can be done in NLogSpace by exploiting the notion of a path decomposition; this non-deterministic algorithm can then be turned into a polynomial backtracking search procedure.

Let us now turn to those of the ideas we use for the DecPup algorithm that can be incorporated into the other problem models: We first observe that cyclic unit graphs are more general solution topologies than paths. Any solution that is a path can be extended to a cycle, but the converse is clearly false. Hence, for a fixed number of units used in the solution, we can assume a fixed cyclic layout of the units throughout the search. By using iterative deepening search (on the number of units used) we can find optimal solutions first.

In this context let us point out that branch-and-bound-search for optimal solutions (again on the number of units used) does not work: e.g. a $K_{6,6}$ graph does not admit solutions with more than three units.

Note also that finding optimal solutions gets more complicated if there are multiple connected components in the input graph. DecPup can then still be used to compute optimal solutions in polynomial time — but only if there are at most logarithmically many connected components in the input graph. Part of the problem is that any two connected components may either have to be assigned to the same, or to two distinct unit graph(s). A priori it is unclear which of the two choices leads to better results. E.g. if we assume that *UnitCap* = 2 then two $K_{3,3}$ should be placed on one cyclic unit graph, while two $K_{6,6}$ must stand alone.

Note that with cycles for unit graphs there are two kinds of rotational symmetry: For any given solution with unit graph $U_1, \dots, U_n, U_1$ there also are identical solutions $U_2, \dots, U_n, U_1, U_2$, etc.; in addition, there is also $U_n, U_{n-1}, \dots, U_1, U_n$. We can break this symmetry without additional computational cost by requiring that

– the first sensor is assigned to unit U_1; and
– the second sensor appears somewhere on the first half of the cycle.

We have prototypically implemented the DecPup algorithm in Java (for connected graphs), and will use it below in the evaluation of the other encodings for the case *InterUnitCap* = 2. The implementation features memoization of no-good units to avoid the rediscovery of unsolvable subproblems, and two-step forward-checking: Checking whether there is enough space for the open neighbours of the current unit on the current plus the next unit (step one), and doing the same for the open neighbours of the open neighbours (step two).

4.2 Adapting the Encodings to *InterUnitCap* = 2

To some extent the ideas presented above can be incorporated into the other problem models: If we use iterative deepening search, then we can assume a fixed cyclic layout of the units for each depth. Then, the connections between units are given, something that greatly simplifies the problem models. It also allows us to use symmetry breaking as defined in section 4.1 above. For example, in the constraint model we can drop the Boolean matrix that tracks the connections between units, and simplify the implicational constraints for a connection (s, z) to

$$s = U_i \rightarrow z \in \{U_{i-1}, U_i, U_{i+1}\} \text{ and } z = U_i \rightarrow s \in \{U_{i-1}, U_i, U_{i+1}\}.$$

The adaptations for Asp and Sat are similar [1].

If we are not doing iterative deepening search, that is, the maximum available number of units in the model is given by the upper bound, then this does not work, as it is not clear where to close the cycle. Especially for the integer programming model this constitutes a challenge: If we use iterative deepening we lose the objective function.

To guide the search, we can, by a simple greedy algorithm, compute an ordering of the variables that ensures that each sensor (or zone) has some predecessor that has already been assigned to a unit; we assume that an arbitrary sensor (or zone) is fixed initially. If variables are assigned in this order then the number of possible unit choices per zone (or sensor) is bounded by three throughout the search, instead of *NoOfUnits*. However, to the best of our knowledge neither integer programming tools nor Asp- or Sat-solvers usually provide this level of control over variable ordering to the user.

5 Evaluation

We have evaluated our encodings on a set of benchmark instances that we received from our partners in industry. All experiments were conducted on a 3 GHz dual core machine with 4 GB RAM running Fedora Linux, release 13 (Goddard). In general in our experiments we have imposed a ten minute time limit for finding solutions.

For the evaluation of the different encodings of the Pup we use the Sat-solver MiniSat v2.0 [14], the constraint logic programming language ECL^{i}PS^e-Prolog v6.0 [7], and Clingo v3.0 [3] from the Potsdam Answer Set Solving Collection (Potassco). For evaluating the integer programming model we have used Cbc v2.6.2 in combination with Clp v1.13.2 from the COIN-OR project [4], and IBM's Cplex v12.1 [5].

In the Asp, Sat and Csp models, as well as in DecPup, we use iterative deepening search for finding optimal solutions, as this has proven to be the most efficient. We did not try this in the integer programming model, as we would lose the objective function in doing so.

The reader is advised to digest the results presented below with caution: We are using both the Sat and the integer programming solvers out of the box, whereas for the Csp model we employ the variable ordering heuristics outlined in the previous section. Moreover, if *InterUnitCap* > 2, for the Asp model we employ the following advanced feature: a portfolio solver Claspfolio, which is a part of Potassco [3], comes with a machine learning algorithm (support vector machine) that has been trained on a large set of Asp programs. Claspfolio analyzes a new Asp program (in our case the Pup program), and configures Clingo to run with options that have already proved successful on similar programs. It is likely that such machine learning techniques could also be developed and fruitfully applied in the other frameworks.

5.1 Experimental Results

InterUnitCap $= 2$*.* All instances are based on rectangular floor plans, and all instance graphs are connected. In all instances there is one zone per room, and by default there are sensors on all doors. Only the grid-* and tri-* instances feature external doors. For an illustration see Figure 2, which shows a rectangular 8×3 floor plan with external doors on two sides of the building.

Apart from that, the instances are structured as follows:

– dbl-* consist of two rows of rooms with all interior doors equipped with a sensor.

– dblv-* are the same, only that there are additional zones that cover the columns.
– tri-* are grids with only some of the doors equipped with sensors. There are additional zones that cover multiple rooms.
– grid-* are not full grids, but some doors are missing, and there are no rooms (zones) without doors.

The runtimes we obtained for the various problem encodings described above are shown in seconds in Table 1 (a "*" indicates a timeout). The Cost column contains the number of units in an optimal solution; a slash "/" in that column indicates that no solution exists.

Table 1. Structured Problems with $InterUnitCap = UnitCap = 2$

Name	$\|S\|$	$\|Z\|$	Edges	Cost	Csp	Sat	DecPup	Asp	Cbc	Cplex
dbl-20	28	20	56	14	0.02	0.48	0.01	0.16	14.12	1.53
dbl-40	58	40	116	29	0.28	2.36	0.05	3.93	224.14	13.58
dbl-60	88	60	176	44	0.42	29.74	0.08	*	*	213.58
dbl-80	118	80	236	59	1.14	*	0.16	*	*	522.50
dbl-100	148	100	296	74	1.89	*	0.41	*	*	*
dbl-120	178	120	356	89	3.21	*	0.39	*	*	*
dbl-140	208	140	416	104	5.01	*	0.59	*	*	*
dbl-160	238	160	476	119	13.94	*	0.71	*	*	*
dbl-180	268	180	536	134	20.07	*	0.87	*	*	*
dbl-200	298	200	596	149	14.4	*	1.08	*	*	*
dblv-30	28	30	92	15	0.09	0.42	65.49	0.26	37.18	2.93
dblv-60	58	60	192	30	0.26	3.15	*	1.94	*	*
dblv-90	88	90	292	45	0.82	12.54	*	27.35	*	*
dblv-120	118	120	392	60	1.85	41.65	*	13.92	*	*
dblv-150	148	150	492	75	3.48	20.97	*	29.54	*	*
dblv-180	178	180	592	90	6.20	44.28	*	54.50	*	*
tri-30	40	30	78	20	1.07	0.79	0.50	0.41	45.17	78.75
tri-32	40	32	85	20	0.64	0.74	*	0.26	55.20	4.66
tri-34	40	34	93	/	21.10	22.77	*	0.89	74.78	5.06
tri-60	79	60	156	40	158.49	315.42	114.08	4.40	*	108.01
tri-64	79	64	170	/	*	379.36	*	43.88	*	76.26
grid-90	50	68	97	34	0.04	4.51	0.03	1.53	*	21.19
grid-91	50	63	97	32	0.10	*	*	0.92	*	16.60
grid-92	50	65	97	33	0.49	*	*	0.87	*	17.40
grid-93	50	58	97	29	0.13	2.68	*	1.75	*	13.41
grid-94	50	66	97	33	0.04	3.66	*	1.61	*	*
grid-95	50	60	97	30	0.02	3.90	0.48	0.97	*	18.34
grid-96	50	62	97	31	0.07	3.30	*	0.87	*	13.62
grid-97	50	64	97	32	0.02	3.67	*	0.86	*	17.90
grid-98	50	59	97	30	0.03	*	*	1.19	*	12.30
grid-99	50	65	97	33	0.03	*	202.48	1.16	*	20.35

Table 2. Structured Problems with *InterUnitCap* = 4 and *UnitCap* = 2

Name	$\lvert S\rvert$	$\lvert Z\rvert$	Edges	Cost	CSP	SAT	ASP	CBC	CPLEX
tri-30	40	30	78	20	0.12	2.40	0.40	182.91	24.79
tri-32	40	32	85	20	0.14	1.91	0.66	270.27	20.84
tri-34	40	34	93	20	*	1.98	0.60	331.29	*
tri-60	79	60	156	40	0.52	*	11.07	*	*
tri-64	79	64	170	40	*	*	7.61	*	*
tri-90	118	90	234	59	1.50	401.44	332.34	*	*
tri-120	157	120	312	79	3.37	*	*	*	*
grid-1	100	79	194	50	*	78.19	31.45	*	*
grid-2	100	77	194	50	*	90.89	18.91	*	*
grid-3	100	78	194	50	*	88.87	25.72	*	*
grid-4	100	80	194	50	*	95.12	24.66	*	*
grid-5	100	76	194	50	*	454.42	48.88	*	*
grid-6	100	78	194	50	*	204.85	9.15	*	*
grid-7	100	79	194	50	*	112.36	12.89	*	*
grid-8	100	78	194	50	*	*	11.89	*	*
grid-9	100	76	194	50	*	91.62	19.71	*	*
grid-10	100	80	194	50	*	545.16	13.54	*	*

InterUnitCap > 2. For the general case we have also tested our encodings on a set of benchmark instances where *InterUnitCap* = 4 that we obtained from our partners in industry:

– tri-* are exactly as before, only with *InterUnitCap* = 4.
– grid-* are as before, only that a bigger number of doors exists.

5.2 Analysis

Any conclusions drawn from our experimental results have to be qualified by the remark that, of course, in every solution framework there are many different problem models, and there is no guarantee that our problem models are the best ones possible.

Let us begin our analysis of the results by highlighting a peculiarity of the PUP: While it is possible to construct instances that require more than the minimum number of units, it is not straight-forward to do so, and such instances also appear to be rare in practice: In our experiments in no solution are there more units than the bare minimum required. It is clear that iterative deepening search thrives on this fact, whereas the integer programming model suffers.

InterUnitCap = 2. The combination of assuming a fixed cyclic unit graph together with iterative deepening search resulted in drastic speedups for the ASP, SAT, and CSP solvers. Symmetry breaking did not have much effect — except on the unsolvable instances.

The ASP and the SAT encoding show broadly similar behavior: Both CLINGO and MINISAT use variations of the DPLL-procedure [6] for reasoning. Oddly, they even both get faster at some point as problem size increases on the dblv-* instances. However, Clingo does significantly better on the grid-like instances. Interestingly, machine learning did not help for the Asp encoding specialized to *InterUnitCap* $= 2$; hence the results shown were obtained using both solvers out of the box.

For the Csp encoding the variable ordering is the key to the good results: Without the variable ordering the Csp model performs quite poorly. The absence of a similar variable selection mechanism from both Asp and Sat in our experiments might explain the surprising superiority of Csp on most benchmarks.

The inconsistent results for DecPup are particularly striking. On the one hand, DecPup performs excellently on the dbl-* instances. But in general, it disappoints. Possibly this might be due to the following: DecPup has a "local" perspective on the problem, that is, it only can see the current and past units; the subsequent units are only created at runtime. In all the other encodings all units are present from the beginning, something which, in one way or another, facilitates propagating the current variable assignment to other units.

The Ip encoding is not yet fully competitive. It particularly struggles with the dblv-* instances. In general, the commercial Cplex is at least one order of magnitude faster than the open source Cbc.

It is also interesting to compare the dbl-* with the dblv-* instances, as the latter are obtained from the former by adding constraints. Both Clingo and MiniSat thrive on the additional constraints, contrary to ECL^{i}PS^e, Cbc, Cplex and DecPup.

InterUnitCap > 2*.* In this setting, for finding solutions the symmetry breaking methods from section 2.6 did increase computation time for the Csp, the Sat, and the Ip model. However, symmetry breaking again does help when proving an instance unsatisfiable. The results in Table 2 were obtained without symmetry breaking.

If Claspfolio's machine learning database is not used to configure options of Clingo, then the two DPLL-based programs again perform quite similar, with Clingo slightly having the edge (results not shown). With machine learning Clingo clearly is the winner, with the main benefits stemming from the following: Use the VSIDS heuristics [15] instead of the BerkMin heuristics [10], and exploit local restarts [16]. Note that MiniSat also uses the VSIDS heuristics.

Interestingly, the Csp-encoding now disappoints. Given that the same variable ordering is used, this may have to be attributed to insufficient propagation when tracking the connections between units.

Again our Ip encoding is not on par yet. But for this encoding comparing the instances tri-30,32,34 in Tables 1 and 2 is particularly instructive: This is basically the same model in both settings, only that in the latter case there are more variables due to the higher upper bound on the number of required units.

6 Future Work

There is still significant work to be done on the Pup: Almost all interesting complexity questions are still open, and a thorough investigation of these questions should eventually lead to better algorithms and encodings for the Pup. It should also be possible to prove better upper bounds, in particular ones that depend on *UnitCap*; especially the integer programming model would benefit from this. It would be interesting to see what the variable ordering heuristics can do for Sat and Asp. More generally, the major challenge is to find stronger problem models in the various frameworks and to improve the implementation of DecPup, the only algorithm guaranteed to run in polynomial time.

Acknowledgment. We greatly appreciate the helpful comments from the anonymous reviewers.

References

1. Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A., Thorstensen, E.: Tackling the Partner Units Problem. Tech. Rep. RR-10-28, Computing Laboratory, University of Oxford (2010), available from the authors
2. Third International Answer Set Programming Competition 2011 (2011), `https://www.mat.unical.it/aspcomp2011/`
3. The Potsdam Answer Set Solving Collection, `http://potassco.sourceforge.net/`
4. COIN-OR CLP/CBC IP solver, `http://www.coin-or.org/`
5. IBM ILOG CPLEX IP solver, `http://www.ibm.com/`
6. Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3) (1960)
7. ECL^{i}PS^e-Prolog, `http://eclipseclp.org/`
8. Falkner, A., Haselböck, A., Schenner, G.: Modeling Technical Product Configuration Problems. In: Proceedings of the Configuration Workshop at ECAI 2010 (2010)
9. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of ICLP 1988 (1988)
10. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. In: Proceedings of DATE 2002 (2002)
11. Hooker, J.N.: Integrated Methods for Optimization. Springer, New York (2006)
12. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic 7(3) (2006)
13. Meringer, M.: Regular Graphs Page, `http://www.mathe2.uni-bayreuth.de/markus/reggraphs.html`
14. Minisat SAT solver, `http://www.minisat.se`
15. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of DAC 2001 (2001)
16. Ryvchin, V., Strichman, O.: Local restarts. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 271–276. Springer, Heidelberg (2008)
17. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence 138(1-2) (2002)
18. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)

Manipulating MDD Relaxations for Combinatorial Optimization

David Bergman, Willem-Jan van Hoeve, and John N. Hooker

Tepper School of Business, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh, PA 15213, U.S.A.
{dbergman,vanhoeve}@andrew.cmu.edu, john@hooker.tepper.cmu.edu

Abstract. We study the application of limited-width MDDs (multi-valued decision diagrams) as discrete relaxations for combinatorial optimization problems. These relaxations are used for the purpose of generating lower bounds. We introduce a new compilation method for constructing such MDDs, as well as algorithms that manipulate the MDDs to obtain stronger relaxations and hence provide stronger lower bounds. We apply our methodology to set covering problems, and evaluate the strength of MDD relaxations to relaxations based on linear programming. Our experimental results indicate that the MDD relaxation is particularly effective on structured problems, being able to outperform state-of-the-art integer programming technology by several orders of magnitude.

1 Introduction

Binary Decision Diagrams (BDDs) [1, 19, 6] provide compact graphical representations of Boolean functions, and have traditionally been used for circuit design and formal verification [17, 19]. More recently, however, BDDs and their generalization Multivalued Decision Diagrams (MDDs) [18] have been used in Operations Research for a variety of purposes, including cut generation [3], vertex enumeration [5], and post-optimality analysis [12, 13].

In this paper, we examine the use of BDDs and MDDs as relaxations for combinatorial optimization problems. Relaxation MDDs were introduced in [2] as a replacement for the domain store relaxation, i.e., the Cartesian product of the variable domains, that is typically used in Constraint Programming (CP). MDDs provide a richer data structure that can capture a tighter relaxation of the feasible set of solutions, as compared with the domain store relaxation. In order to make this approach scalable, MDD relaxations of limited size are applied. Various methods for compiling these discrete relaxations are provided in [14]. The methods described in that paper focus on iterative splitting and edge filtering algorithms that are used to tighten the relaxations. Similar to classical domain propagation, such MDD propagation algorithms have been developed for individual (global) constraints, including inequality constraints, equality constraints, *alldifferent* constraints and *among* constraints [14, 15].

The focus of the current work is the application of limited-width MDD relaxations in the context of optimization problems. We explore two main topics. Firstly, we investigate a new method for building approximate MDDs. We introduce a top-down compilation method based on approximating the set of completions of partially assigned solutions. This procedure differs substantially from the ideas in [2] in that we do not compile the relaxation by splitting vertices, but by merging vertices when the size of the partially constructed MDD grows too large.

Secondly, and more specific to optimization, we introduce a method to improve the lower bound provided by an MDD relaxation. It is somewhat parallel to a cutting plane algorithm in that it "cuts off" infeasible solutions so as to tighten the bound. Unlike cutting planes, however, it can begin with any valid lower bound, perhaps obtained by another method, and tighten it. The bound becomes tighter as more time is invested.

The resulting mechanism is a pure inference algorithm that can be used analogously to a pure cutting plane algorithm. We envision, however, that MDD relaxations would be most profitably used as a bounding technique in conjunction with a branch-and-bound search, much as separation algorithms are used in integer programming. Nonetheless we find in this paper that, even as a pure inference algorithm, MDD relaxation can outperform state-of-the-art integer programming technology on specially structured instances.

One advantage of an MDD relaxation is that it is always easy to solve (as a shortest path problem) whether the original problem is linear or nonlinear. This suggests that MDDs might be most competitive on nonlinear discrete problems. Nonetheless we deliberately put MDDs at a competitive disadvantage by applying them to a problem with linear inequality constraints—namely, to the set covering problem, which is well suited to integer programming methods.

We compare the strength of bounds provided by MDDs with those provided by the linear programming relaxation and cutting planes. We also compare the speed with which MDDs (used as a pure inference method) and integer programming solve the problem. We find that MDDs are much superior to conventional integer programming when the ones in the constraint matrix lie in a relatively narrow band. That is, the matrix has relatively small bandwidth, meaning that the maximum distance between any two ones in the same row is limited.

The bandwidth of a set covering matrix can often be reduced, perhaps significantly, by reordering the columns. Thus MDDs can solve a given set covering problem much more rapidly than integer programming if its variables can be permuted to result in a relatively narrow bandwidth. Algorithms and heuristics for minimum bandwidth ordering are discussed in [20, 7–9, 11, 21–23].

The remainder of the paper is organized as follows. In Section 2 we define MDDs more formally and introduce notation. In Section 3 we describe a new top-down compilation method for creating relaxation MDDs. In Section 4 we present our value enumeration scheme to produce lower bounds. In Section 5 we discuss applying the ideas of the paper to set covering problems. In Section 6 we report on experiments results where we apply the ideas of the paper to set covering problems. We conclude in Section 7.

2 Preliminaries

In this work a *Multivalued Decision Diagram* (MDD) is a layered directed acyclic multi-graph whose nodes are arranged in $n+1$ layers, $L_1, L_2, \ldots, L_{n+1}$. Layers L_1 and L_{n+1} consist of single nodes; the root r and the terminal t, respectively. All arcs in the MDD are directed from nodes in layer j to nodes in layer $j+1$.

In the context of Constraint Satisfaction Problems (CSPs) or Constraint Optimization Problems (COPs), we use MDDs to represent assignments of values to variables. A CSP is specified by a set of constraints $C = \{C_1, C_2, \ldots C_m\}$ on a set of variables $X = \{x_1, x_2, \ldots, x_n\}$ with respective finite domains $D_1, \ldots, D_n$, and a COP is specified by a CSP together with an objective function f to be minimized. By a *solution* to a CSP (COP) we mean an assignment of values to variables where the values assigned to the variables appear in their respective domains. By a *feasible solution* we mean a solution that satisfies each of the constraints in C, and the *feasible set* is the set of all feasible solutions. For a COP, an *optimal* solution is a feasible solution x^* such that for any other feasible solution $\tilde{x}, f(x^*) \leq f(\tilde{x})$.

We use MDDs to represent a set of solutions to a CSP, or COP, as follows. We let the layers $L_1, \ldots, L_n$ correspond to the problem variables $x_1, \ldots, x_n$, respectively. Node $u \in L_j$ has label $\text{var}(u) = j$, representing its variable index. Arc (u, v) with $\text{var}(u) = j$ is labeled with *arc domain* $d_{u,v}$, by an element of the domain of variable x_j, i.e., $d_{u,v} \in D_j$. All arcs directed out of a node must have distinct labels.

A path p from node u_i to node u_k, $i < k$, along arcs $a_i, a_{i+1}, \ldots, a_{k-1}$ corresponds to the assignment of the values d_{a_j} to the variables x_j, for $j = i, i+1, \ldots, k-1$. In particular, we see that any path from the root r to the terminal $t, p = (a_1, \ldots, a_n)$, corresponds to the solution x^p, where $x^p_j = d_{a_j}$. We note that as an MDD is a multi-graph, two paths p_1, p_2, along nodes $r = u_1, \ldots, u_n, t$ may correspond to multiple solutions as there may be multiple arcs from u_j to u_{j+1} corresponding to different assignments of values to the variable x_j.

The set of solutions represented by MDD M is $\text{Sol}(M) = \{x^p | p \in P\}$ where P is the set of paths from r to t. The *width* of layer L_j is given by $\omega_j = |L_j|$, and the *width* of MDD M is given by $\omega(M) = \max_{j \in \{1,2,\ldots,n\}} \omega_j$. The *size* of M is denoted by $|M|$, the number of nodes in M.

For a given CSP $\mathcal{P}$, let $X(\mathcal{P})$ be the set of feasible solutions for $\mathcal{P}$. An *exact* MDD M for $\mathcal{P}$ is any MDD for which $\text{Sol}(M) = X(\mathcal{P})$. A *relaxation* MDD M_{rel} for $\mathcal{P}$ is any MDD for which $\text{Sol}(M_{\text{rel}}) \supseteq X(\mathcal{P})$. For the purposes of this paper, relaxation MDDs are of limited width, in that we require that $\omega_j \leq W$, for some predefined W. This ensures that the relaxation has limited size which is necessary since even for single constrained problems, the feasible set may correspond to an MDD of exponential size (for example inequality constrained problems [4]).

Finally, we note that for a large class of objective functions (e.g., for separable functions), optimizing over the solutions represented by an MDD corresponds to finding a shortest path in the MDD [2]. For example, given a linear objective function $\min cx$, we associate with each arc (u, v) in the MDD a *cost* $c(u, v)$,

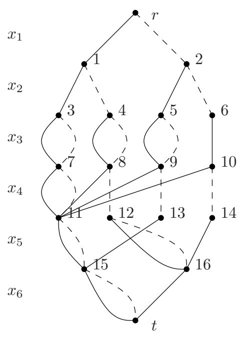


Fig. 1. Exact MDD for Example 1

where $c(u,v) = c_{\text{var}(u)} \cdot d_{u,v}$. Then it is clear that a shortest path from r to t corresponds to the lowest cost solution represented by the MDD.

Example 1. As an illustration, consider the CSP consisting of binary variables $x_1, x_2, \ldots, x_6$, and constraints

$$\begin{aligned} C_1: &\quad x_1 + x_2 + x_3 \geq 1, \\ C_2: &\quad x_1 + x_4 + x_5 \geq 1, \\ C_3: &\quad x_2 + x_4 + x_6 \geq 1. \end{aligned}$$

An exact MDD representation of the feasible set is given in Fig. 1, where arc (u,v) being solid/dashed corresponds to the arc setting var(u) to 1/0.

3 Top-Down MDD Compilation

As discussed above, there are several methods that can be used to construct both exact and approximate MDDs. In this section we propose a new top-down method for creating approximate MDDs.

3.1 Exact Top-Down Compilation

We first discuss an exact top-down compilation method, which is based on the notion of *node equivalence.*

Given a path p from r to u, let $F(p)$ be the set of feasible completions of the corresponding partial assignment. That is, if $(x_1, \ldots, x_k) = (d_1, \cdots, d_k) = d$ is the partial assignment represented by p, then $F(p) = \{y \in D_{k+1} \times \cdots \times D_n | (d,y) \text{ is feasible } \}$. We say that two paths p, p' from r to the same layer are *equivalent* if $F(p) = F(p')$.

Analogously, we define $F(u)$ to be the set of completions at node u, so that $F(u) = \bigcup_{p \in P} F(p)$, where P is the set of paths from r to u. We note that in an exact MDD all paths terminating at a node u are equivalent.

A *node equivalence test* determines when two nodes u, u' on the same layer have the same set of feasible completions. In other words, this test determines when $F(u) = F(u')$. Testing whether two nodes have the same set of feasible completions requires maintaining a *state* I_u at each node [15]. The state of node u should contain all facts about the paths ending at u to run an equivalence test. In addition, it is useful to know when a partial assignment cannot be completed to a feasible solution for a CSP. In such a case, we let the state of such a path, or more generally a node, be $\hat{0}$, to signal that there are no completions of this path/node.

Now, using a properly defined node equivalence test, one can create an exact MDD using Algorithm 1. Given that layers $L_1, \ldots, L_j$ have been created, we examine the nodes in L_j one by one. When examining node u, for each domain value $d \in D_j$ we calculate the new state I_{new} that results from adding $x_j = d$ to the partial paths ending at u. If no other nodes on layer L_{j+1} have the same state (i.e. the same set of feasible completions) we add a new node v to L_{j+1} and the arc (u, v) with arc domain d, and set $I_v = I_{\text{new}}$. If however there is some node $w \in L_{j+1}$ with $I_w = I_{new}$ we know that all paths starting at r, ending at u and having $x_j = d$ will have the same set of feasible completions as w. Therefore, we simply add the arc (u, w) with arc domain d.

Algorithm 1. Top-Down MDD Compilation

```
1: $L_1 = \{r\}$
2: for $j = 1$ to $n$ do
3:   $L_{j+1} = \emptyset$
4:   for all $u \in L_j$ do
5:     for all $d \in D(x_j)$ do
6:       calculate $I_{\text{new}}$, the state for all paths starting at $r$, ending at $u$, and including
         $x_j = d$
7:       if $I_{\text{new}} \neq \hat{0}$ then
8:         if there exists $w \in L_{j+1}$ with $I_w = I_{\text{new}}$ then
9:           add arc $(u, w)$ with $d_{u,w} = d$
10:        else
11:          add node $v$ to $L_{j+1}$
12:          add arc $(u, v)$ with $d_{u,v} = d$
13:          set $I_v = I_{\text{new}}$
14:        end if
15:      end if
16:    end for
17:  end for
18: end for
```

We will be modifying Algorithm 1 later to create approximate MDDs. First, however, we discuss specific exact MDDs for the feasible set *satisfying a single equality constraint.* Such MDDs will be applied in our value enumeration method for tightening lower bounds, presented in Section 4.

Lemma 1. *Let $\mathcal{P}$ be a CSP on n binary variables with the single constraint $\sum_{j=1}^{n} c_j x_j = c$, for a given integer c, and integer coefficients $c_j \geq 0$. An exact MDD for $\mathcal{P}$ has maximum width $c + 1$.*

Algorithm 2. Top-Down Relaxation Compilation

```
1: while $|S_{j+1}| > W$ do
2:     select nodes $u_1, u_2 \in S_{j+1}$
3:     create node $u$
4:     for every arc directed at $u_1$ or $u_2$ redirect arc to $u$ with the same arc domain
5:     $I(u) = I(u_1) \oplus I(u_2)$
6:     $S_{j+1} \leftarrow S_{j+1} \backslash \{u_1, u_2\} \cup \{u\}$
7: end while
```

Proof. We apply Algorithm 1. Given a node u, let p be any path from r to u, and let $a_1, \ldots, a_k$ be the arcs along this path, which set variables $x_1, \ldots, x_k$ to the arc domain values $d_{a_1}, \ldots, d_{a_k}$. We define $I_u = \sum_{j=1}^{k} c_j \cdot d_{a_j}$. Using this label as the state of node u we see that two nodes u and v have the same set of feasible completions if and only if $I_u = I_v$. In addition, if $I_w \geq c + 1$ for some node w, it is clear that all paths from r to w have no feasible completions. Therefore we can have at most $c + 1$ nodes on any layer. □

We note that Lemma 1 is very similar to the classical pseudo-polynomial characterization of knapsack constraints.

3.2 Approximate Top-Down Compilation

In general, an exact MDD representation of all feasible solutions to a CSP may be of exponential size, and therefore generating exact MDDs for combinatorial optimization problems is not practical. In light of this we use relaxation MDDs to approximate the set of feasible solutions. In this section we outline one possible method for generating approximate MDDs, by modifying Algorithm 1.

In order to create a relaxation MDD we merge nodes during the top-down compilation method presented in Algorithm 1 when the width of layer j exceeds a certain preset maximum allotted width W. To accomplish this, we select two nodes and modify their states in a relaxed fashion, ensuring that all feasible solutions will remain in the MDD when it is completed. More formally, if we select nodes u_1 and u_2 to merge, we need to modify their states I_{u_1}, I_{u_2} in such a way as to make them equivalent with respect to the equivalence test used to merge nodes during the top-down compilation. We define a certain *relaxation operation* $\oplus$ on the state of nodes as follows.[1] If for nodes u_1 and u_2 we change their associated states to $I(u_1) \oplus I(u_2)$, any feasible completion of the paths from the root to u_1 and u_2 will remain when the terminal is reached. This is outlined in Algorithm 2, which is to be inserted between lines 17 and 18 in Algorithm 1. In Section 5 we describe such an operation in detail, for set covering problems.

The quality of the relaxation MDD generated using the modification of Algorithm 1 hinges largely on the method used for selecting two nodes to combine. We propose several heuristics for this choice in the following table:

[1] Here we follow the notation $\oplus$ that was used in [15] for their analogous operation for aggregating node information.

Name	Node selection method
H_1	select u_1, u_2 uniformly at random among all pairs in S_{j+1}
H_2	select u_1, u_2 such that $f(u_1), f(u_2) \geq f(v), \forall v \in S_{j+1}, v \neq u_1, u_2$
H_3	select u_1, u_2 such that I_{u_1} and I_{u_2} are *closest* among all pairs in S_{j+1}

The rationale behind each of the methods are the following. Method H_1 calls for randomly choosing which nodes to combine. Randomness often helps in combinatorial optimization and applying it in this context may work as well. H_2 combines nodes that have the greatest shortest path lengths. For this we let $f(u)$ be the shortest path length from the root to u in the partially constructed MDD. Choosing such a pair of nodes allows for approximating the set of feasible solutions in parts of the MDD where the optimal solution is unlikely to lie, and retaining the exact paths in sections of the MDD where the optimal solution is likely to lie. H_3 combines nodes that have similar states. For particular types of states and equivalence tests, we must determine the notion of *closest*. This method is sensible because these nodes will most likely have similar sets of completions, allowing the relaxation to better capture the set of feasible solutions.

4 Value Enumeration

We next discuss the application of MDD relaxations for obtaining lower bounds on the objective function, in the context of COPs. We propose to obtain and strengthen these bounds by means of successive value enumeration. Value enumeration is a method that can be used to increase any lower bound on a COP via a relaxation MDD.

Suppose we have generated a relaxation MDD M_{rel}. We then generate an MDD representing every solution in M_{rel} with objective function value equal to the best lower bound. There are several ways to accomplish this, but in general this MDD can have exponential size. However, for some important cases the MDD representing every solution equal to a particular value has polynomial size.

For example, suppose we have a COP with objective function equal to the sum of the variables, i.e., $f(x) = \sum_{j=1}^{n} x_j$, where we assume that the variable domains are integral. Given a lower bound z_{LB}, the reduced MDD for the set of solutions with objective value equal to z_{LB}, $M_{z_{LB}}$, has width $\omega(M_{z_{LB}}) = z_{LB}+1$, by Lemma 1. The same holds for other linear objective functions as well.

In any case, suppose we have the desired MDD $M_{z_{LB}}$, where $\mathrm{Sol}(M_{z_{LB}})$ is the set of all solutions with objective value equal to z_{LB}. Now, consider the set of solutions $S = \mathrm{Sol}(M_{z_{LB}}) \cap \mathrm{Sol}(M_{\mathrm{rel}})$. As this is the intersection between the solutions represented by the relaxation and every solution equal to the lower bound z_{LB}, showing that there is no feasible solution in S allows us to increase the lower bound.

Constructing an MDD M representing the set of solutions $S = \mathrm{Sol}(M_{z_{LB}}) \cap \mathrm{Sol}(M_{\mathrm{rel}})$ can be done in time $\mathcal{O}(|M_{z_{LB}}|\cdot|M_{\mathrm{rel}}|)$ and has maximum width $\omega(\tilde{M}) \leq \omega(M_{z_{LB}}) \cdot \omega(M_{\mathrm{rel}})$ [6]. As the width of $M_{z_{LB}}$ has polynomial size for certain objective functions and the width of M_{rel} is bounded by some preset W, the width of the resulting MDD will not grow too large in these cases.

The value enumeration scheme proceeds by enumerating all of the solutions in M. If we find a feasible solution, we have found a witness for our lower bounds. Otherwise, we can increase the lower bound by 1. Of course, this method is only practical if we can enumerate these paths efficiently.

Observe that we do not need to start the value enumeration scheme with the value of the shortest path in the original MDD. In fact, we can start with any lower bound. For example, we can use LP to find a strong lower bound and then apply this procedure to any relaxation MDD.

As described above, in order to increase the bound, we are required to certify that none of the paths in M correspond to feasible solutions. Of course this can be done by a naive enumeration of all of the paths in M. However, we use MDD-based CP, as described in [2], in unison with a branching procedure to certify this. In particular we apply MDD filtering algorithms to reduce the size of the MDD $M_{z_{LB}}$, based on the constraints that constitute the COP. In Section 5.3 we will describe a new MDD filtering algorithm that we apply to set covering problems.

5 Application to Set Covering

In this section we describe how to apply the ideas of the paper to set covering problems. We describe a node equivalence test and the state that is necessary to carry out the test. We also describe the operation $\oplus$ that can be used to change the states of the nodes so that we can generate relaxation MDDs.

5.1 Equivalence Test

The well-studied set covering problem is a COP with n binary variables and m constraints, each on a subset C_i of the variables, which require that $\sum_{j \in C_i} x_j \geq 1, i \in \{1, \ldots, m\}$. The objective is to minimize the sum of the variables (or a weighted sum).

The first step in applying our top-down compilation method is defining an equivalence test between partial assignments of values to variables. For set covering problems we do this by equating a set covering instance with its equivalent logic formula. Each constraint C_i can be viewed as a clause $\vee_{j \in C_i} x_j$ and the set covering problem is equivalent to satisfying $F = \bigwedge_i \vee_{j \in C_i} x_j$.

Using this interpretation of set covering problems, one can develop a complete equivalence test by removing clauses that are implied by other clauses. Clause C *absorbs* clause D if all of the literals of C are contained in D. In such a case, satisfying clause C implies that clause D will be satisfied. As an example, consider the two clauses $C = (x_1 \vee x_2)$ and $D = (x_1 \vee x_2 \vee x_3)$. It is clear that if some literal in C is set to `true` then clause D will be satisfied.

Therefore, to develop the equivalence test, for any partial assignment x we delete any clause C_i for which there exists a variable in the clause that is already

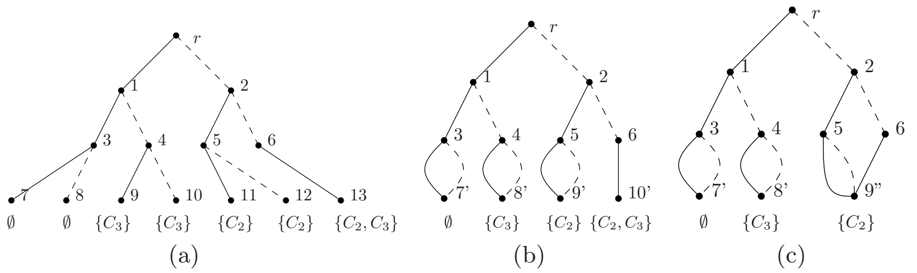


Fig. 2. (a) Exact MDD before combining nodes with the same state, (b) Exact MDD after combining nodes with the same state, (c) MDD after merging node 9' and 10' into 9", making a partially constructed relaxation

set to 1, and then delete all absorbed clauses, resulting in the logical formula $F(x)$. We let I_x be the set of clauses which remain in $F(x)$. Doing so ensures that two partial assignments x^1 and x^2, will have the same set of feasible completions if and only if $I_{x^1} = I_{x^2}$. Note that since each literal is positive in all clauses of a set covering instance, this test can be performed in polynomial time [16].

To create an exact MDD for a set covering instance (using Algorithm 1), we let the state I_u at node u be equal to I_x for the partial assignment given by the arc domains on all paths from the root to u. Two nodes u and v will have the same set of feasible completions if and only if $I_u = I_v$, and so the node equivalence test simply compares I_u with I_v.

Example 2. Continuing Example 1, we interpret the constraints C_1, C_2, and C_3 as set covering constraints. In Fig. 2(a) we see the result of applying the top-down compilation algorithm (following the variable order $x_1, x_2, \ldots, x_6$) and never combining nodes based on their associated states , for the first three layers of the MDD. Below the bottom nodes, we depict the states of the partially constructed paths ending at those nodes. For example, along this path $(r, 2, 5, 11)$, variables x_2 and x_3 are set to 1. Therefore, constraints C_1 and C_3 are satisfied for any possible completion of this path, and so the state at node 11 is C_2. Since node 11 and node 12 have the same state, we can combine these nodes into node 9', as shown in Fig. 2(b). Similarly, nodes 7 and 8 are combined into node 7' and nodes 9 and 10 are combined into node 8'.

5.2 Relaxation Operation

We next discuss our relaxation operator $\oplus$ that is applied to merge two nodes in a layer. For set covering problems, we let $\oplus$ represent the typical set intersection. As an illustration, for the instance in Example 2, suppose we decided that we want to decrease the width of layer 4 by 1. We would select two nodes (in Fig. 2(b) we select nodes 9' and 10') and combine them (making node 9" as seen in Fig. 2(c)), modifying their states to ensure that all feasible paths remain upon completing the MDD. Notice that by taking the intersection of the states of the nodes 9' and 10' we now label 9" with C_2. Before merging the nodes, all partial paths ending at node 10' needed a variable in both constraint C_2 and C_3 to be set to 1. After taking the intersection, we are relaxing this condition, and only require that for all partial paths ending at 9", all completions of this path will set some variable in constraint C_2 to 1, ignoring that this needs to also happen for constraint C_3.

5.3 Filtering

As discussed above, during the value enumeration procedure, it is desirable to perform some MDD filtering to decrease the search space. This filtering can be applied to arc domain values, as described in [2], but also to the states represented in the nodes themselves, as we will describe here in the context of set covering problems.

We associate two 0/1 m-dimensional state variables, $s(v), z(v)$, to each node v in the MDD. The value $s(v)_i$ will be 1 if for all paths from the root to v, there is no variable in constraint C_i which is set to 1. Similarly, $z(v)_i$ will be 1 if for all paths from v to the terminal, there is no variable in C_i which is set to 1.

Finding the values $s(v)_i, z(v)_i$ is easily accomplished by the following simple algorithm. Start with $s(r)_i = 1$ for all i. Now, let node v have parents $u_1, u_2, \ldots, u_k$, and each edge (u_p, u) fixes variable x_j to value $v_p \in \{0, 1\}$. Then,

$$s(v)_i = \prod_{p=1}^{k} s'(u_p)_i,$$

where

$$s'(u_p)_i = \begin{cases} 0 & \text{if } x_j \in C_i \text{ and } v_p = 1 \\ s(u_p)_i & \text{otherwise} \end{cases}$$

The values $z(v)_i$ are calculated in the same fashion, except switching the direction of all arcs in the MDD and starting with $z(t)_i = 1$, where t is the terminal of the MDD.

A node v can now be eliminated whenever there is an index i such that $s(v)_i = z(v)_i = 1$. This is because for all paths from r to v there is no variable in C_i set to 1, and on all paths from v to t, there is no variable in C_i set to 1.

We note here that as in domain store filtering, certain propagators for MDDs are *idempotent*, in that reapplying the filtering algorithm with no additional changes results in no more filtering. The filtering algorithm presented here is not idempotent, i.e, applying it multiple times could result in additional filtering. In our computational experiments we address how this impacts the efficiency of the overall method.

6 Experimental Results

In this section, we present experimental results on randomly generated set covering instances. Our results provide evidence that relaxations based on MDDs perform well when the constraint matrix of a set covering instance has a small bandwidth. We test this by generating random set covering instances with varying bandwidths and comparing solution times via pure-IP (using CPLEX), pure-MDD, and a hybrid MDD-IP method.

In all of the reported results, unless specified otherwise, we apply our MDD-based algorithm until it finds a feasible solution. That is, we solve these set covering problems by continuously improving the relaxation through our value enumeration scheme until we find a feasible (optimum) solution.

6.1 Bandwidth and the Minimum Bandwidth Problem

The *bandwidth* of a matrix A is defined as

$$b_w(A) = \max_{i \in \{1,2,\ldots,m\}} \{ \max_{j,k: a_{i,j}, a_{i,k}=1} \{j - k\}\}.$$

The bandwidth represents the largest distance, in the variable ordering given by the constraint matrix, between any two variables that share a constraint. The smaller the bandwidth, the more structured the problem, in that the variables participating in common constraints are close to each other in the ordering. The *minimum bandwidth problem* seeks to find a variable ordering that minimizes the bandwidth [20, 7–9, 11, 21–23]. This underlying structure, when present in A, can be captured by MDDs and results in good computational performance.

6.2 Problem Generation

To test the statement that MDD based relaxations provide strong relaxations for structured problems, we generate set covering instances with a fixed constraint matrix density d (the number of ones in the matrix divided by $n \cdot m$) and vary the bandwidth b_w of the constraint matrix.

We generate random instances with a fixed number of variables n, constraint matrix density d, and bandwidth b_w, where each row i has exactly $k = d \cdot n$ ones. For constraint i the k ones are chosen uniformly at random from variables $x_{i+1}, x_{i+2}, \ldots, x_{i+b_w}$ As an example, a constraint matrix with $n = 9, d = \frac{1}{3}$ and $b_w = 4$ may look like

$$A = \begin{pmatrix} 1\,1\,0\,1\,0\,0\,0\,0\,0 \\ 0\,1\,1\,1\,0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,1\,1\,0\,0\,0 \\ 0\,0\,0\,1\,0\,1\,1\,0\,0 \\ 0\,0\,0\,0\,1\,0\,1\,1\,0 \\ 0\,0\,0\,0\,0\,0\,1\,1\,1 \end{pmatrix}$$

As b_w grows, the underlying staircase-like structure of the instances dissolves. Hence, by increasing b_w, we are able to test the impact of the structure in the set covering instances on our MDD-based approach.

Consider the case when $b_w = k$. For such problems, as A is totally unimodular [10], the LP optimal solution will be integral, and so the corresponding IP will solve the problem at the root node. Similarly, we show here that the set of feasible solutions can be exactly represented by an MDD with width bounded by $m+1$. In particular, for any node u created during the top-down compilation method, I_u must be of the form $(0, 0, \ldots, 0, 1, 1, \ldots, 1)$. This is because, given any partial assignment fixing the top j variables, if some variable in constraint C_i is fixed to 1, then for any constraint C_k, with $k \leq i$, there must be some variable also fixed to 1. Hence, $\omega(M) \leq m+1$. Therefore, such problems are also easily handled by MDD-based methods. Increasing the bandwidth, however, destroys the totally unimodular property of A and the bounded width of M. Therefore, increasing the bandwidth allows us to test how sensitive the LP and the relaxation MDDs are to changes in the structure of A.

6.3 Evaluating the MDD Parameters

In Section 3.2 we presented three possible heuristics for selecting nodes to merge. In preliminary computational tests, we found that using the heuristic based on shortest partial path lengths, H_2, seemed to provide the strongest MDD relaxations, and so we employ this heuristic.

The next parameter that must be fixed is the preset maximum width W that we allow for the MDD relaxations. Each problem (and even more broadly for each application of MDD relaxations to CSP/COPs) has a different optimal width. To test for an appropriate width for this class of problems, we generate 100 instances with $n = 100, k = 20$ and $b_w = 35$.

In Figure 3(a) we report the average solution time, over the 100 instances, for different maximum allowed widths W. Near $W = 35$ we see the fastest solution times, and hence for the remainder of the experimental testing we fix W at 35. We note here that during our preliminary computational tests, the range of widths that seemed to perform best was $W \in [20, 40]$.

Another parameter of interest is the number of times we allow the filtering algorithm to run before branching. As discussed in Section 5.3 the filtering algorithm presented above for set covering problems is not idempotent and applying the filtering once or for several rounds has different impacts on the solution time. In Figure 3(b) we report solution time versus the number of rounds of filtering averaged over the 100 instances with $W = 35$. Applying the filtering algorithm once yielded the fastest solution times and so we use this for the remainder of the experiments.

6.4 Evaluating the Impact of the Bandwidth

Next we compare the performance of our MDD-based approach with IP. We also compare the performance of these two methods with a hybrid MDD/IP approach. For the hybrid method, the MDD algorithm runs for a fixed amount of time and then passes the lower bound on the objective function to IP as a initial lower bound on the objective function.

We report results for random instances with $n = 250$, $k = 20$ and bandwidth $b_w \in \{22, 24, \ldots, 44\}$ (20 instances per configuration). In Figure 4(a) we show,

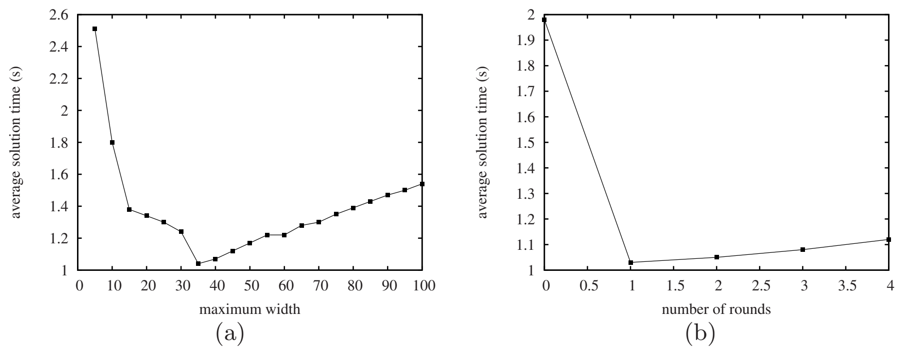


Fig. 3. (a) Maximum width W vs. solution time, (b) Number of rounds of filtering vs. solution time

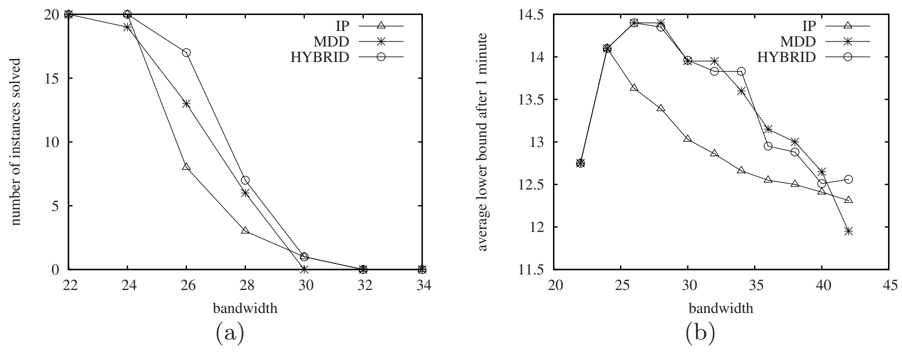


Fig. 4. (a) Number of instances solved in 1 minute for different bandwidths, (b) Average lower bound in 1 minutes for different bandwidths

for increasing bandwidths, the number of instances solved in 60 seconds using the three proposed methods. For the hybrid method, we let the MDD method run for 10 seconds, and then pass the bound z_{LB} given by the MDD method to the IP and let the IP solver run for an additional 50 seconds. In addition, in Figure 4(b) we show, for increasing bandwidths, the best lower bound provided by the three methods after one minute.

For the lower bandwidths, we see that both the MDD-based approach and the hybrid approach outperform IP, with the hybrid method edging out the pure MDD method. As the bandwidth grows, however, the underlying structure that the MDD is able to exploit dissolves, but still the hybrid approach performs best.

6.5 Scaling Up

Here we present results on instances with 500 variables, and again with $k = 20$, to evaluate how the algorithms scale up. We have generated instances for various bandwidths b_w between 21 and 50 (5 random instances per configuration), and

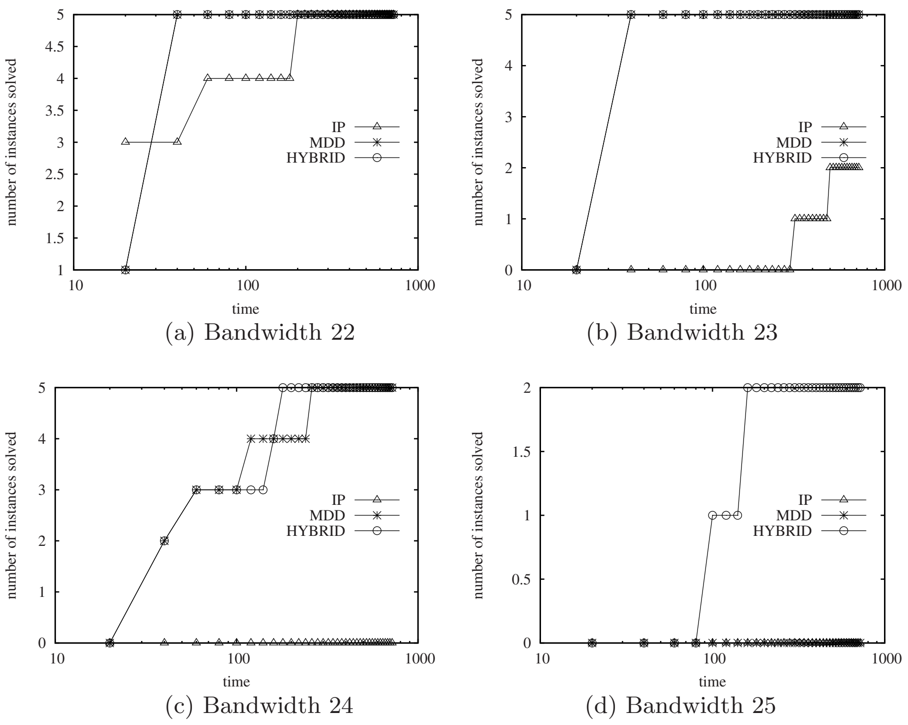
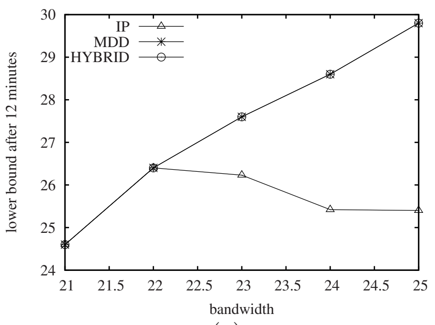


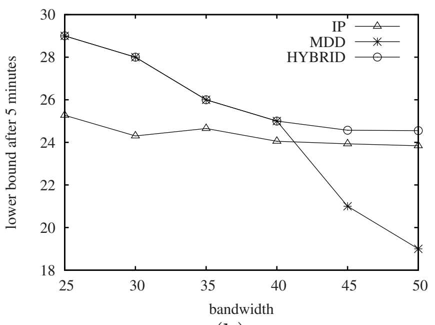
Fig. 5. Performance profile for pure-IP, pure-MDD, and hybrid MDD/IP for instances width various bandwidth. Time is reported in log-scale.

we report the most interesting results corresponding to the 'phase transition', i.e., $b_w \in \{22, 23, 24, 25\}$. We compare the three solution methods, allowing the algorithms to run for 12 minutes.

In the four plots given in Figure 5, we depict the performance profile of the three methods for the different bandwidths. We show for each bandwidth the number of instances solved by time t. As the bandwidth increases, we see that the IP is unable to solve many of the instances that the MDD-based method can, and for $b_w = 25$, neither the pure-IP nor the pure-MDD based methods can solve the instances, while the hybrid method was able to solve 2 of the 5 instances.

Figure 6(a) displays the lower bound given by the three approaches versus time, averaged over the 5 instances. We run the algorithms for 5 minutes and see that the lower bound given by the MDD-based approach dominates the IP bound, especially at small bandwidths. However, as the bandwidth grows, as shown in Figure 6(b), the structure captured by the relaxation MDDs no longer exists and the pure-IP method is able to find better bounds. However, even at the larger bandwidths, the hybrid method provides the best bounds.





(a) (b)

Fig. 6. (a) Bandwidth versus lower bound (12 minute time limit), (b) Larger bandwidths versus lower bound (5 minute time limit)

7 Conclusion

In conclusion, we have examined how relaxation MDDs can help in providing lower bounds for combinatorial optimization problems. We discuss methods for providing lower bounds via relaxation MDDs and provide computational results on applying these ideas to randomly generated set covering problems. We show that in general we can quickly improve upon LP bounds, and even outperform state-of-the-art integer programming technology on problem instances for which the bandwidth of the constraint matrix is limited. Finally, we have shown how a hybrid combination of IP and MDD-based relaxation can be even more effective.

References

1. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers C-27, 509–516 (1978)
2. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A Constraint Store Based on Multivalued Decision Diagrams. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 118–132. Springer, Heidelberg (2007)
3. Becker, B., Behle, M., Eisenbrand, F., Wimmer, R.: BDDs in a branch and cut framework. In: Nikoletseas, S.E. (ed.) WEA 2005. LNCS, vol. 3503, pp. 452–463. Springer, Heidelberg (2005)
4. Behle, M.: On Threshold BDDs and the Optimal Variable Ordering Problem. In: Dress, A.W.M., Xu, Y., Zhu, B. (eds.) COCOA 2007. LNCS, vol. 4616, pp. 124–135. Springer, Heidelberg (2007)
5. Behle, M., Eisenbrand, F.: 0/1 vertex and facet enumeration with BDDs. In: Proceedings of ALENEX. SIAM, Philadelphia (2007)
6. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers C-35, 677–691 (1986)
7. Campos, V., Piñana, E., Martí, R.: Adaptive memory programming for matrix bandwidth minimization. Annals of Operations Research (to appear)
8. Del Corso, G.M., Manzini, G.: Finding exact solutions to the bandwidth minimization problem. Computing 62(3), 189–203 (1999)

9. Feige, U.: Approximating the bandwidth via volume respecting embeddings. J. Comput. Syst. Sci. 60(3), 510–539 (2000)
10. Fulkerson, D.R., Gross, O.A.: Incidence matrices and interval graphs. Pac. J. Math. 15, 835–855 (1965)
11. Gurari, E.M., Sudborough, I.H.: Improved dynamic programming algorithms for bandwidth minimization and the mincut linear arrangement problem. ALGORITHMS: Journal of Algorithms 5 (1984)
12. Hadzic, T., Hooker, J.N.: Postoptimality analysis for integer programming using binary decision diagrams, presented at GICOLAG workshop (Global Optimization: Integrating Convexity, Optimization, Logic Programming, and Computational Algebraic Geometry), Vienna. Technical report, Carnegie Mellon University (2006)
13. Hadzic, T., Hooker, J.N.: Cost-bounded binary decision diagrams for 0-1 programming. Technical report, Carnegie Mellon University (2007)
14. Hadzic, T., Hooker, J.N., O'Sullivan, B., Tiedemann, P.: Approximate Compilation of Constraints into Multivalued Decision Diagrams. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 448–462. Springer, Heidelberg (2008)
15. Hoda, S., van Hoeve, W.-J., Hooker, J.N.: A Systematic Approach to MDD-Based Constraint Programming. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 266–280. Springer, Heidelberg (2010)
16. Hooker, J.N.: Integrated Methods for Optimization. Springer, Heidelberg (2007)
17. Hu, A.J.: Techniques for Efficient Formal Verification Using Binary Decision Diagrams. Technical Report CS-TR-95-1561, Stanford University, Department of Computer Science (1995)
18. Kam, T., Villa, T., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: Multi-valued decision diagrams: Theory and applications. International Journal on Multiple-Valued Logic 4, 9–62 (1998)
19. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell Systems Technical Journal 38, 985–999 (1959)
20. Martí, R., Campos, V., Piñana, E.: A branch and bound algorithm for the matrix bandwidth minimization. European Journal of Operational Research 186(2), 513–528 (2008)
21. Martí, R., Laguna, M., Glover, F., Campos, V.: Reducing the bandwidth of a sparse matrix with tabu search. European Journal of Operational Research 135(2), 450–459 (2001)
22. Piñana, E., Plana, I., Campos, V., Martí, R.: GRASP and path relinking for the matrix bandwidth minimization. European Journal of Operational Research 153(1), 200–210 (2004)
23. Saxe, J.: Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time. SIAM J. Algebraic Discrete Meth. 1, 363–369 (1980)

The AllDifferent Constraint with Precedences[⋆]

Christian Bessiere[1], Nina Narodytska[2], Claude-Guy Quimper[3], and Toby Walsh[2]

[1] CNRS/LIRMM, Montpellier
bessiere@lirmm.fr
[2] NICTA and University of NSW, Sydney
{nina.narodytska,toby.walsh}@nicta.com.au
[3] Université Laval, Québec
claude-guy.quimper@ift.ulaval.ca

Abstract. We propose ALLDIFFPREC, a new global constraint that combines together an ALLDIFFERENT constraint with precedence constraints that strictly order given pairs of variables. We identify a number of applications for this global constraint including instruction scheduling and symmetry breaking. We give an efficient propagation algorithm that enforces bounds consistency on this global constraint. We show how to implement this propagator using a decomposition that extends the bounds consistency enforcing decomposition proposed for the ALLDIFFERENT constraint. Finally, we prove that enforcing domain consistency on this global constraint is NP-hard in general.

1 Introduction

One of the important features of constraint programming are global constraints. These capture common modelling patterns (e.g. "these jobs need to be processed on the same machine so must take place at different times"). In addition, efficient propagation algorithms are associated with global constraints for pruning the search space (e.g. "these 5 jobs have only 4 time slots between them so, by a pigeonhole argument, the problem is infeasible"). One of the oldest and most useful global constraints is the ALLDIFFERENT constraint [1]. This specifies that a set of variables takes all different values. Several algorithms have been proposed for propagating this constraint (e.g. [2,3,4,5,6]). Such propagators can have a significant impact on our ability to solve problems (see, for instance, [7]). It is not hard to provide pathological problems on which some of these propagation algorithms provide exponential savings. A number of hybrid frameworks have been proposed to combine the benefits of such propagation algorithms and OR methods like integer linear programming (see, for instance, [8]). In addition, the convex hull of a number of global constraints has been studied in detail (see, for instance, [9]).

In this paper, we consider a modelling pattern [10] that occurs in many problems involving ALLDIFFERENT constraints. In addition to the constraint that no pair of variables can take the same value, we may also have a constraint that certain pairs of variables are ordered (e.g. "these two jobs need to be processed on the same machine so

[⋆] Supported by the Australian Government's Department of Broadband, Communications and the Digital Economy and the ARC.

must take place at different times, but the first job must be processed before the second"). We propose a new global constraint, ALLDIFFPREC that captures this pattern. This global constraint is a specialization of the general framework that combines several CUMULATIVE and precedence constraints [11,12]. Reasoning about such combinations of global constraints may achieve additional pruning. In this work we propose an efficient propagation algorithm for the ALLDIFFPREC constraint. However, we also prove that propagating the constraint *completely* is computationally intractable.

2 Formal Background

A constraint satisfaction problem (CSP) consists of a set of variables, each with a domain of possible values, and a set of constraints specifying allowed values for subsets of variables. A solution is an assignment of values to the variables satisfying the constraints. We write $\mathcal{D}(X)$ for the domain of the variable X. Domains can be ordered (e.g. integers). In this case, we write $min(X)$ and $max(X)$ for the minimum and maximum elements in $\mathcal{D}(X)$. The *scope* of a constraint is the set of variables to which it is applied. A *global constraint* is one in which the number of variables is not fixed. For instance, the global constraint ALLDIFFERENT$([X_1, \ldots, X_n])$ ensures $X_i \neq X_j$ for $1 \leq i < j \leq n$. By comparison, the binary constraint, $X_i \neq X_j$ is not global.

When solving a CSP, we often use propagation algorithms to prune the search space by enforcing properties like domain, bounds or range consistency. A *support* on a constraint C is an assignment of all variables in the scope of C to values in their domain such that C is satisfied. A variable-value $X_i = v$ is *consistent* on C iff it belongs to a support of C. A constraint C is *domain consistent* (*DC*) iff every value in the domain of every variable in the scope of C is consistent on C. A *bound support* on C is an assignment of all variables in the scope of C to values between their minimum and maximum values (respectively called lower and upper bound) such that C is satisfied. A variable-value $X_i = v$ is *bounds consistent* on C iff it belongs to a bound support of C. A constraint C is *bounds consistent* (*BC*) iff the lower and upper bounds of every variable in the scope of C are bounds consistent on C. Range consistency is stronger than BC but is weaker than DC. A constraint C is *range consistent* (*RC*) iff iff every value in the domain of every variable in the scope of C is bounds consistent on C. A CSP is *DC*/*RC*/*BC* iff each constraint is *DC*/*RC*/*BC*. Generic algorithms exists for enforcing such local consistency properties. For global constraints like ALLDIFFERENT, specialized methods have also been developed which offer computational efficiencies. For example, a bounds consistency propagator for ALLDIFFERENT is based on the notion of Hall interval. A Hall interval is an interval of h domain values that completely contains the domains of h variables. Clearly, variables whose domains are contained within the Hall interval consume all the values in the Hall interval, whilst any other variables must find their support outside the Hall interval.

We will compare local consistency properties applied to logically equivalent constraints. As in [13], we say that a local consistency property Φ on the set of constraints S is *stronger* than Ψ on the logically equivalent set T iff, given any domains, Φ removes all values Ψ removes, and sometimes more. For example, domain consistency on ALLDIFFERENT$([X_1, \ldots, X_n])$ is stronger than domain consistency on

$\{X_i \neq X_j \mid 1 \leq i < j \leq n\}$. In other words, decomposition of the global ALLDIFFERENT constraint into binary not-equals constraints hinders propagation.

3 Some Examples

To motivate the introduction of this global constraint, we give some examples of models where we have one or more sets of variables which take all-different values, as well as certain pairs of these variables which are ordered.

3.1 Exam Time-Tabling

Suppose we are time-tabling exams. A straight forward model has variables for exams, and values which are the possible times for these exams. In such a model, we may have temporal precedences (e.g. part 1 of the physics exam must be before part 2) as well as ALLDIFFERENT constraints on those sets of exams with students in common (e.g. all physics, maths, and chemistry exams must occur at different times since there are students that need to sit all three exams).

3.2 Scheduling

Suppose we are scheduling a single machine with unit-time tasks, subject to precedence constraints and release and due times [14]. A straight forward model has variables for the tasks, and values which are the possible times that we execute each task. In such a model, we have an ALLDIFFPREC constraint on variables whose domains are the appropriate intervals. For example, consider scheduling instructions in a block (a straight-line sequence of code with a single entry and exit point) on one processor where all instructions take the same time to execute. Such a schedule is subject to a number of different types of precedence constraints. For instance, instruction $\mathcal{A}$ must execute before $\mathcal{B}$ if:

Read-after-write dependency: $\mathcal{B}$ reads a register written by $\mathcal{A}$;
Write-after-write dependency: $\mathcal{B}$ writes a register also written by $\mathcal{A}$;
Write-after-read dependency: $\mathcal{B}$ writes a register that $\mathcal{A}$ reads.

Such dependencies give rise to precedence constraints between the instructions.

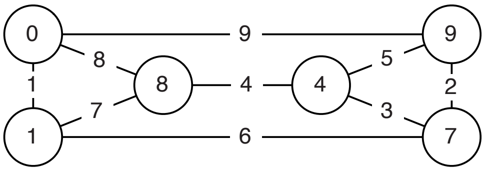
3.3 Breaking Value Symmetry

Many constraint models contain value symmetry. Puget has proposed a general method for breaking any number of value symmetries in polynomial time [15,16]. This method introduces variables Z_j to represent the index of the first occurrence of each value:

$$X_i = j \Rightarrow Z_j \leq i, \qquad Z_j = i \Rightarrow X_i = j$$

Value symmetry on the X_i is transformed into variable symmetry on the Z_j. This variable symmetry is easy to break. We simply need to post precedence constraints on the Z_j. Depending on the value symmetry, we need different precedence constraints.

Consider, for example, finding a graceful labelling of a graph. A graceful labelling is a labelling of the vertices of a graph with distinct integers 0 to e such that the e edges (which are labelled with the absolute differences of the labels of the two connected vertices) are also distinct. Graceful labellings have applications in radio astronomy, communication networks, X-ray crystallography, coding theory and elsewhere. Here is the graceful labelling of the graph $K_3 \times P_2$:



A straight forward model for graceful labelling a graph has variables for the vertex labels, and values which are integers 0 to e. This model has a simple value symmetry as we can map every value i onto $e - i$. In [16], Puget breaks this value symmetry for $K_3 \times P_2$ with the following ordering constraints:

$$Z_0 < Z_1,\ Z_0 < Z_3,\ Z_0 < Z_4,\ Z_0 < Z_5,\ Z_1 < Z_2$$

Note that all the Z_j take different values as each integer first occurs in the graph at a different index. Hence, we have a sequence of variables on which there is both an AllDifferent and precedence constraints.

4 AllDiffPrec

Motivated by such examples, we propose the global constraint:

$$\text{AllDiffPrec}([X_1, \ldots, X_n], E)$$

Where E is a set containing pairs of variable indices. This ensures $X_i \neq X_j$ for any $1 \leq i < j \leq n$ and $X_j < X_k$ for any $(j, k) \in E$. Without loss of generality, we assume that E does not contain cycles. If it does, the constraint is trivially unsatisfiable. It is not hard to see that decomposition of this global constraint into separate AllDifferent and binary ordering constraints can hinder propagation.

Lemma 1. *Domain consistency on the constraint* AllDiffPrec$([X_1, \ldots, X_n], E)$ *is stronger than domain consistency on the decomposition into* AllDifferent$([X_1, \ldots, X_n])$ *and the binary ordering constraints,* $X_i < X_j$ *for* $(i, j) \in E$. *Bounds consistency on* AllDiffPrec$([X_1, \ldots, X_n], E)$ *is stronger than bounds consistency on the decomposition, whilst range consistency on* AllDiffPrec$([X_1, \ldots, X_n], E)$ *is stronger than range consistency on the decomposition.*

Proof: Consider AllDiffPrec$([X_1, X_2, X_3], \{(1, 3), (2, 3)\})$ with $\mathcal{D}(X_1) = \mathcal{D}(X_2) = \{1, 2, 3\}$ and $\mathcal{D}(X_3) = \{2, 3, 4\}$. Then the decomposition into AllDifferent$([X_1, X_2, X_3])$ and the binary ordering constraints, $X_1 < X_3$, and $X_2 < X_3$ is domain consistent. Hence, it is also range and bounds consistent. However, enforcing bounds consistency directly on the global AllDiffPrec constraint will prune 2 from the domain of X_3 since this assignment has no bound support. Similarly, enforcing range or domain consistency will prune 2 from the domain of X_3. □

A simple greedy method will find a bound support for the AllDiffPrec constraint. This method is an adaptation of the greedy method to build a bound support of the AllDifferent constraint. For simplicity, we suppose that E contains the transitive closure of the precedence constraints. In fact, this step is not required but makes our argument easier. First, we need to preprocess variables domains so that they respect the precedence constraints $X_i < X_j$, $(i, j) \in E$: $\min(X_i) < \min(X_j)$ and $\max(X_i) < \max(X_j)$. However, we notice that it is sufficient to enforce a weaker condition on bounds of variables X_i and X_j such that $\min(X_i) \leq \min(X_j)$ and $\max(X_i) \leq \max(X_j)$. If these conditions on variables domains are satisfied then we say that domains are *preprocessed*. Second, we construct a satisfying assignment as follows. We process all values in the increasing order. When processing a value v, we assign v to the variable with the smallest upper bound, u that has not yet been assigned and that contains v in its domain. Suppose, there exists a set of variables that have the upper bound u, so that $X' = \{X_i \mid \mathcal{D}(X_i) = [v, u]\}$. To construct a solution for AllDifferent, we would break these ties arbitrarily. In this case, however, we select a variable that is not successor of any variable in the set X'. Such a variable always exists, as the transitive closure of the precedence graph does not contain cycles. By the correctness of the original algorithm the resulting assignment is a solution. In addition to satisfying the AllDifferent constraint, this solution also satisfies the precedence constraints. Indeed, for the constraint $X_i < X_j$, the upper bound of $\mathcal{D}(X_i)$ is necessarily smaller than or equal to the upper bound of $\mathcal{D}(X_j)$. In the case of equality, we tie break in favor of X_i. Therefore, a value is assigned to X_i before a value gets assigned to X_j . Since we process values in increasing order, we obtain $X_i < X_j$ as required.

Example 1. *Consider* AllDiffPrec$([X_1, X_2, X_3, X_4], \{(1, 3), (2, 3), (1, 4), (2, 4)\})$ *with* $\mathcal{D}(X_1) = \mathcal{D}(X_2) = \{1, 2, 3, 4, 5\}$, $\mathcal{D}(X_3) = \{1, 2, 3\}$ *and* $\mathcal{D}(X_4) = \{2, 3, 4\}$. *First, we preprocess domains to ensure that* $\min(X_i) \leq \min(X_j)$ *and* $\max(X_i) \leq \max(X_j)$, $i \in \{1, 2\}$, $j \in \{3, 4\}$. *This gives* $\mathcal{D}(X_1) = \mathcal{D}(X_2) = \mathcal{D}(X_3) = \{1, 2, 3\}$, $\mathcal{D}(X_4) = \{2, 3, 4\}$. *As in the greedy algorithm, we consider the first value* 1. *This value is contained in domains of variables* X_1, X_2 *and* X_3. *As* $\max(X_1) = \max(X_2) = \max(X_3) = 3$, *by tie breaking we select variables that are not successors of any other variables among variables* $\{X_1, X_2, X_3\}$. *There are two such variables:* X_1 *and* X_2. *We break this tie arbitrarily and set* X_1 *to 1. The new domains are* $\mathcal{D}(X_1) = 1$, $\mathcal{D}(X_2) = \mathcal{D}(X_3) = \{2, 3\}$, $\mathcal{D}(X_4) = \{2, 3, 4\}$. *The next value we consider is* 2. *Again, there exist two variables that contain this value, and they have the same upper bounds. By tie-breaking, we select* X_2. *Finally, we assign* X_3 *and* X_4 *to 3 and 4 respectively.*

We can design a filtering algorithm based on this satisfiability test. By successively reducing a variable domain in halves with a binary search we can filter the lower and upper bounds of a variable domain with $O(\log d)$ tests where d is the cardinality of the domain. Consider, for example, a variable X with the domain $D(X) = [l, u]$. We are looking for a support for $\min(X)$. At the first step we temporally fix the domain of X to the first half so that $D(X) = [l, (u - l)/2]$ and run the bounds disentailment detection algorithm. If this algorithm fails, we halved the search and repeat with the other half. If this algorithm does not fail, we know that there is a value in $[l, (u-l)/2]$ that has a bounds support. Hence, we continue with the binary search within this half. As each test takes $O(n)$ time and there are n variables to prune, the total running time is $O(n^2 \log d)$. In the rest of this paper, we improve on this using sophisticated algorithmic ideas.

5 Bounds Consistency

We present an algorithm that enforces bounds consistency on the ALLDIFFPREC constraint. First, we consider an assignment $X_i = v$ and a partial filtering that this assignment causes. We call this filtering *direct pruning* caused by the assignment $X_i = v$ or, in short, direct pruning of $X_i = v$. Informally, direct pruning works as follows. If X_i takes v then the value v becomes unavailable for the other variables due to the ALLDIFFERENT constraint. Hence, we remove v from the domains of variables that have v as their lower bound or upper bound. Due to precedence constraints, we increase the lower bounds of successors of X_i to $v+1$ and decrease the upper bounds of predecessors of X_i to $v-1$. Note that direct pruning *does not enforce* bounds consistency on either ALLDIFFPREC or the single ALLDIFFERENT constraint. However, direct pruning is sufficient to detect bounds inconsistency as we show below.

Let $P(i)$ and $S(i)$ be the sets of variables that precede and succeed X_i, respectively. We denote the domains obtained after direct pruning of $X_i = v$ as $\mathcal{D}_v^{dp}(X_1), \ldots, \mathcal{D}_v^{dp}(X_n)$, so that for all $j = 1, \ldots, n$:

$$\mathcal{D}_v^{dp}(X_j) = \mathcal{D}(X_j) \setminus \{v\} \text{ if } j \neq i, v \in \{\min(X_j), \max(X_j)\} \tag{1}$$

$$\mathcal{D}_v^{dp}(X_j) = v \text{ if } j = i, \tag{2}$$

$$\mathcal{D}_v^{dp}(X_j) = \mathcal{D}(X_j) \setminus [v, max(X_j)] \text{ if } j \in P(i), \tag{3}$$

$$\mathcal{D}_v^{dp}(X_j) = \mathcal{D}(X_j) \setminus [min(X_j), v] \text{ if } j \in S(i). \tag{4}$$

These bounds could be pruned further but we will first analyze the properties that this simple filtering offers.

Example 2. *Consider* ALLDIFFPREC$([X_1, X_2, X_3], \{(1,2)\})$ *constraint with* $\mathcal{D}(X_1) = \{1,2\}$, $\mathcal{D}(X_2) = \{2,3\}$, $\mathcal{D}(X_3) = \{1,2,3\}$. *For example, an assignment* $X_1 = 2$ *results in the domains:* $\mathcal{D}_2^{dp}(X_1) = \{2\}$, $\mathcal{D}_2^{dp}(X_2) = \{3\}$ *and* $\mathcal{D}_2^{dp}(X_3) = \{1,2,3\}$. *We point out again that we can continue pruning as values* 2 *and* 3 *have to be removed from* $\mathcal{D}_2^{dp}(X_3)$. *However, direct pruning of* $X_1 = 2$ *is sufficient for our purpose. Consider another example. An assignment* $X_3 = 1$ *results in the domains:* $\mathcal{D}_3^{dp}(X_1) = \{2\}$, $\mathcal{D}_3^{dp}(X_2) = \{2,3\}$ *and* $\mathcal{D}_3^{dp}(X_3) = \{1\}$.

Our algorithm is based on the following lemma.

Lemma 2. *Let* ALLDIFFERENT *and precedence constraints be bounds consistent over variables* X, $X_i = v$, $v \in \{\min(X_i), \max(X_i)\}$ *be an assignment of a variable* X_i *to its bound and* $\mathcal{D}_v^{dp}(X_1), \ldots, \mathcal{D}_v^{dp}(X_n)$ *be the domains after direct pruning of* $X_i = v$. *Then,* $X_i = v$ *is bounds consistent iff* ALLDIFFERENT$([X_1, \ldots, X_n])$, *where domains of variables* X *are* $\mathcal{D}_v^{dp}(X_1), \ldots, \mathcal{D}_v^{dp}(X_n)$, *has a solution.*

Proof: Suppose ALLDIFFERENT and the precedence constraints are bounds consistent. As precedence constraints are bounds consistent, we know that for all $(i, j) \in E$, $X_i < X_j$, $\min(X_i) < \min(X_j)$ and $\max(X_i) < \max(X_j)$. Consider direct pruning of $X_i = v$. Note, direct pruning of $X_i = v$ preserves the property of domains being preprocessed. The pruning can only create equality of lower bounds or upper bounds for some precedence constraints. The assignment $X_3 = 1$ demonstrates this situation in Example 2. Direct pruning of $X_3 = 1$ forces lower bounds of X_1 and X_2, that are in the precedence relation, to be equal.

As domains $\mathcal{D}_v^{dp}(X_1), \ldots, \mathcal{D}_v^{dp}(X_n)$ are preprocessed, we know that the greedy algorithm (Section 4) will find a solution of ALLDIFFERENT on the domains $\mathcal{D}_v^{dp}(X_1), \ldots, \mathcal{D}_v^{dp}(X_n)$ that also satisfies the precedence constraints if a solution exists. This solution is a support for $X_i = v$. □

Based on Lemma 2 we prove that we can enforce bounds consistency on the ALLDIFFPREC constraint in $O(n^2)$. However, we start with a simpler and less efficient algorithm to explain the idea . We show how to improve this algorithm in the next section. Given Lemma 2, the most straightforward algorithm to enforce bounds consistency for $X_i = v$ is to assign X_i to v, perform the direct pruning, run the greedy algorithm and, if it fails, prune v. Interestingly enough, to detect bounds disentailment we do not have to run a greedy algorithm for each pair $X_i = v$. If the ALLDIFFERENT constraint and the precedence constraints are bounds consistent, we show that it is sufficient to check that a set of conditions (5)-(10) holds for each interval of values. If these conditions are satisfied then the pair $X_i = v$ is bounds consistent. Hence, for each pair $X_i = v$, $1 \leq i \leq n$, $v \in D(X_i)$, and for each interval we enforce the following conditions. We assume that $\cup_{i=1}^{n} \mathcal{D}(X_i) = [1, d]$. For X_i, $1 \leq i \leq n$, $v \in \mathcal{D}(X_i)$ and for all intervals $[v, v + k]$ and $[v - p, v]$, $k \in [\max(X_i) - v + 1, d - v]$ and $p \in [v - \min(X_i) + 1, v - 1]$, the following conditions have to be satisfied:

$$B^i_{1,v+k} = |\{j \in S(i) | \mathcal{D}(X_j) \subseteq [1, v + k]\}| \tag{5}$$

$$D^i_{v,(v+k)} = |\{j \notin S(i) | \mathcal{D}(X_j) \subseteq [v, v + k]\}| \tag{6}$$

$$B^i_{1,v+k} + D^i_{v,(v+k)} \leq k \tag{7}$$

$$B^i_{v-p,d} = |\{j \in P(i) | \mathcal{D}(X_j) \subseteq [v - p, d]\}| \tag{8}$$

$$D^i_{v-p,v} = |\{j \notin P(i) | \mathcal{D}(X_j) \subseteq [v - p, v]\}| \tag{9}$$

$$B^i_{v-p,d} + D^i_{v-p,v} \leq p \tag{10}$$

Note that we actually do not have to consider all possible intervals. For every variable-value pair $X_i = v$ we consider all intervals $[v, u]$, $u \in [\max(X_i) + 1, d]$ and all intervals $[l, v]$, $l \in [1, \min(X_i) - 1]$. The parameter k (p) is used to slide between intervals $[v, u]$, $u \in [\max(X_i) + 1, d]$ ($[l, v]$, $l \in [1, \min(X_i) - 1]$). Equations (5)–(7) make sure that the number of variables that fall into an interval $[v, u]$, after the assignment X_i to v, is less than or equal to the length of the interval minus 1. Symmetrically, Equations (8)–(10) ensure that the same condition is satisfied for all intervals $[l, v]$. If there exists an interval $[v, u]([l, v])$ that violates the condition for a pair $X_i = v$ then this interval is removed from $D(X_i)$.

Example 3. *Consider* $\text{ALLDIFFPREC}([X_1, X_2, X_3, X_4, X_5], \{(1,2),(1,3)\})$. *Domains of the variables are* $D(X_1) = [1,5]$, $D(X_2) = D(X_3) = [2,6]$ *and* $D(X_4) = D(X_5) = [3,6]$. *Consider a variable-value pair* $X_1 = 3$. *By the direct pruning we get the following domains:* $\mathcal{D}_3^{dp}(X_1) = 3$, $\mathcal{D}_3^{dp}(X_2) = [4,6]$, $\mathcal{D}_3^{dp}(X_3) = [4,6]$, $\mathcal{D}_3^{dp}(X_4) = [4,6]$ *and* $\mathcal{D}_3^{dp}(X_5) = [4,6]$. *The interval* $[4,6]$ *is a violated Hall interval as it contains four variables. We show that Equations* (5)–(6) *detect that the interval* $[3,6]$ *has to be pruned from* $D(X_1)$.

Consider the pair $X_1 = 3$ *and the interval* $[v, v+k]$, *where* $v = 3$, $k = 3$. *We get that* $B^1_{1,6} = |\{j \in \{2,3\} | \mathcal{D}(X_j) \subseteq [1,6]\}| = 2$ $D^1_{3,6} = |\{j \in \{4,5\}) | \mathcal{D}(X_j) \subseteq [3,6]\}| = 2$ *and* $B^1_{1,6} + D^1_{3,6} = 4$ *which is greater than* $k = 3$. *Hence, the interval* $[3,6]$ *has to be removed from* $D(X_1)$.

Theorem 1. *Consider the* $\text{ALLDIFFERENT}[X_1, \dots, X_n]$ *constraint and a set of precedence constraints* $X_i < X_j$. *Enforcing conditions* (5)–(10) *together with bounds consistency on the* ALLDIFFERENT *constraint and the precedence constraints is equivalent to enforcing bounds consistency on the* ALLDIFFPREC *constraint.*

Proof: Suppose conditions (5)–(10) are fulfilled, ALLDIFFERENT and precedence constraints are bounds consistent and the ALLDIFFPREC constraint is not bounds consistent. Let an assignment of a variable X_i to its bound $\max(X_i)$ be an unsupported bound. We denote $\max(X_i)$ v to simplify notations. We recall that we denoted the domains after direct pruning of $X_i = v$ $\mathcal{D}_v^{dp}(X_1), \dots, \mathcal{D}_v^{dp}(X_n)$. By Lemma 2 the $\text{ALLDIFFERENT}([X_1, \dots, X_n])$ constraint where domains of variables X are $\mathcal{D}_v^{dp}(X_1), \dots, \mathcal{D}_v^{dp}(X_n)$ fails. Hence, there exists a violated Hall interval $[l,u]$ such that $|\mathcal{D}_v^{dp}(X_i) \subseteq [l,u]\}| > u - l + 1$.

Note that direct pruning of $X_i = v$ does not cause the pruning of variables in $P(i)$, as all precedence constraints are bounds consistent on the original domains. Next we consider several cases depending on the relative position of the value v and the violated Hall interval on the line. Note that the interval $[l,u]$ was not a violated Hall interval before the assignment $X_i = v$. However, due to direct pruning of $X_i = v$ a number of additional variables domains can be forced to be inside $[l,u]$. Hence, we analyze these additional variables and show that conditions (5)–(10) prevent the creation of a violated Hall interval.

Case 1. Suppose $v \in [l,u]$. As $[l,u]$ is a violated Hall interval, we have that

$$|\{j \in S(i) | \mathcal{D}_v^{dp}(X_j) \subseteq [l,u]]\}| + |\{j \notin S(i) | \mathcal{D}_v^{dp}(X_j) \subseteq [l,u]\}| > u - l,$$

Note that the number of additional variables that fall into the interval $[l,u]$ after setting X_i to v consists only of variables that succeed X_i, such that $\mathcal{D}(X_j) \subseteq [1,u]$. Hence, $|\{j \notin S(i) | \mathcal{D}_v^{dp}(X_j) \subseteq [l,u]\}| = |\{j \notin S(i) | \mathcal{D}(X_j) \subseteq [l,u]\}|$, $|\{j \notin S(i) | \mathcal{D}_v^{dp}(X_j) \subseteq [l,u]\}| = |\{j \in S(i) | \mathcal{D}(X_j) \subseteq [1,u]\}|$ and

$$|\{j \in S(i) | \mathcal{D}(X_j) \subseteq [1,u]\}| + |\{j \notin S(i) | \mathcal{D}(X_j) \subseteq [l,u]\}| > u - l,$$

which violate conditions (5)–(7) for $v = l$ and $k = u - l$.

Case 2. Suppose $v \notin [l, u]$. If $v > u + 1$ or $v < l - 1$, the assignment $X_i = v$ does not force any extra variables to fall into the interval $[l, u]$. Hence, the interval $[l, u]$ is a violated Hall interval before the assignment. This contradicts that AllDifferent is bounds consistent.

Case 3. Suppose $v = u + 1$. In this case the assignment $X_i = v$ does not force any additional variables among successors to fall into $[l, u]$, as $\mathcal{D}_v^{dp}(X_j) \subseteq [u + 2, d]$. Note that there are no successors that are contained in the interval $[1, v]$, because precedence constraints are bounds consistent. Therefore, $|\{j \in S(i)|\mathcal{D}(X_j) \subseteq [l, v]\}| = 0$. Hence, the only additional variables that fall into $[l, u]$ are variables that do not have a precedence relation with X_i and $v = \max(X_j) = u + 1$, so $|\{j|j \notin S(i), \mathcal{D}_v^{dp}(X_j) \subseteq [l, u]\}| = |\{j|j \notin S(i), \mathcal{D}(X_j) \subseteq [l, u + 1]\}|$. As $[l, u]$ is a violated Hall interval, we have

$$|\{j|j \notin S(i), \mathcal{D}(X_j) \subseteq [l, u+1]\}| = |\{j|j \notin S(i), \mathcal{D}_v^{dp}(X_j) \subseteq [l, u]\}| > u - l + 1.$$

This contradicts Equation (10) $|\{j \in S(i)|\mathcal{D}(X_j) \subseteq [l, u + 1]\}| + |\{j|j \notin S(i), \mathcal{D}(X_j) \subseteq [l, u + 1]\}| \leq (u + 1) - l$ as the first term equals 0 in the equation by the argument above.

Case 4. Suppose $v = l - 1$. In this case the set of additional variables that fall into the interval $[l, u]$ consists of two subsets of variables. The first set contains variables that succeed X_i, such that $\mathcal{D}(X_j) \subseteq [l', u]$, $l' < v$ and $\mathcal{D}_v^{dp}(X_j) \subseteq [l, u]$. The second set contains the variables that do not have precedence relation with X_i and $v = max(X_j) = l - 1$. Consider the interval $[l - 1, u]$. As conditions (5)–(7) are satisfied for the interval $[l - 1, u]$, we get that

$$|\{j \in S(i)|\mathcal{D}(X_j) \subseteq [1, u]\}| + |\{j \notin S(i)|\mathcal{D}(X_j) \subseteq [l - 1, u]\}| \leq u - (l - 1),$$

On the other hand, as the $[l, u]$ is violated we have

$$|\{j \in S(i)|\mathcal{D}_v^{dp}(X_j) \subseteq [l, u]\}| + |\{j \notin S(i)|\mathcal{D}_v^{dp}(X_j) \subseteq [l, u]\}| > u - l + 1,$$

We know that $|\{j \notin S(i)|\mathcal{D}(X_j) \subseteq [l - 1, u]\}| = |\{j|j \notin S(i), \mathcal{D}_v^{dp}(X_j) \subseteq [l, u]\}|$ and $|\{j \in S(i)|\mathcal{D}(X_j) \subseteq [1, u]\}| = |\{j \in S(i)|\mathcal{D}_v^{dp}(X_j) \subseteq [l, u]\}|$ by the construction of the direct pruning. This leads to a contradiction between the last two inequalities.

Therefore, the interval $[l, u]$ cannot be a violated Hall interval. Similarly, we can prove the same result for the minimum value of X_j.

The reverse direction is trivial. □

Theorem 1 proves that conditions (5)–(10) together with bounds consistency on the AllDifferent constraint and the precedence constraints are necessary and sufficient conditions to enforce bounds consistency on the AllDiffPrec constraint. The time complexity of enforcing these conditions in $O(nd^2)$, as for each variable we check $O(d^2)$ intervals. This time complexity can be reduced by making an observation, that we do not need to check intervals of length greater than n as conditions are trivially satisfied for such intervals. This reduces the complexity to $O(n^2d)$.

We make an observation that helps to further reduce the time complexity of enforcing these conditions. We denote L the set of all minimum values in variables domains

$L = \cup_{i=1}^{n}\{\min(\mathcal{D}(X_i))\}$ and U the set of all maximum values in variables domains $U = \cup_{i=1}^{n}\{\max(\mathcal{D}(X_i))\}$. Let $[l, u]$ be an interval that violates the conditions. We denote $c_{l,u}$ the amount of violation in this interval: $c_{l,u} = B^i_{1,u} + D^i_{l,u} - (u - l)$.

Observation 1. *Let X_i be a variable and $[v, v + k]$, $v \in \mathcal{D}(X_i)$ be an interval that violates conditions* (5)–(7). *Then there exists a violated interval $[l, u]$ such that $[l, u] \subseteq [v, v + k]$, $l, u \in L \cup U$ and $c_{l,u} > l - v$.*

Proof: Consider a violated interval $[v, v+k]$. In this case $B^i_{1,v+k} + D^i_{v,v+k} > k$. There exists an interval $[l, u] \subseteq [v, v + k]$ such that $l, u \in L \cup U$. We take the largest interval $[l, u]$. Note that such an interval always exists as the interval $[\max(X_i), \max(X_i)]$ is contained inside the interval $[v, v + k]$. The interval $[l, u]$ also violates the conditions, because it contains the same variables. So, we have $B^i_{1,u} + D^i_{l,u} > u - l$. We note that $D^i_{l,u} = D^i_{v,v+k}$ as there are no lower bounds in the interval $[v, l)$. Similarly, there are no upper bounds in the interval $(u, v + k]$. Hence, $B^i_{1,u} = B^i_{1,v+k}$. Therefore, $B^i_{1,u} + D^i_{l,u} > k$. The value $c_{l,u}$ is greater than $k - u + l \geq v + k - v - u + l \geq v + k - u + l - v \geq l - v$ as $u \leq v + k$. □

Observation 1 shows that it is sufficient to check intervals $[v, v+k]$, $\{v, v+k\} \in L \cup U$. We can infer all pruning from these intervals. Let $[l, u]$, $l, u \in L \cup U$ be an interval that violates conditions (5)–(7) for a variable X_i and $c_{l,u}$ be the violation cost. Then we remove the interval $[l - (c_{l,u} - 1), u]$ from $\mathcal{D}(X_i)$, as any interval between $[l - (c_{l,u} - 1), u]$ and $[l, u]$ is a violated interval. A dual observation holds for conditions (8)–(10). This reduces the time complexity of checking (5)–(10) to $O(n^3)$.

Algorithm 1. PruneUpperBounds($X_1, \ldots, X_n$)

```
1  Sort variables such that $\max(\mathcal{D}(X_i)) \leq \max(\mathcal{D}(X_{i+1}))$;
2  for $i \in 1..n$ do
3      Create a disjoint set data structure $T$ with the integers $1..d$;
4      $b \leftarrow \max(\mathcal{D}(X_1)) + 1$;
5      Invariant: $b$ is the smallest value such that there are exactly as many available values
       in the open-interval $[b, \max(\mathcal{D}(X_j)) + 1)$ as there are successors of $X_i$ that have
       been processed.;
6      for $X_j$ in non-decreasing order of upper bound do
7          if $j \notin S(i)$ then
8              $S \leftarrow$ Find($\min(\mathcal{D}(X_j)), T$);
9              $v \leftarrow \min(S)$;
10             Union($v, \max(S) + 1$, T);
11         if $j > 1$ then
12             for $k \in 1..\max(\mathcal{D}(X_j)) - \max(\mathcal{D}(X_{j-1}))$ do
13                 $b \leftarrow \max(Find(b, T)) + 1$;
14             if $Find(v, T) = Find(b, T) \vee v > b \vee j \in S(i)$ then
15                 $b \leftarrow \min(Find(b - 1, T))$;
16         $\max(\mathcal{D}(X_i)) \leftarrow \min(\max(\mathcal{D}(X_i)), b - 1)$;
```

6 Faster Bounds Consistency Algorithm

Observation 1 allows us to construct a faster algorithm to enforce conditions (5)–(10). First, we observe that the conditions can be checked for each variable independently. Consider a variable X_i. We sort all variables X_j, $j = 1, \ldots, n$ in a non-decreasing order of their upper bounds. When processing a variable $X_j, j \notin S(i)$, we assign X_j to the smallest value that has not been taken. When processing a variable $X_j, j \in S(i)$, we store information about the number of successors that we have seen so far. We perform pruning if we find an interval $[l, u]$ such that the number of available values in this interval equals the number of successors in the interval $[1, u]$. We use a disjoint set data structure to perform counting operations in $O(d)$ time.

Algorithm 1 shows a pseudocode of our algorithm. We denote T a disjoint set data structure. The function $Find(v_1, T)$ returns the set that contains the value v_1. The function $Union(v_1, v_2, T)$ joins the values v_1 and v_2 into a single set. We use a disjoint set union data structure [22] that allows to perform $Find$ and $Union$ in $O(1)$ time.

Theorem 1. *Algorithm 1 enforces conditions* (5)–(7) *in $O(nd)$ time.*

Proof: Enforcing conditions (5)–(7) on the ith variable corresponds to the ith loop (line 1). Hence, we can consider each run independently.

We denote I_j a set of values that are taken by non-successors of X_i after the variable X_j is processed. The algorithm maintains a pointer b that stores the minimum value such that the number of available values in the interval $[b, \max(X_j) + 1)$ is equal to $B^i_{1,max(X_j)}$ after the variable X_j is processed.

Invariant. We prove the invariant for the pointer b by induction. The invariant holds at step $j = 0$. Note that the first variable can not be a successor of X_i. Indeed, $b = max(X_1) + 1$ and the interval $[\max(X_1) + 1, \max(X_1) + 1)$ is empty. Let us assume that the invariant holds after processing the variable X_{j-1}.

Suppose the next variable to process is X_j. After we assigned X_j to a value, we move b forward to capture a possible increase of the upper bound from $\max(X_{j-1})$ to $\max(X_j)$ (line 1) and, then, backward if either X_j is a successor of X_i or X_j is a non-successor and X_j takes a value v such that $b \leq v$ (line 1). Note, that when we move b, we ignore values in I_j. To point this out we call steps of b available-value-steps. Thanks to a disjoint set union data structure we can jump over values in I_j in $O(1)$ per step [22].

Moving forward. We move the pointer b on $\max(X_j) - \max(X_{j-1})$ available-value-steps forward. We denote b' a new value of b. The line 1 ensures that the number of available values in the interval $[b', \max(X_j) + 1)$ equals to the number of available values in the interval $[b, \max(X_{j-1}) + 1)$. This operation preserves the invariant by the induction hypothesis.

Moving backward. We consider two cases.

Case 1. X_j is a successor of X_i. In this case, we move b' one available-value-step backward to capture that X_j is a successor (line 1). This preserves the invariant.

Case 2. X_j is not a successor of X_i. Suppose v and b' are in the same set, so that $Find(v, T) = Find(b, T)$. Then we move b' to the minimum element in this set. This step does not change the number of available values between the pointer b' and

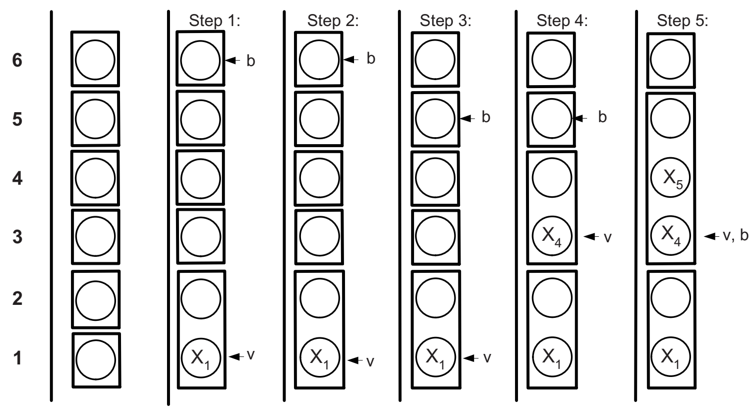


Fig. 1. Algorithm 1 enforces conditions (5)–(7) on the variable X_1

$\max(X_j)$. However, it makes sure that b' stores the minimum possible value. This preserves the invariant.

Suppose v and b' are in different sets. If $v > b'$ then we move b' one available-value-step backward, as v took one of the available values in $[b', \max(X_j) + 1)$. This preserves the invariant. If $v < b'$ then the invariant holds by the induction hypothesis. Hence, the new value of b preserves the invariant.

Note that the length of the interval $[b, \max(X_j) + 1)$ equals the sum of $B^i_{1,\max(X_j)}$ and $D_{b,\max(X_j)}$ due to the invariant. This means that the interval $[b, \max(X_j) + 1)$ violates conditions (5)–(7), as the sum $B^i_{1,\max(X_j)} + D_{b,\max(X_j)}$ has to be less than or equal to the length of the interval $[b, \max(X_j) + 1)$ minus 1.

Soundness. Suppose we pruned an interval $[b - 1, \max(X_j)]$ from $\mathcal{D}(X_i)$ after the processing of the variable X_j. This pruning is sound because the interval $[b, \max(X_j)+1)$ violates conditions (5)–(7).

Completeness. Suppose there exists an interval $[l, u]$ that violates conditions (5)–(7), so that $B^i_{1,u} + D^i_{l,u} > u - l$. However, the algorithm does not prune the upper bound of X_i to $l - 1$. Suppose that $l \in L$, $u \in U$. As the pointer b preserves the invariant, there are exactly $B^i_{1,u}$ available values between $[l, u + 1)$. Hence b points to l and $\max(X_i) \leq l - 1$.

Suppose that $l \notin L$, $u \in U$. We consider the step when the last pruning of the variable X_i occurs. Suppose we processed the variable X_j at this step. The pointer b stores $\max(X_i) + 1$. As b does not move backward in the following steps, we conclude that neither successors nor non-successors with domains that are contained inside the interval $[b, d]$ occur. Hence, $B^i_{1,u} + D^i_{l,u} = B^i_{1,u} + D^i_{\max(X_i)+1,u}$, $\max(X_j) \leq u$, $u \in U$, $l < \max(X_i)$. Hence $[l, u]$ is not a violated interval.

Complexity. At each iteration of the loop (line 1) the pointer b moves $O(d)$ times forward and $O(n)$ times backward. Due to a disjoint set data structure the total cost of the operations is $O(d)$, the functions $Union(v_1, v_2, T)$ and $Find(v_1, T)$ take $O(1)$ [22]. The total time complexity is $O(nd)$. □

We can construct a similar algorithm to Algorithm 1 to enforce conditions (8)–(10) and prune lower bounds.

Example 4. *Consider* $\textsc{AllDiffPrec}([X_1, X_2, X_3, X_4, X_5], \{(1,2),(1,3)\})$ *for Example 3. We show how our algorithm works on this example.*

We represent values in the disjoint set data structure T with circles. We use rectangles to denote sets of joint values. Initially, all values are in disjoint sets. If a variable X_i takes a value v we put the label X_i in the vth circle. Figure 1 shows five steps of the algorithm when processing the variable X_1 (line 1, $i = 1$).

Consider the first step. We set $v = 1$ as $\min(X_1)$ is 1. We join the values 1 and 2 into a single set (line 1). The pointer b is set to $\max(X_1) + 1 = 6$. Consider the second step. We process the variable X_2 which is a successor of X_1. As $\max(X_2) - \max(X_1) = 1$ we move b one available-value-step forward, $b = 7$. However, as X_2 is a successor, we move b available-value-step backward. Hence, $b = 6$. Consider the third step. We process X_3 which is a successor of X_1. As $\max(X_3) - \max(X_2) = 0$ we do not move b forward. However, as X_3 is a successor, we move b available-value-step backward, b is set to 5. Consider the fourth step. We process X_4 which is a non-successor of X_1. The value $\min(X_4)$ is 3. Hence, $v = 3$ and join 3 and 4 into a single set. Consider the fifth step. We process the variable X_5 which is a non-successor of X_1. The value $\min(X_5)$ is 4, as the value 3 is taken by X_4. As values 3 and 4 are in the same set, we do not move v and join $\{3,4\}$ and 5 into a set. Note that v and b are in the same set and we move b to the minimum element in this set. Hence, $b = 3$ and we prune $[3,5]$ from X_1.

The complexity of the algorithm can be reduced to $O(n^2)$. Let L be the set of domain lower bounds sorted in increasing order and let l_{i-1} and l_i be two consecutive values in that ordering. Following [6], we initialize the disjoint set data structures with only the elements in L. We assign a counter c_i to each element l_i initialized to the value $l_i - l_{i-1}$. Line 1 of the algorithm can be modified to decrement the counter of $\max(S)$. The algorithm calls the function *Union* only if the counter of $\max(S)$ is decremented to zero. The algorithm preserves its correctness and since there are at most n elements in L, the factor d in the complexity of the algorithm is replaced by n resulting in a running time complexity of $O(n^2)$.

7 Bounds Consistency Decomposition

We present a decomposition of the $\textsc{AllDiffPrec}$ constraint. For $1 \leq i \leq n$, $1 \leq l \leq u \leq d$ and $u - l < n$, we introduce Boolean variables B_{il} and A_{ilu} and post the following constraints:

$$B_{il} = 1 \iff X_i \leq l \tag{11}$$

$$A_{ilu} = 1 \iff (B_{i(l-1)} = 0 \wedge B_{iu} = 1) \tag{12}$$

$$\sum_{i=1}^{n} A_{ilu} \leq u - l + 1 \tag{13}$$

$$\sum_{j \in S(i)} A_{j,1,u} + \sum_{j \notin S(i)} A_{j,l,u} - B_{i(l-1)} \leq u - l \tag{14}$$

$$\sum_{j \in P(i)} A_{j,l,d} + \sum_{j \notin P(i)} A_{j,l,u} - (1 - B_{iu}) \leq u - l \tag{15}$$

$$\forall j \in S(i), X_i < X_j \tag{16}$$

$$\forall j \in P(i), X_j < X_i \tag{17}$$

Theorem 2. *Enforcing bounds consistency on constraints* (11) *and* (17) *enforces bounds consistency on the corresponding* ALLDIFFPREC *constraint in* $O(n^2d^2)$ *down a branch of the search tree.*

Proof: Constraints (11)–(13) enforce bounds consistency on the ALLDIFFERENT constraint. Constraints (16)–(17) enforce bounds consistency on the precedence constraints. Finally, conditions (8)–(10) are captured by constraints (14) and (15). By Theorem 1, enforcing BC on ALLDIFFERENT, precedence constraints and enforcing conditions (8)–(10) is sufficient to enforce bounds consistency on the ALLDIFFPREC constraint. The time complexity is dominated by $O(nd^2)$ linear inequality constraints (14)–(15). It takes $O(n)$ time to propagate a linear inequality constraint over $O(n)$ Boolean variables down a branch of the search tree. Hence, the total complexity is $O(n^2d^2)$. □

Note that the time complexity of decomposition contains a factor d that we cannot reduce as in the case of the conditions (5)–(10). As we compute the time complexity down a branch of a search tree we have to consider all possible $O(d^2)$ tight intervals that might emerge during the search.

8 Domain Consistency

Whilst enforcing bounds consistency on the ALLDIFFPREC constraint takes just low order polynomial time, enforcing domain consistency is intractable in general (assuming $P \neq NP$).

Theorem 2. *Enforcing domain consistency on* ALLDIFFPREC$([X_1, \ldots, X_n], E)$ *is NP-hard.*

Proof: We give a reduction from 3-SAT. Suppose we have a 3-SAT problem in N variables and M clauses. We consider an ALLDIFFPREC constraint on $2N + 3M$ variables. The first $2N$ variables represent a truth assignment. The next $3M$ variables represent the literals which satisfy each of the clauses. For $1 \leq i \leq N$, the variables X_{2i-1} and X_{2i} have domains $\{i, N + M + i\}$. $X_{2i-1} = i$ corresponds to the case in which we have a truth assignment that assigns x_i to false whilst $X_{2i} = i$ corresponds to the case in which we have a truth assignment that assigns x_i to true. The all different constraint ensures that only one of X_{2i-1} and X_{2i} can be assigned to i. Hence one of these two cases must hold. For $1 \leq i \leq M$, the variables X_{N+3i-2}, X_{N+3i-1} and X_{N+3i} represent the three literals in each clause. The values assigned to these variables will ensure that the truth assignment satisfies at least one literal in each clause. The domains of X_{N+3i-2}, X_{N+3i-1} and X_{N+3i} are $\{N + i, 2N + M + 2i, 2N + M + 2i - 1, \}$. $N + i$ will be the value used to indicate that the corresponding literal satisfies the clause. For each literal in a clause, we add an edge to E to ensure that there is an ordering constraint between one of the first $2N$ variables in the truth assignment section and the corresponding variable in the clause section. For example, suppose the ith clause is $x_j \vee \neg x_k \vee x_l$ then we add 3 edges to E to ensure: $X_{2j} < X_{N+3i-2}$, $X_{2k-1} < X_{N+3i-1}$, and $X_{2l} < X_{N+3i}$. The all different constraint ensures one of X_{N+3i-2}, X_{N+3i-1} and X_{N+3i} takes the smallest value $N + i$, and the ordering constraint then checks that the corresponding literal is set to true. By construction, the ALLDIFFPREC constraint has support iff there is a satisfying assignment to the original 3-SAT problem. □

Note that the proof uses a DAG defined by E that is flat, and does not contain any chains. Hence, enforcing domain consistency on ALLDIFFPREC remains NP-hard without chains of precedences. Note also that SAT remains NP-hard even if each clause has at most 3 literals, and each literal or negated literal occurs at most three times. Hence, a similar reduction shows that enforcing domain consistency on ALLDIFFPREC remains NP-hard even if the degree of nodes in E is at most 3 (that is, we have at most 3 precedence constraints on any variable).

9 Other Related Work

There have been many studies on propagation algorithms for a single ALLDIFFERENT constraint. A domain consistency algorithm that runs in $O(n^{2.5})$ was introduced in [2]. A range consistency algorithm was then proposed in [3] that runs in time $O(n^2)$. The focus was moved from range consistency to bound consistency with [4], who proposed a bounds consistency algorithm that runs in $O(n \log n)$. This was later improved further in [17] and then in [6].

Decompositions that achieve bounds consistency have been given for a number of global constraints. Relevant to this work, similar decompositions have been given for a single ALLDIFFERENT constraint [18], as well as for overlapping ALLDIFFERENT constraints [19]. These decompositions have the property that enforcing bound consistency on the decomposition achieves bounds consistency on the original global constraint.

A number of global constraints have been combined together and specialized propagators developed to deal with these conjunctions. For example, a global lexicographical ordering and sum constraint have been combined together [20]. As a second example, a generic method has been proposed for propagating combinations of the global lexicographical ordering and a family of globals including the REGULAR and SEQUENCE constraints [21].

10 Conclusions

We have proposed a new global constraint that combines together an ALLDIFFERENT constraint with precedence constraints that strictly order given pairs of variables. We gave an efficient propagation algorithm that enforces bounds consistency on this global constraint in $O(n^2)$ time, and showed how this propagator can be simulated with a simple decomposition extends the bounds consistency enforcing decomposition proposed for the ALLDIFFERENT constraint. Finally, we proved that enforcing domain consistency on this global constraint is NP-hard in general. There are many interesting future directions. We could, for example, study the convex hull of the ALLDIFFPREC constraint. Other interesting future work includes studying the combination of precedence constraints with generalizations of the ALLDIFFERENT constraint including the global cardinality constraint and the inter-distance constraint.

References

1. Lauriere, J.L.: ALICE: a language and a program for stating and solving combinatorial problems. Artificial Intelligence 10, 29–127 (1978)
2. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: Proceedings of the 12th National Conference on AI, Association for Advancement of Artificial Intelligence, pp. 362–367 (1994)
3. Leconte, M.: A bounds-based reduction scheme for constraints of difference. In: Proceedings of Second International Workshop on Constraint-based Reasoning, Constraint 1996 (1996)
4. Puget, J.: A fast algorithm for the bound consistency of alldiff constraints. In: 15th National Conference on Artificial Intelligence, Association for Advancement of Artificial Intelligence, pp. 359–366 (1998)
5. Mehlhorn, K., Thiel, S.: Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, p. 306. Springer, Heidelberg (2000)
6. Lopez-Ortiz, A., Quimper, C., Tromp, J., van Beek, P.: A fast and simple algorithm for bounds consistency of the alldifferent constraint. In: Proceedings of the 18th International Conference on AI, International Joint Conference on Artificial Intelligence (2003)
7. Stergiou, K., Walsh, T.: The difference all-difference makes. In: Proceedings of 16th IJCAI, International Joint Conference on Artificial Intelligence (1999)
8. Milano, M., Ottosson, G., Refalo, P., Thorsteinsson, E.: The role of integer programming techniques in constraint programming's global constraints. INFORMS Journal on Computing 14, 387–402 (2002)
9. Williams, H., Yan, H.: Representations of the all different predicate of constraint satisfaction in integer programming. INFORMS Journal on Computing 13, 96–103 (2001)
10. Walsh, T.: Constraint patterns. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 53–64. Springer, Heidelberg (2003)
11. Beldiceanu, N., Bourreau, E., Rivreau, D., Simonis, H.: Solving Resource-constrained Project Scheduling Problems with CHIP. In: 5th International Workshop on Project Management and Scheduling (PMS 1996), Poznan, pp. 35–38 (1996)
12. Simonis, H.: Building industrial applications with constraint programming. In: Comon, H., Marché, C., Treinen, R. (eds.) CCL 1999. LNCS, vol. 2002, p. 271. Springer, Heidelberg (2001)
13. Debruyne, R., Bessière, C.: Some practicable filtering techniques for the constraint satisfaction problem. In: Proceedings of the 15th IJCAI, International Joint Conference on Artificial Intelligence, pp. 412–417 (1997)
14. Garey, M., Johnson, D., Simons, B., Tarjan, R.: Scheduling unit-time tasks with arbitrary release times and deadlines. SIAM J. Comput. 10, 256–269 (1981)
15. Puget, J.-F.: Breaking all value symmetries in surjection problems. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 490–504. Springer, Heidelberg (2005)
16. Puget, J.F.: Symmetry in injective problems. Constraint Programming Letters 3, 1–20 (2007)
17. Mehlhorn, K., Thiel, S.: Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 306–319. Springer, Heidelberg (2002)

18. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.G., Walsh, T.: Decompositions of all different, global cardinality and related constraints. In: Proceedings of 21st IJCAI, International Joint Conference on Artificial Intelligence, pp. 419–424 (2009)
19. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.G., Walsh, T.: Propagating conjunctions of alldifferent constraints. In: Fox, M., Poole, D. (eds.) Proc. of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI 2010). AAAI Press, Menlo Park (2010)
20. Hnich, B., Kiziltan, Z., Walsh, T.: Combining symmetry breaking with other constraints: lexicographic ordering with sums. In: Proceedings of the 8th International Symposium on the Artificial Intelligence and Mathematics (2004)
21. 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)
22. Gabow, H., Tarjan, R.: A linear-time algorithm for a special case of disjoint set union. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing (STOC 1983), pp. 246–251. ACM, New York (1983)

Retail Store Workforce Scheduling by Expected Operating Income Maximization

Nicolas Chapados[1], Marc Joliveau[2], and Louis-Martin Rousseau[2]

[1] Université de Montréal
Montréal, Canada
[2] École Polytechnique de Montréal
Montréal, Canada

Abstract. We address the problem of retail store sales personnel scheduling by casting it in terms of an expected operating income maximization. In this framework, salespeople are no longer only responsible for operating costs, but also contribute to operating revenue. We model the marginal impact of an additional staff by making use of historical sales and payroll data, conditioned on a store-, date- and time-dependent traffic forecast. The expected revenue and its uncertainty are then fed into a constraint program which builds an operational schedule maximizing the expected operating income. A case study with a medium-sized retailer suggests that revenue increases of 7% and operating income increases of 3% are possible with the approach.

Keywords: Shift Scheduling, Statistical Forecasting, Retail, Constraint Programming.

1 Introduction

The objective of work schedule optimization is to allocate the right number of employees at the right time and the right place, to respond to an expected demand, while satisfying organizational constraints and—as much as possible—employee preferences [6]. The ability to forecast future demand constitutes one of the most important ingredients in the quality of the eventual schedule: an ideal schedule is capable of perfectly matching changes in employee demand with commensurate changes in employee supply. In the area of call center management, a coherent methodology has started to crystallize, combining statistical models for call arrival, duration and drop rate forecasting [2,3,12,11,13,1], together with a *dimensioning* aspect that determines the optimal number of employees to schedule in each daily segment to satisfy *quality of service* criteria [7].

The retail sector is another instance of prime economic significance where a combination of statistical modeling and mathematical programming techniques could yield substantial operational efficiencies. As with call centers, there is often considerable uncertainty in the traffic arrival process. However, and in contrast to call centers, the sales staff plays a role in the *generation of operating revenues*, and not only accrues to operating expenses. This means that we can view the introduction of an additional employee in terms of its marginal impact on operating income: the optimal number of employees at any point in time is that which balances revenue increase with additional payroll costs. This is the number that the scheduling process would aim to reach.

For the retail industry, these considerations have received only very little attention from the academic literature [5]. In particular, to the authors' knowledge, sales staff dimensioning based on formal business metrics has not been addressed. This paper introduces a statistical modeling and mathematical programming methodology that intends to bridge this gap. We start by introducing the overall approach, followed by an account of the individual statistical models that constitute the *forecasting* aspect of the methodology. Next, we provide results of a case study with a medium-sized clothing and apparel retailer showing the potential of the approach. We close with a preliminary overview of the constraint programming aspect of the methodology that would introduce the results of forecasting uncertainty into the creation of the final employee schedule, with the goal of increasing schedule robustness.

2 Forecasting in Retail Workforce Management

It is traditional to analyze retail sales in terms of decompositions along more fundamental quantities. A simple one—amenable to robust statistical modeling—is to write the expected sales during period t, S_t, in terms of the number items sold during the period and their average price:

$$\mathbb{E}[S_t \mid E_t, \tilde{T}_t, \mathbf{X}_t] = \mathbb{E}[V_t P_t \mid E_t, \tilde{T}_t, \mathbf{X}_t], \tag{1}$$

where V_t is the *sales volume*, i.e. the number of items sold during period t, and P_t is the average price of an individual item. We condition the expectation on three quantities: E_t is the number of *selling employees*,[1] which we posit has a causal effect on sales—this is the key element linking the forecasting model with workforce optimization. The quantity $\tilde{T}_t$ represents the store traffic during period t (the number of shoppers who can become potential buyers); it can either be the actual traffic, or a *traffic proxy*, somehow related to the traffic but that might be less precisely measured, such as the tick count from unidirectional people counters. Finally, $\mathbf{X}_t$ is a vector of other explanatory variables, such calendar regressors or indicators for special events. For brevity, we denote $\{E_t, \tilde{T}_t, \mathbf{X}_t\}$ by $\mathcal{I}_t$. Expression (1) can be rewritten as

$$\begin{aligned}\mathbb{E}[S_t \mid \mathcal{I}_t] &= \mathbb{E}[V_t \mathbb{E}[P_t \mid V_t, \mathcal{I}_t] \mid \mathcal{I}_t] \\ &= \sum_{v_t} P(V_t = v_t \mid \mathcal{I}_t)\, v_t\, \mathbb{E}[P_t \mid V_t, \mathcal{I}_t],\end{aligned} \tag{2}$$

where the second equation is written assuming that the number of items sold is a discrete quantity. This allows two separate models to be brought to bear: (i) a direct model of

[1] Defined, in a retail store context, as employees assigned to sales and direct customer assistance on the floor; it would exclude, for instance, cashiers that cannot have a direct impact in turning shoppers into buyers, or employees tasked with replenishing shelves if they do not actively seek to help out shoppers.

the conditional expectation of the item price given the number of items sold and other variables $\mathcal{I}_t$,[2] and (ii) a model of the conditional distribution of the number of items sold given only the variables $\mathcal{I}_t$. These models are detailed in section 3.

From this model, a *sales curve* can be established by varying the number of employees across a reasonable range to determine how the store sales are impacted. A further step, assuming homogeneity in staff cost and abilities, is to compute the functional dependency between the operating income (profit) during period t and the number of employees:

$$\mathrm{Op.\,Income}_t(E_t) = \mathbb{E}[S_t \mid E_t, \tilde{T}_t, \mathbf{X}_t] \;-\; E_t W_t, \tag{3}$$

This expression ignores other costs that are not directly affected by the number of employees, and assumes that the gross margin on items sold and an eventual commission rate have been absorbed into S_t. For simplicity, the dependency on other conditioning variables has been suppressed from the notation.

3 Statistical Forecasting Models

The above framework for expected operating income maximization can cast into three separate statistical forecasting problems: (i) traffic estimation, (ii) average price evaluation, (iii) sales volume forecasting. For the first part, we can rely on the extensive methodological toolset proposed for call center traffic forecasting (e.g. [2,3]), whereas we found that for the second, a linear regression model proves sufficient. We cover in more depth the third model.

At a typical small- to medium-sized retail store, the number of items sold during a daily interval (e.g. 30 minutes) will be a small integer. Due to the decomposition of eq. (2), one needs to estimate the entire distribution of sales volume, not only the expected value. Empirically, we find that a standard parametric forms, such as a conditional Poisson distribution, generally provide a bad fit to the realized distribution. We propose to use a more flexible framework, that of *ordinal regression* to estimate the volume distribution.

Ordinal regression models [8,9] attempt to fit measurements that are observed on a categorical *ordinal* scale. Let $Z \in \mathbb{R}$ be an unobserved variable and $V \in \{1, \ldots, K\}$ be defined by discretizing Z according to ordered cutoff points

$$-\infty = \zeta_0 < \zeta_1 < \cdots < \zeta_K = \infty.$$

We observe $V = k$ if and only if $\zeta_{k-1} < Z \leq \zeta_k, k = 1, \ldots, K$. The proportional odds model assumes that the cumulative distribution of V on the logistic scale is modeled by a linear combination of input variables $\mathbf{x}$, i.e.

$$\mathrm{logit} P(V \leq k \,|\, \mathbf{x}) = \mathrm{logit} P(Z \leq \zeta_k \,|\, \mathbf{x}) = \zeta_k - \boldsymbol{\theta}' \mathbf{x},$$

[2] Empirically, it is often noted that the average item price goes down with the number of items sold during a given period; this may be attributable to the propensity of shoppers to buy more items during sales, and the fact that for some store types, for instance clothing and apparel, additional items beyond the first few are often lower-priced accessories that complement main purchases.

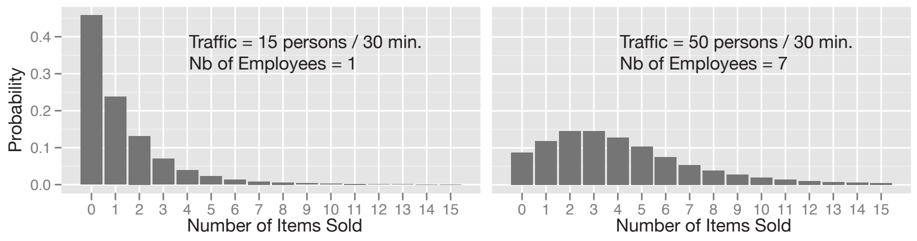


Fig. 1. Distributional sales volume forecast at two operation points. **Left:** With only a single employee in the store, and relatively low store traffic, the most likely number of items sold during a 30-minute period is zero (with 45% probability). **Right:** With seven employees and higher traffic, the distribution shifts rightward, exhibiting a mode between two and three items sold.

where $\boldsymbol{\theta}$ are regression coefficients and $\mathrm{logit}(p) \equiv \log \frac{p}{1-p}$. Model parameters $\{\zeta_i, \boldsymbol{\theta}\}$ can be estimated by maximum likelihood.

In our application, the variable V represents the sales volume during a sub-daily interval (e.g. a 30-minute period). Incorporating the number of employees and the store traffic among the inputs (along with calendar regressors for representing monthly, day-of-week and intraday seasonalities), we obtain a flexible distributional forecast of the sales volume. Figure 1 illustrates two different operation points for a store that was part of our case study (see section 4).

4 Case Study

We carried out a case study of the proposed methodology on data provided by a medium-sized chain of upscale clothing and apparel retail stores. Store locations are present in every major Canadian cities and obey normal retail opening hours and sales cycle, including increased holiday activity and the presence of occasional promotional events. A total of 16 months of historical data was used, covering the 2009–early-2010 period. Confidentiality agreements preclude from giving additional details.

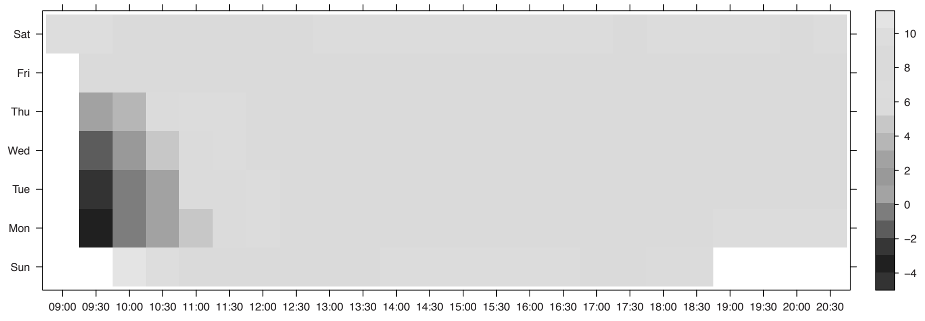


Fig. 2. Average revenue increase by day-of-week and time-of-day across all stores covered by the case study

We applied the forecasting methodology of section 3 to drive an operating income maximization based on eq. (3), for all store locations and time periods covered in the data. Using realized traffic, an improved staffing process would have resulted in an average sales increase of 7% across all stores, resulting in an operating income increase of 3%. Surprisingly, the model also indicates that stores were, on average, slightly *understaffed*, and that those top- and bottom-line increases would have resulted from increased staffing. Figure 2 shows how these revenue increases would have been distributed across the week, on average.

5 Constraint Programming for Schedule Construction

We are now at the stage of building feasible schedules that maximize a store net income, as opposed to the traditional minimization its labor expenses. Given a set of employees E, a set of time periods T and a set of *activities* $A = \{w, r, m, b\}$ (capturing the occupation of an employee, who is either *working*, *resting* at home, taking a *meal* or on *break*), we propose the following Constraint Programming model:

$$\max \sum_{t \in T} R^t_{y^w_t} - \sum_{a \in A} C^a_t y^a_t \tag{4}$$

$$s.t. \quad \texttt{Regular}(\varPi, x^e_{(\texttt{all}\, t \in T)}) \quad \forall e \in E \tag{5}$$

$$\texttt{GCC}(x_t^{(\texttt{all}\, e \in E)}, A, y_t^{(\texttt{all}\, a \in A)}) \quad \forall t \in T \tag{6}$$

$$y^a_t \in \{0, \ldots, |E|\} \quad \forall a \in A, t \in T \tag{7}$$

$$x^e_t \in A \quad \forall e \in E, t \in T \tag{8}$$

The model is based on two sets of variables: x^e_t specifies the *activity* (among those in A) performed by employee e at time t, and y^a_t represents the *number* of employees performing each activity a at time t. The objective function (4) maximizes the revenue forecasted at each time period given the chosen number of working employees (those forecasted values are stored in table R) minus the salary expenses associated to each type of activity (provided in table C). The values in table R are obtained from the statistical forecasting model (*cf.* eq. (2)) for *working* employees; we can assume that the revenue associated with other activities is zero. Constraints (5) enforce that each employee's schedules follows a certain number of regulations, which are captured in $\varPi$ (as proposed in [4,10]), constraints (6) link the x and y variables, while constraints (7)–(8) define the domain of all variables. We plan to implement this straighforward model in the coming weeks in order to present detailed results during the conference.

6 Conclusion

We have proposed a new methodology to forecast to expected revenue function associated to the presence of a given number of employees in a retail store. The use of a Constraint Program, as opposed to an Integer Program for instance, allows to easily incorporate this non-linear function into a shift scheduling model. The overall approach will thus allow to build schedules where employees are considered as a source a revenue rather than a sole expense.

References

1. Aldor-Noiman, S., Feigin, P.D., Mandelbaum, A.: Workload forecasting for a call center: Methodology and a case study. Annals of Applied Statistics 3(4), 1403–1447 (2009)
2. Avramidis, A.N., Deslauriers, A., L'Ecuyer, P.: Modeling daily arrivals to a telephone call center. Management Science 50(7), 896–908 (2004)
3. Brown, L., Gans, N., Mandelbaum, A., Sakov, A., Shen, H., Zeltyn, S., Zhao, L.: Statistical analysis of a telephone call center: A queueing-science perspective. Journal of the American Statistical Association 100(469), 36–50 (2005)
4. Demassey, S., Pesant, G., Rousseau, L.M.: Constraint programming based column generation for employee timetabling. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 140–154. Springer, Heidelberg (2005)
5. Ernst, A., Jiang, H., Krishnamoorthy, M., Owens, B., Sier, D.: An annotated bibliography of personnel scheduling and rostering. Annals of Operations Research 127, 21–144 (2004)
6. Ernst, A., Jiang, H., Krishnamoorthy, M., Sier, D.: Staff scheduling and rostering: A review of applications, methods and models. European Journal of Operational Research 153, 3–27 (2004)
7. Gans, N., Koole, G., Mandelbaum, A.: Telephone call centers: Tutorial, review, and research prospects. Manufacturing and Service Operations Management 5(2), 79–141 (2003)
8. McCullagh, P.: Regression models for ordinal data (with discussion). Journal of the Royal Statistical Society B 42(2), 109–142 (1980)
9. McCullagh, P., Nelder, J.A.: Generalized Linear Models, 2nd edn. Chapman & Hall, London (1989)
10. Quimper, C.G., Rousseau, L.M.: A large neighbourhood search approach to the multi-activity shift scheduling problem. Journal of Heuristics 16, 373–392 (2010)
11. Shen, H., Huang, J.Z.: Forecasting time series of inhomogeneous poisson processes with application to call center workforce management. Annals of Applied Statistics 2(2), 601–623 (2008)
12. Shen, H., Huang, J.Z.: Interday forecasting and intraday updating of call center arrivals. Manufacturing and Service Operations Management 10, 391–410 (2008)
13. Taylor, J.W.: A comparison of univariate time series methods for forecasting intraday arrivals at a call center. Management Science 54(2), 253–265 (2008)

Spatial and Objective Decompositions for Very Large SCAPs

Carleton Coffrin[1], Pascal Van Hentenryck[1], and Russell Bent[2]

[1] Brown University, Providence RI 02912, USA
[2] Los Alamos National Laboratory, Los Alamos NM 87545, USA

Abstract. This paper reconsiders the single commodity allocation problem (SCAP) for disaster recovery, which determines where and how to stockpile a commodity before a disaster and how to route the commodity once the disaster has hit. It shows how to scale the SCAP algorithm proposed in [1] to a geographical area with up to 1,000 storage locations (over a million decision variables). More precisely, the paper shows that spatial and objective decompositions are instrumental in solving SCAP problems at the state scale (e.g., for the state of Florida). The practical benefits of these decompositions are demonstrated on large-scale hurricane disaster scenarios generated by Los Alamos National Laboratory using state-of-the-art disaster simulation tools.

1 Background and Motivation

Every year, considerable human and monetary resources are spent to prepare for, and recover from, seasonal hurricanes. Existing procedures rely on the experience of policy makers but they are often ad-hoc and do not exploit recent progress in optimization to address natural disasters more effectively. Our earlier research [1] demonstrated the benefits of optimization technology to meet the population needs and to reduce storage and transportation costs by using a two-stage stochastic optimization problems with explicit scenarios generated by the National Hurricane Center (NHC) of the National Weather Service in the United States. However, only disasters with up to 100 storage locations (city scale) were considered, although large-scale planning may require as many as 1,000 storage locations (state scale). Indeed, the start-of-the-art algorithm in [1], and its underlying MIP model, have difficulties scaling to problems with 250 storage locations and runs out of memory on larger instances.

This paper shows how to scale the approach for disasters at the state scale using spatial and objective decompositions. The spatial decomposition performs a geographic clustering of the repositories and aggregates the flows across the clusters, thus reducing the number of decision variables considerably. The objective decomposition applies when the SCAP objective function is lexicographic: It separates the decisions taken for meeting the demands and reducing travel time. Experimental results demonstrate the benefits of both approaches on large and very large instances respectively. Given the sizes and complexity of the models considered here, our results are purely empirical: Their practicability is demonstrated by showing improvements on the practice in the field. Note also that the resulting approaches are now deployed and are activated each time a hurricane of category 3 or above threatens the coast of the United States.

The rest of the paper is organized as follows. Section 2 of this paper reviews related work. Section 3 presents a mathematical formulation of the SCAP and Section 4 reviews the approach presented in [1]. Sections 5 and 6 present the novel decomposition techniques. Section 7 reports the experimental results and Section 8 concludes the paper.

2 Previous Work

Humanitarian logistics has been investigated since the 1990s but has received increased attention in recent years due to the increase in major disasters [2,3,4,5]. Humanitarian logistics gives rise optimization problems combining aspects of inventory routing, supply chain management, warehouse location, and vehicle routing, creating novel challenges for existing technology [2,3]. They often combine some, or all, of the following features:

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) [6,7,8,9,1].
2. **Non-Standard Objective Functions** - A makespan time objective in VRPs [6,10,1] or equitability objectives [8].
3. **Arbitrary Side Constraints** - Limited resources, a fixed vehicle fleet [8,1], fixed latest delivery time [6,8], or a insufficient budget [7,11,1].
4. **Stochastic Aspects** - Disasters are inherently unpredictable. Preparations and recovery plans must be robust with respect to many scenarios [7,9,1].

Applications in humanitarian logistics are studied at a variety of scales in space and time. Some problems consider a global scale with time measured in days and weeks [7], while others focus on the minute-by-minute details of delivering supplies from local warehouses directly to the survivors [6,8,1,12]. This paper considers the so-called “last mile” of distribution which involves warehouse selection and customer delivery at the city and state scales.

Humanitarian logistics applications have been mostly formulated as mixed integer programming (MIP) models, which often do not scale to real-world instances [8,6,12]. Moreover, MIP solvers have been shown to have severe difficulties with some of their unique features even when problem sizes are small (e.g., minimizing the latest delivery time in VRPs [10]). Our earlier research [1] demonstrated that hybrid optimization and decomposition methods can yield high-quality solutions to such challenges and scale to real-world instances. This work extends those results and shows that spatial and objective decompositions provide significant scaleability. To the best of our knowledge, this is the first time that SCAPs with over 100 storage locations have been solved.

3 The Single Commodity Allocation Problem (SCAP)

In formalizing SCAPs, a populated area is represented as a graph $G = \langle N, E\rangle$ where N represents the locations of interest to the allocation problem: Sites

Given:
- Repositories: $i \in R$
  - Capacity: RC_i
  - Investment Cost: RI_i
  - Maintenance Cost: RM_i
- Vehicles: $i \in V$
  - Capacity: VC
  - Start Depot: H_i^+
  - End Depot: H_i^-
- Scenario Data: $i \in S$
  - Scenario Probability: P_i
  - Available Sites: $AR_i \subset R$
  - Site Demand: $D_{i,j \in R}$
  - 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 +
- $W_y * MAX_{i \in V}$ 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 commodities must be visited at least once

Fig. 1. The Single Commodity Allocation Problem Specification

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 and the graph edges, E, 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, 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 using weights W_x, W_y, and W_z. Note that the routing objective is to minimize the time of the last delivery, which is required by the Department of Homeland Security in the United States. Minimizing the time of the last delivery is a very difficult aspect of this problem as demonstrated in [10]. However, when solved with a combination of large neighborhood search and constraint programming, the stochastic storage decisions quickly become the most difficult aspect as the number of storage locations increases.

Side Constraints. Each repository $i \in R$ 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. Every repository can act as a warehouse and a customer and its role changes on a scenario-by-scenario basis depending on site availability and demands. Additionally, if a repository is acting as a warehouse for its own demands a vehicle must still visit that location before the stored commodities are available for consumption.

SCAPs also feature a fleet of V vehicles which are homogeneous in terms of their capacity VC. Each vehicle $i \in V$ has a unique starting depot H_i^+ and ending depot H_i^-. Unlike classic vehicle routing problems [13], 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 S different disaster scenarios. Scenario $i \in S$ has an associated probability P_i and specifies the set AR_i of sites which remain intact after the disaster. Moreover, scenario i specifies, for each repository $j \in R$, the demand D_{ij} and site-to-site travel times $T_{i,1..l,1..l}$ (where $l = |N|$) which capture the damages to the transportation infrastructure.

4 The Basic Approach

This section reviews the state-of-the-art algorithm for solving the SCAP problem [1]. This multi-stage algorithm, depicted in Figure 2, decomposes the storage, customer allocation, and routing decisions. The stages and the key decisions of each stage are as follows: (1) *Stochastic Storage*: Which repositories store the commodity and how much do they store? (2) *Customer allocation*: How is the stored commodity allocated to each customer? (3) *Repository routing*: For each repository, what is the best customer distribution plan? (4) *Fleet routing*: How to visit the repositories to minimize the time of the last delivery? The decisions of each stage are considered independently and use the optimization technique most appropriate to their nature. The first two stages are formulated as MIPs, the third stage is solved optimally using constraint programming (CP), and the fourth stage uses large neighborhood search (LNS) and CP.

This work only considers modifications to the Stochastic Storage Model (SSM) and uses identical algorithms for the customer allocation and routing aspects of the problem. Hence, we only review the SSM in detail. The SSM captures the cost and demand objectives precisely but approximates the routing aspects.

```
Multi-Stage-SCAP($SCAP\ \mathcal{G}$)
1  $\mathcal{D} \leftarrow StochasticStorageProblem(\mathcal{G})$
2  for $s \in S$
3  do $\mathcal{C} \leftarrow CustomerAllocationProblem(\mathcal{G}_s, \mathcal{D}_s)$
4      for $w \in R$
5      do $\mathcal{T} \leftarrow RepositoryPathRoutingProblem(\mathcal{G}_s, \mathcal{C}_w)$
6      $\mathcal{I} \leftarrow AggregateFleetRoutingProblem(\mathcal{G}_s, \mathcal{T})$
7      $\mathcal{F}_s \leftarrow PathBasedFleetRoutingProblem(\mathcal{G}_s, \mathcal{T}, \mathcal{I})$
8  return $\mathcal{F}$
```

Fig. 2. The Hybrid Stochastic Optimization Algorithm for Solving SCAPs

Variables:

$Stored_i \in (0, RC_i)$	- Units stored at repository i
$Open_i \in \{0, 1\}$	- Non-zero storage at repository i

Second stage variables for each scenario s:

$Outgoing_{si} \in (0, RC_i)$	- Total units shipped from repository i
$Incoming_{si} \in (0, D_{si})$	- Total units coming to repository i
$Unsatisfied_{si} \in (0, D_{si})$	- Demand not satisfied at repository i
$Sent_{sij} \in (0, RC_i)$	- Units shipped from repository i to repository j

Minimize:

$$W_x \sum_{s \in S} P_s \sum_{i \in R} Unsatisfied_{si} + W_z \sum_{i \in R} (RI_i\ Open_i + RM_i\ Stored_i) +$$
$$W_y \sum_{s \in S} P_s \sum_{i \in R} \sum_{j \in R} T_{sij}\ Sent_{sij} / VC$$

Subject To:

$$\sum_{i \in R} (RI_i\ Open_i + RM_i\ Stored_i) \leq B \qquad (1)$$
$$RC_i\ Open_i \geq Stored_i \qquad \forall i \in R \qquad (2)$$
$$Incoming_{si} + Unsatisfied_{si} = D_{si} \qquad \forall s \in S, i \in R \qquad (3)$$
$$Outgoing_{si} \leq Stored_i \qquad \forall s \in S, i \in R \qquad (4)$$
$$\sum_{j \in R} Sent_{sij} = Outgoing_{si} \qquad \forall s \in S, i \in R \qquad (5)$$
$$\sum_{j \in R} Sent_{sji} = Incoming_{si} \qquad \forall s \in S, i \in R \qquad (6)$$
$$Outgoing_{si} = 0 \qquad \forall s \in S, i \notin AR_s \qquad (7)$$

Fig. 3. The MIP Formulation for the Stochastic Storage Model (SSM)

In particular, the SSM only considers the time to move the commodity from the repository to a customer, not the maximum delivery time. 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

$$W_y \sum_{\langle s,d,q \rangle \in D} T_{sd} \frac{q}{VC}$$

Figure 3 presents the SSM formulation which scales well with the number of disaster scenarios since the number of integer variables only depends on the number of repositories. The meaning of the decision variables is explained in the figure. The objective function sums the unsatisfied demands for each scenario, the investment and maintenance costs, and the shipping costs for each scenario. The second stage costs are obviously multiplied by the scenario probabilities. Constraint (1) captures the budget constraint and constraint (2) ensures that a repository is open if it stores the commodity. Constraint (3) states for each scenario that the unsatisfied demand of a repository is the repository's demand minus the incoming supply (which can include local storage). Constraint (4) expresses that the supply shipped from repository i cannot exceed the amount of commodity stored at repository i. Constraints (5–6) connect the sent, incoming,

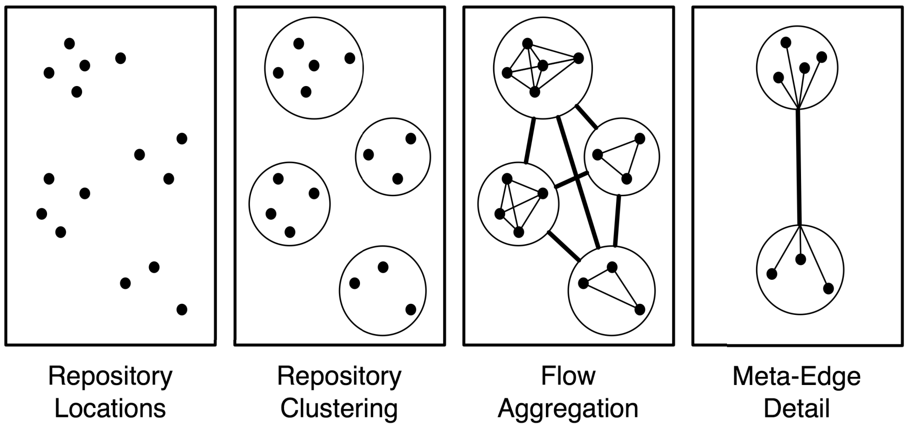


Fig. 4. Storage Clustering and Flow Aggregation

and outgoing variables and constraint (7) ensures that damaged repositories ship no commodity.

The experimental results in [1] indicate that the fleet-routing stage of the algorithm is the dominant factor in the algorithm runtime. However, for instances with more than 100 storage locations, the SSM quickly dominates the runtime (see Section 7 for numerical evidence). The next two sections present two alternative models for the stochastic storage problem that provide significant benefits for scalability. Both stochastic storage models rely on a key observation: In the baseline algorithm (Figure 2), a customer allocation is computed in the SSM and then recomputed in the customer allocation stage (once the uncertainty is revealed). This means, when a customer allocation stage is used, only the storage decisions are a necessary output of the SSM. The two new SSM models achieve better performance by approximating or ignoring the customer allocation in the first-stage storage problem.

5 Spatial Decomposition

In the SSM, the number of variables required for the customer allocation is quadratic in the number of repositories and multiplicative in the number of scenarios (i.e., $|S||R|^2$). The number of variables can easily be over one million when the number of repositories exceeds two hundred. Problems of this size can take up to 30 seconds to solve with a linear-programming solver and the resulting MIP can take several hours to complete. Our goal is thus to reduce the number of variables in the MIP solver significantly, without degrading the quality of the solutions too much.

The Aggregate Stochastic Storage Model (ASSM) is inspired by the structure of the solutions to the baseline algorithm. Customers are generally served by storage locations that are *nearby* and commodities are only transported over large distances in extreme circumstances. We exploit this observation by using a geographic clustering of the repositories. The clustering partitions the set of repositories R into C clusters and the repositories of a cluster $i \in C$ are denoted by CL_i. For a given clustering, we say that two repositories are *nearby* if they are in the same cluster; otherwise the repositories are *far away*. Nearby repositories have a tightly-coupled supply and demand relationship and hence the model needs as much flexibility as possible in mapping the supplies to the demands. This flexibility is achieved by allowing commodities to flow between each pair of repositories within a cluster (as was done in SSM). When repositories are far away, the precise supply and demand relationship is not as crucial since the warehouse to customer relationship is calculated in the customer allocation stage of the algorithm. As a result, it is sufficient to reason about the aggregate flow moving between two clusters at this stage of the algorithm. The aggregate flows are modeled by introducing *meta-edges* between each pair of clusters. If some demand from cluster $a \in C$ must be met by storage locations from cluster $b \in C$, then the sending repositories CL_b pool their commodities in a single *meta-edge* that flows from b to a. The receiving repositories CL_a then divide up the pooled commodities in the *meta-edge* from b to meet all of their demands. Additionally, if each *meta-edge* is assigned a travel cost, the *meta-edge* can approximate the number of trips required between two clusters by simply dividing the total amount of commodities by the vehicle capacity, as is the case for all the other flow edges. Figure 4 visually indicates how to generate the flow decision variables for the clustered problem and how commodities can flow on *meta-edges* between customers in different clusters.

As stated above, the number of variables in the SSM is quadratic in the number of repositories. Given a clustering $CL_{i \in C}$, the number of variables in the clustered storage model is (1) quadratic within each cluster (i.e., $\sum_{i \in C} |CL_i|^2$); (2) quadratic in the number of clusters, (i.e., $|C|^2$); (3) and linear in the repositories connections to the clusters (i.e., $2|R||C|$). The exact number of variables clearly depends on the considered clustering. However, given a specific number $|C|$ of clusters, a lower bound on the number of variables is obtained by dividing the repositories evenly among all the clusters, and the best possible variable reduction on a problem of size n with c clusters and s scenarios is $s\,(\frac{n^2}{c} + 2nc + c^2)$.

Given a clustering $CL_{i \in C}$ and cluster to cluster travel times CT_{scc} for each scenario, the ASSM is presented in Figure 5. The meaning of the decision variables is explained in the figure. The objective function has two terms for the delivery times, one for the shipping between repositories inside a cluster and one for shipping between clusters. Constraints (1–2) are the same as in the SSM model. Constraints (3–4) take into account the fact that the commodity can be shipped from repositories inside the clusters and from clusters. Constraints (5–6) aggregate the outgoing and incoming flow for a cluster, while constraints (7–8) express the damage constraints. Note that the array of variables $Sent_{sij}$ is sparse and only includes variables for repositories inside the same cluster (this is not reflected in the notations for simplicity).

Calculate:

$CS_c = \sum_{i \in CL_c} RC_i$ - Total storage in cluster c

$CD_{sc} = \sum_{i \in CL_c} D_{si}$ - Total demand in cluster c in scenario s

Variables:

$Stored_i \in (0, RC_i)$ - Units stored at repository i

$Open_i \in \{0, 1\}$ - Non-zero storage at repository i

Second stage variables for each scenario s:

$Unsatisfied_{si} \in (0, D_{si})$ - Unsatisfied demands at repository i

$Incoming_{sic} \in (0, D_{si})$ - Units shipped from cluster c to repository i

$Outgoing_{sic} \in (0, RC_i)$ - Units shipped from repository i to cluster c

$Sent_{sij} \in (0, min(RC_i, D_{sj}))$ - Units shipped from repository i to repository j

$Link_{scd} \in (0, min(CS_c, CD_{sd}))$ - Units sent from cluster c to cluster d

Minimize:

$$W_x \sum_{s \in S} P_s \sum_{i \in R} Unsatisfied_{si} + W_z \sum_{i \in R} (RI_i\ Open_i + RM_i\ Stored_i) +$$
$$W_y \sum_{s \in S} P_s \sum_{c \in C} \sum_{i \in CL_c} \sum_{j \in CL_c} T_{sij}\ Sent_{sij}/VC + W_y \sum_{s \in S} P_s \sum_{c \in C} \sum_{d \in C} CT_{scd}\ Link_{scd}/VC$$

Subject To:

$$\sum_{i \in R} (RI_i\ Open_i + RM_i\ Stored_i) \leq B \tag{1}$$
$$RC_i\ Open_i \geq Stored_i \quad \forall i \in R \tag{2}$$
$$\sum_{j \in R} Sent_{sji} + \sum_{c \in C} Incoming_{sic} + Unsatisfied_{si} = D_{si} \quad \forall s \in S, i \in R \tag{3}$$
$$\sum_{j \in R} Sent_{sij} + \sum_{c \in C} Outgoing_{sic} \leq Stored_i \quad \forall s \in S, i \in R \tag{4}$$
$$\sum_{i \in CL_c} Outgoing_{sid} = Link_{scd} \quad \forall s \in S, c \in C, d \in C \tag{5}$$
$$\sum_{i \in CL_d} Incoming_{sic} = Link_{scd} \quad \forall s \in S, c \in C, d \in C \tag{6}$$
$$Sent_{sij} = 0 \quad \forall s \in S, i \notin AR_s, j \in R \tag{7}$$
$$Outgoing_{sic} = 0 \quad \forall s \in S, i \notin AR_s, c \in C \tag{8}$$

Fig. 5. The MIP Formulation for the Aggregate Stochastic Storage Model (ASSM)

6 Objective Decomposition

The ASSM significantly decreases the number of variables but it still requires creating a quadratic number of variables for each cluster. Since this is multiplied by the number of scenarios, the resulting number of variables can still be prohibitive for very large instances. This section presents an objective decomposition which applies when the objective is lexicographic, i.e., when policy makers set the values of the weights such that $W_x \gg W_y \gg W_z$, which is often the case in practice. Let us contemplate what this means for the behavior of the model algorithm as the budget parameter B is varied. With a lexicographic objective, the model will first try to meet as many demands as possible. If the demands can be met, it will reduce delivery times until it cannot be reduced further or the

Calculate:

$SD_s = \sum_{i \in R} D_{si}$ - Total demand in scenario s

Variables:

$Stored_i \in (0, RC_i)$ - Units stored at repository i

$Open_i \in \{0, 1\}$ - Non-zero storage at repository i

$Used_s \in (0, SD_s)$ - Units used in scenario s

Minimize:

$$\sum_{s \in S} P_s \, (SD_s - Used_s)$$

Subject To:

$$RC_i \; Open_i \geq Stored_i \qquad \forall i \in R \quad (1)$$

$$\sum_{i \in AR_s} Stored_i \geq Used_s \qquad \forall s \in S \quad (2)$$

$$\sum_{i \in R} (RI_i \; Open_i + RM_i \; Stored_i) \leq B \quad (3)$$

Fig. 6. Phase 1 of the Lexicographic Stochastic Storage Model (LSSM-1)

budget is exhausted. As a result, the optimization with a lexicographic objective exhibits three phases as B increases. In the first phase, the satisfied demands, routing times, and costs increase steadily. In the second phase, the satisfied demands remain at a maximum, the routing times decrease, and the costs increase. In the last phase, the satisfied demands remain at a maximum, the routing times remain at a minimum, and the costs plateau even when B increases further. The experimental results from [1] confirm this behavior.

The Lexicographic Stochastic Storage Model (LSSM) assumes that the objective is lexicographic and solves the first phase with a much simpler (and faster) model. The goal of this phase is to use the available budget in order to meet the demands as best possible and it is solved with a two-stage stochastic allocation model that ignores the customer allocation and delivery time decisions. Since each scenario s has a total demand SD_s that must be met, it is sufficient to maximize the expected amount of demands that can be met, conditioned on the stochastic destruction of storage locations. Figure 6 presents such a model. The meaning of the decision variables is explained in the figure.

During the first phase, the model in Figure 6 behaves similarly to the SSM for a lexicographic objective. But the model does not address the delivery times at all, since this would create a prohibitive number of variables. To compensate for this limitation, we use a second phase whose idea can be summarized by the following greedy heuristic: "if all the demands can be met, use the remaining budget to store as much additional commodity as possible to reduce delivery times". This greedy heuristic is encapsulated in another MIP model (LSSM-2) presented in Figure 7. LSSM-2 utilizes the remaining budget while enforcing the decisions of the first step by setting the lower bound of the $StoredEx_i$ variables to the value of the $Stored_i$ variables computed by LSSM-1. This approximation is rather crude but produces good results on actual instances (see Figures 9 and 10 in Section 7). Our future work will investigate how to improve this formulation by taking account of customer locations, while still ignoring travel distances.

Variables:

$StoredEx_i \in (Stored_i, RC_i)$ - Units stored at repository i

$OpenEx_i \in \{0, 1\}$ - Non-zero storage at repository i

Maximize:

$$\sum_{i \in R} StoredEx_i$$

Subject To:

$$RC_i \; OpenEx_i \geq StoredEx_i \quad \forall i \in R \quad (1)$$

$$\sum_{i \in R} (RI_i \; OpenEx_i + RM_i \; StoredEx_i) \leq B \quad (2)$$

Fig. 7. Phase 2 of the Lexicographic Stochastic Storage Model (LSSM-2)

```
Greedy-Truck-Agent (GTA)()
1  while there exists some commodity to be picked up and 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 commodity from the nearest warehouse
5  goto final destination
```

Fig. 8. The Agent-based SCAP Algorithm Simulating the Practice in the Field

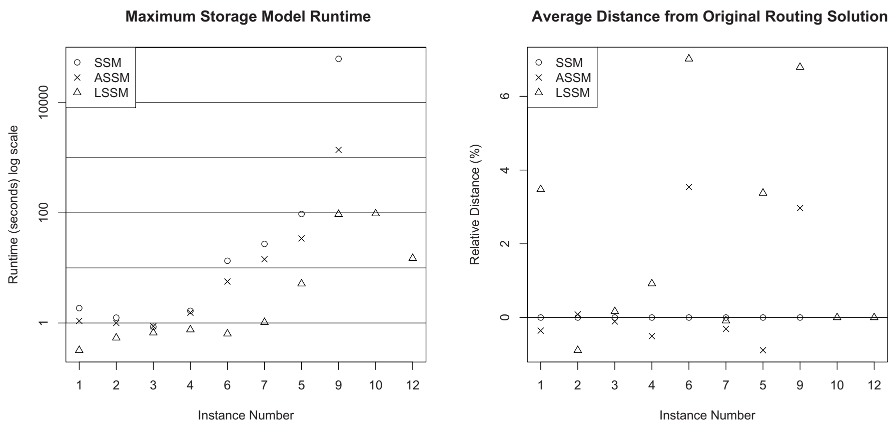


Fig. 9. Runtime and Quality Tradeoffs of SSM, ASSM, and LSSM

The resulting approach is less flexible than the SSM and ASSM approaches because it ignores the weighting factors W_x, W_y, and W_z. However, it produces a significant increase in performance by decreasing the number of decision variables from quadratic to linear. The asymptotic reduction is essential for scaling the algorithm to very large instances. Note that it is well-known in the goal-programming community that lexicographic multi-objective programs can be solved by a series of single-objective problems [14]. The sub-objectives are considered in descending importance and, at each step, one sub-objective is optimized in isolation and side constraints are added to enforce the optimization of the previous steps. Our decomposed storage model follows the same schema, except that the second step is necessarily approximated due to its size.

7 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 [15]. The problem sizes and algorithm parameters are presented in Table 1. The *Trip Lower Bounds* are simply the total amount of commodities that are shipped divided by the vehicle capacity. These values are included because they are a good metric for the routing difficulty of a benchmark. The amount of commodities that need to be moved can vary significantly from scenario to scenario. Therefore, we present both the smallest and the largest trip bounds across all the scenarios. Benchmarks 3 and 6 feature scenarios where the hurricane misses the region; this results in the minimum trip bound 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. The algorithm parameters include a runtime cap for the client allocation and fleet routing subproblems (defined in [1]), and the number of clusters that will be used in the ASSM model. All of the experimental results have fixed values of W_x, W_y, and W_z satisfying the field constraint $W_x \gg W_y \gg W_z$ and we vary the value of the budget B to evaluate the algorithm (as was done in [1]). The results are consistent across multiple weight configurations, although there are variations in the problem difficulties.

The Algorithm Implementation and the Baseline Algorithm. The algorithms were implemented in the COMET system [16] and the experiments were run on Intel Xeon CPU 2.80GHz machines running 64-bit Linux Debian. To validate our results, we compare our proposed storage models with those of the previous

Table 1. Benchmark Statistics and Algorithm Parameters (timeouts in seconds)

Benchmark	$\|R\|$	$\|V\|$	$\|S\|$	Min Trip Lower Bound	Max Trip Lower Bound	CA Timeout	Fleet Timeout	Clusters
BM1	25	4	3	6	27	30	10	4
BM2	25	5	3	60	84	30	20	4
BM3	25	5	3	0	109	30	20	4
BM4	30	5	3	35	109	30	20	4
BM5	100	20	3	82	223	90	200	4
BM6	25	5	18	0	140	30	20	4
BM7	30	10	18	7	23	30	20	4
BM9	250	10	18	7	23	250	90	10
BM10	500	20	18	13	45	-	180	-
BM12	1000	20	3	64	167	-	300	-

Table 2. Runtime Statistics in Seconds for the Baseline Algorithm (SSM)

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	89.89	21.90	39.96	13.01	**0.9293**	**0.4670**	9.257	0.1746	10.057	10.13
BM2	169.1	35.93	66.02	10.47	**0.5931**	**0.2832**	16.67	0.1956	19.26	20.00
BM3	98.58	14.51	61.07	13.79	**0.3557**	**0.1748**	7.225	0.1050	12.04	13.33
BM4	184.2	26.25	68.76	5.163	**0.8892**	**0.3940**	21.24	0.2075	19.58	20.00
BM5	1308	62.01	520.5	32.70	**46.70**	**21.31**	90.87	1.225	128.0	200.0
BM6	723.5	58.76	75.34	3.079	**5.165**	**3.076**	10.81	0.1281	13.35	15.56
BM7	832.0	97.05	75.13	13.31	**16.15**	**5.153**	5.500	0.4509	19.31	20.00
BM9	16123	13661	11108	13459	**10672**	**13458**	143.7	1.377	65.97	90.00

Table 3. Runtime Statistics in Seconds for the Aggregated Model (ASSM)

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	89.77	22.19	39.25	13.28	**0.5464**	**0.2389**	9.043	0.1791	10.04	10.12
BM2	169.4	35.91	65.93	10.49	**0.4846**	**0.1850**	16.81	0.2084	19.26	20.00
BM3	98.73	14.49	61.15	13.81	**0.3986**	**0.1609**	7.245	0.1092	12.05	13.33
BM4	182.8	24.28	69.74	3.822	**0.6950**	**0.3717**	20.88	0.2122	19.56	20.00
BM5	1266	70.41	487.4	35.18	**18.40**	**7.700**	90.88	0.8691	123.9	200.0
BM6	714.86	59.04	73.28	1.032	**3.130**	**1.041**	10.57	0.09642	13.27	15.56
BM7	823.6	98.79	67.95	12.99	**8.849**	**2.666**	5.479	0.4475	19.28	20.00
BM9	6377	803.6	1184	363.8	**747.7**	**363.5**	153.9	0.8491	66.72	90.00

study [1]. Our routing time results also include the solution from the Greedy Truck Agent (GTA) algorithm proposed in [1] which mimics how actual decision makers operate in the field and thus provides a sense of improvement and scale in solution quality. The agent-based algorithm uses the same storage model but builds a routing solution without any optimization. Each vehicle works independently to deliver as much commodity as possible using the algorithm in Figure 8.

Baseline Efficiency Results. Table 2 depicts the runtime results for the baseline algorithm in [1]. In particular, the table reports, in 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), customer allocation (CA), the repository routing (RR), the aggregate fleet routing (AFR), and the 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 for 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 problems with 100 repositories or less. However, benchmark 9 (250 repositories) clearly indicates that the runtime of the storage model has exploded and becomes the dominating factor of the algorithm. Benchmarks 10 and 12 are unsolvable due to memory issues: These models require over 3,000,000 variables.

Table 4. The Number of Variables in the SSM and ASSM

Benchmark	BM1	BM2	BM3	BM4	BM5	BM6	BM7	BM9	BM10	BM12
SSM	1875	1875	1875	2700	30000	11250	16200	1125000	-	-
ASSM	1116	1206	1248	1470	12246	7344	9576	237420	-	-
Lower Bound	1101	1101	1101	1443	9948	6606	8658	204300	-	-

Table 5. Degradation of the Routing Times Compared to the SSM Results

Benchmark	BM1	BM2	BM3	BM4	BM5	BM6	BM7	BM9	BM10	BM12
ASSM Change(%)	-0.356	0.0834	-0.108	-0.504	-0.887	3.54	-0.308	2.95	-	-
GTA Change(%)	56.4	43.1	73.7	52.0	64.0	92.2	55.5	259	-	-

ASSM Quality and Efficiency Results. Table 3 depicts the improvement of our ASSM for the SCAP algorithm. Observe in column *STO* that ASSM runs about twice as fast on benchmarks 1 through 7 and 14 times faster on benchmark 9. The clustering was obtained using ten samples of the k-means algorithm. The sample with the smallest mean sum is used in the clustered storage model. The distances between clusters are calculated on a scenario-by-scenario basis using the average distance between all pairs of points in each cluster. The runtime benefits of the clustering algorithm are largely due to the reduction in the number of variables in the model. Section 5 analyzed the variable reduction and pointed out that the reduction is tightly coupled with the clustering. Due to geographic considerations in these instances, the clustering exhibits great variation from instance to instance and it is important to report the actual reduction in problem size. Table 4 presents the number of variables of the SSM and the ASSM, as well as the lower bound on the number of variables. Observe that the benefits become more significant as the problem size grows and the runtime results confirm this.

Table 5 describes the relative changes in routing times when using ASSM instead of SSM. The quality degradation of the GTA algorithm is also presented to provide a sense of scale. Because the ASSM is a courser approximation of the travel times, some decrease in routing quality is expected. Fortunately, the reduction in quality is not significant and negligible when compared to GTA. It is also surprising that sometimes the clustering model improves the quality of the routing solution. This is a result of the fact that the travel time objective is only approximated in *all* of the stochastic storage models. When there are large distances between nodes, the ASSM meta-edges provide a more accurate estimate of the number of trips needed between two clusters. Unfortunately, the ASSM still suffers from the same memory issues as the SSM on very large instances and is unable to solve benchmarks 10 and 12.

LSSM Quality and Efficiency Results. Table 6 depicts the performance improvement of the LSSM and the consistent reduction in runtime of the storage model (*STO*), which runs about 1,000 times faster than the SSM on benchmark 9 on the most difficult configuration and 200 times faster on average. Benchmarks 10 and 12 are now in the scope of the solution method. Note that, due to the

Table 6. Runtime Statistics in Seconds for the Lexicographic Model (LSSM)

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	94.97	24.38	41.32	12.79	**0.11**	**0.06**	11.46	0.19	9.819	10.00
BM2	169.3	36.04	65.59	10.43	**0.16**	**0.08**	16.86	0.24	19.25	20.00
BM3	98.85	14.31	60.97	13.76	**0.17**	**0.09**	7.28	0.13	12.12	13.34
BM4	183.8	24.26	69.15	3.53	**0.25**	**0.19**	21.27	0.25	19.59	20.00
BM5	1240	69.07	468.6	33.70	**2.12**	**0.86**	90.94	0.81	120.6	200.0
BM6	719.8	56.91	70.46	0.08	**0.35**	**0.09**	11.06	0.09	13.23	15.56
BM7	810.5	100.8	59.54	13.51	**0.70**	**0.15**	5.40	0.37	19.17	20.00
BM9	5859	458.6	488.9	26.32	**53.84**	**27.30**	164.8	0.70	65.92	90.00
BM10	32048	1708	1921	108.9	**17.34**	**22.02**	1385*	8.66	175.7	180.0
BM12	18201	73.11	6146	46.54	**14.12**	**0.22**	5485*	14.28	227.3	300.0

Table 7. Degradation of the Routing Times Compared to the SSM Results

Benchmark	BM1	BM2	BM3	BM4	BM5	BM6	BM7	BM9	BM10	BM12
LSSM Change(%)	3.47	-0.891	0.165	0.919	3.38	7.02	-0.0884	6.77	-	-
GTA Change(%)	61.1	42.5	74.5	53.5	67.0	103	55.5	295	78.5	91.5

enormous size of benchmarks 10 and 12, the customer allocation stage does not return a feasible solution within 1000 seconds. To resolve this difficulty, we simply ignore the integer variables, solve the LP relaxation, and round the results. Table 6 indicates that solving the LP relaxation of these problems can take over ten minutes. The MIP solver is also terminated whenever the optimality gap is smaller than 0.05% to stabilize its runtime behavior on the largest instances.

Table 7 describes the relative change in routing times from the SSM. The quality degradation of the GTA algorithm is also presented to provide a sense of scale. Because the LSSM has no information about the travel time, some decrease in routing quality is expected. Again, it is impressive that the reduction is so small (especially when compared with the GTA algorithm). This model is the first that can solve benchmarks 10 and 12, and thus can report the relative improvements over the GTA algorithm. However, it cannot report the relative change compared to the SSM because that model cannot solve these instances. Some policy makers may be concerned by the 7.0% increase in delivery time in benchmark 6 and may prefer to use the SSM. However, some types of disasters require immediate response where every minute is valuable. In those extreme situations, the ASSM and LSSM provide a much faster alternative to the SSM. Our results thus allow policy makers to choose on a case-by-case basis which is preferable: A more immediate response or a higher quality solution.

The lack of information about travel time is an advantage for the memory usage of the LSSM. Only three pieces of the problem specification need to be considered, the repository information, scenario demands, and scenario damages. This resolves the memory issues faced by the other models since the travel times can be handled at the scenario level and not globally. This allows the LSSM to scale to the largest benchmarks. Figure 9 visually summarizes the runtime and quality tradeoffs of the SSM, ASSM, and LSSM. The left graph shows how the

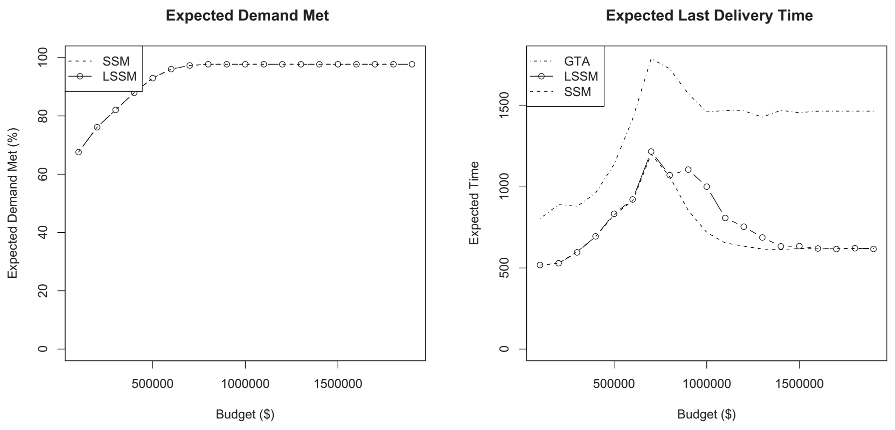


Fig. 10. Behavior of SSM and LSSM on Benchmark 6

runtime of all the storage models varies as the number of repositories increases. The logarithmic scale illustrates the significant time savings of ASSM and LSSM over SSM. The right graph shows the change in the routing quality when ASSM and LSSM are used instead of SSM. Despite the significant reduction in runtime, the degradation of solution quality is no more than 6% on average.

Behavioral Analysis of LSSM. The LSSM ignores the algorithm parameters W_x, W_y, and W_z, and implicitly assumes the field constraint $W_x \gg W_y \gg W_z$. Although the other storage models are more flexible in this regard, all the storage models are configured for this field constraint for the purpose of this study. This means that the storage decisions for the LSSM will be exactly the same as the SSM until all of the demands are met. Once all of the demands are satisfied, the LSSM will degrade because it cannot determine how to use additional funds to decrease the delivery time. However, as the budget increases, it will approach the same solution as the SSM because these solutions correspond to storing commodities at all of the repositories. Figure 10 presents the experimental results on benchmark 6 which exhibits this behavior most dramatically (other benchmarks are less pronounced and are 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. We can see that, as the satisfied demand increases, the routing times of both algorithms are identical until the total demand is met. At that point, the routing times diverge as the travel distance becomes an important factor in the objective, but they re-converge as the budget approaches its maximum and all of the repositories are storing commodities. These results confirm our behavioral expectation. The experimental results also demonstrate that the degradation of the decomposed model is not significant when compared to the choices made by the GTA algorithm, representing the practice in the field.

8 Conclusion

This paper studied the scalability of the SCAP problem in the field of humanitarian logistics. The SCAP models the strategic planning process for disaster recovery with stochastic last-mile distribution. The paper proposed two new stochastic storage models that produce high quality solutions to real-world benchmarks that were hitherto unsolvable. The algorithms use spatial and objective decompositions to exploit the problem structure and speedup stochastic storage decisions. The experimental results on water allocation benchmarks indicate that the algorithms are: (1) practical from a computational standpoint; (2) produce significant scalability over previous work; (3) deliver better performance than existing relief delivery procedures. This work is currently deployed at Los Alamos National Laboratory and is activated every time a hurricane of category 3 or above threatens the United States in order to aid federal organizations such as the Department of Energy and the Department of Homeland Security in preparing for, and responding to, disasters.

References

1. Van Hentenryck, P., Bent, R., Coffrin, C.: Strategic Planning for Disaster Recovery with Stochastic Last Mile Distribution. In: [17], pp. 318–333
2. Wassenhove, L.V.: Humanitarian aid logistics: supply chain management in high gear. Journal of the Operational Research Society 57(1), 475–489 (2006)
3. Beamon, B.: Humanitarian relief chains: Issues and challenges. In: 34th International Conference on Computers & Industrial Engineering, pp. 77–82 (2008)
4. United-States Government: The Federal Response to Hurricane Katrina: Lessons Learned (2006)
5. Fritz Institute.: Fritz Institute Website (2008), `http://www.fritzinstitute.org`
6. Barbarosoglu, G., Ozdamar, L., Cevik, A.: An interactive approach for hierarchical analysis of helicopter logistics in disaster relief operations. European Journal of Operational Research 140(1), 118–133 (2002)
7. Duran, S., Gutierrez, M., Keskinocak, P.: Pre-positioning of emergency items worldwide for care international. Interfaces (2009) (to appear)
8. Balcik, B., Beamon, B., Smilowitz, K.: Last mile distribution in humanitarian relief. Journal of Intelligent Transportation Systems 12(2), 51–63 (2008)
9. Gunnec, D., Salman, F.: A two-stage multi-criteria stochastic programming model for location of emergency response and distribution centers. In: INOC (2007)
10. Campbell, A.M., Vandenbussche, D., Hermann, W.: Routing for relief efforts. Transportation Science 42(2), 127–145 (2008)
11. Griffin, P., Scherrer, C., Swann, J.: Optimization of community health center locations and service offerings with statistical need estimation. IIE Transactions (2008)
12. Gunes, C., van Hoeve, W.J., Tayur, S.: Vehicle routing for food rescue programs: A comparison of different approaches. In: [17], pp. 176–180
13. Toth, P., Vigo, D.: The Vehicle Routing Problem. SIAM Monographs on Discrete Mathematics and Applications, Philadelphia, Pennsylvania (2001)

14. Ignizio, J.P.: A review of goal programming: A tool for multiobjective analysis. The Journal of the Operational Research Society 29(11), 1109–1119 (1978)
15. FEMA: FEMA HAZUS Overview (2010), `http://www.fema.gov/plan/prevent/hazus`
16. Dynadec, Inc.: Comet 2.1 User Manual (2009), `http://dynadec.com/`
17. Lodi, A., Milano, M., Toth, P. (eds.): CPAIOR 2010. LNCS, vol. 6140. Springer, Heidelberg (2010)

Upgrading Shortest Paths in Networks

Bistra Dilkina, Katherine J. Lai, and Carla P. Gomes

Computer Science Department, Cornell University
{bistra,klai,gomes}@cs.cornell.edu

Abstract. We introduce the Upgrading Shortest Paths Problem, a new combinatorial problem for improving network connectivity with a wide range of applications from multicast communication to wildlife habitat conservation. We define the problem in terms of a network with node delays and a set of node upgrade actions, each associated with a cost and an upgraded (reduced) node delay. The goal is to choose a set of upgrade actions to minimize the shortest delay paths between demand pairs of terminals in the network, subject to a budget constraint. We show that this problem is NP-hard. We describe and test two greedy algorithms against an exact algorithm on synthetic data and on a real-world instance from wildlife habitat conservation. While the greedy algorithms can do arbitrarily poorly in the worst case, they perform fairly well in practice. For most of the instances, taking the better of the two greedy solutions accomplishes within 5% of optimal on our benchmarks.

1 Introduction

Many applications in areas as diverse as VLSI circuit design, QoS routing, and traffic engineering involve designing networks under constrained shortest paths and budget limits. For example, in a transportation network, a key goal is to connect major cities via short routes to better serve the bulk of the traffic. In a multicast communication setting where a single node is broadcasting to a set of subscribers, it is important to minimize the latency, or the shortest path delays between the source node and all the subscribers. In wildlife conservation, our motivating application from computational sustainability [8], the landscape connectivity between important habitat patches is measured as the length of the shortest path in terms of landscape resistance to animal movement. Maintaining good landscape connectivity, i.e. short resistance paths, is key to resilient wildlife populations in an increasingly fragmented habitat matrix.

In this work, we introduce a new general network improvement problem relevant in such settings. The problem is defined with respect to a network with node delays where the delay between a pair of nodes is the shortest path delay in the network, while the overall delay of the network is measured as the average delay among a designated set of node pairs. Given a set of node upgrade actions with respective costs and upgraded node delays, we seek to choose the best possible upgrade strategy in terms of minimizing total upgrade cost and resulting overall network delay. We refer to this problem as the *Upgrading Shortest Paths Problem.* We consider two optimization settings. In the budget-constrained setting, the goal is to find an upgrade strategy such that the total upgrade cost does not exceed a given budget B, and the resulting upgraded network has minimum overall delay over all possible strategies which obey the budget constraint. On the other hand, in the delay-constrained setting, the goal is to find a minimum-cost set of nodes to be upgraded so that the overall delay in the resulting network meets a given bound D.

Some network improvement problems have been studied previously. In particular, most of the previous work has concentrated on the edge-delay variant where either edges can be upgraded directly, or nodes are to be upgraded, effectively upgrading all the edges incident to the upgraded nodes. Many of the studies assume a particular relationship between the delays and the upgraded delays. For example, if a node v is upgraded, then the delay of each edge incident to v reduces by a factor x where $0 \leq x < 1$, and if both endpoints of an edge are upgraded, then its delay reduces by a factor of x^2. Paik et al. [15] introduce several NP-hard network improvement problems under this upgrade model and unit costs, including the minimum-cost network improvement problem subject to a maximum delay constraint over all pairs of nodes in graph. Krumke [11] studies a similar network improvement problem but the constraint is on the total delay of the minimum spanning tree of the resulted upgraded network.

Although the edge-delay setting has been studied more than its node counterpart, placing the delays on the nodes can be more appropriate in certain applications. For example, in telecommunications, expensive equipment such as routers and switches are at the nodes of the underlying network. Unfortunately, while one can easily reduce an edge-weighted version to a node-weighted version, the reverse does not usually hold for undirected graphs, and hence it is desirable to work directly on node-weighted problems in undirected graphs. In this work, we address the more general node-weighted variant.

Landscape Connectivity

Although the network optimization problem considered here is very general, the main motivating application for our work is in Conservation Planning.

Habitat fragmentation is one of the principal threats to biodiversity. The focus of much ecology research is to quantify *landscape connectivity* [17], a measure of the degree to which the landscape facilitates or impedes movement among habitat patches. The landscape is represented as a set of small parcels or pixels, each of which has a resistance value that gives the species-specific cost of moving through particular landscape features. Resistance models are inferred by relating landscape characteristics to genetic distance between individuals at different locations [4] or to radio-collar movement data. Under the Least-Cost Path model, the connectivity between two designated habitat patches is measured as the length of the shortest resistance-weighted path between them [16].

Preserving and restoring connectivity for broad-scale ecological processes, such as dispersal and gene flow, has become a major conservation priority [3]. While conservation biology has historically set conservation objectives and plans irrespective of their cost, multiple studies in recent years have shown that systematic conservation planning is the right approach. It is possible to achieve conservation objectives at a fraction of the cost (or achieve higher targets for the same cost) if the conservation and management costs are formally considered at the outset of the planning process [13, 10]. Decision-support tools to design efficient budget-constrained conservation strategies are needed and yet still largely lacking.

By reducing the problem of maximizing landscape connectivity to the Upgrading Shortest Paths problem, we provide conservation planners with a tool to evaluate trade offs between costs and connectivity benefits as well as generate conservation plans with formal optimality guarantees. In particular, we can model the pixels or parcels of land as nodes in the graph, and edges are drawn between parcels that share boundaries. The resistance of each parcel is the corresponding node delay, and its upgraded delay is the predicted effective resistance of the parcel if it were under conservation management. Given pairs of important habitat patches (i.e. existing conserved areas or subpopulations), solving the combinatorial optimization problem designs a conservation strategy that maximizes the resulting landscape connectivity.

Recently, the related problem of Wildlife Corridor Design was studied in [2, 9, 5]. In the optimization model used for designing wildlife corridors, the goal is to maximize the total utility of the set of bought parcels while ensuring that the parcels connect a designated set of reserves and that the total cost does not exceed a specified budget. By enforcing connectivity of the purchased parcels, it in effect pessimistically assumes that any land parcel that is not bought for the wildlife corridor is no longer usable by the wildlife. In reality, choosing not to buy a piece of land may not significantly impact whether wildlife will still be able to use the land. In the landscape connectivity conservation problem discussed in this work, each land parcel may contribute to the connectivity of the terminals, whether or not it has been bought. The benefit of buying a piece of land is reflected by decreasing the land's effective resistance.

Our Contributions

In this paper, we introduce the Upgrading Shortest Paths problem, a new combinatorial problem for improving network connectivity in many real-world applications. We show that this problem is NP-hard. We give a formulation of the problem as a multicommodity flow mixed integer program for solving it to optimality, as well as two fast greedy algorithms. We tested these approaches on various synthetically generated planar graph instances and a real-world instance from conservation planning, and we found that our MIP formulation scales surprisingly well to instances with hundreds of nodes. While the greedy algorithms can perform arbitrarily badly even in planar graphs, they performed fairly well, coming within 5% of optimal on most of these test instances. One interesting phenomenon we observed is that the hardness of the instances is very much correlated with the nature and magnitude of the generated upgraded delay values for the nodes. Changes in node delays that were large in magnitude and varying greatly from node to node resulted in longer running times for the MIP and larger optimality gaps for the greedy algorithms.

The paper is organized as follows. First, we formally define the Upgrading Shortest Paths Problem. Second, we characterize its computational complexity. Third, we describe the three solution approaches. Finally, in the experiments section we study their typical case behavior on a synthetic dataset and present results for an instance derived from a real conservation planning setting.

2 The Upgrading Shortest Paths Problem

2.1 Problem Definition

We can define an instance of the decision version of the Upgrading Shortest Paths (USP) problem as follows.

Definition 1 (The Upgrading Shortest Paths Problem)

Given: *an undirected graph $G = (V, E)$, a set of terminal pairs $P \subseteq V \times V$, a cost function on the nodes $c : V \to \mathbb{R}^+$, a delay function $d : V \to \mathbb{R}^+$, a delay function $d' : V \to \mathbb{R}^+$ where $d'(v) \leq d(v)$ for all $v \in V$, a budget value $B \geq 0$, and a delay value $D \geq 0$.*

Find: *a set of nodes $V' \subseteq V$ such that $\sum_{v \in V'} c_v \leq B$, and the average shortest path for pairs in P is at most D when evaluated under the effective delays:*

$$\hat{d}(v) = \begin{cases} d'(v) & \text{if } v \in V' \\ d(v) & \text{otherwise} \end{cases} \tag{1}$$

For convenience, we also define T to be the set of all terminals, or set of nodes that appear in at least one pair $p \in P$. We can also define the following two variations of the USP problem.

Definition 2 (Budget-constrained USP Problem). *The delay value D is not given as an input, and the objective is to find a set of nodes $V' \subseteq V$ such that $\sum_{v \in V'} c_v \leq B$, and the average shortest path for pairs in P is minimized.*

Definition 3 (Delay-constrained USP Problem). *The budget value B is not given as an input, and the objective is to find a set of nodes $V' \subseteq V$ such that the average shortest path for pairs in P is at most D, and the total cost $\sum_{v \in V'} c_v$ is minimized.*

2.2 Computational Complexity

We now show that the two variants of the USP problem are NP-hard.

Theorem 1. *The budget-constrained Upgrading Shortest Paths Problem is NP-hard.*

Proof. To show that budget-constrained USP is NP-hard, we use a reduction from the knapsack problem which is NP-hard [7]. In a knapsack instance, we are given items indexed $\{1, \ldots, n\}$ with sizes $\{c_1, \ldots, c_n\}$ and values $\{d_1, \ldots, d_n\}$. The goal is to find some subset S that maximizes $\sum_{i \in S} d_i$ subject to the constraint that $\sum_{i \in S} c_i \leq B$, where B is the capacity of the knapsack.

Let G be a path graph with endpoints s and t, and n interior points v_i, one for each item in the knapsack instance. Note that the only shortest path between s and t is the entire path. We can now construct a USP instance with the graph G, one terminal pair (s, t), and a budget value of B. The nodes s and t have zero cost and delay. Each intermediate node v_i has cost c_i, delay d_i, and upgraded delay of 0. We can now map a set of items S exactly to the set of nodes in G that represent them, and this set of nodes has total cost $\sum_{i \in S} c_i$ and improves the total shortest path length by $\sum_{i \in S} d_i$ when bought. Since the optimal solution to the USP instance minimizes the shortest path length while satisfying the budget constraint, it in effect finds the set of nodes with the maximum total decrease in delay, thus exactly solving the knapsack instance. Therefore, the budget-constrained USP is NP-hard. Instances that involve more complicated graphs than a simple path graph can be viewed as having multiple knapsack instances to choose from, and instances with more than one demand pair may have overlapping knapsack instances where items are bought once but may contribute to multiple knapsacks.

Theorem 2. *The delay-constrained USP problem is NP-hard and can only be approximated within an $\Omega(\log |V|)$ factor unless P=NP.*

Proof. To show this result directly, we use a reduction from set cover which has the same hardness results [1]. In a set cover instance, we are given a universe of elements $U = \{1, \ldots, n\}$, a family $\mathcal{S}$ of candidate sets S_j each of which has a cost c_j. The goal is to find a family of sets $\mathcal{C} \subseteq \mathcal{S}$ such that they cover all of the elements, i.e. $\cup_{S \in \mathcal{C}} S = U$, and such that the total cost of the sets in $\mathcal{C}$ is minimized. We can construct a USP instance where there is a zero-delay node v_i for each element i in U, and the terminal pairs set P is composed of all pairs of these nodes. Each set S_j is similarly represented by a node u_j with delay 1, cost c_j and upgraded delay 0. Each node u_j is connected to the nodes v_i for which $i \in S_j$ as well as every other node u_k. Thus the shortest path delay between any two distinct nodes is at least 1, and upgrading a set of nodes u_j such that every node v_i is adjacent to at least one of these nodes decreases all of the delays to 0. Thus if we set the target average delay D to 0 and minimize the cost necessary to achieve this delay, we are exactly solving the set cover problem, and the cost of the nodes in the optimal solution is equal to the cost of the sets in the set cover instance.

3 Solution Methods

In this section, we present an exact method for solving the two variations of the USP problem using a MIP formulation as well as two greedy algorithms for the budget-constrained variation. To evaluate the quality of an approximation algorithm or a heuristic, it is standard to calculate the optimality gap of a solution by taking the difference between the approximate and exact solutions and dividing this result by the optimal value. However, this is a problematic and uninformative measure for the budget-constrained problem. For example, if the best upgraded shortest paths all have delay 0 and a heuristic finds a nearly-optimal solution of average delay ϵ, the heuristic still has an infinite optimality gap. For the sake of evaluating the performance of our solution methods, for the rest of this paper we will regard the objective function for the budget-constrained problem as maximizing the improvement in the average shortest path delay.

For all of the methods we present, we can prune the search space by eliminating all nodes $v \in V$ for which upgrading the node will never improve the delay between any terminal pair in P. To find these nodes, we first calculate single-source shortest paths from each terminal to the rest of the nodes in both the graph with no upgrades and the graph with all nodes upgraded. A node v will never improve the delay for a terminal pair p if the shortest path for p with no upgrades is shorter than the shortest possible path passing through v in the fully upgraded graph. If this condition holds for all terminal pairs, then under no upgrade strategy would v ever improve the objective, and hence we can safely prune it.

3.1 Mixed Integer Programming

We can solve the Upgrading Shortest Paths problem exactly by formulating it as a mixed integer program (MIP). We use a multicommodity flow formulation for computing the shortest delay path for each terminal pair $p = (s, t) \in P$. For this formulation, we transform the given undirected graph G to a directed graph G' where delays now appear on the edges instead of the nodes. Each node v in the graph is replaced by two nodes, the "in" node v^- and the "out" node v^+, that are then connected with two parallel edges directed from v^- to v^+. We refer to these edges as the "original node edge" e_v and the "upgraded node edge" e'_v, and their delays are set to the original and upgraded delays of the node v, respectively. Each undirected edge $\{u, v\}$ in the graph becomes two directed edges (u^+, v^-) and (v^+, u^-) with delay 0. See Figure 1 for an example of an edge in G and its corresponding subgraph in G'. We can now state the flow formulation for the constructed graph G'.

We now describe the variables used in our formulation:

- x_v: binary variable indicating whether node $v \in V$ is to be upgraded.
- *cost*: the total cost of all upgraded nodes.
- f_{pe}: continuous variable indicating the flow of commodity p on edge e, i.e. whether edge e is chosen to be on the shortest path for the terminal pair p.
- f_{pv}: continuous variable indicating the flow of commodity p on edge e_v. In an integral solution, this indicates whether the original node v is chosen to be on the shortest path for the terminal pair p.
- f'_{pv}: continuous variable indicating the flow of commodity p on edge e'_v. In an integral solution, this indicates whether the upgraded node v is chosen to be on the shortest path for the terminal pair p.
- $delay_p$: variable for the effective shortest path delay for terminal pair p.
- $avgdelay$: variable for the delay over all terminal pairs.

The full MIP for the budget-constrained problem is shown in Equations (2)-(18). The delay-constrained MIP is a simple modification of this MIP where the objective function minimizes *cost* instead, and Constraint (16) is replaced by the constraint $avgdelay \leq D$. We use Constraints (3)-(10) to model each terminal pair's shortest delay path as a multicommodity flow problem. For each terminal pair (s, t), Constraints (3)-(8) force the nodes s and t to be the source and sink of one unit of flow, respectively. We use $\delta^-(v^-)$ to indicate the set of incoming edges to the node v^- and $\delta^+(v^+)$ to indicate the set of outgoing edges from node v^+. The next constraints (9)-(10) enforce flow conservation through the rest of the nodes in the graph. The total delay for a terminal pair p is equal to the sum of delays of each edge e, scaled by the flow f_{pe} going through it (Constraint (13)).

Constraints (11)-(12) ensure that if a node v is chosen to be upgraded, only the upgraded node edge e'_v can carry flow; the original node edge e_v is not to be used. Similarly, if a node v is not chosen to be upgraded, only the original node edge e_v can be used to carry flow. Constraints (14) and (15) compute the total cost of the upgraded nodes and the average delay of all terminal pairs, respectively. Constraint (17) enforces that the upgrade decision variables are binary, and Constraint (18) enforces that the flow variables are all non-negative.

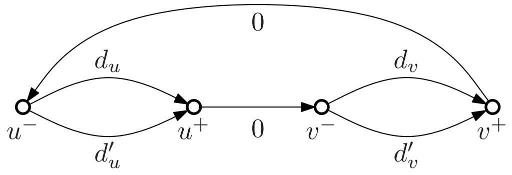


Fig. 1. The representation of nodes u, v, and an undirected edge between them in the new directed graph G'. The delay of each edge is labeled.

$$\begin{aligned}
&\min \quad avgdelay && && (2)\\
&\text{s.t. } f_{ps} + f'_{ps} = 1 && \forall p = (s,t) \in P && (3)\\
&\sum_{e \in \delta^-(s^-)} f_{pe} = 0 && \forall p = (s,t) \in P && (4)\\
&f_{ps} + f'_{ps} = \sum_{e \in \delta^+(s^+)} f_{pe} && \forall p = (s,t) \in P && (5)\\
&f_{pt} + f'_{pt} = 1 && \forall p = (s,t) \in P && (6)\\
&\sum_{e \in \delta^-(t^-)} f_{pe} = f_{pt} + f'_{pt} && \forall p = (s,t) \in P && (7)\\
&0 = \sum_{e \in \delta^+(t^+)} f_{pe} && \forall p = (s,t) \in P && (8)\\
&\sum_{e \in \delta^-(v^-)} f_{pe} = f_{pv} + f'_{pv} && \forall p = (s,t) \in P, \forall v \neq s,t \in V && (9)\\
&f_{pv} + f'_{pv} = \sum_{e \in \delta^+(v^+)} f_{pe} && \forall p = (s,t) \in P, \forall v \neq s,t \in V && (10)\\
&f'_{pv} \le x_v && \forall p = (s,t) \in P, \forall v \neq s,t \in V && (11)\\
&f_{pv} \le 1 - x_v && \forall p = (s,t) \in P, \forall v \neq s,t \in V && (12)\\
&delay_p = \sum_{v \in V} [d(v) f_{pv} + d'(v) f'_{pv}] && \forall p \in P && (13)\\
&cost = \sum_{v \in V} c(v) x_v && && (14)\\
&avgdelay = \frac{1}{|P|} \sum_{p \in P} delay_p && && (15)\\
&cost \le B && && (16)\\
&x_v \in \{0,1\} && \forall v \in V && (17)\\
&f_{pe}, f_{pv}, f'_{pv} \ge 0 && \forall p \in P, e \in E, v \in V && (18)
\end{aligned}$$

For both of the minimization problems, we implement pruning by setting $x_v = 0$ for all nodes $v \in V$ for which upgrading the node will never improve the delay between any terminal pair in P. For each node v that will never improve the delay for some particular pair $p \in P$, we add the constraint $f'_{pv} = 0$.

3.2 A Naive Greedy Algorithm

One naive approach for the budget-constrained USP problem is to take the current shortest paths between terminal pairs and upgrade them as much as possible. This cuts down on the search space a great deal. Intuitively, the greedy algorithm sorts the nodes in decreasing order by their heuristic *value* and attempts to upgrade each node in the list with what is left of the budget. To define the value for each node, the greedy algorithm first sets the values of every

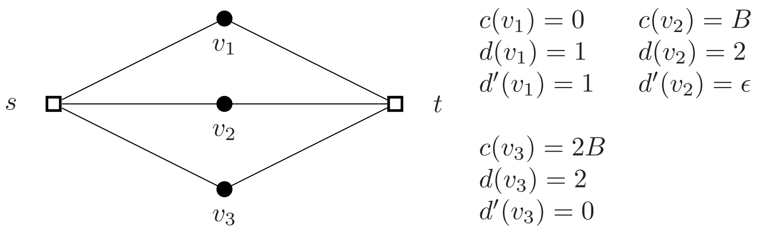


Fig. 2. In this example, the naive greedy algorithm will only examine the nodes v_1 and v_3 since they are part of the current and best possible paths. Under a budget constraint of B, the naive greedy algorithm will make no improvement to the delay even though upgrading v_2 would decrease it to an arbitrarily small $\epsilon > 0$.

node to 0. Then, for each pair of terminals $p = (s, t)$, it adds $(d(v) - d'(v))/c(v)$ to the value of each node v on the shortest path between s and t. The total running time is that of running Dijkstra's shortest paths algorithm from each terminal, sorting the eligible nodes, and adding them in linear time. In total this algorithm takes $O(|T|(|E| + |V| \log |V|))$ time, where $T = \{t : \exists(s, t) \in P\}$ is the set of terminals that show up in some terminal pair.

A similar alternative to the way this heuristic cuts down on its search space is to consider only the nodes on the shortest paths that would exist if the entire graph were upgraded; these are the best possible paths if the budget were infinite. It is a simple matter to run both heuristics and take the better result; we will call this combined approach the *Naive Greedy* algorithm. As with many heuristics, this algorithm does not have a provable guarantee. In fact, it can do arbitrarily poorly as shown in the example in Figure 2.

3.3 An Iterative Greedy Algorithm

Further improvement on the naive greedy algorithm can be gained by considering nodes that are not considered by the naive greedy algorithm. After pruning and eliminating the nodes that could never improve the delay for any terminal pair (as described earlier in Section 3.1), we again assign a heuristic value to each eligible node. Here, we redefine a node's value to be the change in the average shortest path delay if we were to upgrade that one node, divided by its cost. The new greedy algorithm iteratively upgrades the node with the highest value and recomputes the remaining nodes' values before upgrading the next node. After the algorithm exhausts the budget, it is possible that some of the nodes it chose to upgrade no longer improve the solution. As such, it removes these unnecessary nodes from the solution and starts over again but with the current set of upgraded nodes and the leftover budget. We can repeat this process until there is no longer any improvement made on the objective function, or we can set a limit to the number of times that this is run. We will call this the *Iterative Greedy* algorithm, and more detailed pseudocode is outlined in Algorithm 1.

The time complexity of this greedy algorithm is dominated by calls to the function `CalcShPaths()`. Calculating single-source shortest paths for all of the

Algorithm 1. The Iterative Greedy Algorithm

```
Input: input of the USP instance, a parameter NumIters, and the subroutines
  – CalcShPaths($G$, $T$, $d$, $d'$, V'): shortest path delays from the nodes $t \in T$ to all
    other nodes $v \in V$ assuming the nodes in V' have been upgraded
  – CalcAvgDelay(pathDists, $P$): average delay for pairs in $P$
  – CalcImpr(pathDists, $P$, $v$): improvement in average delay for $P$ if $v$ is upgraded
Output: A set V' $\subseteq V$ to upgrade
1  V' $\leftarrow \emptyset$
2  spent $\leftarrow 0$
3  for $i \leftarrow 1$ to NumIters do
4      Q $\leftarrow V -$ V'
5      pathDists $\leftarrow$ CalcShPaths($G$, $T$, $d$, $d'$, V')
6      startAvgSP $\leftarrow$ CalcAvgDelay(pathDists, $P$)
7      while Q $\neq \emptyset$ do
8          foreach $v \in$ Q do
9              if spent $+ c(v) \leq B$ then value($v$) $\leftarrow$ CalcImpr(pathDists,$P$,$v$)$/c(v)$
10             else Q $\leftarrow$ Q $- \{v\}$
11         if Q $\neq \emptyset$ then
12             Let $v \in$ Q be the node for which value($v$) is maximum
13             V' $\leftarrow$ V' $+ \{v\}$
14             Q $\leftarrow$ Q $- \{v\}$
15             spent $\leftarrow$ spent $+ c(v)$
16             pathDists $\leftarrow$ CalcShPaths($G$, $T$, $d$, $d'$, V')
17     deleted $\leftarrow$ false
18     avgSP $\leftarrow$ CalcAvgDelay(pathDists, $P$)
19     foreach $v \in$ V' do
20         if avgSP = CalcAvgDelay(CalcShPaths($G$, $T$, $d$, $d'$, V' $- \{v\}$), $P$)
           then
21             V' $\leftarrow$ V' $- \{v\}$
22             spent $\leftarrow$ spent $- c(v)$
23             deleted $\leftarrow$ true
24     if deleted = false or avgSP = startAvgSP then return V'
25 return V'
```

terminals is implemented by running Dijkstra's algorithm $|T|$ times using Fibonacci heaps, which takes a total of $O(|T|(|E| + |V|\log|V|))$ time. In each iteration of the greedy algorithm, i.e. each iteration of the for loop starting at Line 3, this function is called $O(|V'|)$ times. Having computed the shortest paths, the function `CalcAvgDelay()` takes $O(|P|)$ time to look up the shortest path delay for each terminal pair. The function `CalcImpr()` needs to calculate the upgraded delays of the shortest paths that must pass through v. This can be done in $O(1)$ time for each terminal pair (for a total of $O(|P|)$ time for the function) by adding up the shortest path delays from the node v to the two terminals, removing the delay $d(v)$ from both paths, and adding the upgraded delay $d'(v)$. Since the running times of these other functions and the various loops are all dominated by the running time of the shortest-paths computations, the total running time for each iteration of the greedy algorithm can be bounded above by $O(|V||T|(|E| + |V|\log|V|))$.

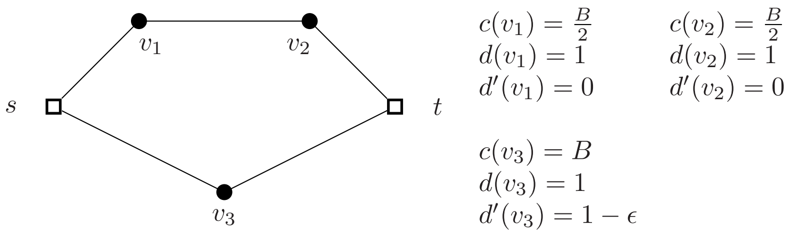


Fig. 3. In this example, there is one terminal pair $p = (s, t)$, and the initial shortest path length is 1 via node v_3. Under a budget constraint of B, the greedy algorithm ignores both nodes v_1 and v_2 in favor of v_3.

In terms of performance guarantees, this greedy algorithm can also perform arbitrarily poorly. Greedy algorithms occasionally have provable guarantees in some problems such as maximizing submodular functions [14, 6, 12]. Submodular functions capture settings where the payoff for choosing some set of items exhibits diminishing returns, i.e. they are functions f that satisfy $f(A \cup B) \leq f(A) + f(B) - f(A \cap B)$. In a recent result, Lin and Bilmes [12] show that it is possible to use a modification of our iterative greedy approach to get a constant approximation when maximizing a submodular function subject to a budget constraint. Their algorithm takes the better of the two solutions of a) performing the greedy algorithm and b) choosing the single element x for which $c(x) \leq B$ and $f(x)$ is maximized. Unfortunately, their small modification fails to work here because the budget-constrained USP problem, when posed as a maximization problem under the choice of V', is not submodular. We give an example in Figure 3. Buying either of the nodes v_1 or v_2 alone does not decrease the shortest path length, but buying both of them decreases it by 1 (and is in fact the optimal solution). The iterative greedy algorithm would preferentially add nodes that immediately improve the objective function, so it would choose v_3 and use up the entire budget in the process. Choosing the one element that improves the objective the most also gives the same result. Since this is only an improvement of ϵ to the optimal improvement of 1, the performance of the greedy algorithm can be made arbitrarily worse by setting ϵ to be an arbitrarily small but positive value.

4 Experimental Results

We implemented and tested the greedy algorithms against the exact solution provided by using a MIP solver. To test these algorithms, we created synthetic problem instances on square grid graphs. The initial delay and cost of each node was chosen uniformly at random from the range [50, 1000]. Three terminal pairs were chosen from a set of four randomly chosen terminals (two set in opposite corners of the graph) by taking the edges in the all-pairs shortest-paths minimum spanning tree on the terminals. Three approaches were used to model the upgraded delay function:

scaled. Each upgraded delay value is some scalar factor times the original delay value, i.e. $d'(v) = cd(v)$ for some $c \in [0, 1]$.

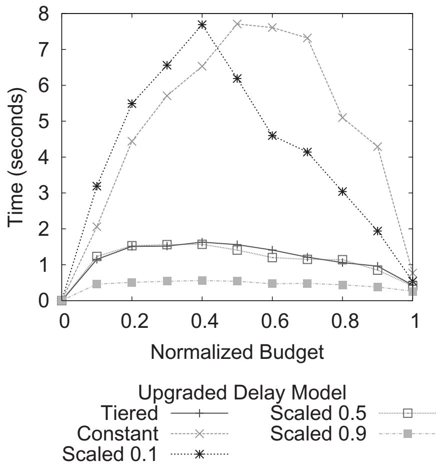
constant. Each upgraded delay is equal to the same constant 50.

tiered. For nodes with delay value $d(v)$ in the range (500, 1000), the upgraded delay value is 500, and those with $d(v)$ in the range [100, 500], $d'(v)$ is set to 75.

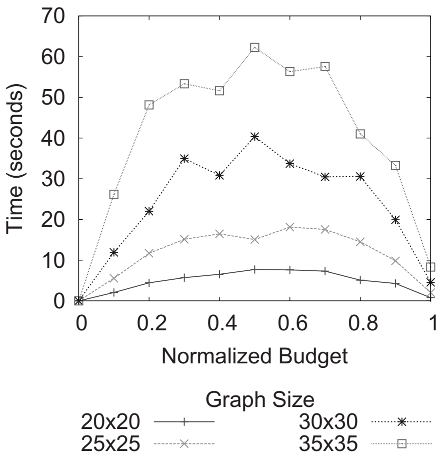
We first tested the performance of solving the MIP encoding exactly by using IBM ILOG CPLEX 11 on 100 instances of 20 by 20 grid graphs (400 nodes) with pruning as described earlier. We varied the budget value B between 0 and $B_{\max}$, the total budget necessary to achieve the shortest possible delays. Each value $B_{\max}$ is specific to the instance and is calculated by solving the budget minimization MIP. As shown in Figure 4a, the problem exhibits easy-hard-easy behavior as the budget is increased. It is notable on these instances that the easy-hard-easy trend is most pronounced for the **constant** model and the **scaled** model for $c = 0.1$. As a general trend, instances where the change in node delays are larger and vary a great deal are harder than instances where the new node delays represent very little change. The MIP scaled surprisingly well for larger grid graphs, as can be seen in Figure 4b.

4.1 Greedy Algorithm Performance

The naive greedy algorithm does not always perform very well, though it sometimes outperforms the iterative greedy algorithm when the budget nears $B_{\max}$. The iterative greedy algorithm performed very well on these randomly generated



(a) All models



(b) **constant**

Fig. 4. (a) Median MIP running times for different upgraded delay models on 100 instances of 20x20 grid graphs with 3 terminal pairs. The budget ranges from 0 to 100% of the maximum budget necessary for the instance. (b) Median MIP running times for different grid graph sizes under the **constant** upgraded delay model.

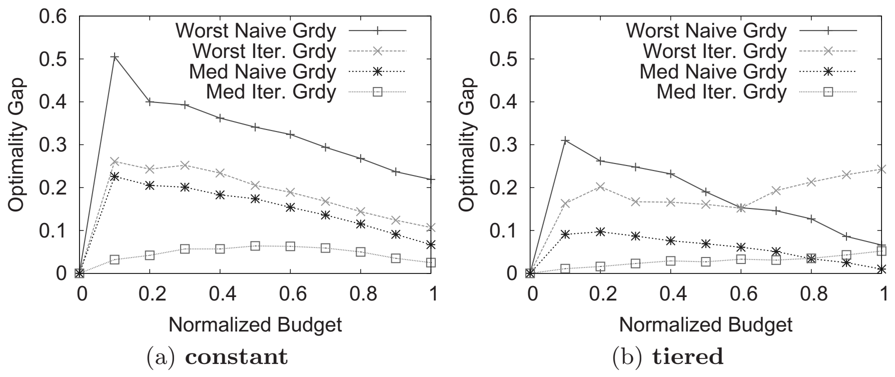


Fig. 5. The worst and median performances of the greedy algorithms are given from running on 100 instances of 20x20 grid graphs on two of the upgraded delay models

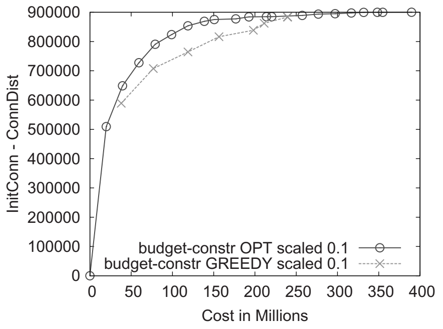
instances. Both the median and mean performance of the algorithm on 100 samples were within 5% of optimal for all of the upgraded delay models. As expected, there were occasionally instances where the algorithm did poorly, though given the nature of our synthetic instances, this did not occur for many instances, nor was the result ever found to be worse than 60% of optimal. Figure 5 shows the average and worst case performances of the greedy algorithm on our data set.

By the heuristic nature of the greedy algorithms, both of the greedy implementations were very fast. In our experiments, the naive greedy algorithm finished in at most 0.02 seconds, and the iterative greedy algorithm finished in at most 0.5 seconds.

4.2 Results on Grizzly Bear Data

We apply our solution approaches to data derived from a real conservation setting. We use the data for the grizzly bear corridor design problem studied in [2]. The goal in this work was to ensure connectivity between three major national conservation parks with existing grizzly populations. The data was compiled by Dr. Jordan Suter and is given in terms of habitat suitability, or *utility*, values and costs for different land parcels in the geographical area surrounding the three wildlife reserves. For each land parcel, we generated landscape resistance values that were inversely correlated with their utility values. In many ecological studies, habitat suitability and resistance are treated as complementary values. Hence, we compute the resistance of nodes on the same scale as the utilities $resist(v) = \min_{u \in V} util(u) + \max_{u \in V} util(u) - util(v)$.

At a 10 by 10 km pixel resolution, the resulting network has 3299 parcels (nodes) and 3 terminal pairs (connecting the three reserves). We solved the 10km grizzly instance for different resistance models for both the budget-constrained and delay-constrained formulations. Figure 6 presents results from the **scaled** model for $c = 0.1$. The other resistance models behaved qualitatively similarly, although this was the most computationally demanding setting for the MIP formulation. The graph on the left plots the Pareto frontier between cost and



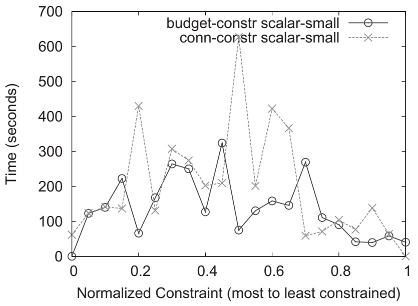


Fig. 6. Results for the 10km grizzly instance for the scaled 0.1 resistance model. Left plot shows the tradeoff between the cost spend and the improvement in delay achieved by the optimal as well as the iterative greedy solutions. Right plot shows the running time of both the budget-constrained and delay-constrained formulations as a function of the tightness of the respective constraint.

delay, i.e. the tradeoff curve of improvement in average terminal pair delay as we increase the budget allowed for upgrades. Such analysis can provide important insight for conservation planning as a small fraction of the maximum budget is enough to achieve more than half of the connectivity improvement. The graph on the right plots the computation time versus the normalized constraint for both constrained variants of the problem. For all resistance models, the minimum cost delay-constrained problem usually required more time to solve to optimality than the minimum delay budget-constrained variant. While the wildlife corridor design problem cannot be solved to optimality within hours for this instance [5], the respective Upgrading Shortest Paths problem on the same graph is solvable to optimality in a practical time frame.

5 Conclusions and Future Work

In this paper, we introduced the USP problem, a new NP-hard combinatorial problem for improving network connectivity in real-world applications. We also provided a MIP formulation that scaled very well with the size of the graph. This was a surprisingly positive result given the bad scaling behavior of MIP formulations for many other combinatorial network design problems. This is also a very practical result because in the context of conservation planning, problem instances are usually quite large, on the order of thousands of nodes. The greedy algorithms provided very high quality solutions in practice and can be used for extremely large instances.

The introduction of the USP problem also opens up several interesting open problems. Our greedy algorithms perform well but can do arbitrarily poorly. An open research direction is to design approximation algorithms with provable performance guarantees. It would also be interesting to study exactly why this MIP scales so well as compared to other combinatorial network design problems.

In the context of conservation planning, our model can also be generalized to capture other features such as multiple species of wildlife that have different resistance values for the same land parcel. We can also study the generalized model where each node can have different upgraded delay values available at different costs. This would model more fine-tuned applications where there is a discrete spectrum of actions that can be taken to decrease the inherent delay or resistance of a node.

Acknowledgments

This research was supported by NSF Expeditions in Computing award for Computational Sustainability (Grant 0832782) and by the USDA Forest Service, Rocky Mountain Research Station (10-JV-11221635-24). Katherine J. Lai is supported by a Graduate Research Fellowship from the National Science Foundation.

References

1. Alon, N., Moshkovitz, D., Safra, S.: Algorithmic construction of sets for k-restrictions. ACM Trans. Algorithms 2, 153–177 (2006)
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. Crooks, K.R., Sanjayan, M. (eds.): Connectivity Conservation. Cambridge University Press, Cambridge (2006)
4. Cushman, S.A., McKelvey, K.S., Schwartz, M.K.: Use of empirically derived source-destination models to map regional conservation corridors.. Conservation Biology 23(2), 368–376 (2009)
5. Dilkina, B., Gomes, C.P.: Solving connected subgraph problems in wildlife conservation. In: Lodi, A., Milano, M., Toth, P. (eds.) CPAIOR 2010. LNCS, vol. 6140, pp. 102–116. Springer, Heidelberg (2010)
6. Feige, U., Mirrokni, V.S.: Maximizing non-monotone submodular functions. In: FOCS, pp. 461–471 (2007)
7. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York (1979)
8. Gomes, C.P.: Computational Sustainability: Computational methods for a sustainable environment, economy, and society. The Bridge, National Academy of Engineering 39(4) (Winter 2009)
9. Gomes, C.P., van Hoeve, W.-J., Sabharwal, A.: Connections in networks: A hybrid approach. In: Trick, M.A. (ed.) CPAIOR 2008. LNCS, vol. 5015, pp. 303–307. Springer, Heidelberg (2008)
10. Joseph, L.N., Maloney, R.F., Possingham, H.P.: Optimal allocation of resources among threatened species: a project prioritization protocol. Conservation Biology 23(2), 328–338 (2009)
11. Krumke, S.: Improving Minimum Cost Spanning Trees by Upgrading Nodes. Journal of Algorithms 33(1), 92–111 (1999)

12. Lin, H., Bilmes, J.: Multi-document summarization via budgeted maximization of submodular functions. In: Human Language Technology Conference, NAACL/HLT (2010)
13. Naidoo, R., Balmford, A., Ferraro, P.J., Polasky, S., Ricketts, T.H., Rouget, M.: Integrating economic costs into conservation planning. Trends in Ecology & Evolution 21(12), 681–687 (2006)
14. Nemhauser, G.L., Wolsey, L.A., Fisher, M.L.: An analysis of approximations for maximizing submodular set functions-I. Mathematical Programming 14(1), 265–294 (1978)
15. Paik, D., Sahni, S.: Network upgrading problems. Networks 26(1), 45–58 (1995)
16. Singleton, P.H., Gaines, W.L., Lehmkuhl, J.F.: Landscape permeability for large carnivores in washington: a geographic information system weighted-distance and least-cost corridor assessment. Res. Pap. PNW-RP-549: U.S. Dept. of Agric., Forest Service, Pacific Northwest Research Station (2002)
17. Taylor, P.D., Fahrig, L., Henein, K., Merriam, G.: Connectivity is a vital element of landscape structure. Oikos 73, 43–48 (1993)

Parallel Machine Scheduling with Additional Resources: A Lagrangian-Based Constraint Programming Approach

Emrah B. Edis[1] and Ceyda Oguz[2]

[1] Dokuz Eylül University – Department of Industrial Engineering, Buca, 35160, Izmir, Turkey
emrah.edis@deu.edu.tr
[2] Koç University - Department of Industrial Engineering, Sariyer, 34450, Istanbul, Turkey
coguz@ku.edu.tr

Abstract. This study deals with an unrelated parallel machine scheduling problem with one additional resource type (e.g., machine operators). The objective is to minimize the total completion time. After giving the integer programming model of the problem, a Lagrangian relaxation problem (LRP) is introduced by relaxing the constraint set concerning the additional resource. A general subgradient optimization procedure is applied to a series of LRPs to maximize the lower bound for the original problem. To generate efficient upper bounds for the original problem, a constraint programming (CP) model is applied by taking the LRP solutions as input regarding the machine assignments. For the problem, a pure CP model is also developed to evaluate its performance. All the solution approaches are evaluated through a range of test problems. The initial computational results show that Lagrangian-based CP approach produces promising results especially for larger problem sizes.

Keywords: scheduling, parallel machines, additional resources, Lagrangian relaxation, constraint programming.

1 Introduction

Parallel machine scheduling (PMS) with additional resources is a significant area of research and involves scheduling a set of jobs over a discrete time horizon, where each job requires some constant amount of a limited cumulative resource over its processing time. This study deals with an unrelated (i.e., heterogeneous) PMS problem with one additional resource type (e.g., machine operators) which has an arbitrary (i.e., not fixed) but limited size. Given n jobs, m unrelated parallel machines, discrete integer resource requirements of jobs, res_i (i =1,…,n), processing time of job i on machine j (j=1,…,m), p_{ij}, and available size of the single additional resource, R; the aim is to schedule the jobs on the machines with the objective of minimizing total completion time without preemption and subject to additional resource constraints.

So far a number of researchers deal with PMS problems with additional resources. While most of them assume that machines are identical [e.g., 1,2,3] or dedicated [e.g., 4, 5], only few studies deal with uniform or unrelated machines [e.g., 6]. Except a few [e.g., 7, 8], all the studies aim to minimize makespan (i.e., the completion time of the last job). In terms of solution approaches, a number of studies give polynomial-time algorithms for some special cases, e.g., two or three machines, unit processing times, or 0/1 resource requirements, [e.g., 2,4,7]. Realizing that many problems are NP-hard, researchers focus on exact and heuristic algorithms. Exact algorithms [e.g., 9, 10] are relatively few due to the combinatorial nature of the problem. Problem-based heuristics [e.g., 9] and metaheuristic approaches [e.g., 11] are also not many.

Lagrangian relaxation (LR) may also be a suitable technique for PMS problems with additional resources. For unit processing times, Ventura and Kim [7] and Edis et al. [8] propose Lagrangian-based heuristic algorithms by relaxing the resource constraints and obtain efficient results. On the other hand, constraint programming (CP) is another technique to be used in especially sequencing and scheduling applications. In the literature, CP and LR are integrated in different manners (e.g., CP-based LR for the automatic recording problem [12]). It is also shown that solving relaxed sub-problems (e.g., machine assignment and resource allocation) with integer programming (IP) techniques, and handling sequencing and scheduling sub-problems by CP techniques generate more efficient results [13,14,15,16]. For minimizing makespan, Edis and Ozkarahan [17] proposed a combined IP/CP model for a PMS problem with additional resources and machine eligibility restrictions. Hooker [18] proposed a logic based Benders decomposition (LBBD) approach for a different class of PMS problems. The LBBD is performed by partitioning the original problem into a relaxed IP master problem and a series of CP scheduling sub-problems. The information obtained by solving sub-problems is given back into the IP model by Benders' cuts. Since the resource constraints are related to individual machines rather than across all machines, such problems can easily be decomposed into independent single machine sub-problems each of which can be handled individually by CP.

In this paper, on the other hand, we focus on a PMS problem with a common additional resource across all machines and propose a Lagrangian-based CP approach (LBCPA). Within LBCPA, by utilizing a standard subgradient optimization algorithm, a series of Lagrangian relaxation problems (LRPs) (obtained by relaxing the resource constraints) are solved and the infeasible solutions of LRPs are converted into feasible ones by a CP model. The rest of the paper is organized as follows. Section 2 gives the IP model of the problem together with its LRP. Section 3 presents the details of LBCPA and gives a pure CP model for the original problem. Finally, the computational results, conclusions and further research issues are given in Section 4.

2 Integer Program and its Lagrangian Relaxation Problem

With the given scheduling horizon length, T, the IP model is presented as follows:

$$\text{Minimize } \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{t=p_{ij}}^{T} t\, x_{ijt} \tag{1}$$

$$\text{s.t. } \sum_{j=1}^{m}\sum_{t=p_{ij}}^{T} x_{ijt} = 1 \qquad i=1,\dots,n \tag{2}$$

$$\sum_{i=1}^{n}\sum_{s=t}^{t+p_{ij}-1} x_{ijs} \le 1 \qquad j=1,\dots,m;\ t=1,\dots,T \tag{3}$$

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{s=t}^{t+p_{ij}-1} res_i x_{ijs} \le R \qquad t=1,\dots,T \tag{4}$$

$$x_{ijt} \in \{0,1\} \qquad i=1,\dots,n;\ j=1,\dots,m;\ t=1,\dots,T \tag{5}$$

In the above formulation, $x_{ijt} = 1$ if job i is assigned to machine j and completes its processing at time t, and $x_{ijt} = 0$ otherwise. The objective function (1) aims to minimize the total completion time of the jobs. Constraint set (2) states that each job should certainly be completed at one machine. Constraint set (3) ensures that only one job can be processed on any machine at any time interval. Constraint set (4) ensures that, at any time interval, the total resource consumption of jobs cannot exceed the available resource size, R. Finally, all x_{ijt} decision variables are binary as stated in (5).

In the formulation (1)-(5), constraint set (4) complicates the entire problem and removing it converts the problem to an easier one. Therefore, constraint set (4) is dualized so that the associated LRP can be solved more easily:

$$\text{(LRP) Minimize } \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{t=p_{ij}}^{T} t\, x_{ijt} + \sum_{t=1}^{T}\lambda_t \left[\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{s=t}^{t+p_{ij}-1} res_i x_{ijs} - R\right] \tag{6}$$

$$\text{s.t. } (2), (3) \text{ and } (5)$$

In (6), $\lambda = [\lambda_t]$ is the set of nonnegative Lagrangian multipliers for constraint set (4) which determine the tightness of the lower bound. In the proposed approach, a subgradient optimization procedure is applied to maximize the lower bound.

3 Lagrangian-Based Constraint Programming Approach

In the proposed approach, while solving a series of LRPs to maximize the lower bound, a CP model is applied to each infeasible solution of LRP to create feasible schedules with efficient upper bounds. The infeasibility of the solutions will be due to violating the resource capacity constraints (4) which are relaxed in the LRP. The CP scheduling model takes the job-machine assignment (*machine*$_i$) and processing time (*duration*$_i$) information from the LRP. The CP scheduling model is developed in OPL 6.3$^{\text{TM}}$ [19] using its special framework designed for solving scheduling problems:

$$\text{Minimize } \sum_{i=1}^{n} endof(job_i) \tag{7}$$

$$\text{s.t. } noOverlap(job_i | machine_i = j) \quad j = 1,\dots,m \tag{8}$$

$$\sum_{i=1}^{n} pulse(job_i, res_i) \le R \tag{9}$$

In formulation (7)-(9), job_i is defined as an interval variable whose position in time is unknown to the scheduling problem. The objective function (7) aims to minimize the total completion time of jobs. Constraint set (8) ensures that the jobs on each machine do not overlap. Here, noOverlap is an OPL scheduling constraint used to prevent intervals in a sequence from overlapping. Constraint (9) states that total amount of additional resource consumed at each time should not exceed the available amount, R. Pulse is an elementary cumulative function which covers the usage of a cumulative or renewable resource when an activity increases the resource usage at its start and decreases usage when it releases the resource at its end time. [19]

On the other hand, a pure CP model is also developed with two reasons. Firstly, the subgradient procedure requires an initial upper bound which can easily be obtained by a pure CP model. Secondly, we want to see the performance of a pure CP model for the original problem. This CP model is also developed in OPL 6.3:

$$\text{Minimize} \sum\nolimits_{i=1}^{n} \sum\nolimits_{j=1}^{m} endof(act_{ij}) \tag{10}$$

$$\text{s.t.} \quad alternative(job_i, \text{all}_j\ act_{ij}) \qquad i = 1,\ldots,n \tag{11}$$

$$noOverlap(all_i\ act_{ij}) \qquad j = 1,\ldots,m \tag{12}$$

$$\sum\nolimits_{i=1}^{n} pulse(job_i, res_i) \leq R \tag{13}$$

The formulation (10)-(13) includes additional optional interval variables [20] act_{ij} that can be left unperformed to handle the allocation of jobs to machines. The objective function (10) aims to minimize the total completion time of jobs. Constraint set (11) ensures that each job should be processed on exactly one of the parallel machines. Constraint sets (12) and (13) are similar to constraint sets (8) and (9).

The objective value of the first feasible solution (i.e., z^0) obtained from the pure CP model of (10)-(13) is used as an initial upper bound (UB^0) of the subgradient optimization procedure. In addition, LRP and IP models require a feasible period length, T, which is also obtained from the first solution of the pure CP model (i.e., T^0). The details of the subgradient optimization procedure are given below.

1. Set the initial values: $T = T^0$; $\lambda_t^1 = 0$ for all t; $r = 1$; $LB^0 = 0$; $UB^0 = z^0$.
2. Solve LRP with λ_t^r. Let $Z_D(\lambda^r)$ be the optimal objective function value of LRP.
3. Update the lower bound: $LB^r = \{\max LB^{r-1}, Z_D(\lambda^r)\}$.
4. If LRP generates a feasible solution, stop. Otherwise, generate a feasible solution with objective function value $z(\bar{x}^r)$ by using the proposed CP scheduling model.
5. Update the upper bound: $UB^r = \min\{UB^{r-1}, z(\bar{x}^r)\}$
6. If $(UB^r - LB^r) < \varepsilon$, STOP. Otherwise, go to Step 7.
7. Calculate the subgradients $G_t^r = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{s=t}^{t+p_{ij}-1} res_i x_{ijs} - R$; t= 1, …, T;
8. Determine the step size : $T^r = \pi^r (UB^r - LB^r) / \sum_{t=1}^{T} G_t^{r\,2}$
9. Update λ_t using: $\lambda_t^{r+1} = \max\{0, \lambda_t^r + T^r G_t^r\}$ t= 1, …, T
10. Set the iteration number ($r = r + 1$), and go to Step 2.

After a preliminary computational test, it was found that the best convergence is obtained by setting $\pi^1 = 2$. If LB^r does not improve during five consecutive iterations, π^r is divided by two. In Step 6, if $(UB^r - LB^r)$ becomes less than ε = 1, the procedure is terminated, otherwise, it terminates when $\pi^r \leq 0.005$.

4 Computational Results and Conclusion

In the computational study, two sets of instances containing 15 and 30 jobs with five machines were considered. Processing time, p_{ij}, and the resource requirements, res_i, were drawn from uniform distributions of integers between [10,20] and [0,5], respectively. The resource size, R, was taken as 8, 10 and 12. For each combination of these levels, five test problems were solved. For all solution methods, run-time limits of 300 and 600 seconds were set for 15-job and 30-job test problems, respectively. In addition, one-second run-time limit was set for CP scheduling model (7)-(9) which produces very quick and efficient feasible results. The LBCPA was coded in IBM ILOG OPL 6.3 [19] by using its scripting language. OPL 6.3 uses ILOG CPLEX 12.1 [21] for solving IP models and ILOG CP Optimizer 2.3 [19] for solving CP models. All problems were run on a Core 2 Duo 2.2 GHz, 2 GB RAM computer. To compare the performance of the proposed approach, the results of the pure IP and pure CP models were also obtained. Table 1 presents the summary of computational results. The gap percent values of all solution methods were calculated based on the lower bounds derived by the IP model (i.e., the best lower bounds found by CPLEX 12.1).

For all test problems, increasing R makes the problem easier to be solved by the IP model. Among three methods, the pure CP model gives the worst performance for all experimental points in terms of the average gap percent since it tries to make machine assignment and scheduling decisions together. LBCPA, on the other hand, takes job-machine assignment from LRP and obtains the sequence from CP; hence, it performs better than the pure CP model. Furthermore, LBCPA outperforms the IP model when the problem is stricter (e.g., the problems with smaller R). In all problems with $R = 8$, LBCPA performs better than the IP model. For 30-job test problems, LBCPA results in 2.28% gap, while the IP model gives 3.00% in average. Consequently, LBCPA gives efficient results in larger problems with restricted size of additional resource.

Table 1. Summary of Computational Results

Parameters				**Pure IP Model**				**Pure CP Model**		**Lagrangian-based CP Approach (LBCPA)**					
n	***m***	***R***	**Avg. *T***	**Avg. LB**	**Avg. UB**	**Avg. CPU Time (sec.)**	**Avg. Gap % (# of opt.)**	**Avg. UB**	**Avg. Gap % (# of opt.)**	**Avg. z^0**	**Avg. Lagr. LB**	**Avg. UB**	**Avg. # of Iter.**	**Avg. CPU Time (sec.)**	**Avg. Gap % (# of opt.)**
15	5	8	96.6	446.2	450.2	73.41^+	0.76 (3)	451.2	1.07 (1)	578.4	429.9	449.2	116.4	199.07	0.57 (3)
		10	71.0	393.5	396.0	25.79^+	0.56 (3)	401.2	1.92 (0)	490.4	384.1	397.8	112.0	174.31	1.00 (1)
		12	65.4	365.6	365.6	15.66	0.00 (5)	371.8	1.69 (0)	465.4	359.4	366.6	80.0	128.02	0.28 (3)
Average (15 jobs)				**401.8**	**403.9**	**38.29^+**	**0.44 (11)**	**408.07**	**1.56 (1)**	**511.4**	**391.1**	**404.5**	**102.8**	**167.13**	**0.62 (7)**
30	5	8	165.6	1412.4	1493.4	*	5.40 (0)	1483.6	5.01 (0)	2003.6	1398.2	1463.4	91.4	534.93^+	3.42 (0)
		10	126.2	1260.4	1287.2	*	2.06 (0)	1315.8	4.30 (0)	1708.2	1253.5	1288.0	106.0	*	2.12 (0)
		12	109.6	1193.8	1213.8	29.01^+	1.54 (2)	1239.4	3.72 (0)	1542.4	1190.1	1210.6	87.8	330.23^+	1.31 (1)
Average (30 jobs)				**1288.9**	**1331.5**	**29.01^+**	**3.00 (2)**	**1346.2**	**4.35 (0)**	**1751.4**	**1280.6**	**1320.7**	**95.1**	**412.11^+**	**2.28 (1)**

* Specified run-time limits are exceeded for all five test instances in the group.
+ Average CPU times are computed for only instances which the corresponding solver does not time out.

For the future study, the lower bounds produced by the subgradient optimization procedure of the LBCPA and time efficiency of LRPs should be improved.

Acknowledgements. This study is partially supported by a post-doctoral research grant from TÜBİTAK (Scientific and Technological Research Council of Turkey).

References

1. Blazewicz, J., Ecker, K.: A linear time algorithm for restricted bin packing and scheduling problems. Operations Research Letters 2(2), 80–83 (1983)
2. Blazewicz, J., Kubiak, W., Röck, H., Szwarcfiter, J.: Minimizing mean flow time with parallel processors and resource constraints. Acta Informatica 24, 513–524 (1987)
3. Ventura, J.A., Kim, D.: Parallel machine scheduling about an unrestricted due date and additional resource constraints. IIE Transactions 32, 147–153 (2000)
4. Kellerer, H., Strusevisch, V.A.: Scheduling parallel dedicated machines under a single non-shared resource. European Journal of Operational Research 147, 345–364 (2003)
5. Kellerer, H., Strusevisch, V.A.: Scheduling problems for parallel dedicated machines under multiple resource constraints. Discrete Applied Mathematics 133, 45–68 (2004)
6. Kovalyov, M.Y., Shafransky, Y.M.: Uniform machine scheduling of unit-time jobs subject to resource constraints. Discrete Applied Mathematics 84, 253–257 (1998)
7. Ventura, J.A., Kim, D.: Parallel machine scheduling with earliness-tardiness penalties and additional resource constraints. Computers and Operations Research 30, 1945–1958 (2003)
8. Edis, E.B., Araz, C., Ozkarahan, I.: Lagrangian-based solution approaches for a resource-constrained parallel machine scheduling problem with machine eligibility restrictions. In: Nguyen, N.T., Borzemski, L., Grzech, A., Ali, M. (eds.) IEA/AIE 2008. LNCS (LNAI), vol. 5027, pp. 337–346. Springer, Heidelberg (2008)
9. Blazewicz, J., Kubiak, W., Martello, S.: Algorithms for minimizing maximum lateness with unit length tasks and resource constraints. Discrete Applied Mathematics 42, 123–138 (1993)
10. Kellerer, H., Strusevisch, V.A.: Scheduling parallel dedicated machines with the speeding-up resource. Naval Research Logistics 55(5), 377–389 (2008)
11. Li, Y., Wang, F., Lim, A.: Resource constraints machine scheduling: A genetic algorithm approach. In: 2003 Congress on Evolutionary Computation, vol. 1-4, pp. 1080–1085 (2003)
12. Sellmann, M., Fahle, T.: Constraint Programming Based Lagrangian Relaxation for the Automatic Recording Problem. Annals of Operations Research 118, 17–33 (2003)
13. Darbi-Dowman, K.D., Little, J., Mitra, G., Zaffalon, M.: Constraint logic programming and integer programming approaches and their collaboration in solving an assignment scheduling problem. Constraints 1, 245–264 (1997)
14. Smith, B.M., Brailsford, S.C., Hubbard, P.M., Williams, H.P.: The progressive party problem: integer linear programming and constraint programming compared. Constraints 1, 119–138 (1997)
15. Darbi-Dowman, K.D., Little, J.: Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. Informs Journal on Computing 10(3), 276–286 (1998)
16. Lustig, I.J., Puget, J.F.: Program does not equal program: Constraint programming and its relationship to mathematical programming. Interfaces 31(6), 29–53 (2001)
17. Edis, E.B., Ozkarahan, I.: A combined integer/constraint programming approach to a resource-constrained parallel machine scheduling problem with machine eligibility restrictions. Engineering Optimization 43(2), 135–157 (2011)

18. Hooker, J.N.: Planning and scheduling to minimize tardiness. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 314–327. Springer, Heidelberg (2005)
19. IBM, ILOG OPL IDE 6.3., User's Manual, IBM Corp. (2009)
20. Laborie, P., Rogerie, J.: Reasoning with conditional time-intervals. In: Proceedings of the Twenty-First International FLAIRS Conference, pp. 555–560 (2008)
21. IBM, ILOG CPLEX 12.1., User's Manual, IBM Corp. (2009)

Branch-Cut-and-Propagate for the Maximum k-Colorable Subgraph Problem with Symmetry

Tim Januschowski[1,⋆] and Marc E. Pfetsch[2]

[1] Cork Constraint Computation Centre
Computer Science Department, University College Cork, Ireland
janus@cs.ucc.ie
[2] Institute for Mathematical Optimization, TU Braunschweig
Pockelsstr. 14, 38106 Braunschweig, Germany
m.pfetsch@tu-bs.de

Abstract. Given an undirected graph and a positive integer k, the maximum k-colorable subgraph problem consists of selecting a k-colorable induced subgraph of maximum cardinality. The natural integer programming formulation for this problem exhibits two kinds of symmetry: arbitrarily permuting the color classes and/or applying a non-trivial graph automorphism gives equivalent solutions. It is well known that such symmetries have negative effects on the performance of constraint/integer programming solvers.

We investigate the integration of a branch-and-cut algorithm for solving the maximum k-colorable subgraph problem with constraint propagation techniques to handle the symmetry arising from the graph. The latter symmetry is handled by (non-linear) lexicographic ordering constraints and linearizations thereof. In experiments, we evaluate the influence of several components of our algorithm on the performance, including the different symmetry handling methods. We show that several components are crucial for an efficient algorithm; in particular, the handling of graph symmetries yields a significant performance speed-up.

1 Introduction

Symmetry in integer programs (IPs) and constraint programs (CPs) has been recognized to harm the performance of solution algorithms for a long time. One reason for this unfavorable effect is that symmetric solutions appear repeatedly in the search tree, without giving new information about optimal solutions. Furthermore, in the IP-setting, symmetries usually lead to weak bounds of the linear programming relaxation. In recent years, several (successful) methods to handle symmetry have been developed, see Margot [31] for an overview.

A particular problem in which symmetry arises is the maximum k-colorable subgraph problem, which is defined as follows: Given an undirected graph and a positive integer k, find a largest (induced) subgraph that can be colored with k colors, i.e., we can assign one of at most k colors to each node in the subgraph,

⋆ Tim Januschowski is supported by an Embark Scholarship of the Irish Research Council for Science, Engineering and Technology.

such that two adjacent nodes do not receive the same color. Any k-colored subgraph can be transformed into another, equivalent k-colored subgraph by arbitrarily permuting the color classes or by applying a graph automorphism. This gives rise to color and graph symmetry in a natural IP-formulation of the maximum k-colorable subgraph problem.

One way to tackle symmetry is a polyhedral approach. For instance, so-called *orbitopes* can be used to handle the symmetry arising from permuting the color classes. These polytopes where introduced in [22]. In particular, a complete description via *shifted column inequalities* (SCIs) was given, and an efficient separation algorithm for SCIs was developed. A fast algorithm to perform constraint propagation on orbitope structures was investigated in [21]. We will skip details on these techniques, because they are not of central importance for this paper. They will, however, be used in the computational experiments and are crucial for fast algorithms.

The symmetry arising from automorphisms of the graph, however, has not yet been treated from a polyhedral viewpoint. In this paper, we investigate methods based both on constraint propagation and linear inequalities to handle such symmetry, together with the orbitope approach for color symmetries. To the best of our knowledge, no previous work on combining polyhedral symmetry handling techniques from IP with CP symmetry handling approaches has appeared in the literature. We use the maximum k-colorable subgraph problem as a prototype application for this investigation. This line of research has been started in [18], where we investigate the interaction of the problem-specific polyhedral structure of the maximum k-colorable subgraph problem with orbitopes.

After discussing related work and giving basic definitions, we investigate a symmetry handling approach based on a lexicographic ordering constraint in Section 2.1. For special structures of the automorphism group, one can strengthen the original constraint, for instance if a transposition is part of the automorphism or a symmetric group is acting on a subgraph. In Section 2.2, we study domain propagation of so-called structural symmetry breaking constraints to handle symmetries that arise from a combination of graph and color symmetries. Section 3 describes components of a branch-cut-and-propagate algorithm to solve the maximum k-colorable subgraph problem, i.e., we combine a branch-and-cut with a constraint programming algorithm to solve a constraint integer program, see [4,6]. In Section 4, we report on computational experiments, which show the effectiveness of this approach. For example, we show that the usage of symmetric subgroups of the automorphism group lead to a significant improvement of the algorithm performance.

1.1 Related Work

Margot [31] gives an excellent overview of symmetry handling methods in IP and CP. We thus keep this section brief and only highlight some related work.

The handling of symmetries has been extensively discussed in the CP literature. Puget [38] first introduced a symmetry breaking constraint, and Crawford et al. [9] propose general symmetry breaking constraints. The addition of constraints for the handling of row and column symmetry has received considerable attention, see, e.g., [10,13,23]. Gent et al. [16] present an overview.

In integer programming, the handling of symmetries by adding constraints was proposed by Méndez-Díaz and Zabala [33] for the graph coloring problem. General IP-methods to handle symmetries via pruning in the search tree have been developed by Margot [28,29,30] and Ostrowski et al. [37,26].

The maximum k-colorable subgraph problem appears to have rarely been studied in the literature, supposedly, because it is closely connected to both the more prominent graph coloring problem and the stable set problem. See [18] for more details as well as a list of applications and relevant IP-techniques for the maximum k-colorable subgraph problem.

1.2 Basic Notation and Definitions

We use the following notation throughout the article.

For an integer $n \geq 1$, let $[n] := \{1, 2, \ldots, n\}$. For $x \in \mathbb{R}^{[n]\times[k]}$ and $S \subseteq [n]\times[k]$, we write $x(S) := \sum_{(i,j)\in S} x_{ij}$. We use row_i for the set $\{(i,1),(i,2),\ldots,(i,k)\}$ and col_j for the set $\{(1,j),(2,j),\ldots,(n,j)\}$. By $\text{row}_i(x)$ we denote the ith row of x and $\text{col}_j(x)$ denotes the jth column of x. By $\mathbf{0}$ we denote a zero-matrix of appropriate dimensions.

Any permutation of a finite set can be written using disjoint cycles. A *cycle* $(a_1 a_2 \ldots a_k)$ refers to the mapping $a_1 \mapsto a_2 \mapsto \ldots \mapsto a_k \mapsto a_1$; k is the *length* of the cycle, and a cycle of length k is called a *k-cycle*. A 2-cycle is a *transposition*.

Let $G = (V, E)$ be a simple undirected graph with node set V and edge set E, where $n := |V|$. We will often need an order of the nodes in V and thus assume that $V := \{1, \ldots, n\}$; in particular, we can directly compare nodes, e.g., $u < v$ for $u, v \in V$. We denote by $G[V']$ the subgraph induced by $V' \subseteq V$. Thus, the set of edges of $G[V']$ is $\{e \in E \,:\, e \subseteq V'\}$. For a node v, the *neighborhood* $\Gamma(v)$ of node v contains all nodes adjacent to v, i.e.,

$$\Gamma(v) := \{u \in V \,:\, \{u, v\} \in E\}.$$

The *closed neighborhood* of v is $\overline{\Gamma}(v) := \Gamma(v) \cup \{v\}$. The *degree* of v is defined as $\delta(v) := |\Gamma(v)|$. A *clique* is a set of nodes $C \neq \varnothing$, such that for all $u, v \in C$, $u \neq v$, we have $\{u, v\} \in E$. A *stable set* is a set of nodes $S \neq \varnothing$ such that for all $u, v \in S$, we have $\{u, v\} \notin E$.

For a positive integer k, G is *k-colorable* if we can assign to each node in G a color (number) in $[k]$ such that adjacent nodes do not have the same color.

Definition 1. *For a positive integer k and a graph G, the* maximum k-colorable subgraph problem *consists of finding a set $V' \subseteq V$ such that $G[V']$ is k-colorable and V' has maximum cardinality.*

The maximum k-colorable subgraph problem is NP-hard and hard to approximate, see [18] for a detailed discussion.

Note that the restriction to simple graphs for the maximum k-colorable subgraph problem is without loss of generality: parallel edges can be replaced by a single edge, and nodes with loops will never be part of V'.

We consider the following IP-formulation for the maximum k-colorable subgraph problem.

$$
\begin{aligned}
(\mathrm{IP}_k(G)) \quad & \max \sum_{v \in V} \sum_{j \in [k]} x_{vj} && && (1)\\
& x_{uj} + x_{vj} \le 1 && \forall \{u,v\} \in E,\ j \in [k] && (2)\\
& x(\mathrm{row}_v) \le 1 && \forall v \in V && (3)\\
& x_{vj} \in \{0,1\} && \forall v \in V,\ j \in [k]. && (4)
\end{aligned}
$$

Let x be a solution of (2)–(4). Because of the *packing inequalities* (3), $x(\mathrm{row}_v)$ is 1 if and only if node v is in the selected subgraph (i.e., colored). Thus, the objective function represents the cardinality of the selected subgraph.

The *k-colored subgraph polytope* corresponding to the maximum k-colorable subgraph problem is

$$\mathrm{P}_k(G) := \mathrm{conv}\{x \in \{0,1\}^{V \times [k]} \ : \ x \text{ satisfies (2) and (3)}\}.$$

A *symmetry* of an IP is a permutation of the variables that maps feasible solutions to feasible solutions with the same objective function value. They are variable solution symmetries in the sense of Cohen et al. [8]. The symmetries of an IP form the *symmetry group*. The symmetry group partitions the feasible solutions into disjoint orbits. We say that the addition of a set of constraints leads to *partial/complete* symmetry handling with respect to the symmetry group if for every orbit of solutions at least/exactly one solution is preserved.

The maximum k-colorable subgraph problem exhibits two basic types of symmetries: color and graph symmetries. In [18], we concentrated on handling color symmetry polyhedrally via orbitopes. Color symmetries form a symmetric group $\mathfrak{S}_k$ of degree k that operates by permuting the columns of x in $(\mathrm{IP}_k(G))$. In this paper, we concentrate on graph symmetry handling.

2 Graph Symmetry Handling

Each element of the automorphism group $\mathrm{Aut}(G)$ of the graph G yields a symmetry of $(\mathrm{IP}_k(G))$ by permuting the rows of x accordingly. Graph and color symmetries differ in certain aspects as we discuss in the following.

First, unlike color symmetries, graph automorphisms may not yield symmetries in the weighted version of the maximum k-colorable subgraph problem, where each node receives a weight and the goal is to maximize the sum of the weights of the colored nodes. Recall that this article only considers the unweighted version.

Second, the color symmetries of $(\mathrm{IP}_k(G))$ are known (but could also be efficiently be computed, see [7]). In contrast, finding graph automorphisms (and hence graph symmetries) is at least as hard as the graph isomorphism problem, which has an open complexity status (see Garey and Johnson [15] and Johnson [20]). More precisely, the problem of detecting whether a graph admits a non-trivial automorphism is graph-isomorphism complete.

Third, whereas color symmetries always form a symmetric group, any group can be the automorphism group of a graph, see, e.g., Frucht [14].

2.1 Lexicographic Graph Symmetry Handling

Crawford et al. [9] propose constraints to handle general symmetries for Boolean satisfiability problems that we can adapt for our purposes as follows. For every symmetry $\phi \in \operatorname{Aut}(G)$, we have the following lexicographic ordering constraint.

$$[\operatorname{row}_1(x), \dots, \operatorname{row}_n(x)] \geq_{\text{lex}} [\operatorname{row}_{\phi(1)}(x), \dots, \operatorname{row}_{\phi(n)}(x)]. \tag{5}$$

If we add (5) for all $\phi \in \operatorname{Aut}(G)$, then the graph symmetries are completely handled. In particular, the addition of (5) comes with the guarantee that from every orbit of solutions, the lexicographically largest solution is preserved, see [9]. Note that in (5), it suffices to consider only nodes v for which $\phi(v) \neq v$.

Remark 1. When combining color and graph symmetry handling, one needs to order the rows and the columns of x in the same lexicographic fashion (i.e., decreasingly or increasingly). Orbitope symmetry handling for the color symmetries uses a lexicographically decreasing order on the columns. Hence, a lexicographically decreasing order on the rows should be used as well. Otherwise, one may lose entire orbits of solutions, see Flener et al. [10].

Constraints (5) are non-linear. A standard linearization is:

$$\sum_{i\in[n]} \sum_{j\in[k]} 2^{(n+1-i)k-j}\, x_{ij} \geq \sum_{i\in[n]} \sum_{j\in[k]} 2^{(n+1-i)k-j}\, x_{\phi(i),j}\,. \tag{6}$$

Inequalities (6) have coefficients from 1 to 2^{nk-1}. Even for small n and k, this will cause severe numerical difficulties for IP-solvers [31].

We therefore do not use inequalities (6), but directly perform constraint propagation on (5) as follows (see Frisch et al. [13] for domain filtering for (5) in a pure CP context). Assume that x_{ij} becomes fixed during search for some $i \in [n]$, $j \in [k]$; the implications from fixings of $x_{\phi(i),j}$ are similar. Further, assume that x_{st} and $x_{\phi(s),t}$ are fixed for all $s < i$ and all $t \in [k]$, and that

$$[\operatorname{row}_1(x), \dots, \operatorname{row}_{i-1}(x)] = [\operatorname{row}_{\phi(1)}(x), \dots, \operatorname{row}_{\phi(i-1)}(x)].$$

If x_{ij} is fixed to 1, this implies that $x_{\phi(i),\ell} = 0$ for $\ell < j$ (otherwise Constraint (5) would be violated). If x_{it} is fixed to 0 for all $t \in [j]$, then we can fix $x_{\phi(i),t} = 0$ for all $t \in [j]$.

To provide further propagation steps, we need the following notation. We denote by $\check{x}$ and $\hat{x}$ the matrix where all unfixed variables in x are set to 0 and 1, respectively. If $\operatorname{row}_{i+1}(\hat{x}) <_{\text{lex}} \operatorname{row}_{\phi(i+1)}(\check{x})$, we derive stronger fixings. It follows that $\operatorname{row}_i(\tilde{x}) >_{\text{lex}} \operatorname{row}_{\phi(i)}(\tilde{x})$ must hold for every feasible solution $\tilde{x}$ compatible with the fixings, otherwise Constraint (5) is violated. If x_{ij} is fixed to 1, then we may, additionally to the above fixings, fix $x_{\phi(i),j}$ to 0. If x_{it} is fixed to 0 for all $t \in [j]$, then we may, additionally to the above fixings, fix $x_{\phi(i),j+1}$ to 0.

If the automorphism group has a particular structure, this general approach can be specialized. We discuss several examples in the following.

Graph Transpositions: One particularly simple kind of graph automorphisms are transpositions, which frequently occur in the automorphism group of a graph G. For a transposition $\phi = (u\ v) \in \mathrm{Aut}(G)$ (we always assume $u < v$), we can simplify (5) to:

$$\mathrm{row}_u(x) \geq_{\mathrm{lex}} \mathrm{row}_v(x), \tag{7}$$

and we can simplify (6) to:

$$\sum_{j \in [k]} 2^{k-j} x_{uj} \geq \sum_{j \in [k]} 2^{k-j} x_{vj}. \tag{8}$$

This inequality can be strengthened by using the packing inequalities:

$$\sum_{\ell \in [j]} x_{u\ell} \geq \sum_{\ell \in [j]} x_{v\ell}, \qquad \forall\, j \in [k]. \tag{9}$$

Constraints (8) and (9) are logically equivalent, because they exclude all but the lexicographically largest solutions with respect to $(u\ v)$. However, polyhedrally speaking, Inequalities (9) dominate (8); they also avoid large coefficients.

Composition of Transpositions: Consider the case in which $\mathrm{Aut}(G)$ contains a composition of (disjoint) transpositions

$$\phi = (u_1\ u_2)(u_3\ u_4) \dots (u_{\ell-1}\ u_\ell),$$

such that $u_i \neq u_{i+1}$ for all $i \in [\ell - 1]$.

In this case, we can strengthen the above mentioned general domain propagation follows. We can assume w.l.o.g. that $u_1 = \min\{u_1, u_2, \dots, u_\ell\}$. In case $\{u_1, u_2\} \in E$, then $x_{u_2,1} = 0$ must hold. For the sake of contradiction, assume that there exists a feasible solution $\tilde{x}$ with $\tilde{x}_{u_2,1} = 1$. It follows that $\tilde{x}_{u_1,1} = 0$ due to $\{u_1, u_2\} \in E$. Then, however, $\tilde{x}$ violates (5); this proves the claim.

Symmetric Groups: In order to handle symmetric (sub)groups, we need the following basic fact.

Lemma 1. *Let $(u\ v) \in \mathrm{Aut}(G)$ with $\{u, v\} \in E$. Then, for any $(v\ w) \in \mathrm{Aut}(G)$, we have $\{v, w\} \in E$.*

Proof. By definition, $\Gamma(u) = \Gamma(v)$ for $(u\ v) \in \mathrm{Aut}(G)$. Because $(u\ w) = (u\ v)(v\ w)(u\ v)$, it follows that $\Gamma(w) = \Gamma(u)$. Since $v \in \Gamma(u) = \Gamma(w)$, we have $\{v, w\} \in E$. $\square$

It follows that a symmetric subgroup of the automorphism group either acts on a stable set or on a clique. If it acts on a clique, we can use orbitopal fixing, see [21], as an efficient domain filtering algorithm; in particular, for a symmetric group acting on the clique $C = \{u_1, u_2, u_3, \dots, u_c\}$, we may fix $x_{u_i,j} = 0$ for all $i = 2, \dots, c$ and $j = 1, \dots, k(i-1)$ due to the lexicographic ordering.

Lemma 2. *Let a symmetric subgroup act on a stable set S. Then adding*

$$x_{uj} = x_{vj} \quad \textit{for all } j \in [k],\ u, v \in S, \tag{10}$$

to $(\mathrm{IP}_k(G))$ *does not change the optimal value.*

Proof. If no optimal solution colors a node in S, the claim holds trivially. Otherwise, assume that there exists a solution $\tilde{x}$ with $\tilde{x}_{uj} = 1$ for some $u \in S$ and $j \in [k]$. Consider any node $v \in S \setminus \{u\}$. By assumption, we have $(u\ v) \in \mathrm{Aut}(G)$, i.e., $\Gamma(u) = \Gamma(v)$. Since $\tilde{x}$ is a valid coloring, there exists no $w \in \Gamma(v) = \Gamma(u)$ such that $\tilde{x}_{wj} = 1$. Since by assumption $\{u, v\} \notin E$, (re)coloring v with color j yields a valid coloring. Since v was arbitrary, we conclude that any node in S can be colored with the same color. Moreover, if $\tilde{x}$ was optimal the resulting coloring is optimal and satisfies (10). □

We note that the equations (10) do not conflict with Constraints (5) for optimal solutions: Every solution that fulfills (10) has lexicographically ordered rows for nodes in S and thus fulfills (5). Moreover, the proof of Lemma 2 shows that if a solution fulfills (5) but not (10), the solution is either not optimal or can be partially recolored such that it fulfills (10).

Arbitrary Groups: In the general case, we cannot exploit any particular structure of the automorphism group. In particular, since $\mathrm{Aut}(G)$ may become very large, it is in general not efficient to consider (5) for all available automorphisms. Thus, we only consider the generators of $\mathrm{Aut}(G)$ that NAUTY [32] outputs.

2.2 Combinations of Graph and Color Symmetry

So far, we have focused on the independent handling of graph and color symmetries: orbitopes handle the color symmetries completely, and (5) can handle graph symmetries completely. As the following example shows, there may, however, be combinations of graph and color symmetries (*product symmetries*) that are left unhandled, even if we restrict attention to transpositions (see also Flener et al. [10]). In this section we deal with methods to handle product symmetries.

Example 1. Consider a graph with three isolated nodes $\{1, 2, 3\}$, $k = 3$, and the two solutions $[1, 0, 0; 0, 1, 0; 0, 1, 0]$ and $[1, 0, 0; 1, 0, 0; 0, 1, 0]$ (with the obvious interpretation as matrices). Both solutions have lexicographically ordered rows and columns, but are symmetric via a combination of graph and color symmetries.

Note that the difference between the number of solutions in the case with and without complete handling of product symmetries can be exponential in the size of G, even if graph and color symmetries are handled completely, see [23].

Flener et al. [11] present constraints that provide complete symmetry handling for product symmetries, where the graph automorphism group is a union of symmetric groups. We need some notation in order to present this result. We partition the set of nodes V into sets $V_1, \dots, V_s$ such that a symmetric subgroup $\mathfrak{S}_p$ of $\operatorname{Aut}(G)$ acts on V_p for all $p \in [s]$. Let $V_p = \{v_1^p, \dots, v_{q_p}^p\}$ with $v_1^p < v_2^p < \dots < v_{q_p}^p$. We require the lexicographic ordering

$$\operatorname{row}_{v_i^p}(x) \geq_{\text{lex}} \operatorname{row}_{v_{i+1}^p}(x), \tag{11}$$

for all $i \in [q_p - 1]$ and all $p \in [s]$. We introduce *frequency variables*

$$f_j^p := \sum_{v \in V_p} x_{vj} \in \mathbb{Z}_+,\ j \in [k],\ p \in [s], \tag{12}$$

determining the number of nodes in V_p colored with color j. Finally, we introduce the constraints

$$(f_j^1, f_j^2, f_j^3, \dots, f_j^s) \geq_{\text{lex}} (f_{j+1}^1, f_{j+1}^2, f_{j+1}^3, \dots, f_{j+1}^s),\ j \in [k-1]. \tag{13}$$

We refer to Constraints (11) and (13) as *structural symmetry breaking* (SSB) constraints. Note that variables f_j^p are only used for ease of exposition; we do not use them in our implementation. Flener et al. [11] proved that SSB constraints completely handle all symmetries of $(\text{IP}_k(G))$.

Unfortunately, SSB constraints do not necessarily imply a lexicographic ordering of the columns as one easily verifies. This impedes a combination of orbitope symmetry handling with SSB constraints in the general case. However, the following result shows how to relabel the nodes such that orbitope symmetry handling and SSB constraints do not conflict.

Proposition 1. *Let G be a graph such that a symmetric subgroup $\mathfrak{S}_p$ of $\operatorname{Aut}(G)$ acts on V_p for $p = 1, \dots, s$. Let the node labeling of G be such that*

$$v_1^1 < \dots < v_{q_1}^1 < v_1^2 < \dots < v_{q_2}^2 < v_1^3 < \dots < v_1^s < \dots < v_{q_s}^s.$$

Then SSB constraints imply a lexicographically decreasing order on the columns.

Proof. For the sake of contradiction, assume the existence of a solution $(\tilde{x}, \tilde{f})$ that fulfills (13) and has two columns j, $j+1$ such that $\operatorname{col}_j(\tilde{x}) <_{\text{lex}} \operatorname{col}_{j+1}(\tilde{x})$. Let i be the smallest row index in which $\operatorname{col}_j(\tilde{x})$ and $\operatorname{col}_{j+1}(\tilde{x})$ differ and $\tilde{x}_{ij} = 0$, $\tilde{x}_{i,j+1} = 1$. Let node i be in partition V_p.

By assumption, $\tilde{x}_{rj} = \tilde{x}_{r,j+1}$ for $r < i$. Because of the packing inequalities, it follows that $\tilde{x}_{rj} = \tilde{x}_{r,j+1} = 0$. Thus, $\tilde{f}_j^q = \tilde{f}_{j+1}^q = 0$ for all $q \in [p-1]$, due to the node labeling.

Within partition V_p, we have $\tilde{x}_{rj} = 0$ for all $r \in V_p$ with $r < i$. Moreover, node i is colored with color $j+1$. Thus, $\tilde{x}_{it} = 0$ for all $t \in [j]$ by the packing inequalities. Since the rows within a partition are lexicographically decreasingly ordered, it follows that $\tilde{x}_{rt} = 0$ for all r in V_p with $r > i$ and all $t \in [j]$. Therefore, we have shown that $\tilde{x}_{vj} = 0$ for all $v \in V_p$. It follows that $0 = \tilde{f}_j^p < \tilde{f}_{j+1}^p \geq 1$. Thus, $\tilde{x}$ violates an SSB constraint. $\square$

The node labeling/order is important for the maximum k-colorable subgraph problem and has a significant influence on the solving time. We shall verify in experiments, whether the additional symmetry handling by SSB constraints outweighs the effect of restricted node orderings.

We directly perform domain filtering for SSB constraints as follows (see Flener et al. [11] for stronger domain filtering in a pure CP context). First, for Constraint (11), we use the already mentioned domain propagation of Constraint (5). Second, we propagate Constraint (13) as follows.

Assume variable x_{ij} becomes fixed, with corresponding frequency variable f_j^p. Assume further that all variables x_{vj} and $x_{v,j+1}$ for all $v \in V_q$ are fixed such that $f_j^q = f_{j+1}^q$ for all $q < p$.

First, consider the case where all variables $x_{v,j+1}$ with $v \in V_p$ are fixed. Then $x(V_p \times \{j+1\}) = f_{j+1}^p$, and because of (13), it follows that $\tilde{x}(V_p \times \{j\}) \geq f_{j+1}^p$ must hold for any feasible $\tilde{x}$ that is compatible with the fixings. Assume that $\hat{x}(V_p \times \{j\}) = \ell < f_{j+1}^p$ (see Section 2.1 for a definition of $\hat{x}$). If there are $(f_{j+1}^p - \ell)$ unfixed variables with indices in $V_p \times \{j\}$, we can fix all of them to 1. If there are strictly fewer, the constraint is violated; if there are more, we do nothing.

Moreover, consider the case where $\hat{x}(V_{p+1} \times \{j\}) < \check{x}(V_{p+1} \times \{j+1\})$, then $f_j^p > f_{j+1}^p$ must hold for all solutions compatible with the fixings. If there are $(f_{j+1}^p - \ell + 1)$ unfixed variables with indices in $V_p \times \{j\}$, we can fix all of them to 1. If there are strictly fewer, the constraint is violated; if there are more, we do nothing. We can deduce similar domain filterings for variables in $V_p \times \{j-1\}$ based on the fixing of x_{ij}.

3 A Branch-Cut-and-Propagate Algorithm

In this section, we describe the main components of a branch-cut-and-propagate algorithm for the maximum k-colorable subgraph problem: preprocessing, node labeling heuristics, branching rules, (valid) inequalities, and node domination inequalities. Additionally, we use the constraints for graph symmetries as described in Section 2 and conflict analysis, see Achterberg [3,4]. We assume familiarity with the branch-and-cut approach, see, e.g., Nemhauser and Wolsey [36] for background.

At several places, we use TCLIQUE, a specialized branch-and-bound method for the maximum weighted clique problem, which is available in SCIP [39]. By working on the complement graph, TCLIQUE can solve the maximum weighted stable set problem as well.

3.1 Preprocessing

Preprocessing tries to reduce the size of the graph, before the main optimization is performed. For the maximum k-colorable subgraph problem, we can remove nodes with low degree as follows. If a node v with $\delta(v) < k$ exists, we can always color this node with a color that is not used by its neighbors. Hence, we can delete this node from graph G and adapt the objective function.

3.2 Node Labeling Heuristics

Before setting up formulation ($\mathrm{IP}_k(G)$), one can reorder (relabel) the nodes, which has a big impact on the performance of solution algorithms. We investigate the following node labelings:

- Sort nodes w.r.t. increasing/decreasing node-degree.
- Choose a large clique as the first nodes in the order and then sort w.r.t. increasing/decreasing node-degree.
- Sort the nodes such that nodes in the same symmetric group have consecutive node labels (see Proposition 1).
- Sort the nodes randomly.

3.3 Branching Rules

One possibility is to branch on nodes as done in [34], i.e., we choose a node in the graph and generate a branch for each color which is still available.

- *Uncolored node branching*: choose a node with highest number of non-available colors. Ties are broken according to the degree of the nodes in the uncolored subgraph, i.e., the number of neighbors whose color is not fixed.
- *Uncolored Sewell's rule*: choose a node as above. Ties are broken according to the rule of Sewell [40]: The node whose coloring induces the smallest number of colorings for its neighborhood is selected.

As an alternative, we branch on the first fractional *variable* in the row-wise ordering (*first index branching*), see [21]. In the *truncated first index branching* variant, we branch on the first fractional variable x_{ij} for which the current LP-solution has no 1 in positions $(j, j), \dots, (i-1, j)$. If no such variable has been found, we use the standard (strong-)branching rule of SCIP, see [5]. These two variable branching rules try to support the lexicographic ordering of the columns.

3.4 Valid Inequalities

In order to generate additional cutting planes, we implemented the following separation algorithms.

- Let S be a set of nodes in G, and let $\alpha(S)$ be the maximum size of a stable set in $G[S]$. Then, the inequality

$$\sum_{v \in S} x_{vj} \leq \alpha(S) \tag{14}$$

  is a valid inequality for $\mathrm{P}_k(G)$ for all $j \in [k]$. We use a heuristic algorithm to separate (14) with small S. If S is a clique, (14) are called *clique inequalities*; we use TCLIQUE to separate clique inequalities. If C is an odd cycle in G, then we refer to (14) as an *odd cycle inequality.* We separate odd cycle inequalities with an algorithm due to Grötschel et al. [17].

- The *neighborhood inequality* for a node v is given by

$$\sum_{u \in \Gamma(v)} x_{uj} + r x_{vj} \leq r \qquad \forall\, j \in [k],$$

  where $r = \alpha(\Gamma(v))$, see [34,35]; r is computed via TCLIQUE. These inequalities are added to $\mathrm{IP}_k(G)$ from the start.
- *Shifted Column Inequalities* (SCIs) for orbitopes are separated with the linear-time separation algorithm described in [22].
- *Clique Shifted Column Inequalities*, see [18,22], and *Packing-Clique Inequalities*, see [18], are separated heuristically.

3.5 Dominated Nodes

Additional inequalities can be added at the start by considering dominated nodes. A node v *dominates* another node u if $\overline{\Gamma}(v) \supseteq \Gamma(u)$. We say that v *strictly* dominates u if $\Gamma(v) \setminus \{u\} \supsetneq \Gamma(u) \setminus \{v\}$. Dominated nodes appear in the context of the stable set problem, see, e.g., [12], or the graph coloring problem, see, e.g., [27]. In fact, in graph coloring, we can remove dominated nodes from the graph if $\{u, v\} \notin E$, because they can always be colored with the same color as the dominating nodes. In the maximum k-colorable subgraph problem one can show that this is not the case (see [19] for an example). We can, however, generate inequalities based on the following result.

Theorem 1. *Let $u, v \in V$, $u \neq v$, be a pair of nodes with $\overline{\Gamma}(v) \supseteq \Gamma(u)$.*

1. *If $\{u, v\} \in E$, then adding the following inequality does not change the optimal value of* $(\mathrm{IP}_k(G))$*:*

$$\sum_{j \in [k]} x_{vj} \leq \sum_{j \in [k]} x_{uj}. \tag{15}$$

2. *If $\{u, v\} \notin E$, then adding the following inequalities does not change the optimal value of* $(\mathrm{IP}_k(G))$*:*

$$x_{vj} \leq x_{uj} \quad \forall\, j \in [k]. \tag{16}$$

We refer to [19] for a proof. The basic ideas are as follows. In the first case, for any optimal solution coloring v, there exists an optimal solution that colors u. In the second case, for any optimal solution that colors v, there exists an optimal solution such that u receives the same color as v. Note that both kinds of inequalities may cut off feasible/optimal solutions.

If node u dominates v and v dominates u, this induces a graph transposition $(u\ v) \in \mathrm{Aut}(G)$. For graph transpositions, we have already provided stronger inequalities: Inequalities (9) for $\{u, v\} \in E$ and Equation (10) for $\{u, v\} \notin E$. In particular, we can add (9) and (10) for *all* such pairs u, v with $u < v$.

Avoiding Conflicts between Constraints: Adding Inequalities (15) for all pairs of (strictly) dominated/dominating nodes may cut off all optimal solutions.

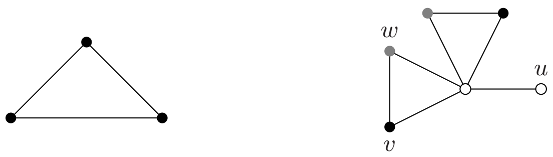


Fig. 1. *Left:* All pairs of nodes u, v dominate each other, i.e., $\overline{\Gamma}(v) \supseteq \Gamma(u)$. *Right:* Node u is dominated by nodes v and w. For $k = 2$, an optimal 2-colored subgraph is depicted in gray and black. It contains u, v, and w. However, we cannot color node u with the same color as v *and* w.

The left-hand side of Figure 1 shows an example: If $k = 2$, including (15) for all pairs cuts off all optimal solutions. Similarly, we cannot add all Inequalities (16), see the right-hand side of Figure 1.

In order to avoid these difficulties, we construct a directed graph D. For every pair of nodes u, v, such that v strictly dominates u, we add a directed edge (v, u). We obtain the following result (a proof can be found in [19]).

Lemma 3. *Let A be a subset of arcs of D such that A is a branching, i.e., A is acyclic and the in-degree of every node is at most one. Then at least one optimal solution remains when adding* (15) *and* (16) *for all arcs in A.*

Similarly, a naïve combination of constraints for graph transposition or dominated neighborhood inequalities may remove all optimal solutions in $(\text{IP}_k(G))$. Using an analogous construction as above guarantees that at least one optimal solution remains.

4 Computational Results

We implemented a branch-cut-and-propagate algorithm as described above in C++ based on SCIP 2.0.0, see [4,39]. We use CPLEX 12.1 as the LP-solver. The experiments were run on a linux cluster, in which each node has 8 Intel Xeon CPUs with 2.66GHz; we only use four CPUs per node to avoid timing issues.

To obtain a test set, we selected graphs from the Color02 symposium [2] and the clique part of the Second DIMACS Implementation Challenge [1] benchmarks; we add the complement of some of the clique graphs. Finally, we included instances for the wave length assignment problem, see [24,25]; only instances for which the chromatic number is strictly greater than 20 were considered for $k = 20$. In total, we selected 74 combinations of graphs and numbers of colors; see Table 1 for the Color02 and DIMACS graphs (see [19] for more details on the benchmark). The instances were chosen such that our default version (described below) can prove optimality of each instance within about one hour.

In our experiments, we use a time-limit of two hours. We initialize the algorithm with an optimal solution in order to minimize the effect of heuristics. Table 2 sums up the experiments. We report the number of instances that could

Table 1. Partial list of benchmark instances. On the left part are Color02 instances; on the top right DIMACS clique instances, and on the bottom right complement graphs of DIMACS clique instances.

Name	k_1	k_2	k_3	k_4
1-FullIns_4	3			
1-Insertions_4	3			
2-FullIns_4	3			
3-FullIns_4	5	6		
4-FullIns_3	3			
4-FullIns_4	5	6	7	
5-FullIns_3	3			
5-FullIns_4	5	6	7	8
DSJC125.9	4	5	6	
DSJC250.9	3	4		
DSJR500.1c	3	4	5	
DSJR500.1	8	9	10	11
myciel5	4	5		
myciel6	3			
queen6_6	6			

Name	k_1	k_2	k_3	k_4	k_5	k_6
san200_0.9_1	4					
c-fat500-2	15	17	20	22	25	
c-fat500-10	10	15	17	20	22	25
gen200_p0.9_55	4					
c-fat200-1	10					
gen200_p0.9_44	4					
sanr200_0.9	4					
san200_0.9_2	4					
MANN_a27	15	17	20	22	25	
johnson8-4-4c	4					
c-fat500-1c	4	5	6	6		
c-fat200-2c	7	8				
c-fat200-1c	6	7	8	9		
hamming6-4c	4	5				

not be solved within two hours (column '>2h'), and we report the shifted geometric mean[1] of the number of nodes in the search tree as well as the shifted geometric mean of the CPU time in seconds (columns 'nodes' and 'time', respectively). If an instance times out, its running time is evaluated as 2h, and we use the number of nodes produced within 2h for the computation of the geometric means.

Our default settings are as follows:

Preprocessing: Remove nodes with low degree smaller than k.
Node labeling: Choose a clique as the first nodes in the labeling, then label nodes with decreasing degree.
Cutting Planes: Neighborhood inequalities are added to $\mathrm{IP}_k(G)$, stable set inequalities (14) are separated heuristically in the root node, and clique inequalities are separated at every fifth depth of the tree.
Symmetry Handling: Use orbitopal fixing for color symmetries.
Branching Rule: Use the uncolored Sewell's rule.

We separate cutting planes other than clique inequalities only at the root node; this seems to be generally superior for our test set. In the following experiments, we add/remove features to the default settings in order to asses the importance of the features on the overall performance.

[1] The shifted geometric mean of values $t_1, \dots, t_n$ 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.

Table 2. Summary of experiments

Variant	>2h	nodes	time
default	0	251.0	104.7
OFSCI	1	275.4	127.7
noOFSCI	2	329.3	111.2
noOFnoSCI	16	1462.5	449.5
CliqueSCI	9	284.2	238.8
PackingClique	0	275.3	117.7
IncDeg	19	1246.0	480.9
DecDeg	20	1526.7	431.2
CliqueIncDeg	2	378.7	135.7
SymGOrder	14	848.2	312.5
Random	17	1269.7	445.3

Variant	>2h	nodes	time
firstIndex	1	288.5	126.2
TruncFirstIndex	0	192.0	108.8
UCVertex	0	268.1	109.4
OddCycle	0	296.9	145.3
DomNode	1	292.2	112.9
DomNodeG$\mathfrak{S}$	0	245.1	85.7
G$\mathfrak{S}$	0	249.1	84.0
GGen	0	218.0	86.9
G$\mathfrak{S}$SymGOrder	14	832.5	273.8
GGenSymGOrder	13	748.1	267.4
SSB	13	869.9	270.2
GGenSSB	13	808.5	274.6

Experiment 1: Color Symmetry Handling. In the first experiment, we evaluate the effect of different degrees of color symmetry handling: separating SCIs with/without orbitopal fixing (OFSCI/noOFSCI in Table 2), no color symmetry handling (noOFnoSCI), additionally separating clique shifted column inequalities (CliqueSCI), and packing-clique inequalities (PackingClique). The default is to not separate SCIs, but perform orbitopal fixing.

Any form of color symmetry handling clearly outperforms the variant without color symmetry handling. The default settings are the fastest. Orbitopal fixing as a pure constraint propagation algorithm outperforms the variants that separate shifted column inequalities (the same is observed in [21]), which can be explained by the fact that separating SCIs seems to yield more difficult LPs. The strengthened inequalities (PackingClique/CliqueSCI) do not yield an improvement over the default or variant 'noOFSCI'.

Experiment 2: Node Labelings. Here we investigate the effect of the node labelings on the performance. Our results clearly indicate their high impact. The default settings are the best choice. The other variants increasing/decreasing degree (IncDeg/DecDeg), keep symmetry groups together (SymGOrder), and random ordering (Random) all perform much worse. Note that sorting a clique to the front, as done in the default settings, has a high positive impact.

Experiment 3: Branching Rules. We compare the two vertex based branching rules 'uncolored vertex' (UCVertex) and 'uncolored Sewell' (default) with the variable branching rules 'first index' (FirstIndex) and 'truncated first index' (TruncFirstIndex). Here, 'first index' has the worst performance, while the other variants are not far apart w.r.t. their running time. The variant 'truncated first index' produces the fewest number of nodes in the tree, but is not the fastest, probably because of the time used for strong-branching.

Experiment 4: Cutting Planes. In this experiment, we compare separating clique inequalities (default) with the additional separation of odd cycle inequalities at the root node (OddCycle). Clearly, separating odd cycle inequalities does not yield an improvement to the default settings.

Experiment 5: Graph Symmetry Handling. In the final experiment, we compare different forms of graph symmetry handling and dominated node inequalities. We evaluate handling dominated nodes (DomNodes), symmetric subgroups of the graph symmetry group (G$\mathfrak{S}$), their combination (DomNodeG$\mathfrak{S}$), generator symmetry (GGen) which subsume 'G$\mathfrak{S}$', symmetric subgroups with SSB Constraints (SSB), and combining generator symmetry with SSB Constraints (GGenSSB). Whenever we use SSB constraints, we use the node labeling 'SymGOrder' in order to be able to combine SSB Constraints with orbitopal fixing. We further present the results for G$\mathfrak{S}$ and GGen w.r.t. the labeling 'SymGOrder' (G$\mathfrak{S}$SymGOrder/GGenSymGOrder).

Handling dominated nodes deteriorates the performance of the default settings; however, handling symmetric subgroups additionally yields an improvement on the default version. The overall best settings are obtained by handling symmetric subgroups only. The gain in time over the default version is approximately 20% in running time. The strongest form of graph symmetry handling (GGen) produces the least of amount of search nodes for the default branching rule; however, the running time is slightly worse than handling symmetric subgroups only.

Any graph symmetry handling in combination with labeling 'SymGOrder' clearly outperforms the variant with labeling 'SymGOrder' without graph symmetry handling. However, the handling of additional product symmetries via SSB constraints does not seem to have a positive impact on the solving time. The difference in running time is fairly small for all graph symmetry handling with labeling 'SymGOrder'. Overall they are inferior to graph symmetry handling with the default labeling, e.g., version 'G$\mathfrak{S}$' vs. 'G$\mathfrak{S}$SymGOrder'.

5 Conclusion and Future Work

The branch-cut-and-propagate algorithm that we presented in this paper provides an efficient way to solve a symmetric formulation of the maximum k-colorable subgraph problem. The performance of the algorithm can be substantially improved by handling symmetries via a combination of inequalities and domain propagation. In particular, we have demonstrated that handling graph symmetries in this way is very useful.

Future work will involve a more detailed computational analysis of the results in this paper. Moreover, one could identify (strong) linearizations for lexicographic ordering constraints for general graph symmetries and SSB constraints; this would help to polyhedrally handle additional symmetries.

References

1. Second DIMACS implementation challenge: Maximum clique, graph coloring, and satisfiability (1993), `ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/clique/`
2. COLOR 2002 – computational symposium: Graph coloring and its generalizations (2002), `http://mat.gsia.cmu.edu/COLOR02`
3. Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optimization 4(1), 4–20 (2007)
4. Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation 1(1) (2009)
5. 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)
6. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: A new approach to integrate CP and MIP. In: Trick, M.A. (ed.) CPAIOR 2008. LNCS, vol. 5015, pp. 6–20. Springer, Heidelberg (2008)
7. Berthold, T., Pfetsch, M.E.: Detecting orbitopal symmetries. In: Operations Research Proceedings 2008, pp. 433–438. Springer, Heidelberg (2009)
8. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K., Smith, B.: Symmetry definitions for constraint satisfaction problems. Constraints 11, 115–137 (2006)
9. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR 1996), pp. 148–159. Morgan Kaufmann, San Francisco (1996)
10. 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)
11. Flener, P., Pearson, J., Sellmann, M.: Static and dynamic structural symmetry breaking. Annals of Mathematics and Artificial Intelligence 57(1), 37–57 (2009)
12. Fomin, F.V., Grandoni, F., Kratsch, D.: A measure & conquer approach for the analysis of exact algorithms. Journal of the ACM 56:25:1–25:32 (2009)
13. Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Propagation algorithms for lexicographic ordering constraints. Artificial Intelligence 170, 803–834 (2006)
14. Frucht, R.: Herstellung von Graphen mit vorgegebener abstrakter Gruppe. Compositio Mathematica 6, 239–250 (1938)
15. Garey, M.R., Johnson, D.S.: Computers and Intractability. A Guide to the Theory of NP-Completeness. W. H. Freeman and Company, New York (1979)
16. Gent, I.P., Petrie, K.E., Puget, J.-F.: Symmetry in constraint programming. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, pp. 329–376. Elsevier, Amsterdam (2006)
17. Grötschel, M., Lovász, L., Schrijver, A.: Geometric Algorithms and Combinatorial Optimization, 2nd edn. Algorithms and Combinatorics, vol. 2. Springer, Heidelberg (1993)
18. Januschowski, T., Pfetsch, M.E.: The maximal k-colorable subgraph problem and orbitopes (2010) (preprint), `http://www.optimization-online.org/DB_HTML/2010/11/2821.html`

19. Januschowski, T., Pfetsch, M.E.: Branch-cut-and-propagate for the maximum k-colorable subgraph problem (2011) (preprint), `http://www.optimization-online.org/DB_HTML/2011/02/2909.html`
20. Johnson, D.S.: The NP-completeness column: An ongoing guide. V. Journal of Algorithms 3, 381–395 (1982)
21. Kaibel, V., Peinhardt, M., Pfetsch, M.E.: Orbitopal fixing. In: Fischetti, M., Williamson, D.P. (eds.) IPCO 2007. LNCS, vol. 4513, pp. 74–88. Springer, Heidelberg (2007)
22. Kaibel, V., Pfetsch, M.E.: Packing and partitioning orbitopes. Mathematical Programming 114(1), 1–36 (2008)
23. Katsirelos, G., Narodytska, N., Walsh, T.: On the complexity and completeness of static constraints for breaking row and column symmetry. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 305–320. Springer, Heidelberg (2010)
24. Koster, A.M.C.A., Ruepp, S.: Benchmarking RWA strategies for dynamically controlled optical networks. In: Proceedings of the Thirteenth International Telecommunications Network Strategy and Planning Symposium (NETWORKS 2008), pp. 1–14 (2008)
25. Koster, A.M.C.A., Scheffel, M.: A routing and network dimensioning strategy to reduce wavelength continuity conflicts in all-optical networks. In: Proceedings of the International Network Optimization Conference (INOC 2007), Spa, Belgium (2007)
26. Linderoth, J., Ostrowski, J.P., Rossi, F., Smriglio, S.: Orbital branching. In: Fischetti, M., Williamson, D.P. (eds.) IPCO 2007. LNCS, vol. 4513, pp. 104–118. Springer, Heidelberg (2007)
27. Lucet, C., Mendes, F., Moukrim, A.: Pre-processing and linear-decomposition algorithm to solve the k-colorability problem. In: Ribeiro, C.C., Martins, S.L. (eds.) WEA 2004. LNCS, vol. 3059, pp. 315–325. Springer, Heidelberg (2004)
28. Margot, F.: Pruning by isomorphism in branch-and-cut. Mathematical Programming 94(1), 71–90 (2002)
29. Margot, F.: Small covering designs by branch-and-cut. Mathematical Programming 94(2-3), 207–220 (2003)
30. Margot, F.: Symmetric ILP: Coloring and small integers. Discrete Optimization 4(1), 40–62 (2007)
31. Margot, F.: Symmetry in integer linear programming. In: Jünger, M., Liebling, T., Naddef, D., Nemhauser, G.L., Pulleyblank, W., Reinelt, G., Rinaldi, G., Wolsey, L. (eds.) 50 Years of Integer Programming 1958–2008, ch. 17, pp. 647–681. Springer, Heidelberg (2010)
32. McKay, B.D.: Practical graph isomorphism. In: Congressus Numerantium, pp. 45–87 (1981)
33. Méndez-Díaz, I., Zabala, P.: A polyhedral approach for graph coloring. Electronic Notes in Discrete Mathematics 7, 178–181 (2001)
34. Méndez-Díaz, I., Zabala, P.: A branch-and-cut algorithm for graph coloring. Discrete Applied Mathematics 154(5), 826–847 (2006)
35. Méndez-Díaz, I., Zabala, P.: A cutting plane algorithm for graph coloring. Discrete Applied Mathematics 156(2), 159–179 (2008)
36. Nemhauser, G.L., Wolsey, L.A.: Integer and combinatorial optimization. Wiley-Interscience, New York (1988)
37. Ostrowski, J., Linderoth, J., Rossi, F., Smriglio, S.: Orbital branching. Mathematical Programming 126(1), 147–178 (2011)

38. Puget, J.-F.: On the satisfiability of symmetrical constrained satisfaction problems. In: Komorowski, J., Raś, Z.W. (eds.) ISMIS 1993. LNCS, vol. 689, pp. 350–361. Springer, Heidelberg (1993)
39. SCIP. Solving Constraint Integer Programs, `http://scip.zib.de`
40. Sewell, E.C.: An improved algorithm for exact graph coloring. In: Johnson, D.S., Trick, M. (eds.) Cliques, Coloring, and Satisfiability. Second DIMACS Implementation Challenge. Proceedings of a Workshop held at DIMACS, 1993. Ser. Discrete Math. Theor. Comput. Sci., vol. 26, pp. 359–373. AMS, DIMACS (1996)

Climbing Depth-Bounded Adjacent Discrepancy Search for Solving Hybrid Flow Shop Scheduling Problems with Multiprocessor Tasks

Asma Lahimer[1], Pierre Lopez[1], and Mohamed Haouari[2]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31077 Toulouse Cedex 4, France
Université de Toulouse, UPS, INSA, INP, ISAE, UT1, UTM, LAAS,
F-31077 Toulouse Cedex 4, France
{asma.lahimer,pierre.lopez}@laas.fr

[2] INSAT, Institut National des Sciences Appliquées et de Technologie
Centre Urbain Nord BP 676 - 1080 Tunis Cedex, Tunisie
mohamed.haouari@insat.rnu.tn

Abstract. This paper considers multiprocessor task scheduling in a multistage hybrid flow-shop environment. The problem even in its simplest form is NP-hard in the strong sense. The great deal of interest for this problem, besides its theoretical complexity, is animated by needs of various manufacturing and computing systems. We propose a new approach based on limited discrepancy search to solve the problem. Our method is tested with reference to a proposed lower bound as well as the best-known solutions in literature. Computational results show that the developed approach is efficient in particular for large-size problems.

Keywords: Hybrid flow shop scheduling, Multiprocessor tasks, Discrepancy search.

1 Introduction

Flow shop scheduling refers to a manufacturing facility in which all jobs visit the production machines in the same order. In hybrid flow shop scheduling, the jobs serially traverse stages following the same production route, and must be assigned to one of the parallel machines composing each stage. The hybrid flow shop scheduling problem with multiprocessor tasks is itself a generalization of the hybrid flow shop problem, allowing tasks to be processed on more than one processor in a given stage, at a time. It can also be viewed as a specific case of the resource-constrained project scheduling problem (RCPSP).

Many applications of hybrid scheduling problems with multiprocessor tasks can be found in various manufacturing systems (*e.g.*, work-force assignment in [6], transportation problem with recirculation in [4]), as well as in some computer systems (*e.g.*, real-time machine-vision [8]).

Hybrid flow shop scheduling problem with multiprocessor tasks has received considerable attention from researchers and has been solved by various approaches, *e.g.* genetic algorithms [16], tabu search, and ant colony system [21].

Motivated by the success of discrepancy search for solving shop scheduling problems, in particular hybrid flow shop [2], [3], we propose in this paper a new approach based on discrepancy search to solve the hybrid flow shop problem with multiprocessor tasks.

2 Problem Definition

The hybrid flow shop scheduling problem with multiprocessor tasks can be formally described as follows: A set $\mathbf{J}=\{1,2,\ldots,n\}$ of n jobs, have to be processed in m stages. Hence, a job is a sequence of m tasks (one task for each stage). Each stage $\mathbf{i}=\{1,2,\ldots,m\}$ consists of m_i identical parallel processors. In a stage i, the job j requires simultaneously $size_{ij}$ processors. That is, $size_{ij}$ processors selected at stage j are required for processing job j for a period of time equal to the processing time requirement of job j at stage i, namely p_{ij}. The objective is to minimize the *makespan* ($C_{\max}$), that is, the completion time of all tasks in the last stage. According to the classical 3-field notation in production scheduling, the problem is denoted by Fm(m_1,..,m_m)$|size_{ij}|C_{\max}$.

3 Discrepancy Search

3.1 General Statement

Limited discrepancy search (LDS) was introduced in 1995 by Harvey and Ginsberg [10]. This seminal method can be considered as an alternative to the branch-and-bound procedure, backtracking techniques, and iterative sampling. From an optimization view-point this technique is similar to variable neighbourhood search. Discrepancy search has been further extended in the literature [9,13] to become Local Branching applied to Mixed-Integer Programs (MIPs) and Constraint Programming (CP). The neighbourhood in local branching is so defined using the spirit of limited discrepancy search.

Indeed, it starts from an initial global instantiation suggested by a given heuristic and successively explores branches with increasing discrepancies from it, in order to obtain a solution (in a satisfaction context), or a solution of better performance (in an optimization context). A discrepancy is associated with any decision point in a search tree where the choice goes against the heuristic. For convenience, in a tree-like representation the heuristic choices are associated with left branches while right branches are considered as discrepancies. Figure 1 illustrates the spirit of LDS. At *kth* iteration, solutions having discrepancies between 0 and k are visited. The first line in the figure illustrates the order its branches are visited while the second line shows the number of discrepancies associated with each solution. Since LDS proposition in 1995, several variants were suggested, among them, Improved Limited Discrepancy Search (ILDS) [14], Depth-bounded Discrepancy Search (DDS) [23], Discrepancy-Bounded Depth First Search [1] and Climbing Discrepancy Search (CDS) [15].

In the following sections, we focus on those methods that inspired our approach, in particular DDS and CDS.

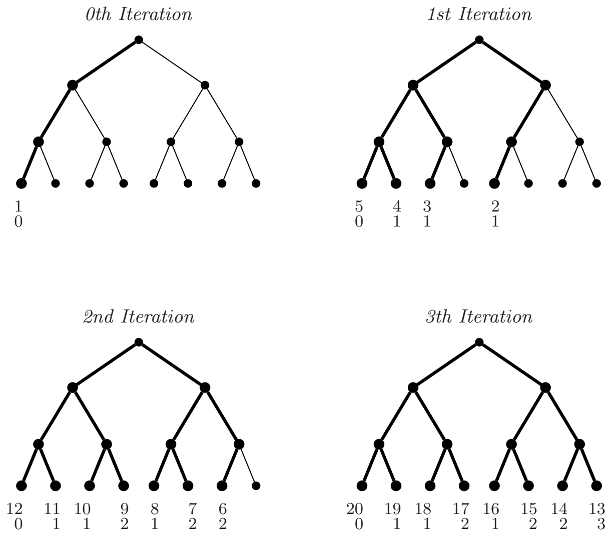


Fig. 1. Limited Discrepancy Search

3.2 Depth-Bounded Discrepancy Search

Depth-bounded Discrepancy Search (DDS) developed in [23], is an improved LDS that prioritizes discrepancies at the top of the tree to correct early mistakes first. This assumption is ensured by means of an iteratively increasing bound on the tree depth. Discrepancies below this bound are prohibited. DDS starts from an initial solution. At *ith* iteration, it explores those solutions on which discrepancies occur at a depth not greater than i.

3.3 Climbing Discrepancy Search

Climbing Discrepancy Search (CDS) is a local search method adapted to combinatorial optimization problems proposed in [15]. CDS starts from an initial solution that would be dynamically updated. Indeed, it visits branches progressively until a better solution is reached. Then, the initial solution is updated and the exploration process is restarted.

4 Proposal: Climbing Depth-Bounded Adjacent Discrepancy Search

4.1 CDADS: Main Features

To stick to the problem under consideration, we now consider an optimization context. We propose CDADS (Climbing Depth-bounded Adjacent Discrepancy Search) method, that is a combination of a depth-bounded discrepancy search and a climbing discrepancy search. We also assume that, if several discrepancies

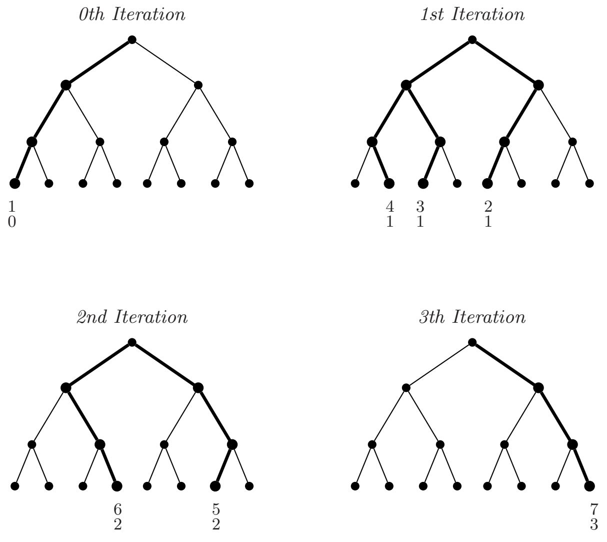


Fig. 2. Depth-bounded Ajacent Discrepancy Search

occur in the construction of a solution, these discrepancies are necessarily *adjacent* in the list of successive decisions. CDADS starts from an initial solution obtained by a given heuristic, and explores its neighborhood progressively, according to the depth-bounded discrepancy search strategy. Hence, a limit depth d is fixed. Discrepancies below this bound are prohibited. At *ith* iteration, we allow i discrepancies above the limit level d.

When considering solutions with more than one discrepancy, we require these discrepancies are achieved consecutively, that means a solution consists of discrepancies that happen one after the other. This assumption of adjacency considerably limits the search space. We also consider that the initial solution is generated by a 'good' heuristic. Thus, only the immediate neighborhood of a discrepancy may receive an additional discrepancy. Even if we are aware that other strategies for limiting the search space could be envisaged (focusing for example on given subsets of discrepancies), we make the bet that only performing adjacent discrepancies is promising. We then obtain a truncated DDS based on adjacent discrepancies, DADS (Depth-bounded Adjacent Discrepancy Search). This approach is illustrated by an example on a binary tree of depth 3 (see Figure 2).

At the starting point, DADS visits the initial solution recommended by the heuristic. For convenience, we assume that left branches follow the heuristic. At first iteration, DADS visits leaf nodes at the depth limit with exactly one discrepancy. The first line shown under the branches reports the visit order of considered solution, while the second line illustrates the number of discrepancies made in each solution. The 2nd iteration allows to exploring more solutions with two discrepancies with respect to the adjacency assumption. In this representation, the maximum depth bound is taken to be 3. If now, we limit the depth to two levels, several branches would not be retained, namely the branches 4, 6, and 7 would not be visited by DADS.

Going back to the optimization issue, CDADS merges the DADS strategy with a CDS exploration principle, that is the initial solution used by DADS is dynamically updated when a best solution is found, and the exploration process is restarted.

4.2 Heuristics

CDADS is strongly based on the quality of the initial solution. Thus, we carried out an experimental comparison between various priority rules presented in the literature [21], [17]. We considered the most effective heuristics to multiprocessor task hybrid flow shop scheduling. The four selected rules are:

- **SPT** (Shortest Processing Time), which ranks jobs according to the ascending order of their processing times;
- **SPR** (Shortest Processing Requirement), which ranks jobs according to the ascending order of their processing requirement;
- the **Energy** rule, considering first the jobs with the smallest energy (where the energy of an operation j at a stage i is evaluated by $p_{ij} \times size_{ij}$); and
- **NSPT_LastStage** (Normalized SPT applied at the last stage). For this latest rule, Şerifoğlu and Ulusoy [21] propose to schedule jobs according to their ranking index (RI_j) defined by:

$$RI_j = \frac{\max_k\{p_{mk}\} - p_{mj} + 1}{\max_k\{p_{mk}\} + 1}.$$

In Table 1, the selected priority rules are ranked according to their percentage of best solutions found, that is, performance.

4.3 Schedule Generation Scheme

Schedule generation schemes (SGSs) are widely used in solving preemptive problems. We distinguish between serial SGS and parallel SGS. These two heuristics ensure task scheduling based on a given priority rule. Hence, tasks are selected one after the other and a start time is fixed for each one.

Table 1. Heuristic selection

Priority Rule	Performance (%)
NSPT_LastStage	27
Energy	25
SPT	17
SPR	14

Serial SGSs are introduced in [12]. At each iteration, the first available task in ζ is selected, where ζ is the priority list recommended by the priority rule. The selected task is scheduled as soon as possible with respect to both resource constraints and precedence constraints.

Parallel SGSs developed in [5], suggest a chronological procedure in scheduling tasks. At each time t, a set ζ_t of tasks being scheduled is defined: this set contains unscheduled tasks that can be processed at t without breaking neither precedence constraints nor resource constraints. If we consider that $\underline{t}$ is the first time where $\zeta_{\underline{t}} \neq \emptyset$, the first task in the priority list ζ belonging to $\zeta_{\underline{t}}$ is performed at $\underline{t}$. The same process is applied until all tasks are scheduled. The two schemes depicted above may appear similar. However, the schedule they generate are different: a serial SGS provides an active schedule while a parallel SGS generates a non-delay schedule.

In the scheduling theory, Sprecher *et al.* [22] show that the set of active schedules includes at least one optimal solution. On the contrary, non-delay schedules may eliminate all optima.

Concerning our method CDADS, we do not enumerate all possible solutions, so even serial SGSs may exclude all optimum solutions. Furthermore, in practice, parallel SGSs are known for their operational efficiency. Hence, we opt for the implementation of a parallel SGS which has been proved, moreover, to be more efficient in our experimental studies.

4.4 Lower Bound

For efficiency purpose, we join CDADS with an evaluation of lower bounds at each node. The proposed lower bound is based on lower bounds previously presented in [16]. Even though our bound presented below is largely taken from this latest reference, the mathematical expression was revised in two places to be more accurate. Thus, we suggest this formula:

$$LB = \max(LB^s, LB^j)$$

where LB^j is a job-based lower bound similar to the one suggested in [16]: $LB^j = \max_{j \in J}(\sum_{i=1}^{m} p_{ij})$; and LB^s is a stage-based lower bound: $LB^s = \max_{i=1..m} LB(i)$.

For this latter bound, we claim that:

$$LB(i) = \begin{cases} \max[M_1(i), M_2(i), \max_{j \in J}(p_{ij})] + \min_{j \in J}(\sum_{l=i+1}^{m} p_{lj}) \ , & \forall i = 1 \\ \min_{j \in J}(\sum_{l=1}^{i-1} p_{lj}) + \max[M_1(i), M_2(i), \max_{j \in J}(p_{ij})] + \min_{j \in J}(\sum_{l=i+1}^{m} p_{lj}) \ , & \forall i = 2..m-1 \\ \min_{j \in J}(\sum_{l=1}^{i-1} p_{lj}) + \max[M_1(i), M_2(i), \max_{j \in J}(p_{ij})] \ , & \forall i = m \end{cases}$$

where
$$M_1(i) = \left\lceil \frac{1}{m_i} \sum_{j \in J} (p_{ij} size_{ij}) \right\rceil$$
and
$$M_2(i) = \sum_{j \in A_i} p_{ij} + \frac{1}{2} \sum_{j \in B_i} p_{ij} \ ,$$
with
$$A_i = \{j | size_{ij} > \frac{m_i}{2}\}$$
and
$$B_i = \{j | size_{ij} = \frac{m_i}{2}\}.$$

Justification of the expression of $LB(i)$.

We assume that only non-delay task scheduling is considered.

The first term of $LB(i)$ gives a lower bound on the beginning of every job $j \in J$ on any machine of stage i.

The last term can be explained accordingly, since it is associated with the minimal required time to achieve the processing of every job j on all the subsequent stages of stage i.

The middle term concerns the processing of jobs on stage i. $M_1(i)$ stands for the mean stage load for job preemptive scheduling, while $M_2(i)$ reviews two different situations for partitionning the jobs according to their resource requirement. Set A_i consists of jobs that must be processed sequentially (resource requirement greater than the half of the resource capacity m_i). Set B_i groups together the jobs having a resource requirement exactly equal to the half of the resource capacity. Obviously, a job belonging to A_i and another job belonging to B_i must also be processed sequentially. The added term $\max_{j \in J}(p_{ij})$ (that is one of the revisions) contributes to maximize the evaluation of stage load on a considered stage i, especially when some jobs having high processing time are being scheduled.

This justifies the validity of the bound. □

5 Computational Study

5.1 Test Beds

For comparison purpose, we assess the performance of CDADS on instances of Oğuz's benchmark available on her home page: `http://home.ku.edu.tr/coguz/public_html/`. This benchmark is widely used in the literature [20], [11], [18].

The number of jobs is taken to be $n = 5, 10, 20, 50, 100$ and the number of stages m takes its value from the set $\{2, 5, 8\}$. The benchmark considers two types of problems, “Type-1” and “Type-2”. In ‘Type-1’ instances, the number of processors m_i available at each stage i (resource capacity) is randomly determined from the set $\{1, \ldots, 5\}$, while in ‘Type-2’ m_i is fixed to 5 processors for every stage i. In fact, ‘Type-2’ instances are globally more flexible than ‘Type-1 instances’. For each combination of n and m, and for each type, 10 instances are randomly generated, which leads a total of 300 instances. The processing time of each job j in stage i (p_{ij}) and its processing requirement ($size_{ij}$) are integer and are randomly generated from sets $\{1, \ldots, 100\}$ and $\{1, \ldots, m_i\}$, respectively.

The algorithm implementing CDADS was coded in C++ and run on an Intel core 2 Duo 2 GHz PC. The maximum CPU time is set to 60 seconds. The exploration is also stopped when CDADS reaches a given lower bound on the makespan. Obviously, if CDADS misses the optimal solution, the best-found solution when the maximum CPU time is reached, is then taken to be the problem solution.

5.2 Restart Policy

For the computational study, we have then retained four priority rules to generate the initial solutions (see Section 4.2). That is why whe have introduced a restart policy to benefit from these heuristics. At a starting point, we use the best rule, that is the NSPT_LastStage. However, if no improvement is noticed during the CDADS search, we restart the process with another solution obtained by applying the next rule “Energy” that could lead a more efficient solution for this specific instance, and so on.

The restart policy is limited by the size of the heuristics pool: restarts are then allowed at most four times, since we have selected four rules. At each restart k (starting from $k = 0$), we increase the number of maximum nodes that can be visited according to a geometrical series $nbrNodes \times f^k$, where f is fixed to 1.3 and *nbrNodes* varies linearly with the problem size (the number of jobs n; for example for $n = 20$ we fix *nbrNodes* to 2000 nodes). Hence the search space is expanded at each restart.

5.3 Results

We tested two strategies for applying discrepancy: Top First and Bottom First. In the Top First exploration, discrepancies at the top of the tree are privileged while the Bottom First strategy favors discrepancies at the bottom. Computational study shows that CDADS is really more efficient with a Top First strategy (then contradicting – for the problem at hand – the statement of relative indifference of discrepancy order by [19]). Thus, the results shown below refer to this latter strategy.

Table 2 gives for each configuration (n: number of jobs, and m: number of stages) and each type, the average percentage deviation (*%dev*) and the average CPU time. The average percentage deviation is measured in two ways:

- For small problems, solutions are compared to the optimal solutions ($C^*_{\max}$ denotes the optimum makespan):

$$\frac{C_{\max} - C^*_{\max}}{C^*_{\max}} \times 100;$$

- For larger problems, solutions found by the CDADS are compared to the lower bound (LB):

$$\frac{C_{\max} - LB}{LB} \times 100.$$

As explained in Section 5.2, CDADS is run four times on each of the selected priority rules (NSPT_LastStage, Energy, SPT, SPR) for each instance. The best solution is taken to be the CDADS solution for the corresponding problem. According to findings of [21], the Fm(m_1,...,m_m)|$size_{ij}$|$C_{\max}$ problem and its symmetric have the same optimal makespan. Referring to this property, we apply a two-directional planning (forward schedule and backward schedule).

From Table 2, it is observed that the average percentage deviation is higher for 'Type-1' instances. Globally, *%dev* is 1.66% for 'Type-1' problems and 6.39% for 'Type-2' problems. This increase can be linked to several assumptions: the lower bound becomes less effective as m_i increases in 'Type-2' instances and so the average percentage deviation would be higher. Another explanation can also be considered: the number of processors are fixed in 'Type-2' problems, that is $m_i = 5$, and the scheduling problem becomes more difficult to solve for CDADS.

Table 2. CDADS performance

		'Type-1' Problems		'Type-2' Problems	
n	*m*	*%dev*	*CPU(s)*	*%dev*	*CPU(s)*
5	2	0	< 0.1	0	< 0.1
	5	0.21	< 0.1	0.46	< 0.1
	8	1.71	< 0.1	0.5	< 0.1
10	2	0	< 0.1	1.72	< 0.1
	5	0.66	0.4	6.44	< 0.1
	8	8.47	< 0.1	9.61	0.2
20	2	0.05	0.1	3.34	3.1
	5	2.57	1.1	7.97	1.3
	8	5.11	0.2	15	1.3
50	2	0.49	2.3	1.74	4.2
	5	0.54	5	8.2	13.5
	8	1.62	6.8	12.42	33.4
100	2	0.08	11.1	3.32	22.8
	5	1.5	13.6	10.75	40.9
	8	1.86	11	14.33	47.3
Global average		1.66	3.44	6.39	10.53

Results show the behavior of our approach with variations of n and m. For a given n, the average percentage deviation increases with increasing m. Indeed, the problem difficulty increases when m increases and the obtained solution is further away from the lower bound. On the other hand, for a given number of stages m, increasing n has no significant effect on the average percentage deviation, as the effectiveness of CDADS is independent of the number of jobs: the stability of our method seems to be not linked to the number of jobs n, since for a given m (*e.g.*, $m = 8$), in 'Type-1' problems, when n increases from 50 jobs to 100 jobs, the average percentage deviation increase slightly (from 1.62% to 1.86%). It also can be noticed, that in some cases, increasing n results in a decrease in the deviation value (for the configuration $n = 20, m = 8$ the $\%dev$ is taken to be 5.11%, and is evaluated to 1.62% for $n = 50, m = 8$). Apparently, the lower bound becomes more effective with n increasing.

From the experimental studies, it can be observed that CDADS converges quickly. The average CPU time varies between less than 0.1 seconds and 47.3 seconds. The computational cost is more important in 'Type-2' instances, confirming the difficulty of these problems. Similarly, for a fixed m, increasing n leads to CPU time increase. Conversely, when n is fixed, increasing m increases the CPU time.

5.4 Comparison of CDADS Solutions with State-Of-the-Art Results

Table 3 presents the results of CDADS on $\%dev$, the average percentage deviation (as well as a synthesis of the average CPU time for all instances, in the last line of the table). Furthermore, it shows the results obtained by Jouglet *et al.* in [11]. These results are the most recent and the best-known solutions in literature. Thus, we have compared the results of CDADS with GA (genetic algorithm), CP (constraint programming), and MA (memetic algorithm). The CP approach is based on a branch-and-bound algorithm with constraint propagation techniques. The applied techniques are based on processing disjunctive constraints, edge-finding and energetic reasoning from the implementation in Ilog Scheduler. Concerning the memetic algorithm, it incorporates the constraint programming into a genetic algorithm as its local search engine. We disregard the results published by Ercan *et al.* [16] given inconsistency encountered. We contrast our results only versus those presented in [11]. However, we omit the average deviation published in this latest paper due to detected miscalculation (induced by Ercan *et al.*'s errors). Hence, we recalculated the average percentage deviation for all methods given in [11]. The maximum CPU time is fixed at 900 seconds for GA, CP, and MA.

As revealed in Table 3 (and as already noticed in Table 2), on the whole, the total average of $\%dev$ obtained by CDADS is 1.66% and 6.39% for the 'Type-1' and 'Type-2' problems, respectively. Compared to the corresponding averages of 2.27% and 7.28% achieved by GA, and the corresponding values of 5.39% and 11.92 % obtained by CP, CDADS outperforms the GA and CP algorithms. Furthermore, CDADS was clearly superior to CP especially for larger instances ($n = 50$ and $n = 100$).

Table 3. Comparing average percentage deviation (and CPU time)

		'Type-1' Problems				'Type-2' Problems			
n	*m*	*CDADS*	*GA*	*CP*	*MA*	*CDADS*	*GA*	*CP*	*MA*
5	2	0	0.29	0	0	0	1.23	0	0
	5	0.21	1.35	0	0	0.46	1.44	0	0
	8	1.71	4.15	0	0	0.5	2.38	0	0
10	2	0	0	0	0	1.72	2.83	1.72	1.75
	5	0.66	1.64	0	0	6.44	7.8	6.1	5.67
	8	8.47	9.38	10.32	8.02	9.61	10.87	8.37	8.8
20	2	0.05	0.44	2.59	0.66	3.34	3.7	6.72	3.43
	5	2.57	3.49	10.85	2.78	7.97	9.57	22.86	9.57
	8	5.11	5.69	17.98	5.32	15	17.26	28.52	16.02
50	2	0.49	0.63	2.79	0.49	1.74	2.76	6.54	2.21
	5	0.54	0.59	5.3	0.51	8.2	10.95	20.01	10.32
	8	1.62	2.17	14.42	1.71	12.42	15.89	30.06	17.25
100	2	0.08	0.15	1.96	0.07	3.32	3.05	5.68	2.7
	5	1.5	2.5	5.19	2.33	10.75	14.95	19.13	14.37
	8	1.86	1.99	9.47	2.15	14.33	20.06	23.15	17.83
Global average		**1.66**	2.27	5.39	1.6	**6.39**	7.28	11.92	8.32
Average CPU(s)		**3.44**	879.93	320.3	326.01	**10.53**	879.08	423.09	511.27

As depicted in the table, MA finds slightly better solutions in 'Type-1' problems, that is 1.60% is obtained by MA while CDADS gives an average deviation percentage of 1.66%. Overall, CDADS outperforms significantly MA, as CDADS results are at 6.39% from optimal solutions (or lower bounds) for 'Type-2' problems against 8.32% for MA.

To further assess the effectiveness of CDADS, we measure the number of improved known solutions. It can be seen from Table 4 that CDADS improves 75 known solutions among the 300 tested instances. Thus, the rate of improvement reaches 25%. The results also outline that most improvements are spotted in large instances ($n = 50, 100$), see figure 3. No significant improvements are noticed for small instances ($n = 5, 10$) since all optimal solutions for these problems are known.

In this study, we also compare the convergence of algorithms. It can be seen from the last line of Table 3, that CDADS outperforms the genetic algorithm (GA), constraint programming (CP), and the memetic algorithm (MA). Indeed, CDADS takes between less than 0.1 seconds (for small problems) and 47.3 seconds (for large problems) to find their solutions, while methods proposed in [11] converge much more slower [0.7 sec, 900 sec]. Even all results were obtained under different computational budgets, we can conclude that CDADS

Table 4. Number of improved solutions

n	‘Type-1’ Problems	‘Type-2’ Problems
5	0	0
10	1	0
20	5	10
50	8	20
100	8	23
total	22	53

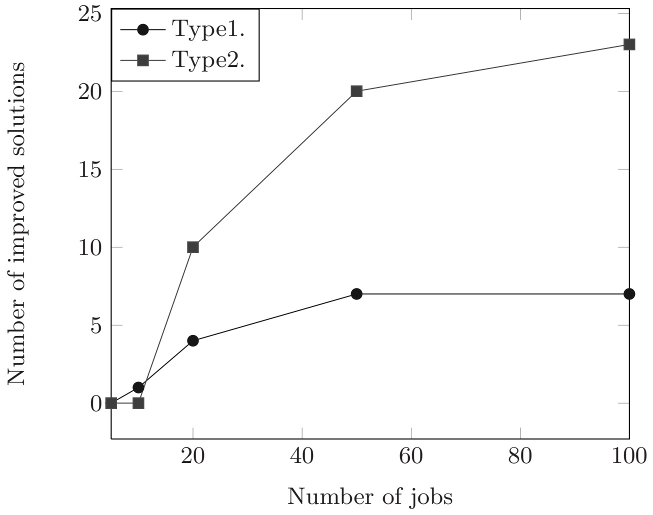


Fig. 3. Variation of the number of improved solutions with the number of jobs

demonstrates fast convergence. Indeed, according to Dongarra’s normalized coefficients [7], our machine is approximately only 3.5 times faster than the machine used by Jouglet *et al.*

6 Conclusions

In this paper, the hybrid flow shop problem with multiprocessor tasks is addressed by means of a discrepancy search method. The proposed method, Climbing Depth-bounded Adjacent Discrepancy Search (CDADS), is based on adjacent discrepancies. We selected several heuristics to generate the initial solution. A lower bound is also proposed to lead a more efficient search. Compared to the best-known results in the literature, CDADS provides better solutions in little CPU time.

In the very short-term, it would be beneficial for our study to explore the impact of adjacent discrepancies *vs.* other strategies for limiting the search space. Also, we would consider the application of CDADS to simpler problems like classical hybrid flow shop ($size_{ij} = 1, \ \forall\, i, j$), widely studied in the literature. Another expected aim would be to adapt the proposed implementation of discrepancy search to more general scheduling problems, in particular the Resource-Constrained Project Scheduling Problem, which still remains one of the most challenging problems in large-scale scheduling.

References

1. Beck, J.C., Perron, L.: Discrepancy-bounded depth first search. In: Proceedings of CPAIOR 2000, pp. 8–10 (2000)
2. Ben Hmida, A., Haouari, M., Huguet, M.-J., Lopez, P.: Solving two-stage hybrid flow shop using climbing depth-bounded discrepancy search. Computers and Industrial Engineering 60(2), 320–327 (2010)
3. Ben Hmida, A., Huguet, M.-J., Lopez, P., Haouari, M.: Climbing depth-bounded discrepancy search for solving hybrid flow shop scheduling problems. European Journal of Industrial Engineering 1(2), 223–243 (2007)
4. Bertel, S., Billaut, J.-C.: A genetic algorithm for an industrial multiprocessor flow shop scheduling problem with recirculation. European Journal of Operational Research 159(3), 651–662 (2004)
5. Brooks, G., White, C.: An algorithm for finding optimal or near optimal solutions to the production scheduling problem. Journal of Industrial Engineering 16, 34–40 (1965)
6. Chen, J., Lee, C.-Y.: General multiprocessor task scheduling. Naval Research Logistics 46, 57–74 (1999)
7. Dongarra, J.: Performance of various computers using standard linear equations software. Technical report, University of Tennessee (2009)
8. Ercan, M.F., Fung, Y.-F.: Real-time image interpretation on a multi-layer architecture. In: Proceedings of IEEE TENCON 1999, pp. 1303–1306 (1999)
9. Fischetti, M., Lodi, A.: Local branching. Mathematical Programming 98, 23–47 (2003)
10. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 1995), Montréal, Québec, Canada, vol. 1, pp. 607–615 (August 1995)
11. Jouglet, A., Oğuz, C., Sevaux, M.: Hybrid flow-shop: a memetic algorithm using constraint-based scheduling for efficient search. Journal of Mathematical Modelling and Algorithms 8, 271–292 (2009)
12. Kelley Jr, J.E.: The critical-path method: Resources planning and scheduling. In: Thompson, G.L., Muth, J.F. (eds.) Industrial Scheduling, pp. 347–365. Prentice-Hall, Englewood Cliffs (1963)
13. Kiziltan, Z., Lodi, A., Milano, M., Parisini, F.: CP-based local branching. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 847–855. Springer, Heidelberg (2007)
14. Korf, R.E.: Improved limited discrepancy search. In: Proceedings of the 13th National Conference on Artificial Intelligence (AAAI 1996), Portland, OR, vol. 1, pp. 286–291 (August 1996)

15. Milano, M., Roli, A.: On the relation between complete and incomplete search: an informal discussion. In: Proceedings of CPAIOR 2002, Le Croisic, France, pp. 237–250 (2002)
16. Oğuz, C., Ercan, M.F.: A genetic algorithm for hybrid flow-shop scheduling with multiprocessor tasks. Journal of Scheduling 8, 323–351 (2005)
17. Oğuz, C., Fung, Y.-F., Ercan, M.F., Qi, X.-T.: Parallel genetic algorithm for a flow shop problem with multiprocessor tasks. In: International Conference on Computational Science, Berlin, Heidelberg, pp. 548–559 (2003)
18. Oğuz, C., Zinder, Y., Ha Do, V., Janiak, A., Lichtenstein, M.: Hybrid flow shop scheduling problems with multiprocessor task systems. European Journal of Operational Research 152, 115–133 (2004)
19. Prosser, P., Unsworth, C.: LDS: testing the hypothesis. Technical Report DCS TR-2008-273, Dept of Computing Science, University of Glasgow (2008)
20. Şerifoğlu, F.S., Ulusoy, G.: Multiprocessor task scheduling in multistage hybrid flow-shops: A genetic algorithm approach. European Journal of Operational Research 55(5), 504–512 (2004)
21. Şerifoğlu, F.S., Ulusoy, G.: Multiprocessor task scheduling in multistage hybrid flow-shops: An ant colony system approach. International Journal of Production Research 44(16), 3161–3177 (2006)
22. Sprecher, A., Kolisch, R., Drexl, A.: Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem. European Journal of Operational Research 80(1), 94–102 (1995)
23. Walsh, T.: Depth-bounded discrepancy search. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), Nagoya, Japan, vol. 2, pp. 1388–1395 (August 1997)

On Counting Lattice Points and Chvátal-Gomory Cutting Planes

Andrea Lodi[1], Gilles Pesant[2], and Louis-Martin Rousseau[2]

[1] DEIS, Università di Bologna
andrea.lodi@unibo.it
[2] CIRRELT, École Polytechnique de Montréal
{Gilles.Pesant,Louis-Martin.Rousseau}@polymtl.ca

Abstract. The paper investigates the relationship between counting the lattice points belonging to an hyperplane and the separation of Chvátal-Gomory cutting planes. In particular, we show that counting can be exploited in two ways: (i) to strengthen the cuts separated, e.g., by Gomory classical procedure, and (ii) to heuristically evaluate the effectiveness of those cuts and possibly select only a subset of them. Empirical results on a small set of 0-1 Integer Programming instances are presented.

Keywords: Counting; Lattice points; Cutting Planes; Cut selection.

1 Introduction

We consider the *Integer Linear Program* (ILP) problem

$$\min\{c^T x : Ax \le b, x \ge 0 \text{ integer}\} \tag{1}$$

where A is a $m \times n$ matrix, $b \in I\!R^m$, and $c \in I\!R^n$ and we assume that (A, b) is an integer matrix. In addition, we assume all variables are upper bounded and the bound constraints are included in the system $Ax \le b$. We consider two associated polyhedra:

$$P := \{x \in I\!R^n_+ : Ax \le b\} \tag{2}$$
$$P_I := conv\{x \in Z^n_+ : Ax \le b\} = conv(P \cap Z^n). \tag{3}$$

Cutting plane generation is one of the most useful ingredients for solving ILPs and, more generally, Mixed Integer Linear Programming (MILP) problems, i.e., the general problems in which some of the variables are not restricted to assume integer values (see, Lodi [11] and Linderoth and Lodi [10] for dedicated surveys on MILP computation and software, respectively). Roughly speaking, a cutting plane, or cut in short, is a (redundant) linear inequality which is valid for P_I, i.e., does not cut off any feasible solution of the ILP at hand, and is violated by some solutions in P. In particular, the *separation* problem

> given a feasible solution $x^* \in P$, find a linear inequality $\alpha^T x \ge \alpha_0$ which is valid for (3), i.e., satisfied by all feasible solutions $\bar{x}$ of the system (1), while it is violated by x^*, i.e., $\alpha^T x^* < \alpha_0$, or prove that none exists

is iteratively solved and the cuts are added to P to improve the approximation.

A famous class of cutting planes for ILP is the classical *Chvátal-Gomory* (CG) cut family [9,6]. A CG cut is a valid inequality for P_I of the form

$$\lfloor u^T A \rfloor x \leq \lfloor u^T b \rfloor , \tag{4}$$

where $u \in R^m_+$ is called the CG *multiplier vector*, and $\lfloor \cdot \rfloor$ denotes lower integer part. It was proved by Eisenbrand [8] that separation of a generic $x^* \in P$ is NP-hard, but, in the special case in which x^* is fractional vertex of P associated with a certain basis B (say) of (A, I), Gomory [9] has shown that it can be cut off by the CG cut in which u is chosen as the i-th row of B^{-1}, where i is the row associated with any fractional component of x^*.

Roughly speaking, a CG cut is derived by: (i) combining with non-negative multipliers other valid constraints, i.e., $u^T Ax \leq u^T b$ is clearly valid for P_I, as well as its weakened version $\lfloor u^T A \rfloor x \leq u^T b$; and (ii) shifting the inequality $\lfloor u^T A \rfloor x \leq u^T b$ towards the interior of the polyhedron until it touches the first integral point, so as to obtain $\lfloor u^T A \rfloor x \leq \lfloor u^T b \rfloor$. The validity of shifting argument is guaranteed by the integrality of x.

It is easy to see that the integer points encountered by the inequality of step (i) above during the shifting of step (ii) do not necessarily belong to P_I.

Contribution of the paper. Although for decades the research focus for (M)ILPs has been on finding new families of cutting planes and separate them efficiently, nowadays it seems that the focus might be shifted on the *cut selection*, i.e., choosing within a large set of cutting planes to be separated or already separated, a small(er) set with the simultaneous goals of (1) improving the approximation, and (2) keeping the size of the LP relaxation small (to speed up the LP computation and avoid numerical troubles). A few attempts have been made in the literature in this direction (see, e.g., [1,2,3,15]) but the topic is mostly unexplored. In this paper we discuss ideas on cut selection (for the special case of CG cuts while extensions are briefly discussed in Section 4) originated by the recent and sophisticated work on counting lattice points developed in several research areas (see, e.g., Issue 81 of *Optima* [5]). The basic (simple) idea is as follows: we want to use the number of integer points to which a CG cut is simultaneously tight (satisfied at equality) as a quality measure of the cut itself, i.e., to discriminate among the CG cuts solving the separation problem at a given iteration which are the most promising to be part of the continuous relaxation of the next iteration. In particular, at a first glance, it seems reasonable to think that the higher the number of integer points the better, but we will show in the computational section that it might make sense to select (also) cuts with few tight integer points. This is conceivable for two reasons. First, the way in which we count the number of lattice points explicitly uses variable bounds. However, it might very well be that not only a tight integer point does not belong to P_I (as anticipated above) but in addition does not satisfy the bounds. If this is the case for all lattice points tight to a cut, then our counting procedure gives us a natural way of strengthening the cut by further shifting it.We have noticed that such a cut becomes generally strong. Second, we empirically observed that cuts tight to few integer points are much sparser than those tight to many.

In other words, selecting cuts tight to few lattice points keeps the LP relaxation sparse, which is a known criterion to control numerics and speed up the LP computation.

The paper is organized as follows. In Section 2 we present the algorithm we use for counting lattice points and we show how to possibly exploit the counting information in the cut generation and selection. In Section 3 we present preliminary computational results on ILPs from the literature. Some short conclusions are drawn in Section 4 outlining open questions and potential research directions.

2 Counting for Cutting

Given an inequality in the form (4), we are interested in counting the number of lattice points to which the inequality is tight, i.e., that satisfy the inequality at equality. We restrict the description to the case in which the variables of the ILP (1) are bounded. In such a case, the problem reduces to count the number of feasible solutions of the system

$$\alpha^T x = \beta, \ell \le x \le u, x \text{ integer}, \tag{5}$$

where α, β, ℓ, u are integer vectors.

In his paper on propagating knapsack constraints using dynamic programming, Trick [13] mentions that counting the number of solutions of knapsack constraints can be done in pseudo-polynomial time through a simple recursion. Our approach is a direct adaptation of the latter except that since coefficients (and possibly variables) can be negative, we first compute the largest and smallest possible partial sums of the left hand side of (5) which can be completed to reach β (note that this depends on the ordering of the terms). We then create data structures of the appropriate size and restrict the state space expansion.

The algorithm previously outlined returns the number of feasible solutions of (5) for each of the cuts which solves the separation problem at a given iteration. We use the information in two ways.

First, if the number of lattice points tight to a specific inequality is 0, the cut can be strengthened by reducing its right hand side. More precisely, the optimal right hand side is the solution of the ILP

$$\max\{\alpha^T x : \alpha^T x \le b, \ell \le x \le u \text{ integer}\}, \tag{6}$$

which in the special case $x \in \{0,1\}^n$ reduces to the classical *Subset Sum* Problem. However, we do not solve (6) directly to strengthen the cuts since the same recursion used for counting can be used repeatedly with different tentative values for the right hand side. We note that a more general version of this strengthening argument is considered in [7].

Second, the number of tight lattice points is then used as a quality measure for cut selection. We investigated the selection of the cut with the *largest* number of tight points and the reverse option, that is choosing the cut which is tight to the *smallest* number of lattice points. In case it is a cut which is not tight to any (option above), the cut is strengthened before being added.

We end the section by noting that counting does not come computationally for free. Without going into the details because of the lack of space (see [13,12]), the counting algorithm is clearly more expensive than most of the currently used cut selection procedures. However, here we are mainly interested in learning something about the characteristics of good cuts to add, while the tradeoff with respect to the computational effort will be considered in a future work.

3 Empirical Results

We have conducted a preliminary set of computational experiments with a pure cutting plane algorithm on 7 0-1 ILP instances from the MIPLIB 3.0 as shown in Table 1. In particular, the algorithm iteratively solves the separation problem by the classical procedure of Gomory [9] by generating a so-called *round* of CG cutting planes from the tableau, selects "some" of the separated cuts and adds them to the LP relaxation. An LP is obviously solved at each iteration.

In the first experiment we tested the impact of the strengthening procedure on the quality of the cuts. It is well known (see, e.g., [4]) that the best method for using CG cuts (and, more generally, Gomory Mixed-Integer cuts) is to add them in rounds, i.e., one cut for each of the integer-constrained variable which is basic and fractional in the current tableau. On the 7 instances in Table 1 we could strengthen cuts only for problems `p0033` and `p0548` in 1 round. Precisely, for `p0033` we strengthened 4 over 7 cuts and because of that the %gap dropped from 17.61% to 8.18. For `p0548` instead we strengthened 4 over 50 cuts, which only slightly improved the %gap from 45.61% to 45.33.

In the second and third experiments we looked specifically at cut selection. Namely, the results in Table 2(I) refer to the comparison between three versions of the cutting plane procedure in which, at each round, the most violated cut (v.most) or the cut with the *largest* number of tight lattice points (v.max) or the cut with the *smallest* number of tight lattice points (v.min) is selected. Only one cut per iteration for 20 iterations. It is clear from the results that selecting the cut tight to the largest number of lattice points does not pay off while the version tight to the smallest number of points is competitive with the classical most violated cut selection. In fact, it is never significantly worse and twice much better. This could be explained by the fact of being "blocked" by few(er) lattice

Table 1. Seven 0-1 ILP instances from the MIPLIB 3.0. (The %gap is computed as $(opt - lb.init)/opt$, where *lb.init* is the initial value of the LP relaxation and *opt* is the value of the optimal integer solution.)

name	preprocessed	initial lower bound value (*lb.init*)	optimal solution value (*opt*)	%gap
harp2	Yes	-74,232,132.35	-73,899,798.00	0.45
mod008	No	290.93	307.00	5.23
p0033	No	2,520.57	3,089.00	18.40
p0201	No	6,875.00	7,615.00	9.72
p0282	No	176,867.50	258,411.00	31.56
p0548	Yes	4,533.50	8,691.00	47.84
lseu	Yes	947.96	1,120.00	15.36

Table 2. Percentage gap closed: (I) three versions of cut selection, one cut per iteration, (II) two versions of cut selection, two cuts per iteration. (The %gap closed is computed as $(lb.v - lb.init)/(opt - lb.init) * 100$, where *lb.init* and *opt* are from Table 1 and *lb.v* is the final lower bound value of a generic version v.)

	%gap closed		
name	v.most	v.max	v.min
harp2	4.90	0.08	0.80
mod008	6.41	1.53	3.28
p0033	0.50	0.17	54.00
p0201	15.57	6.46	15.97
p0282	0.21	0.01	0.02
p0548	3.23	0.01	3.57
lseu	5.83	3.51	41.56
average	5,24	1,68	17,03

(I)

	%gap closed	
name	v2.most	v2.tight
harp2	6.11	9.50
mod008	5.97	3.78
p0033	46.65	54.53
p0201	15.76	18.70
p0282	0.23	0.02
p0548	3.51	3.58
lseu	8.37	40.98
average	12,37	18,73

(II)

points can be interpreted positively: in the limit a *unique* tight lattice point can be the optimal vertex of P_I.

Finally, we compared a version selecting the two most violated cuts (v2.most) with one selecting the two cuts with largest and smallest number of tight points (v2.tight). The results reported in Table 2(II) show that on 5 over 7 of the instances using the counting information helps to close more gap. Clearly, this is a very limited set of experiments and more extensive computation is needed.

4 Conclusions

We have outlined and preliminary tested some ideas for using information associated with counting lattice points to select (and strengthen) cutting planes. We have developed and applied the method to Chvátal-Gomory cuts.

We have not discussed the computational cost of the cut selection rule based on counting lattice points. That is clearly higher than that of most of the currently used cut selection procedures. However, the focus of the paper was mostly on learning something about the characteristics of good cuts to add, while the tradeoff with respect to the computational effort will be considered in a future work. One natural option to reduce such an effort is approximate counting.

The idea of using a counting information in cutting planes can be somehow naturally extended to other families of cuts. In particular, it is well known that CG cuts are a special case of *split cuts*, where one part of the disjunction has an empty intersection with P. Of course, one could continue testing tightness to the cut itself (although the coefficients of the cut might be fractional, thus making the counting algorithm of Section 2 not applicable) but it might be possible to consider the tightness to the disjunction itself.

Finally, we observe that Zanarini and Pesant [16] and Pryor and Chinneck [14] have recently used counting information for branching (as opposed to cutting) but in a rather different way: in [16] the branching is shown to be most efficient when based on the highest solution density while in [14] the smallest solution density is clearly dominating. This seems to suggest that the way of effectively exploiting the counting information in both branching and cutting is not clear yet and thus need to be further investigated.

Acknowledgements

Part of this work has been done during the visits the authors exchanged within the Italy-Quebec Executive Programme 12 “Algorithms and systems for the operational planning in industry and services”. The support to the second author by the Institute of Advanced Studies of the University of Bologna is also acknowledged. The authors are grateful to Matteo Fischetti and Juliane Dunkel for interesting discussions on the topic.

References

1. Achterberg, T.: SCIP: Solving Constraint Integer Programs. Mathematical Programming Computation 1, 1–32 (2008)
2. Achterberg, T.: LP Basis Selection and Cutting Planes. Research Talk @ MIP (2010), `http://www2.isye.gatech.edu/mip2010/program/program.pdf`
3. Andreello, G., Caprara, A., Fischetti, M.: Embedding $\{0, \frac{1}{2}\}$-Cuts in a Branch-and-Cut Framework: A Computational Study. INFORMS Journal on Computing 19, 229–238 (2007)
4. Balas, E., Ceria, S., Cornuéjols, G., Natraj, N.: Gomory Cuts Revisited. Operations Research Letters 19, 1–9 (1996)
5. Caprara, A., Lodi, A., Scheinberg, K. (eds.): Counting and Estimating Lattice Points. Optima, vol. (81), `http://www.mathprog.org/Optima-Issues/optima81.pdf`
6. Chvátal, V.: Edmonds polytopes and a hierarchy of combinatorial problems. Discrete Mathematics 4, 305–337 (1973)
7. Dunkel, J.: On Gomory-Chvátal Cutting Planes, the Elementary Closure, and a Strengthened Closure for Polytopes in the Unit Cube. PhD thesis. MIT, Cambridge, MA (2011)
8. Eisenbrand, F.: On the membership problem for the elementary closure of a polyhedron. Combinatorica 19, 297–300 (1999)
9. Gomory, R.E.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the AMS 64, 275–278 (1958)
10. Linderoth, J.T., Lodi, A.: MILP Software. In: Cochran, J.J. (ed.) Wiley Encyclopedia of Operations Research and Management Science, vol. 5, pp. 3239–3248. Wiley, Chichester (2011)
11. Lodi, A.: MIP computation. In: Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A. (eds.) 50 Years of Integer Programming 1958-2008, pp. 619–645. Springer, Heidelberg (2009)
12. Pesant, G., Quimper, C.-G.: Counting solutions of knapsack constraints. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 203–217. Springer, Heidelberg (2008)
13. Trick, M.A.: A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. Annals of Operations Research 118, 73–84 (2003)
14. Pryor, J., Chinneck, J.W.: Faster Integer-Feasibility in Mixed-Integer Linear Programs by Branching to Force Change. Computers & OR 38, 1143–1152 (2011)
15. Wesselmann, F., Suhl, U.H.: Implementation techniques for cutting plane management and selection. Technical Report Universität Paderborn (2007)
16. Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics. Constraints 14, 392–413 (2009)

Precedence Constraint Posting for Cyclic Scheduling Problems

Michele Lombardi, Alessio Bonfietti, Michela Milano, and Luca Benini

DEIS, University of Bologna,
Viale del Risorgimento 2, 40136 Bologna, Italy
{michele.lombardi2,alessio.bonfietti,michela.milano,luca.benini}@unibo.it

Abstract. Resource constrained cyclic scheduling problems consist in planning the execution over limited resources of a set of activities, *to be indefinitely repeated*. In such a context, the iteration period (i.e. the difference between the completion time of consecutive iterations) naturally replaces the makespan as a quality measure; exploiting inter-iteration overlapping is the primary method to obtain high quality schedules. Classical approaches for cyclic scheduling rely on the fact that, by fixing the iteration period, the problem admits an integer linear model. The optimal solution is then usually obtained iteratively, via linear or binary search on the possible iteration period values. In this paper we follow an alternative approach and provide a port of the key Precedence Constraint Posting ideas in a cyclic scheduling context; the value of the iteration period is not a-priori fixed, but results from conflict resolution decisions. A heuristic search method based on Iterative Flattening is used as a practical demonstrator; this was tested over instances from an industrial problem obtaining encouraging results.

Keywords: Precedence Constraint Posting, Resource Constrained Scheduling, Cyclic Scheduling.

1 Introduction

Cyclic scheduling problems arise in a wide variety of real-life contexts, when a set of activities have to be repeated an very large, possibly unknown number of times. In this case a schedule cannot be represented by enumerating start times of all activities, as this would require a huge amount of storage, and a convenient abstraction is to assume that activities will execute an infinite number of times *following a periodic schedule*. In this context the notion of optimality is related to the period of the schedule. A minimal period corresponds to the highest number of activities carried out in average over a large time window.

One of the most successful approaches for solving non cyclic resource constrained scheduling problems is Precedence Constraint Posting (PCP, [23,5,19]) that basically adds precedence constraints to the project graph to avoid resource over-usage. In this paper, we explore how to apply the PCP solution method to the cyclic scheduling problems. Porting the key PCP concepts to the cyclic scheduling domain is however a non-trivial task and requires a number of generalizations. We provide basic definitions concerning resource profiles, Minimal Critical Sets (MCS) and Resolvers that apply to the cyclic case.

As a second contribution of the paper, we implement a PCP solver by porting to the cyclic domain the Iterative Flattening method [4,3]; this is a heuristic algorithm that repeatedly attempts to build improving solutions by iteratively selecting a MCS and resolving the conflict through the addition of a precedence constraint. We have experimented the proposed approach on a set of industrial benchmarks taken from the field of loop scheduling problems extracted by a compiler. The results are very promising (despite the simplicity of this first implementation) and open perspectives for future improvements.

2 Problem Description

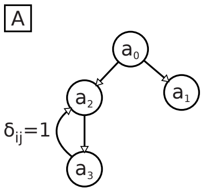
Cyclic Scheduling can be considered as an extension of classical Resource Constrained Project Scheduling (RCPSP), as it involves deciding the execution order of a set of activities subject to temporal and resource constraints; unlike in the RCPSP, however, the set of activities is *indefinitely repeated over time.* The regularity of the process flow allows one to partially overlap consecutive executions of the activity set, allowing better resource utilization.

Formally, the problem can be defined over a directed graph (referred to as Project Graph) $\mathcal{G} = \langle A, E \rangle$, where A is the set of activities a_i and E is a set of arcs (a_i, a_j), representing temporal dependencies. We refer as *iteration* to a full execution of the graph; the *makespan* is the time span between the start of the first activity and the end of the last one in a single iteration; the *period* is the difference between the start time of the same task in consecutive iterations; the *completion frequency* is the inverse of the period. Figure 1A shows an example of a Project Graph.

We assume activities are non-interruptible and have fixed duration d_i. Similarly, a minimum time lag d_{ij} is associated to each arc and constrains the minimum time distance between the end of a_i and the beginning of a_j *in specific iterations* (say iteration k and h); in particular a parameter δ_{ij} known as *height* or *delay* specifies the iteration offset (i.e. the difference $h - k$). Formally, let s_i^k be the start time of a_i in iteration k, then an arc (a_i, a_j) enforces:

$$s_j^{k+\delta_{ij}} \geq s_i^k + d_i + d_{ij} \tag{1}$$

A



B

	d_i
a_0	1
a_1	7
a_2	1.5
a_3	1.5

	δ_{ij}	d_{ij}
(a_0, a_1)	0	0
(a_0, a_2)	0	0
(a_2, a_3)	0	0
(a_3, a_2)	1	0

C

	r_{i0}
a_0	0
a_1	1
a_2	1
a_3	0

$c_0=2$

Fig. 1. A simple Project Graph and related data

it is convenient to think about the precedence constraint and the delay as iteration k of a_i "providing input" to iteration $k + \delta_{ij}$ of activity a_j. Unlike many approaches, we assume $\delta_{ij} \in \mathbb{Z}$, rather than $\in \mathbb{N}$. Figure 1B reports durations, delay and time lag values for the Project Graph on the left; note all d_{ij} are 0 for sake of simplicity.

The presence of delay allows cycles to be consistently defined in the project graph, since some of the precedence constraints along the circular path will target activities from different iterations; in general, any cycle with a *strictly positive* sum of arc delay admits a feasible assignment of start times. As an example, the graph from Figure 1 contains a single cycle (involving nodes a_2 and a_3), with total delay equal to 1 (as δ_{32}); if no delay was specified, there would be no chance to satisfy the cyclic dependence. Note that cyclic scheduling enables cycle treatment, but does not require the graph to actually contain cycles.

Each activity a_i requires some non-negative amount r_{ih} of resource r_h from a set R (see Figure 1C). Each resource has limited capacity c_h, so that the total consumption due to activities *from any iteration* cannot exceed c_h at any point of time. Formally:

$$\sum_{i,k \,:\, s_i^k \leq t < s_i^k + d_i} r_{ih} \leq c_h \qquad \forall \text{ time instant } t, \ \forall r_h \in R \tag{2}$$

The problem objective is to minimize the average period; given a reference activity a_i, this can be formally defined as:

$$\lambda_i = \lim_{k \to \infty} \frac{\sum_{h=1}^{k} s_i^h - s_i^{h-1}}{k} = \lim_{k \to \infty} \frac{s_i^k}{k} \tag{3}$$

If no activity is unnecessarily delayed, the choice of a_i does not influence the result [13], so that the period has no dependence on i; hence, we always write λ in the followings. Each cycle L in the graph sets a lower bound on the average period; this is given by the so-called *cycle ratio*, i.e. the total cycle length over the total delay. Therefore, the Maximum Cycle Ratio MCR (or *iteration bound* [22]) of the graph provides a inherent bound on the minimum achievable period; let this be λ^*:

$$\lambda^* = \max_{L \in \mathcal{G}} \frac{\sum_{(a_i,a_j) \in L} d_i + d_{ij}}{\sum_{(a_i,a_j) \in L} \delta_{ij}} \tag{4}$$

the MCR value can be computed in polynomial time by means of specialized algorithms [7,12]; for a graph with n nodes and m arcs the typical runtime is $O(nm \log n)$ or worse. The λ^* value for the graph in Figure 1 is given by the only cycle contained in the graph: $\lambda^* = (d_2 + d_{2,3} + d_3 + d_{3,2})/(\delta_{2,3} + \delta_{3,2}) = 3$.

2.1 Cyclic and Periodic Scheduling

A solution to a cyclic scheduling problem (i.e. a *schedule*) is an assignment of start times to each activity a_i in each iteration k; as a consequence, a schedule has in principle infinite size. A cyclic schedule is periodic if the start times follow a static pattern, repeated with a fixed period; in such case s_i^k can be rewritten as follows:

$$s_i^k = s_i^0 + k \cdot \lambda \tag{5}$$

where λ has the meaning from Equation (3). In this section we disregard resource constraints (they will be considered later on); under such assumption, periodic schedules have two fundamental properties: 1) there exists a feasible schedule if and only if there exists a periodic schedule [21,14]; 2) the periodic schedule with the minimum period has $\lambda = \lambda^*$ and is therefore optimal [6]. From now on, we focus on periodic schedules and on feasible assignments for the start times at iteration 0; for sake of simplicity, *the notation s_i will be used in place of s_i^0.*

The problem of finding an optimal periodic schedule satisfying all *temporal* constraints can be formalized as follows:

$$\begin{array}{llll} \text{(P0)} & \min & \lambda & \\ & \text{subject to:} & s_j + \delta_{ij}\lambda \geq s_i + d_i + d_{ij} & \forall (a_i, a_j) \in E \quad (6) \\ & & s_i \geq 0 & \forall a_i \in A \\ & & \lambda \geq 0 & \end{array}$$

where Constraints (6) are derived by combining Equation (1) and (5). Note that P0 is a linear program and could be solved in principle via any general LP technique. It is however more common (and efficient) to use dedicated MCR computation algorithms (see [16,12,7]). Solving problem P0 provides no consistency guarantee on resource constraints; as a matter of fact, handling resource constraints is usually the toughest issue in any cyclic scheduling method.

3 Related Work

The cyclic scheduling literature mainly arises from industrial and computing contexts. The former includes mass production, chemical and hoist scheduling problems, the latter includes parallel processing, software pipelining and dataflow mapping problems in embedded systems. While there is a considerable body of work on cyclic scheduling in the OR literature, the problem has not received much focus from the AI community ([10] is one of the few approaches). A subclass of cyclic scheduling (targeted by most of the existing approaches) is the so-called modulo scheduling [20], where the start times and the period λ are required to assume integer values (this is not the case for our method).

Several heuristic and complete approaches have been proposed for cyclic scheduling. A heuristic algorithm called *iterative modulo scheduling* is proposed in [25] and generates near-optimal schedules. An interesting heuristic approach (*SCAN*) based on a time-indexed ILP model is presented in [2]. Both methods compute a schedule for a single iteration, which is characterized in terms of its makespan and period. The makespan dictates the size of the model, so that schedules with a relatively small period could be difficult to compute due to a possibly high makespan. Our approach is not time indexed and does not suffer from this issue.

Advanced complete formulations are proposed in [11] by Eichenberger and in [8] by Dupont de Dinechin; both the approaches are based on a time-indexed ILP model; the former exploits a decomposition of start times to overcome the issue with large makespan values, while the latter has no such advantage, but provides a better LP relaxation. In [1] the authors report an excellent overview of the state-of-the-art formulations and present a new model issued from Danzig-Wolfe Decomposition. Other good overviews of complete methods can be found in [14,9].

To the best of our knowledge, most of the state-of-the-art approaches are based on iteratively solving resource subproblems obtained by fixing the period value; fixing λ allows solving the resource constrained cyclic scheduling problem via an integer linear program (while modeling λ as an explicit decision variable yields non-linear models). The obvious drawback is that a resource constrained scheduling problem needs to be repeatedly solved for different λ values. In this paper, we provide an alternative method which does not require the iterative solution of NP-hard subproblems.

4 Cyclic Precedence Constraint Posting

Precedence Constraint Posting (PCP, see [23,5]) is a scheme for solving scheduling problems with limited resources. Roughly speaking, the method relies on a *constraint model* to ensure temporal consistency, while resource constraints are tackled by iterative 1) identification of possible conflicts and 2) their resolution through the addition of temporal constraints, so that resource constraints are progressively turned into temporal constraints. A solution is provided in the form of a conflict free, temporal consistent *augmented graph*. The main idea roots back to [18,17] from the stochastic scheduling domain and was successfully applied in an AI context in [4,3,24,19].

The method is very appealing in a cyclic context, where an optimal solution can be computed in polynomial time (although less efficiently that in non-cyclic scheduling) if no resource constraint is taken into account. This section describes how the key PCP concepts can be generalized to a cyclic scheduling context.

4.1 Concurrency and Requirement Functions

A resource conflict arises when the cumulative usage of overlapping tasks exceeds the capacity; in cyclic scheduling, computing the resource usage requires to take into account overlapping iterations. Figure 2A shows a periodic schedule for the graph from Figure 1; as one can observe, not only activities from different iterations may overlap, but different iterations of the same activity may be running at the same time instant. In the figure, the schedule *period* is 3, while the *makespan* is marked by the end of a_1 and has value 8.

We devise a formal characterization of the resource usage of an activity, by restricting focus to an arbitrary λ length time span $[k \cdot \lambda, (k+1) \cdot \lambda[$, with k sufficiently large for the schedule to have entered a fully periodic behavior: this always happens when $(k+1) \cdot \lambda$ is greater than the schedule makespan

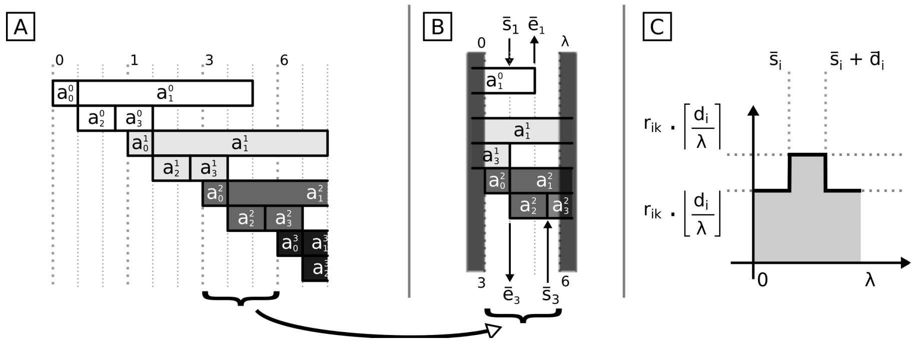


Fig. 2. A) a periodic schedule; B) a periodic schedule on the modular interval; C) modular requirement function

(see Figure 2B). Due to periodicity, the interval is representative of the steady schedule behavior; in the followings, it is referred to as *modular interval* $[0, \lambda[$. A local start time within the modular interval can be associated to each activity via decomposition; namely, we can write:

$$s_i = \bar{s}_i + \beta_i \cdot \lambda \tag{7}$$

with $\bar{s}_i$ in $[0, \lambda[$ and $\beta_i \in \mathbb{N}$; equivalently, we have $\bar{s}_i = \{s_i/\lambda\} \cdot \lambda$, where the notation $\{\cdot\}$ denotes the fractional part; in the example from Figure 2B, we have $\bar{s}_0 = 0, \bar{s}_1 = \bar{s}_2 = 1, \bar{s}_3 = 2.5$. In the followings, we refer to $\bar{s}_i$ as *modular start time* of a_i, because of the analogy between the start time decomposition and the modulo operation over $\mathbb{Z}$. Since s_i represents the start time of iteration 0 of a_i, the β_i value in Equation (7) refers to the number of full periods elapsed before a_i is scheduled for the first time; in the schedule from Figure 2 all β_i are zero, since all s_i are in the interval $[0, \lambda[$. Similarly, we can define a modular end value:

$$\bar{e}_i = \left\{ \frac{s_i + d_i}{\lambda} \right\} \cdot \lambda \quad \text{or, equivalently: } s_i + d_i = \bar{e}_i + \eta_i \cdot \lambda \tag{8}$$

where η_i has a similar meaning to β_i; in Figure 2 we have $\eta_1 = 2, \eta_3 = 1$, while all other η_i values are 0. Observe that $\bar{e}_i$ can be higher *or lower* than $\bar{s}_i$, as one can see in Figure 2.

The number of concurrent executions of each activity within a period can be characterized by relying on modular start/end values; in particular, in the interval $[0, \lambda[$, at least $\lfloor d_i/\lambda \rfloor$ iterations of activity a_i are *always* executing. An extra iteration should be added if the time instant under analysis falls between the modular start and the modular end. Formally, we introduce a concurrency function $\#a_i(t)$ with values in the time interval $[0, \lambda[$ and such that:

$$\#a_i(t) = \left\lfloor \frac{d_i}{\lambda} \right\rfloor + pulse(a_i, t) \tag{9}$$

and:

$$pulse(a_i, t) = \begin{cases} 1 & \text{if } \bar{s}_i \leq t < \bar{e}_i \text{ or } t < \bar{e}_i < \bar{s}_i \text{ or } \bar{e}_i < \bar{s}_i \leq t \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Function $\#a_i(t)$ ranges in between $\lfloor d_i/\lambda \rfloor$ and $\lceil d_i/\lambda \rceil$ and is therefore constant if d_i is an integer multiple of λ. Equation 9 shows how the concurrency function results from the superimposition of a constant and a *pulse* function; the latter is triggered by a new arrival of a_i and makes a step down when a *previous* iteration of a_i ceases execution. The concurrency function allows a *modular requirement function* $\bar{r}_{ik}(t)$ to be easily defined as $\bar{r}_{ik}(t) = r_{ik} \cdot \#a_i(t)$. Figure 2C provides a pictorial representation of the requirement function. Note that for λ sufficiently large, $\lfloor d_i/\lambda \rfloor$ is 0, $\bar{s}_i = s_i$, $\bar{e}_i = s_i + d_i$ and $\bar{r}_{ik}(t)$ boils down to the rectangular function used in classical scheduling.

4.2 Minimal Critical Sets

In non-cyclic scheduling a Critical Set (CS) is a subset S of activities, collectively overusing a resource r_h in case of simultaneous execution. A Minimal Critical Set (MCS) is CS such that none of its proper subsets is a CS. A precedence constraint posted over a pair of activities a_i, a_j in the same MCS prevents the conflict: hence, by focusing on MCS, one can avoid adding useless precedence constraints. In a cyclic context, several iterations of the same activity may be running concurrently; therefore, we propose to define an MCS as a *multiset*, i.e. a set where objects can appear several times.

Definition 1 (Generalized Critical Set and Minimal Critical Set). *A Generalized Critical Set (GCS) is a multiset M of activities in A, such that, given the current temporal constraints:*

1. *$card(a_i, M)$ iterations of each activity a_i may overlap*
2. *the set of overlapping executions causes an over-usage:* $\exists r_h \mid \sum_{a_i \in M} r_{ih} \cdot card(a_i, M) > c_h$

A Generalized Minimal Critical Set is a multiset M such that none of its proper sub-multisets is a GCS.

where $card(a_i, M)$ gives the cardinality of a_i in M; we write $a_i \in M$ iff $card(a_i, M) > 0$; we write $M' \subseteq M$ iff, $\forall a_i \in M'$, it holds $card(a_i, M') \leq card(a_i, M)$. In the followings, we always use the term Critical Set and Minimal Critical Set in the generalized sense. With reference to Figure 2, $\{a_1, a_1, a_1, a_2\}$ is a CS, while $\{a_1, a_1, a_2\}$ or $\{a_1, a_1, a_1\}$ are MCS; note that a MCS may even consist of several iterations of a single activity.

4.3 MCS Resolvers

In a PCP approach, critical sets are cleared by adding carefully designed constraints, referred to as *resolvers*. Specifically, we adopt the following definition, holding for both classical and generalized MCS:

Definition 2 (Resolver). *Let M be an MCS; a* resolver *for M is any temporal constraint ρ which prevents the multiset from satisfying Definition 1.*

A simple pairwise precedence constraints (i.e. added arc) is an example of valid resolver in non-cyclic scheduling; other types of constraints could be employed as well [15]. A good resolver should allow to efficiently test consistency after it is added to the temporal model: this is the reason why simple precedence constraints are by far the most common resolvers in the literature. In second place, it should be possible to encode an optimal solution as a set of resolvers, so as not to loose the chance of achieving optimality.

By direct application of Definition 2 we get a necessary and sufficient condition, defining a resolver in the most general form:

$$\forall t \in [0, \lambda[: \bigvee_{a_i \in M} [\#a_i(t) < card(a_i, M)] \tag{11}$$

which basically states that for each time instant t in the modular interval, at least one of the concurrency functions for activities in the MCS must be strictly less then the cardinality in M. Equivalently:

$$\forall t \in [0, \lambda[: \sum_{a_i \in M} \#a_i(t) < \sum_{a_i \in M} card(a_i, M) \tag{12}$$

Unfortunately, Constraint 12 does not allow an efficient consistency check, hence a suitable decomposition should be devised. Observe that:

Statement 1. *Let L be a cycle; let us refer as A_L to the set of activities it includes and as Δ_L to its total delay, then L does not allow more than Δ_L iterations of activities in A_L to run concurrently; the same restriction cannot be achieved by employing fewer arcs or a larger delay.*

A formal proof is omitted due to lack of space, but the statement follows quite intuitively from the semantic of arc delays described in Section 2. Now:

Theorem 1. *Given a feasible schedule according to Constraint* (12)*, it is possible to identify a collection $\mathcal{L}$ of unary and binary loops L such that any feasible schedule modification according to $\mathcal{L}$ is feasible according to Constraint* (12)*.*

Proof. It is sufficient to prove that any infeasible schedule modification according to Constraint (12) is infeasible according to precedence constraints in $\mathcal{L}$. For each activity a_i, let us introduce a self-loop with delay $\delta_{ii} = \lceil d_i/\lambda \rceil$; then, for each pair of activities a_i, a_j such that a_i precedes a_j in the modular interval (i.e. $\bar{e}_i \leq \bar{s}_j$), we introduce a binary cycle with $\delta_{ij} = \beta_j - \eta_i$ and $\delta_{ji} = \beta_i - \eta_j + 1$; let the collection of those cycles be $\mathcal{L}$. By construction, all precedence constraints enforced by $\mathcal{L}$ are satisfied. In order to violate Constraint (12), any modification of the target schedule should either increase the amount of overlapping executions for a single activity a_i or break a modular precedence relation; due to Statement 1, this would respectively result in the violation of a unary or binary loop constraint in $\mathcal{L}$. □

Basically, Theorem 1 shows that a properly designed binary or unary loop "covers" part of the feasible space of Constraint (12). By iterating the process for the infinitely many possible valid schedules according to (12), we obtain:

Theorem 2. *Constraint* (12) *is equivalent to a infinite-size disjunction of unary and binary cycles defined over activities in M.*

Hence, *we can focus on resolvers formulated as unary or binary cycles* without loosing the chance of achieving optimality; moreover, adding a cycle to the Project Graph still allows to efficiently compute the best achievable period.

Unary resolver: A unary resolver L over a_i with $\Delta_L < card(a_i, M)$ delay can be built by adding a self-arc (a_i, a_i) with $\delta_{ii} = card(a_i, M) - 1$; any smaller δ_{ii} value would unnecessarily over-constrain the partial solution.

Binary resolver: Building a binary resolver over two activities a_i and a_j requires adding arcs (a_i, a_j), (a_j, a_i) and deciding their delay value so that:

$$\delta_{ij} + \delta_{ji} = card(a_i, M) + card(a_j, M) - 1 \tag{13}$$

where we recall $\delta_{ij}, \delta_{ji} \in \mathbb{Z}$. This means that *binary cycle resolvers are parametric and an infinite number of possible resolvers can be defined between a pair of activities*. In this work we use a simple heuristic to decide the delay distribution; a deeper investigation of the parameter space is left for future work.

5 Implementing the Cyclic PCP

5.1 Cyclic Iterative Flattening

As a practical demonstrator for the framework proposed, we realized a heuristic PCP solver for period minimization in resource constrained cyclic scheduling. In particular, our solver is a porting of the Iterative Flattening method to cyclic scheduling.

Iterative Flattening (iFlat) is a heuristic solution method for makespan minimization on the RCPSP. The approach is introduced in [4,3] and makes use of a Temporal Constraint Network to enforce consistency of precedence and time window constraints. In the basic version, iFlat performs a fixed number of iterations; during each of them, a solution is tentatively built by repeatedly 1) selecting a MCS and 2) resolving the conflict through the addition of a precedence constraint; no backtracking is performed. The MCS and resolver selection is randomized so that each iteration explores a different portion of the search space. MCS identification is done [24,23] either by computing so-called resource usage envelopes, or by scanning the contention peaks arising in the earliest start time schedule (computed by disregarding resource constraints).

The main scheme of our approach matches that of basic iFlat and is reported in Algorithm 1; as one can see, we adopt the peak based MCS identification procedure. Unlike the original iFlat, the presented approach lacks support for time windows, deadlines and maximum time lag constraints, which is planned for future research. In the following, each step will be further detailed.

Algorithm 1. (Cyclic) Iterative Flattening

Input: a problem instance defined over a graph $\mathcal{G} = \langle A, E \rangle$; a number of iterations n
Output: a conflict free augmented graph

```
1: for i = 0 to n-1 do
2:   while at least an MCS exists do
3:     compute contention peaks and sample an MCS
4:     sample a resolver for the MCS
5:     add the resolver and check temporal consistency
6:   if the current $\lambda^*$ improves the best one then
7:     store the current solution and set current $\lambda^*$ as threshold
8: return the best solution
```

5.2 Temporal Consistency Check

Since we do not consider time windows or deadline constraint on specific activities, checking temporal consistency amounts to compute an optimal (resource unaware) periodic schedule and test whether the resulting period λ^* is less than the current threshold. We recall that an optimal schedule is obtained by solving problem P0 from Section 2.1 via MCR computation.

Our MCR algorithm is a variant of the CYCLE class, as described in [12]. The key idea is that, for any fixed λ^0, P0 becomes a longest path computation problem on a cyclic graph; in detail, each arc (a_i, a_j) has a modified length equal to $d_i + d_{ij} - \delta_{ij} \cdot \lambda^0$. At the end of the computation, the longest path to a_i defines the activity earliest start value (i.e. s_i). By starting from a guess value $\lambda^0 \leq \lambda^*$ (i.e. super-optimal) the longest path computation either proves feasibility (hence we deduce $\lambda^0 = \lambda^*$) or identifies an infeasible cycle; in this case, the cycle provides a new lower bound $\lambda^1 > \lambda^0$, which can be used as a new guess to reiterate the process.

Some heuristics can be employed to find a good initial period guess. Here, however, we simply start with $\lambda^0 = 0$; whenever a resolver is added and a new schedule needs to be computed (step 5 from Algorithm 1), we use the λ^* of the current graph to prime the process, making the computation incremental. This is a peculiar advantage of the CYCLE algorithm class, which could not be exploited with the (otherwise faster) algorithm by Young-Tarjan-Orlin [26].

The longest path computation can be performed via Bellmann-Ford like algorithms[1]; specifically, the method we use is reported in Algorithm 2 and is designed to avoid repeated traversals of graph cycles. The procedure keeps a set of nodes to be visited (referred to as Q); for each activity a_i, the algorithm also stores the set $V(a_i)$ of tasks on the critical path to a_i. The output is an assignment of start times s_i and a period bound λ^1 (in case of feasibility) or the bound alone (in case of infeasibility). Algorithm 2 is initialized by setting all start times to 0; the corresponding critical paths contain no node and have 0 delay (line 1); all tasks are enqueued to be processed ($Q \leftarrow T$).

At each step an arbitrary task is picked from the Q set (line 5); the choice does not compromise correctness; here, we simply pick the lowest index activity

[1] The simpler Dijkstra algorithm is not an option due to the presence of cycles.

Algorithm 2. Find Start Times (longest paths)

Input: the current augmented graph and λ guess λ^0
Output: a lower bound on λ^* and an assignment of start times
Data structures:
- $\forall a_i \in A$: $V(a_i)$ is the set of nodes on the critical path
- $\forall a_i \in A$: s_i is the start time of iteration 0
- $\forall a_i \in A$: δ_i is the delay of the current critical path to a_i
- Q is the set of activities to be visited
- λ^1 is the period lower bound being computed

```
1: $\forall a_i \in A$: $V(a_i) \leftarrow \emptyset$, $s_i \leftarrow 0$, $\delta_i \leftarrow 0$
2: $Q \leftarrow A$, $\lambda^1 \leftarrow 0$
3: while $Q \neq \emptyset$ do
4:    pick an arbitrary activity $a_i$ from $Q$
5:    $Q \leftarrow Q \setminus \{a_i\}$, $V(a_i) \leftarrow V(a_i) \cup \{a_i\}$
6:    for all arcs $(a_i, a_j) \in E$ having $a_i$ as source do
7:       if $a_j$ is not in $V(a_i)$ then
8:          if $s_j < s_i + d_i + d_{ij} - \delta_{ij} \cdot \lambda^0$ then //$a_i$ is on the critical path to $a_j$
9:             $s_j \leftarrow s_i + d_i + d_{ij} - \delta_{ij} \cdot \lambda^0$ //update the critical path to $a_j$
10:            $\delta_j \leftarrow \delta_i + \delta_{ij}$ //update total delay to $a_j$
11:            $V(a_j) \leftarrow V(a_i)$ //transfer set of visited nodes
12:            $Q \leftarrow Q \cup \{a_j\}$ //enqueue $a_j$
13:      else //a cycle $L$ has been identified
14:         let $\Delta_L := \delta_i - \delta_j + \delta_{ij}$ //total delay for cycle $L$
15:         let $D_L := s_i + d_i + d_{ij} - s_j + (\delta_i - \delta_j) \cdot \lambda^0$ //total length of cycle $L$
16:         $\lambda^1 \leftarrow \max(\lambda^1, D_L/\Delta_L)$ //update bound with cycle ration of $L$
17: if $\lambda^1 > \lambda^0$ then
18:    return the bound $\lambda^1$ and no start time
19: else
20:    return all the $s_i$ and the bound $\lambda^1$
```

in Q. Then, a_i is marked as part of longest path to itself: this is needed to deal with self-loops in the Project Graph.

Next, all the successors a_j of task a_i are processed; if the successor is not part of the longest path to a_i (line 7) and if arc (a_i, a_j) sets a stronger bound on the start time s_j (line 8), then: 1) s_j is updated, 2) the total delay δ_j is updated and 3) the longest path to a_j becomes $V(a_i)$; finally, a_j is marked as an activity to be processed (at line 12). If a_j is found to be part of the longest path to a_i (line 13), a loop L has been identified; in such a case: 1) either the constraint corresponding to arc (a_i, a_j) is feasible (and the current λ^0 guess is confirmed); or 2) the constraint is infeasible, and we can determine the minimum λ which would provide feasibility.

This is done by computing the Cycle Ratio of the identified loop; in particular, the total duration D_L of activities and arcs in the loop can be computed as shown at line 15; the total delay on the loop is referred to as Δ_L, hence the cycle ratio is D_L/Δ_L (line 16) and the bound λ^1 is updated. If at the end of the process (when Q becomes empty) λ^1 is higher than the current throughput guess λ^0 an infeasibility has been detected and the algorithm repeats (line 18). Otherwise, all nodes have been assigned a feasible start time and a valid lower bound on the period has been computed. Minor modifications are done in the actual implementation to improve the runtime; as for any other CYCLE algorithm, known bounds on the asymptotic complexity are very loose.

5.3 Detecting Contention Peaks and MCS

Here, we define a contention peak as a maximal set of overlapping iterations of activities *in the current schedule*, collectively overusing a resource r_k. Each contention peak is a multiset and corresponds to a (Generalized) CS occurring in a specific schedule. In our algorithm, we choose the (Generalized) MCS to be resolved by: 1) identification of all the contention peaks for each resource r_k; 2) extraction of an MCS from each peak by randomly reducing the cardinality of activities in the multi-set, until minimality is met; 3) choosing an MCS from the whole pool uniformly at random. Of course some clever heuristics may be used to bias the choice towards promising MCS: this is left for future research.

Our peak detection procedure (in Algorithm 3) is an extension of the one presented in [23] for non-cyclic schedules; the method operates on the modular interval $[0, \lambda[$ and is designed to take into account the peculiar profile of the modular requirement function $\bar{r}_{ik}(t)$, consisting of a constant and a *pulse* component (see Equation 9 in Section 4.1). The main idea is to focus on the maximal overlapping of the *pulse* functions; in Figure 3 one can check how the individual modular requirement functions $\bar{r}_{1,0}$ and $\bar{r}_{2,0}$ contribute to the usage profile of resource r_0 in the modular interval $[0, \lambda[$ in our example problem. Peaks in the resource usage are due to the *pulse* components; specifically, the schedule has a single maximal peak at time 1.

The peak detection algorithm processes activities by increasing modular start time (i.e. pulse start time) and keeps track via a set X of pulses currently in execution. New pulses are added to X when they start (i.e. $\bar{s}_i = t$, line 4) and removed from X when they are over (i.e. $\bar{e}_i \leq t$). The X set is initialized before the main loop so as to include pulses that cross the λ boundary (i.e. such that $\bar{e}_i < \bar{s}_i$, in line 2). A set Rm is used to prevent the crossing activities from being removed twice from X. A peak is collected whenever some activity needs to be removed from X; in such a case, the peak contains each activity with the cardinality specified by the value of the concurrency function $\#a_i(t)$. The final step of the algorithm always processes time λ and collects the last peak.

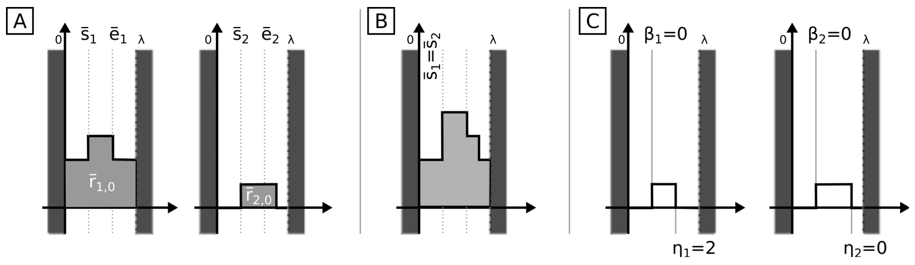


Fig. 3. A,B) effect of combining different usage functions; C) activity pulses

Algorithm 3. (Cyclic) Peak Detection

Input: a schedule and a target resource r_k
Output: a list of detected peaks
Data structures:
- Q: vector of activities requiring r_k
- X: set of executing activities in execution at the current time instant
- Rm: set of activities removed from X

```
 1: sort activities in $Q$ by increasing modular start time $\bar{s}_i$
 2: $X$ = set of activities in $Q$ such that $\bar{e}_i < \bar{s}_i$
 3: for i = 0 to $|Q|$ do
 4:    if $i < |Q|$ then $a_i$ = i-th activity in $Q$; current time $t = \bar{s}_i$
 5:    else current time $t = \lambda$
 6:    if there is any activity $a_j \in X$ such that $\bar{e}_j \leq t$ and $a_j \notin Rm$ then
 7:       if $\sum_{a_j \in Q} \bar{r}_{jk}(t) > c_k$ then
 8:          build a peak with all activities $a_j$ in $Q$, each with cardinality $\#a_j(t)$
 9:    remove from $X$ all activities $a_j$ such that $\bar{e}_j \leq t$ and $a_j \notin Rm$
10:    add to $Rm$ all the removed activities
11:    if $i < |Q|$ then add $a_i$ to $X$
12: return the list of detected peaks
```

5.4 Sampling and Adding a Resolver

Once an MCS M is selected, we sample a binary or unary resolver uniformly at random. The original iFlat algorithm can exploit temporal constraint propagation to identify trivially infeasible resolvers; since our temporal model does not provide any actual propagation mechanism (i.e. there is no time window for any activity), the sampled resolver is tested with one-step look ahead: in case the λ^* value of the resulting graph is higher then the current threshold, the resolver is discarded and a new one is sampled. In the worst case, the process stops when there are no more resolvers and the current algorithm iteration ends.

As described in Section 4.3, cyclic resolvers are either unary or binary cycles. While the delay distribution for a unary resolver is fixed; binary resolvers have parametric delay. Here, we use a heuristic to deterministically choose a distribution, given an ordered pair of activities a_i, a_j. The main underlying idea is that a binary cycle prevents the two involved activities a_i, a_j from simultaneously reaching a maximum concurrency; intuitively, this means to prevent the *pulse* components from the corresponding $\#a_i(t)$ function from overlapping. This can be done by either forcing $\bar{e_i}$ to "precede" $\bar{s_i}$, or vice-versa.

We recall the end of a_i pulse is associated to the iteration number η_i, while the beginning of a_j pulse to the iteration number β_j. In our approach, when posting a resolver on the ordered pair a_i, a_j we always heuristically set the delay of the added arc (a_i, a_j) to:

$$\delta_{ij} = \beta_j - \eta_i \tag{14}$$

intuitively, this amounts to force iteration β_j of a_j to wait for the end of iteration η_i of a_i. The δ_{ji} delay is fixed according to Equation (13). Figure 3D shows the β and η values for (the pulses of) a_1 and a_2; their computation is done as described in Section 4.1. When posting a resolver between a_1 and a_2 for the MCS $\{a_1, a_1, a_2\}$, our heuristics would set δ_{ij} to $0 - 2 = -2$ and δ_{ji} to $2 - \delta_{ij} = 4$.

5.5 Experimental Results

The cyclic iFlat approach has been implemented in C++ and tested on a benchmark consisting of industrial problems from the instruction scheduling domain [8]. In particular, the instances represent loop scheduling problems extracted by the compiler for the ST200 processor, by STMicroelectronics; each activity represent an instruction, requiring one or more CPU components for its execution. All instructions have unary duration; the maximum δ_{ij} on the whole benchmark is 4; the maximum d_{ij} is 3. With the objective to make the benchmark more challenging, in [1] the authors replaced the original resource consumption with a random number, thus providing a modified data set.

The benchmark is employed in [1] to perform a through comparison of ILP based complete approaches; given a large time limit (604800 seconds) the compared solvers found the optimal solution for almost all the instances. Our experiment focus is on assessing how close to the known optima the PCP heuristics can go; to this end, it must be mentioned that in the original problem the start times and the period were required to be integer (i.e. it was actually a modulo scheduling problem). Our approach does not make such an assumption and has in principle the chance to find better solutions; nevertheless, since all durations are unary, the optimal values found in [1] are a pretty reliable estimate of the minimal period values our heuristics could ever achieve.

All experiments are performed on an Intel Core 2 T7200, by running 1200 iFlat iterations for each instance. Table 1 shows the result of the evaluation for both the data sets (industrial and modified). Next to the instance name, the number of activities and arcs is reported. Then, for each data set, the table shows the optimal λ value (for the corresponding modulo scheduling problem), the best λ found by our heuristics within one second and the sequence number of the iteration when such solution was found. The table content is completed by the best overall λ, the iteration and the time when it was found, the total solution time. Some of the instances from the original paper are missing, due to problems during the conversion to our solver input format. For the instances reporting a "—" in the *opt* column an optimal solution was not found by the reference ILP solvers.

A number in bold face highlights the cases where our heuristics hits the actual optimum; in general cyclic iFlat obtains very good results, reaching optimal or close to optimal solutions within 1 second. In a few cases (e.g. gsm-st231.14) the heuristics appears to beat the optimal solver: this is only a consequence of the integrality requirement assumed by the complete approach used as a reference. The modified instances are remarkably more difficult than their counterparts

Table 1. Results for the benchmark from [1]; *: the reported optimum is for for the modulo scheduling problem, where the start times and the period are constrained to be integer

			INDUSTRIAL INSTANCES							MODIFIED INSTANCES						
				1 sec		best					1 sec		best			
inst	acts	arcs	opt*	λ	it	λ	it	time	tot. time	opt*	λ	it	λ	it	time	tot. time
adpcm-st231.1	86	405	21	22.00	13	**21.00**	121	4.17	39.04	—						
adpcm-st231.2	142	722	40	43.00	20	41.00	158	6.38	40.46	—						
gsm-st231.2	101	462	26	27.00	5	**26.00**	98	3.89	47.39	—						
gsm-st231.6	30	130	7	**7.00**	19	7.00	19	0.05	4.02	27	**27.00**	63	27.00	63	0.37	7.26
gsm-st231.7	44	192	11	**11.00**	4	11.00	4	0.03	9.58	41	**41.00**	53	41.00	53	0.88	18.98
gsm-st231.9	34	154	28	**28.00**	1	28.00	1	0.00	0.35	32	**32.00**	59	32.00	59	0.22	3.63
gsm-st231.10	10	42	4	**4.00**	0	4.00	0	0.00	0.26	8	**8.00**	41	8.00	41	0.01	0.33
gsm-st231.11	26	137	20	**9.00**	0	9.00	0	0.00	0.08	24	**24.00**	4	24.00	4	0.01	4.58
gsm-st231.12	15	70	8	**8.00**	40	8.00	40	0.02	0.63	13	**13.00**	65	13.00	65	0.03	0.63
gsm-st231.13	46	210	19	**19.00**	0	19.00	0	0.00	0.57	43	**43.00**	24	43.00	24	0.44	22.54
gsm-st231.14	39	176	10	**9.50**	17	9.50	17	0.12	9.21	33	36.00	17	35.00	774	9.96	15.42
gsm-st231.15	15	70	8	**8.00**	3	8.00	3	0.00	0.63	12	**12.00**	585	12.00	585	0.28	0.62
gsm-st231.16	65	323	16	**16.00**	15	16.00	15	0.52	36.48	—						
gsm-st231.17	38	173	9	**9.00**	20	9.00	20	0.14	8.21	33	34.00	3	**33.00**	262	2.80	12.46
gsm-st231.19	19	86	8	**8.00**	14	8.00	14	0.01	0.87	15	**15.00**	149	15.00	149	0.25	1.79
gsm-st231.20	23	102	6	**5.33**	186	5.33	186	0.52	3.35	20	**20.00**	48	20.00	48	0.17	4.06
gsm-st231.21	33	154	18	**18.00**	1	18.00	1	0.00	0.33	30	**29.00**	211	29.00	211	0.72	3.92
gsm-st231.22	31	146	18	**18.00**	1	18.00	1	0.00	0.30	29	**29.00**	36	29.00	36	0.15	3.10
gsm-st231.25	60	273	16	**16.00**	38	16.00	38	0.42	13.12	—						
gsm-st231.29	44	192	11	**11.00**	4	11.00	4	0.03	9.45	42	**42.00**	6	42.00	6	0.17	19.26
gsm-st231.30	30	130	7	**7.00**	19	7.00	19	0.05	4.02	25	26.00	0	**25.00**	461	2.88	7.52
gsm-st231.31	44	192	11	**11.00**	4	11.00	4	0.02	9.58	39	41.00	14	40.00	284	4.66	18.94
gsm-st231.32	32	138	15	**15.00**	0	15.00	0	0.00	0.10	30	**30.00**	13	30.00	13	0.05	8.11
gsm-st231.33	59	266	15	**15.00**	11	14.50	324	3.06	11.25	—						
gsm-st231.34	10	42	4	**4.00**	2	4.00	2	0.00	0.25	7	**7.00**	82	7.00	82	0.03	0.32
gsm-st231.35	18	80	6	**6.00**	13	6.00	13	0.01	0.67	14	15.00	33	**14.00**	1109	1.07	1.15
gsm-st231.36	31	143	10	**10.00**	28	10.00	28	0.05	1.99	24	27.00	8	26.00	757	3.56	5.77
gsm-st231.39	26	118	8	**8.00**	10	8.00	10	0.01	1.84	21	22.00	105	22.00	105	0.43	4.79
gsm-st231.40	21	103	10	**10.00**	17	10.00	17	0.02	1.13	17	**17.00**	32	17.00	32	0.06	2.00
gsm-st231.41	60	315	18	20.00	54	18.00	139	2.06	17.75	—						
gsm-st231.42	23	102	6	**5.33**	186	5.33	186	0.51	3.31	18	19.00	109	**18.00**	882	2.65	3.69
gsm-st231.43	26	115	8	9.00	8	9.00	8	0.01	0.16	20	22.00	52	21.00	1153	1.93	2.01

and set tougher challenges to iFlat, which nevertheless obtains pretty good results. The delay distribution heuristics from Section 5.4 seems to be a key factor to get high quality solutions.

6 Conclusion and Future Work

The main contribution of this work is the generalization of the key PCP concepts to the cyclic scheduling domain; in this context, the use of PCP avoids the repeated resolution of NP-hard subproblem, which is a common trait of most state of the art approaches. As a practical demonstrator, we implemented a simple and yet effective cyclic version of the Iterative Flattening heuristics.

Many interesting research directions remain open: the parameter space of binary resolvers should be characterized so as to narrow the range of possible delay distribution choices; resource and temporal propagation techniques for cyclic problems should be defined; effective MCS selection heuristics should be tested and other search schemes investigated.

References

1. Ayala, M., Artigues, C.: On integer linear programming formulations for the resource-constrained modulo scheduling problem (2010)
2. Blachot, F., de Dinechin, B.D., Huard, G.: SCAN: A heuristic for near-optimal software pipelining. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 289–298. Springer, Heidelberg (2006)
3. Cesta, A., Oddi, A., Smith, S.F.: Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems. In: Proc. of AAAI/IAAI, pp. 742–747 (2000)
4. Cesta, A., Oddi, A., Smith, S.F.: Scheduling multi-capacitated resources under complex temporal constraints. In: Maher, M.J., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, p. 465. Springer, Heidelberg (1998)
5. Cesta, A., Oddi, A., Smith, S.F.: A Constraint-Based Method for Project Scheduling with Time Windows. Journal of Heuristics 8(1), 109–136 (2002)
6. Chrétienne, P.: Transient and limiting behavior of timed event graphs. RAIRO Techniques et Sciences Informatiques 4, 127–192 (1985)
7. Dasdan, A.: Experimental analysis of the fastest optimum cycle ratio and mean algorithms. ACM Transactions on Design Automation of Electronic 9(4), 385–418 (2004)
8. de Dinechin, B.D.: From machine scheduling to VLIW instruction scheduling. ST Journal of Research 1(2), 1–35 (2004)
9. de Dinechin, B.D., Artigues, C., Azem, S.: Resource-Constrained Modulo Scheduling, ch. 18. ISTE, London (2010)
10. Draper, D.L., Jonsson, A.K., Clements, D.P., Joslin, D.E.: Cyclic scheduling. In: Proc. of IJCAI, pp. 1016–1021. Morgan Kaufmann Publishers Inc., San Francisco (1999)
11. Eichenberger, A.E., Davidson, E.S.: Efficient formulation for optimal modulo schedulers. ACM SIGPLAN Notices 32(5), 194–205 (1997)
12. Georgiadis, L., Golberg, A.V., Tarjan, R.E., Werneck, R.F.: An experimental study of minimum mean cycle algorithms. In: Proc. of ALENEX. Citeseer (2009)
13. Ghamarian, A.H., Geilen, M., Stuijk, S., Basten, T., Theelen, B.D., Mousavi, M.R., Moonen, A.J.M., Bekooij, M.: Throughput Analysis of Synchronous Data Flow Graphs. In: Proc. of ACSD, pp. 25–36 (2006)
14. Hanen, C., Munier, A.: Cyclic scheduling on parallel processors: an overview, ch. 4. Wiley, Chichester (1994)
15. Heilmann, R.: A branch-and-bound procedure for the multi-mode resource-constrained project scheduling problem with minimum and maximum time lags. European Journal of Operational Research 144(2), 348–365 (2003)
16. Howard, R.A.: Dynamic Programming and Markov Processes. Wiley, New York (1960)
17. Igelmund, G., Radermacher, F.J.: Algorithmic approaches to preselective strategies for stochastic scheduling problems. Networks 13(1), 29–48 (1983)
18. Igelmund, G., Radermacher, F.J.: Preselective strategies for the optimization of stochastic project networks under resource constraints. Networks 13(1), 1–28 (1983)
19. Laborie, P.: Complete MCS-Based Search: Application to Resource Constrained Project Scheduling. In: Proc. of IJCAI, pp. 181–186. Professional Book Center (2005)

20. Lam, M.: Software pipelining: An effective scheduling technique for VLIW machines. In: Proc. of ACM SIGPLAN 1988, vol. 23, pp. 318–328. ACM, New York (1988)
21. McCormick, S.T., Rao, U.S.: Some complexity results in cyclic scheduling. Mathematical and Computer Modelling 20(2), 107–122 (1994)
22. Parhi, K.K., Messerschmitt, D.G.: Rate-optimal fully-static multiprocessor scheduling of data-flow signal processing programs. In: Proc. of ISCAS, vol. 217, pp. 1923–1928. IEEE, Los Alamitos (1989)
23. Policella, N., Cesta, A., Oddi, A., Smith, S.F.: From precedence constraint posting to partial order schedules: A CSP approach to Robust Scheduling. AI Communications 20(3), 163–180 (2007)
24. Policella, N., Smith, S.F., Cesta, A., Oddi, A.: Generating Robust Schedules through Temporal Flexibility. In: Proc. of ICAPS, pp. 209–218 (2004)
25. Rau, B.R.: Iterative modulo scheduling: An algorithm for software pipelining loops. In: Proc. of MICRO, pp. 63–74. ACM, New York (1994)
26. Young, N., Tarjan, R., Orlin, J.: Faster Parametric Shortest Path and Minimum Balance Algorithms. Networks 21, 205–221 (2002)

A Probing Algorithm for MINLP with Failure Prediction by SVM

Giacomo Nannicini[1,⋆], Pietro Belotti[2], Jon Lee[3], Jeff Linderoth[4,⋆⋆], François Margot[1,⋆⋆⋆], and Andreas Wächter[3]

[1] Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA
{nannicin,fmargot}@andrew.cmu.edu

[2] Dept. of Mathematical Sciences, Clemson University, Clemson, SC
pbelott@clemson.edu

[3] IBM T. J. Watson Research Center, Yorktown Heights, NY
{jonlee,andreasw}@us.ibm.com

[4] Industrial and Systems Eng., University of Wisconsin-Madison, Madison, WI
linderoth@wisc.edu

Abstract. Bound tightening is an important component of algorithms for solving nonconvex Mixed Integer Nonlinear Programs. A *probing* algorithm is a bound-tightening procedure that explores the consequences of restricting a variable to a subinterval with the goal of tightening its bounds. We propose a variant of probing where exploration is based on iteratively applying a truncated Branch-and-Bound algorithm. As this approach is computationally expensive, we use a Support-Vector-Machine classifier to infer whether or not the probing algorithm should be used. Computational experiments demonstrate that the use of this classifier saves a substantial amount of CPU time at the cost of a marginally weaker bound tightening.

1 Introduction

A Mixed Integer Nonlinear Program (MINLP) is a mathematical program with continuous nonlinear objective and constraints, where some of the variables are required to take integer values. Without loss of generality, we assume that the problem is a minimization problem. MINLPs naturally arise in numerous applied problems, see e.g. [1,2]. In this paper, we address nonconvex MINLPs where neither the objective function nor the constraints are required to be convex — a class of problems typically difficult to solve in practice. An exact solution method for nonconvex MINLPs is Branch-and-Bound [3], where lower bounds are obtained by convexifying the feasible region using under-estimators, often linear inequalities [4,5]. The convexification depends on the variable bounds,

⋆ Supported by an IBM grant and by NSF grant OCI-0750826.
⋆⋆ Supported by U.S. Department of Energy grant DE-FG02-08ER25861 and by NSF grant CCF-0830153.
⋆⋆⋆ Supported by NSF grant OCI-0750826.

with tighter bounds resulting generally in a tighter convexification. As such, bound tightening is an important part of any MINLP solver.

Probing is a bound-tightening technique often applied to Mixed Integer Linear Programs (MILPs) [6]. The idea is to tentatively fix a binary variable to 0 and then to 1, and use the information obtained to strengthen the linear relaxation of the problem. Similar techniques have been applied to MINLPs as well [5]. In this paper, we propose a probing technique based on truncated Branch-and-Bound searches. Let $\bar{z}$ be the objective value of the best solution of the original problem found so far. In each Branch-and-Bound search, we choose a variable, say x_i, and impose $x_i \in S$, where S is a subinterval of the current domain of x_i. In addition, we add a constraint bounding the objective value of the solution to at most $\bar{z}$. If that problem is infeasible, we can discard S from the domain of x_i. On the other hand, if we are able to solve the modified problem to optimality, with an optimal value $\bar{z}^* < \bar{z}$, we update $\bar{z}$ and can again discard S from the domain of x_i. Details on the choice of x_i and S are given in Section 3.

This probing algorithm potentially requires a significant amount of CPU time. To limit this drawback, we use a Support Vector Machine (SVM) classifier [7] before performing a Branch-and-Bound search, to predict the success or failure of the search. If we conclude that the probing algorithm is unlikely to tighten the bounds on the variable, we skip its application. Machine learning methods have been used in the OR community for various tasks, such as parameter tuning [8] and solver selection [9]. In this paper, machine learning is used to predict failures of an algorithm based on characteristics of its input data. The features on which the SVM prediction is based are problem and subinterval dependent, and are related to the outcome of the application of a fast bound-tightening technique (Feasibility-Based Bound Tightening [5]) using the same subinterval.

We provide preliminary computational results to assess the practical efficiency of the approach. The experiments show that the proposed probing algorithm is very effective in tightening the variable bounds, and it is helpful for solving MINLPs with Branch-and-Bound. By using SVM to predict failures of the probing algorithm, we save on average 30% of the total bound-tightening time, without much deterioration of the quality of the bounds.

The rest of this paper is organized as follows. In Section 2, we introduce the necessary background. In Section 3, we describe the probing algorithm. In Section 4, we discuss how we can integrate a machine learning method in our algorithm to save CPU time. In Section 5, we provide computational testing of the proposed ideas and Section 6 has conclusions.

2 Background

A function is *factorable* if it can be computed in a finite number of simple steps, starting with model variables and real constants, using elementary unary and binary operators. We consider an MINLP of the form:

$$\left.\begin{array}{rl} \min & f(x) \\ s.t. & g_j(x) \leq 0 \quad \forall j \in M \\ & x_i^L \leq x_i \leq x_i^U \quad \forall i \in N \\ & f(x) \leq \bar{z} \\ & x_i \in \mathbb{Z} \quad \forall i \in N_I, \end{array}\right\} \qquad \mathcal{P}$$

where f and g_j are factorable functions, $N = \{1, \ldots, n\}$ is the set of variable indices, $M = \{1, \ldots, m\}$ is the set of constraint indices, $x \in \mathbb{R}^n$ is the vector of variables with lower/upper bounds $x^L \in (\mathbb{R} \cup \{-\infty\})^n$, $x^U \in (\mathbb{R} \cup \{+\infty\})^n$, and $\bar{z}$ is an upper bound on the optimal objective value, which can be infinite. The variables with indices in $N_I \subset N$ are constrained to take on integer values in the solution.

A Linear Programming (LP) based Branch-and-Bound algorithm can be used to solve $\mathcal{P}$ [4]. In such a method, subproblems of $\mathcal{P}$ are generated by restricting the variables to reduced interval domains, $[\bar{x}^L, \bar{x}^U] \subset [x^L, x^U]$. A key step is the creation of an LP relaxation of the feasible region of a subproblem, which we refer to as *convexification*. This convexification is used to obtain a lower bound on the optimal objective value of the subproblem. In general, the tighter the variable bounds, the tighter the convexification, and the stronger the resulting lower bound. Therefore, bound-tightening techniques aim to deduce improved variable bounds implied by the constraint structure of the subproblem, and are widely used by existing software, such as `Baron` [10] and `Couenne` [11], for the solution of MINLPs.

A commonly used bound-tightening procedure is Feasibility-Based Bound Tightening (FBBT), which uses a symbolic representation of the problem in order to propagate bound changes on a variable to other variables. For instance, suppose that $\mathcal{P}$ contains the equation $x_3 = x_1 + x_2$, with variable bounds $x_1 \in [0, 1]$, $x_2 \in [0, 3]$, $x_3 \in [0, 4]$; if we tighten the bounds on x_2 and restrict this variable to the interval $[1, 2]$, then we can propagate the change to x_3 and impose $x_3 \in [1, 3]$. A full description of FBBT can be found in [12,13].

The other aspects of the Branch-and-Bound algorithm are similar to those of any Branch-and-Bound for solving MILPs; see [5] for more details.

3 The Probing Algorithm

In this section we describe the probing algorithm to increase the lower bound on variable x_i, where the current bounds on that variable are $x_i \in [x_i^L, x_i^U]$ with $x_i^L > -\infty$. The special case $x_i^L = -\infty$ is treated below. The probing algorithm for decreasing the upper bound is similar. For simplicity, we describe the procedure applied to the root node $\mathcal{P}$.

Let ℓ and u be such that $x_i^L \leq \ell \leq u \leq x_i^U$. We denote by $\mathcal{P}_i[\ell, u]$ the problem obtained from $\mathcal{P}$ by adding the constraint $x_i \in [\ell, u]$. For $s > 0$, an *s-probing iteration* for x_i consists of the following: set $\ell = x_i^L$, $u = \min\{x_i^L + s, x_i^U\}$, and perform a Branch-and-Bound search on $\mathcal{P}_i[\ell, u]$ *with a time limit*. If we have then proved that $\mathcal{P}_i[\ell, u]$ is infeasible, we update the lower bound $x_i^L \leftarrow u$. If we are able to solve $\mathcal{P}_i[\ell, u]$ to optimality, finding a solution with objective value z^*, we update the best incumbent value $\bar{z} \leftarrow z^*$ and the lower bound $x_i^L \leftarrow u$. In both cases, the s-probing iteration is deemed a success. Otherwise, it is a failure.

The Aggressive Probing algorithm for variable x_i (see Algorithm 1) has an initial value for s as input and runs an s-probing iteration. While an exit condition is not met, if the s-probing iteration is successful, the value of s is doubled and a new s-probing iteration is executed. If an s-probing iteration fails, the value of s is halved and a new s-probing iteration is performed.

For integer variables, we round the probing interval endpoints appropriately. Additionally, if the search on $P_i[\ell, u]$ is completed and u is integer, we set $x_i^L \leftarrow u + 1$ instead of $x_i^L \leftarrow u$.

With sufficiently large time limits, Algorithm 1 will provide an optimality certificate if $\bar{z}$ is the optimal objective value. If run to completion, the algorithm proves that no better solution exists with x_i in the interval $[x_i^L, x_i^U]$.

The special case $x_i^L = -\infty$ is handled as follows. Define a positive number B. We perform a Branch-and-Bound search on $\mathcal{P}_i[-\infty, -B]$. If $\mathcal{P}_i[-\infty, -B]$ is proved infeasible or solved to optimality within the time limit, we set $x_i^L \leftarrow -B$ and execute Algorithm 1. Otherwise, we conclude that x_i^L cannot be tightened. In our experiments, we use $B = 10^{10}$.

Two details of the algorithm still need to be specified: the exit condition and the initial choice of s. We use two exit conditions: a maximum CPU time for the application of Aggressive Probing, and a maximum number of consecutive failed s-probing iterations. We experimented with several choices for the initial value of s that took into account the distance between the variable bounds and the solution of the LP relaxation (or a feasible solution to $\mathcal{P}$, if available). But the results were not better than a simpler method that seems to work well: the initial value of s is chosen to be a small, fixed value `size`, depending on the variable type. In our experiments, we use `size` $= 0.5$ for continuous variables, `size` $= 1.0$ for general integer variables, and `size` $= 0.0$ for binary variables.

The good performance of the update strategy and the initial choice for the value of s lies in the dynamic and geometric adjustment of the interval length during Aggressive Probing. If the initial interval can easily be proven infeasible, the s-probing iteration will terminate very quickly, typically with a single application of FBBT, or at the root node by solving the LP relaxation. In this case, because the interval size is increased in a geometric fashion, in a few iterations we will reach the scale that is needed for the probing interval to be "not trivially infeasible". On the other hand, using a small interval size yields better chances of completing the s-probing iteration within the time limit, in case $\mathcal{P}_i[\ell, u]$ is difficult to solve even with u close to ℓ.

In our experiments, we set the time limit for the Branch-and-Bound search during an s-probing iteration to $\min\{2/3\, \texttt{time\_limit}, \texttt{time\_limit} - \mathit{current\_time}\}$. This avoids investing all of the CPU time in the first probing iteration in cases where the initial interval-size guess is too large.

Algorithm 1. The Aggressive Probing algorithm

```
Input: variable index $i$, time_limit, size, max_failures
Set $s \leftarrow$ size, $fail \leftarrow 0$,
while $current\_time <$ time_limit and $fail <$ max_failures and $x_i^L < x_i^U$ do
  $\ell \leftarrow x_i^L$
  Set $u \leftarrow \min\{\ell + s, x_i^U\}$; if $x_i$ is integer constrained, round $u \leftarrow \lfloor u \rfloor$
  Execute limited-time Branch-and-Bound on $\mathcal{P}_i[\ell, u]$
  if solution $\bar{x}$ found then
    Set $\bar{z} \leftarrow \min\{\bar{z}, f(\bar{x})\}$
  if search complete then
    if $x_i$ is integer constrained then
      Set $x_i^L \leftarrow u + 1$
    else
      Set $x_i^L \leftarrow u$
    Set $s \leftarrow 2s$, $fail \leftarrow 0$
  else
    Set $s \leftarrow s/2$, $fail \leftarrow fail + 1$
```

4 Support Vector Machine for Failure Prediction

Applying Aggressive Probing to tighten all variables of even a moderately-sized MINLP can take a considerable amount of CPU time. We would like to avoid wasting time in trying to tighten the bounds of a variable for which there is little hope of success.

Observe that if a probing subproblem $\mathcal{P}_i[\ell, u]$ is not solved to optimality, we make no progress towards bound tightening and the CPU time invested in that probing iteration is wasted (unless a feasible solution better than the incumbent $\bar{z}$ is found: in this case, the improved solution is kept). To avoid this situation, we suggest to use the degree of success in applying the fast FBBT algorithm on $\mathcal{P}_i[\ell, u]$ as a factor in deciding whether or not to run the expensive Aggressive Probing algorithm. Note that FBBT is the first step of the Branch-and-Bound algorithm used in Aggressive Probing, so this does not require additional work. Our hypothesis is that if the constraint $x_i \in [\ell, u]$ used during an s-probing iteration does not result in tighter bounds on other variables when FBBT is used, then $\mathcal{P}_i[\ell, u]$ is approximately as difficult as $\mathcal{P}$, so the limited-time Branch-and-Bound algorithm is likely to fail. This intuition is confirmed by empirical tests: on 84 nontrivial MINLP instances taken from various sources, we perform Aggressive Probing with `max_failures` $= 10$ to tighten lower and upper bound of all variables, processing them in the order in which they appear in the problem, with a time limit of 1 minute per variable and a total time limit of 1 hour per instance. We record, for each s-probing iteration, whether FBBT is able to use the probing interval $x_i \in [\ell, u]$ to tighten the bounds on other variables. We observe that, in 587 cases out of 11,747, no stronger bounds are obtained. In 528 of these 587 cases (90%), the subsequent Branch-and-Bound search on $\mathcal{P}_i[\ell, u]$ could not be completed within the time limit. Therefore, the success of FBBT in using $x_i \in [\ell, u]$ to tighten other variable bounds indeed gives an indication of the difficulty of solving $\mathcal{P}_i[\ell, u]$.

Supported by this observation, we use the following strategy. Before applying Branch-and-Bound to $\mathcal{P}_i[\ell, u]$, we execute FBBT and compute a measure of the bound reduction obtained on the variables $x_j, j \in N \backslash \{i\}$. Based on this measure, we use an algorithm to decide whether to perform Branch-and-Bound on $\mathcal{P}_i[\ell, u]$. In the remainder of this section we discuss our choice of the bound reduction measure and the decision method.

4.1 Measuring the Effect of FBBT

Several bound-reduction measures are possible. Because our aim is to save CPU time, the bound-reduction measure computation should be fast. A simple way of measuring the bound reduction obtained for the variables x_j, $j \in N \setminus \{i\}$, is to count the number of tightened variables, and for each of these, to compute the interval reduction: $\gamma(x_j) = 1 - (\tilde{x}_j^U - \tilde{x}_j^L)/(x_j^U - x_j^L)$, where $\tilde{x}^L$ (resp. $\tilde{x}^U$) are the vectors of variable lower (resp. upper) bounds after applying FBBT, and x^L, x^U are those of the original problem $\mathcal{P}$. (Infinity is treated like a number in the following way: $1/\infty = 0, \infty/2\infty = 0.5$.) Hence, to quantify the bound reduction associated with the application of FBBT on the problem $\mathcal{P}_i[\ell, u]$, we use a vector $(\eta, \rho) \in [0, 1]^2$, where η is the fraction of tightened variables, and ρ is the average value of $\gamma(x_j)$ over all x_j that were successfully tightened.

4.2 Support Vector Machines

Once a vector (η, ρ) is computed for variable x_i, the decision of whether to perform Branch-and-Bound is taken by a predictor trained by a Support Vector Machine (SVM). While it could be argued that SVM is not really required for classifying our 2-dimensional data, we use SVM for three reasons. First, in our experiments SVM performs better than a predictor based on a simple Gaussian model for the data, see Section 5.2. Second, our future research efforts will utilize additional input features besides (η, ρ) (see Section 6 for details); therefore, the flexibility and extensibility of SVM is desirable. Finally, SVM is a parameterized method allowing better control of the trade-off between Precision and Recall of the classifier (see below for details) than simpler methods. Because we are interested in a classifier with high Precision and good Recall, the ability to tune the classifier is an advantage.

Next, we provide a brief description of the basic concepts behind SVM; see [14,15] for a comprehensive introduction to the topic. Given training data $\mathcal{D} = \{(z_i, y_i) : z_i \in \mathbb{R}^p, y_i \in \{-1, 1\}, i \in 1, \ldots, q\}$, SVM seeks an affine hyperplane that separates the points with label -1 from those with label 1 by the largest amount, i.e., the width of the strip containing the hyperplane and no data point in its interior is as large as possible.

In its simplest form, the associated optimization problem can be written as:

$$\left.\begin{array}{lll} \min & \|h\|_2 & \\ s.t. & y_i(h^\top z_i - b) \geq 1 & \forall (z_i, y_i) \in \mathcal{D} \\ & h \in \mathbb{R}^p, b \in \mathbb{R}, & \end{array}\right\} \qquad \text{(SVM)}$$

where the hyperplane is defined by $h^\top z = b$. Instead of seeking a separating hyperplane in $\mathbb{R}^p$, which may not exist, SVM implicitly maps each data point into a higher dimensional *feature space* where linear separation may be possible. The mapping is implicit because we do not need explicit knowledge of the feature space. In the optimization problem (SVM), we express the separating hyperplane in terms of the training points z_i (see e.g., [15]), and substitute the dot-products between vectors in $\mathbb{R}^p$ with a possibly nonlinear *kernel function* $K : \mathbb{R}^p \times \mathbb{R}^p \mapsto \mathbb{R}$. The kernel function can be interpreted as the dot-product in the higher-dimensional space. The separation hyperplane in the feature space translates into a nonlinear separation surface in the original space $\mathbb{R}^p$. Furthermore, SVM handles data that is not separable in the feature space by using a *soft margin*, i.e., allowing the optimal separation hyperplane to misclassify some points, imposing a penalty for each misclassification. The outcome of the SVM training algorithm is a subset V of $\{z_i : \exists y \in \{-1, 1\} \text{ with } (z_i, y) \in \mathcal{D}\}$ with corresponding scalar multipliers $\alpha_v : v \in V$, and a scalar b. The elements of V are called *support vectors*. To classify a new data point $w \in \mathbb{R}^p$, we compute the value of $\sum_{v \in V} \alpha_v K(v, w) - b$ and use its sign to classify w. Hence, the complexity of storing an SVM model depends on the number of support vectors $|V|$, and the time required to classify a new data point depends on $|V|$ and K.

Commonly used kernel functions are:

- linear: $K(u, v) = u^\top v$,
- polynomial: $K(u, v) = (\lambda u^\top v + \beta)^d$,
- radial basis: $K(u, v) = e^{-\lambda \|u - v\|^2}$,

where λ, β and d are input parameters. Problem-specific kernel functions can be devised as well. Another commonly adjusted tuning parameter is the misclassification cost ω, which determines the ratio between the penalty paid for misclassifying an example of label 1 and the penalty paid for misclassifying an example of label -1. The ratio ω can be adjusted to handle unbalanced data sets where one class is much more frequent than the other.

4.3 Aggressive Probing Failure Prediction with SVM

In this section we assume that we have an SVM model trained on a data set of the form $\mathcal{D} = \{(\eta_i, \rho_i, y_i) : (\eta_i, \rho_i) \in [0, 1]^2, y_i \in \{-1, 1\}, i = 1, \dots, q\}$, where each point corresponds to an s-probing iteration, η_i, ρ_i are as defined in Section 4.1, and $y_i = 1$ if the limited-time Branch-and-Bound search applied to the corresponding probing subproblem did not complete, $y_i = -1$ otherwise. Generating the set $\mathcal{D}$ and computational experiments with model training will be discussed in Section 5.

Given such an SVM model, we proceed as follows. At an s-probing iteration corresponding to problem $\mathcal{P}_i[\ell, u]$, we apply FBBT and compute the resulting $w_j = (\eta_j, \rho_j)$ as described in Section 4.1. If $w_j = (0, 0)$, FBBT could not tighten the bounds on any variable; in this case, as discussed at the beginning of Section 4, we do not execute Branch-and-Bound and continue to the subsequent probing iteration as if the s-probing iteration failed. If $w_j \neq (0, 0)$, we predict the label y_j of w_j using our SVM classifier. The Branch-and-Bound search on $\mathcal{P}_i[\ell, u]$ is thus executed only if the predicted label is $y_j = -1$; otherwise, we continue with the algorithm as if the s-probing iteration failed.

Note that we could apply the SVM classifier even on points of the form $(0, 0)$. However, in our experiments this point was always labeled as 1 by the tested SVM models, therefore we save CPU time by not running the SVM predictor. Additionally, we exclude points $(0, 0, y_i)$ from the data set $\mathcal{D}$ on which the model is trained; this yields an additional advantage that will be discussed in Section 5.

5 Computational Experiments

We implemented Aggressive Probing within `Couenne`, an open-source Branch-and-Bound solver for nonconvex MINLPs [11]. We are mainly interested in applying our probing technique to difficult instances $\mathcal{P}$ to improve a Branch-and-Bound search; thus, in our implementation Aggressive Probing reuses as much previously computed information as possible. The root node of each probing subproblem $\mathcal{P}_i[\ell, u]$ is generated by modifying the root node of $\mathcal{P}$, changing variable bounds and possibly generating new linear inequalities to improve the convexification, so that the problem instance is read and processed only once. The branching strategy of `Couenne` was set to `Strong Branching` [5,16] in all experiments.

We utilized LIBSVM [17], a library for Support Vector Machines written in C. Given the availability of LIBSVM's source code, it could be efficiently integrated within `Couenne` for our tests. The experiments were conducted on a 2.6 GHz AMD Opteron 852 machine with 64GB of RAM, running Linux.

5.1 Test Instances

The test instances are a subset of MINLPLib [18], a freely available collection of convex and nonconvex MINLPs. We excluded instances with more than 1,000 variables and instances for which the default version of `Couenne` took more than 2 minutes to process the root node, or ran into numerical problems. Additionally, we excluded the instances for which Aggressive Probing was able to find the optimal solution and provide an optimality certificate in less than 2 hours. These are easy instances that can be quickly solved by default `Couenne`, therefore there is no need for expensive bound-tightening methods. We are left with 32 instances, which are listed in Table 1.

5.2 Training the SVM Classifier

As a first step in training an SVM to classify failures of the probing algorithm, we obtained a large-enough set of training examples. We used a superset of the test problems described in Section 5.1, including some additional problems from MINLPLib as well as problems from [19] with less than 1,000 variables, giving a total of 84 instances. We applied Aggressive Probing on all variables, with a time limit of 30 seconds for each s-probing iteration, 1 minute for each variable bound, and 2 hours per problem instance. We did not include data for s-probing iterations started with less than 20 seconds of CPU time left within the time limit, or iterations in which a feasible MINLP solution was discovered during the Branch-and-Bound search. For the remaining s-probing iterations, we recorded the values of (η_i, ρ_i) (see Section 4.1) and a label $y_i = 1$ if the probing iteration fails, $y_i = -1$ if it succeeds. The reason for excluding s-probing iterations performed with less than 20 seconds of CPU time left is that they are likely to fail simply because they are not given enough time to complete, regardless of the difficulty of the s-probing subproblem. Similarly, we excluded s-probing iterations in which an improved solution was found, because such a discovery cannot be predicted by only considering the s-probing subproblem, yet it can be used to infer tighter variable bounds through FBBT, therefore making $\mathcal{P}_i[\ell, u]$ easier than initially estimated. This yields a data set $\mathcal{D}$ with $q = 11,747$ data points that can be used for training, as explained in Section 4.3. Eliminating all points with $(\eta_i, \rho_i) = (0, 0)$ (see end of Section 4.3) leaves $11,160$ points, of which $4,186$ have the label $y_i = 1$. By removing the points $(0, 0)$ from the training set, the number of support vectors in the final model is likely to be smaller.

It is known that SVM is very sensitive to its algorithm settings, hence a grid search on a set of input parameters is typically applied in order to find the values that yield the best performance on the input data. We tested three types of kernel functions: linear, polynomial, and radial. For each of these kernel functions, we performed grid search on the input parameters (see end of Section 4.2), using the following values: $\lambda = 2^k$ with $k = -3, \ldots, 2$, $\beta = 2^k$ with $k = -3, \ldots, 2$, $d = 1, \ldots, 5$, and $\omega = 2^k$ with $k = -3, \ldots, 4$. Each parameter was considered only when appropriate; e.g., d was used for the polynomial kernel only. Overall, we tested 1,057 combinations of input parameters.

In our first set of experiments pertaining to the training of an SVM on $\mathcal{D}$, we performed 3-fold cross validation. We trained the model on 2/3 of $\mathcal{D}$, and used the remaining 1/3 to estimate the performance of the model. The resulting model consisted of 4,500 to 5,000 support vectors. Such a large number of support vectors would yield a slow classifier and may indicate overfitting. To obtain a model with fewer support vectors, we first attempted ν-classification [14], without success. In the end, training the model on a small subset of the full data set $\mathcal{D}$ was found to be very effective in reducing the number of support vectors, without deterioration in the accuracy of the model; this approach has been used before in the machine learning community [20].

With this setup, experiments for testing parameter values of the model were performed as follows. We randomly selected 10 different training sets, each one containing 1/10 of the full data set $\mathcal{D}$. Each training experiment, corresponding to a set of input parameter values, was performed on each of the 10 training sets, and the performance of the resulting models was evaluated on the 9/10 of $\mathcal{D}$ that was not used for training. To measure performance, we use Precision and Recall, commonly defined as follows:

$$\text{Precision: } TP/(TP + FP), \qquad \text{Recall: } TP/(TP + FN),$$

Table 1. Performance of Aggressive Probing, with and without failure prediction by SVM. We report, after probing (columns "Probing") and after applying FBBT and recomputing the convexification (columns "Probing + FBBT + Conv."): the fraction of tightened variables with finite bounds, and the average bound reduction. We also report the amount of optimality gap closed by the new convexification, and the total probing time.

			Without SVM						With SVM					
	# vars		Probing		Probing + FBBT + Conv.				Probing		Probing + FBBT + Conv.			
Instance	orig	total	Tght. %	Red. %	Tght. %	Red. %	Gap %	Time	Tght. %	Red. %	Tght. %	Red. %	Gap %	Time
csched2	401	582	2.00	5.05	3.44	7.12	97.76	36474.7	1.25	4.74	1.20	17.50	97.76	288.2
fo7_2	115	253	12.17	25.00	13.04	43.62	0.00	11133.1	12.17	7.87	13.04	35.37	0.00	6731.2
fo7	115	253	10.43	31.84	9.88	57.82	0.00	11129.9	10.43	5.87	9.88	43.28	0.00	5487.6
fo8_ar2_1	145	371	46.90	50.12	38.81	47.23	0.00	16372.5	46.90	49.99	38.81	47.15	0.00	13494.5
fo8_ar25_1	145	371	46.90	50.97	38.81	47.70	0.00	16373.4	46.90	50.22	38.81	47.24	0.00	13523.9
fo8_ar3_1	145	371	51.72	48.53	46.63	41.42	0.00	16448.0	52.41	46.31	48.25	39.41	0.00	14474.2
fo8_ar5_1	145	371	49.66	47.91	42.59	43.50	0.00	16385.6	49.66	48.06	42.59	43.60	0.00	14715.3
fo8	147	325	9.52	24.80	8.92	55.21	0.00	14186.1	8.16	3.23	8.31	46.29	0.00	6107.0
fo9_ar2_1	181	462	47.51	49.45	39.18	46.19	0.00	20435.2	47.51	46.50	39.18	44.30	0.00	13768.2
fo9_ar25_1	181	462	47.51	49.57	39.18	46.16	0.00	20439.2	47.51	46.45	39.18	44.18	0.00	13821.3
fo9_ar3_1	181	462	49.72	50.20	42.42	44.66	0.00	20433.2	49.72	45.41	42.42	41.80	0.00	17838.2
fo9_ar4_1	181	462	49.72	48.85	42.42	43.70	0.00	20443.4	49.72	44.90	42.42	41.48	0.00	17977.8
fo9_ar5_1	181	462	49.72	48.26	42.42	43.29	0.00	20437.1	49.72	44.95	42.42	41.44	0.00	18037.8
fo9	183	406	8.20	16.81	7.88	52.66	0.00	17546.6	8.20	2.50	7.88	45.12	0.00	7021.7
lop97icx	987	1393	0.00	0.00	0.00	0.00	0.00	120602.0	0.00	0.00	0.00	0.00	0.00	21.1
nvs23	10	64	90.00	92.07	100.00	98.41	97.02	1080.8	90.00	91.80	100.00	98.38	97.10	1080.7
nvs24	11	76	90.91	88.25	100.00	97.47	94.96	1201.0	90.91	88.25	100.00	97.47	94.96	1201.2
o7_2	115	253	19.13	26.50	19.37	46.38	0.00	11292.3	19.13	26.50	19.37	46.38	0.00	10191.1
o7_ar4_1	113	290	54.87	47.15	48.28	47.16	0.00	12918.8	54.87	47.25	48.28	47.22	0.00	12918.4
o7	115	253	17.39	28.12	16.21	53.76	0.00	11294.3	17.39	27.97	16.21	53.67	0.00	6872.0
o8_ar4_1	145	371	59.31	45.35	52.29	47.52	0.00	16696.9	59.31	45.80	52.29	47.81	0.00	16697.2
o9_ar4_1	181	462	56.35	45.76	49.13	45.96	0.00	20717.9	56.35	45.76	49.13	45.96	0.00	20474.3
space25a	384	502	17.19	38.16	32.67	30.61	0.00	36746.0	16.93	36.64	29.28	32.22	0.00	36704.2
tln12	169	361	35.50	35.67	37.95	34.10	0.00	19243.3	35.50	35.67	37.95	34.10	0.00	17973.4
tln5	36	81	41.67	33.33	62.96	44.86	0.00	3997.8	41.67	33.33	62.96	44.86	0.00	3997.8
tln6	49	109	61.22	28.33	71.56	41.24	0.00	5532.1	61.22	28.33	71.56	41.24	0.00	5619.6
tln7	64	141	43.75	25.60	64.54	32.90	0.00	7285.5	43.75	25.60	64.54	32.90	0.00	7281.3
tls12	813	1285	14.39	67.42	32.14	48.11	0.00	129660.0	14.39	67.42	32.14	48.11	0.00	128592.0
tls6	216	359	2.78	100.00	25.07	45.60	0.00	18514.3	2.78	100.00	25.07	45.60	0.00	18401.0
water4	196	319	25.51	56.54	40.13	50.49	41.54	18305.9	27.55	60.80	41.69	53.47	50.47	18013.8
waterx	71	174	14.08	25.38	32.18	20.77	34.83	7926.7	15.49	23.08	34.48	19.36	34.83	7874.8
waterz	196	319	15.31	62.02	25.71	55.10	21.47	18637.7	14.80	67.28	24.14	60.07	29.35	18617.8
Avg.	197.41	388.28	23.92	43.53	30.50	45.65	12.11	22496.6	23.90	40.58	30.32	44.59	12.64	15494.3

where TP is the number of True Positives, i.e., the examples with label 1 that are classified with label 1 by the model. FP is the number of False Positives, i.e., the examples with label -1 that are classified with label 1. Finally, FN is the number of False Negatives, i.e., the examples with label 1 that are classified with label -1. Intuitively, Precision is the fraction of data points labeled 1 by the classifier that are indeed of class 1, whereas Recall is the fraction of class 1 data points processed by the classifier that are correctly labeled as 1. Overall, for each set of training parameters, we have 10 values for Precision and 10 values for Recall. We compute the average and the standard deviation of these values, and use them to choose the best set of parameters.

Results of this experiment are summarized in Figure 1. Each point represents the average values of Precision and Recall corresponding to a set of parameter values. When producing the figure, we eliminated points for which the standard deviation of either Precision or Recall was more than 1/4 of its mean, because these points correspond to experiments with unreliable results.

Figure 1 shows the trade-off between Precision and Recall that can be achieved by varying the learning parameters. We are interested in the set of Pareto optima with respect to these two criteria. Most Pareto optima are obtained with a polynomial kernel, and the remaining with a radial-basis kernel. The linear kernel yields inferior results, implying that the data set is difficult to separate in the original space. There are points with very high Precision ($> 85\%$) but low Recall that represent "conservative" classifiers: very few probing iterations are labeled as 1 (failure), but in that case the classifier is almost always correct. Such a classifier is of limited value for our purpose. We are more interested in the region with roughly 80% Precision and 60% Recall: approximately 60% of the unsuccessful probing iterations in the test set are predicted correctly, while keeping good Precision. These models use a polynomial kernel with degree $d \geq 3$ and $\omega = 1$; additionally, we found that using $\beta = 1, 2$, or 4 seems effective. These models have between 500 and 800 support vectors. The standard deviation of Precision and Recall for all models achieving a Pareto optimum is fairly small, typically less than 2. Therefore, we can assume that the performance of the SVM model does not depend heavily on the particular subset of $\mathcal{D}$ that is used for training.

For comparison, we also fit a simple 2-dimensional Gaussian model. In this model, each class is assumed to be normally distributed, and a 2-dimensional Gaussian model is fit to each class using maximum-likelihood estimation. Then, we classify points in the corresponding test set by computing the probability that they are generated by the two normal distributions and by picking the class that maximizes this probability. Over the 10 training/test sets, the Gaussian model gives a classifier with mean Precision 47.90%, standard deviation 0.91, mean Recall 87.47%, standard deviation 1.31. This performance is comparable to a particular choice of parameters of SVM to obtain high Recall and low Precision. Computational experiments (not reported in detail) demonstrate that using the Gaussian model leads to weaker bound tightening compared to SVM, with no saving of CPU time on average.

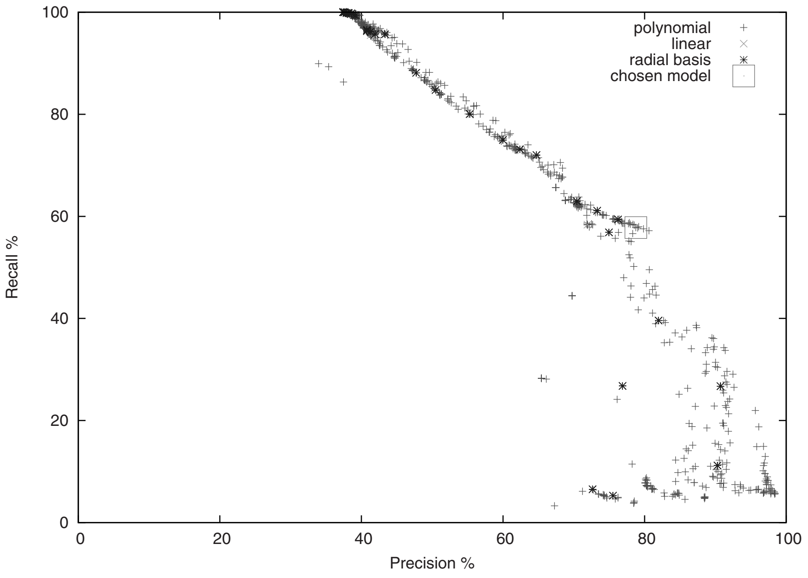


Fig. 1. Average values of Precision and Recall for all tested combinations of training input parameters

Based on these results, we use an SVM model trained with a polynomial kernel of degree 4, $\lambda = 4$, $\beta = 4$, $\omega = 1$ for the experiments in the remainder of this section; this model has 580 support vectors yielding fast classification. Note that 580 is almost half the size of the training set, suggesting that some overfitting might occur. However, the model shows good performance on examples not included in the training set.

5.3 Testing the Probing Algorithm

In this section, we discuss the effect of applying Aggressive Probing on a set of difficult MINLPs. In addition to FBBT, we also use Optimality-Based Bound Tightening (OBBT) [12]. This bound-tightening technique maximizes and minimizes the value of each variable over the convexification computed by `Couenne` at the root node, and uses the optimal values as variable bounds. For each test instance, we first apply FBBT and OBBT. Then, for each variable, we apply Aggressive Probing to tighten both the lower and upper bounds, with a time limit of 60 seconds per variable, and 36 hours per instance. The parameter `max_failures` is set to 10. Variables are processed in the order in which they are stored in `Couenne`. Note that `Couenne` uses a standardized representation of the problem where extra variables, called *auxiliary* variables, are typically added to represent expressions in the original problem formulation [5]. In our experiments, to limit CPU time, OBBT and Aggressive Probing are applied only to original variables; in principle, both can be applied to auxiliary variables without modification.

After Aggressive Probing has been applied to all of the original variables or the global time limit is reached, we record the fraction of tightened variables η, and the average bound reduction ρ, as described in Section 4.1. Then, we apply an additional iteration of FBBT to propagate the new bounds and generate convexification inequalities. This gives a strengthened convexification $\mathcal{C}'$ of $\mathcal{P}$ that is compared to the initial one, $\mathcal{C}$. We record the fraction of variables for which at least one bound could be tightened in $\mathcal{C}'$, as well as the average bound reduction ρ of the tightened variables. Additionally, we compute the percentage of the optimality gap of $\mathcal{C}$ that is closed by $\mathcal{C}'$, i.e., $(z(\mathcal{C}') - z(\mathcal{C}))/(z(\mathcal{P}) - z(\mathcal{C}))$, where $z(\mathcal{C})$ is the optimal objective value of $\mathcal{C}$, and $z(\mathcal{P})$ is the value of the best known solution for the particular instance. The value of $z(\mathcal{P})$ for each instance was obtained from the MINLPLib website.

Results are reported in Table 1. The fraction of tightened variables is relative to the number of original variables for "Probing". For "Probing + FBBT + Conv.", it is relative to the total number of variables, because auxiliary variables can also be tightened after bound propagation through FBBT. The fraction of tightened variables takes into account variables with finite bounds only. Infinite variable bounds are tightened to a finite value only for the three `water` instances, independent of whether SVM is used.

First, we discuss the effect of Aggressive Probing alone. Table 1 shows that the effect of probing is problem-dependent; for example, for `lop97icx`, no variable is tightened by our algorithm, and for `nvs23` and `nvs24`, more than 90% of the variables are tightened. On average, approximately 25% of the original variables are tightened by Aggressive Probing, and after applying FBBT, approximately 30% of the total number of variables (original plus auxiliary) gained tighter bounds. The average bound reduction is close to 50%. The amount of optimality gap closed by adding convexification inequalities after tightening the bound is largely problem dependent as well. The new convexification is much stronger for the `water`, `nvs` and `csched2` instances, but for the remaining instances, the optimality gap is unchanged. This is probably due to the geometry of the initial convexification, for which the LP solution is extremely difficult to cut off without branching, so that no optimality gap is closed by Aggressive Probing. In summary, on all but one test instance, Aggressive Probing is able to provide better variable bounds compared to traditional bound-tightening techniques (FBBT followed by OBBT). This comes at a large computational cost, but may be worth the effort for some difficult instances that cannot be solved otherwise, or when parallel computing resources provide a large amount of CPU power.

Comparing the Aggressive Probing algorithm with and without SVM for failure prediction, we observe on average 30% of computing time saving when using SVM, while the number of tightened variables and average bound tightening is only slightly weaker. CPU time savings are problem dependent: the difference can be huge (`csched2a`, `lop97icx`), or negligible. In only two cases (`nsv24` and `tln6`), using SVM for failure prediction results in an overall longer probing time, but the increase is negligible. Summarizing, using an SVM model to predict likely failures of the Aggressive Probing algorithm leads to CPU

Table 2. Number of successful and failed s-probing iterations recorded by applying Aggressive Probing on the full test set of Table 1

	# s-prob. iter.	
	Success	Failure
Without SVM	1600	18131
With SVM	1634	8998

Table 3. Results on `water` instances with Branch-and-Bound, with and without Aggressive Probing. We report the percentage of optimality gap closed at the end of the optimization process, the number of nodes, and the total CPU time. When Aggressive Probing is used, we additionally report the optimality gap closed by probing at the root (after recomputing the convexification) and the corresponding CPU time.

	Without Aggr. Probing			With Aggr. Probing + SVM				
	Final			Probing Root		Final		
Instance	Gap %	Nodes	Time	Gap %	Time	Gap %	Nodes	Time
`water4`	100.00	1751046	20252.1	50.47	18013.8	100.00	88902	28977.1
`waterx`	30.31	589477	86415.6	34.83	7874.8	67.11	582046	86393.3
`waterz`	77.78	11088079	86399.6	29.35	18617.8	100.00	1033027	45082.8

time savings that depend on the problem instance at hand and are sometimes very large, sometimes moderate, while variable bounds are tightened by almost the same amount.

Table 2 reports the total number of successful and failed s-probing iterations performed over all test instances. The use of an SVM classifier decreases the number of failed s-probing iterations by a factor two, and increases the percentage of successful s-probing iterations from 8% to 15%. These improvements come at essentially no cost.

5.4 Branch-and-Bound After Probing

The main purpose of a bound-tightening technique is to improve the performance of a Branch-and-Bound search. In this section, we report Branch-and-Bound experiments with and without Aggressive Probing on a few selected instances. Table 1 indicates that the probing algorithm proposed in this paper may be effective on the three `water` instances. Therefore, we execute the Branch-and-Bound algorithm of `Couenne` on these instances with a time limit of 24 hours, using the variable bounds obtained after applying FBBT and OBBT at the root node. Then we perform the same experiment using the variable bounds provided by Aggressive Probing with SVM for failure prediction. Results are reported in Table 3, where we include the time spent by probing in the total CPU time.

The `water4` instance is solved with and without Aggressive Probing; Branch-and-Bound without probing is 30% faster, but it explores 20 times as many nodes. Thus, probing is very effective in reducing the size of the enumeration tree. The `waterx` instance remains unsolved after 24 hours. However, employing Aggressive Probing yields a much better lower bound when the time limit is reached (we close an additional 37% of optimality gap). Finally, `waterz` is not solved by Branch-and-Bound unless Aggressive Probing is used. Due to tighter variable bounds, we can solve the instance to optimality in approximately 12 hours, whereas it is unsolved in 24 hours (with 1.2 million active nodes and 23% optimality gap left) if Aggressive Probing is not employed. To the best of our knowledge, an optimality certificate for the solutions to the `water4` and `waterz` has not been provided previously; the optimal solutions of these instances have objective values 910.8821 and 907.0169, respectively.

6 Conclusions

In this paper, we presented a bound-tightening technique for MINLP that uses truncated Branch-and-Bound searches. Computational tests demonstrate that our Aggressive Probing algorithm is able to tighten the variable bounds on many instances, even after other bound-tightening techniques have been applied. Because Aggressive Probing can easily be carried out in parallel, it is well-suited for leveraging parallel environments to solve difficult MINLPs .

The main drawback of the method is its large computational cost. By using an SVM classifier, we predict success or failure of each iteration of Aggressive Probing, where the prediction is based on features of the algorithm's input. Skipping the probing iterations likely to fail saves on average 30% of the computing time. Currently, the prediction is based on only two easy-to-compute features of the input problem, but we plan to extend the model by taking into account more features, such as the maximum allowed time for the probing iteration, and a measure of its effectiveness from previous iterations.

Computational experiments demonstrate that, on some instances, the overall Branch-and-Bound search is improved using our bound-tightening algorithm. In particular, we are able to solve a difficult MINLPLib instance, `waterz`, for the first time, and we obtain better lower bounds (or a smaller enumeration tree) for other instances.

References

1. Biegler, L., Grossmann, I., Westerberg, A.: Systematic Methods of Chemical Process Design. Prentice-Hall, Upper Saddle River (NJ) (1997)
2. Floudas, C.: Global optimization in design and control of chemical process systems. Journal of Process Control 10, 125–134 (2001)
3. Tawarmalani, M., Sahinidis, N.: Exact algorithms for global optimization of mixed-integer nonlinear programs. In: Pardalos, P., Romeijn, H. (eds.) Handbook of Global Optimization, vol. 2, pp. 65–86. Kluwer Academic Publishers, Dordrecht (2002)
4. Tawarmalani, M., Sahinidis, N.: Global optimization of mixed integer nonlinear programs: A theoretical and computational study. Mathematical Programming 99, 563–591 (2004)
5. Belotti, P., Lee, J., Liberti, L., Margot, F., Wächter, A.: Branching and bounds tightening techniques for non-convex MINLP. Optimization Methods and Software 24(4-5), 597–634 (2008)

6. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. ORSA Journal on Computing 6(4), 445–455 (1994)
7. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning 20, 273–297 (1995)
8. Hutter, F., Hoos, H., Leyton-Brown, K.: Automated configuration of mixed integer programming solvers. In: Lodi, A., Milano, M., Toth, P. (eds.) CPAIOR 2010. LNCS, vol. 6140, pp. 186–202. Springer, Heidelberg (2010)
9. Markót, M.C., Schichl, H.: Comparison and automated selection of local optimization solvers for interval global optimization methods. Technical report, Faculty of Mathematics, University of Vienna
10. Sahinidis, N.: Baron: Branch and reduce optimization navigator, user's manual, version 4.0 (1999), `http://archimedes.scs.uiuc.edu/baron/manuse.pdf`
11. Belotti, P.: Couenne: a user's manual. Technical report, Lehigh University (2009)
12. Shectman, J., Sahinidis, N.: A finite algorithm for global minimization of separable concave programs. Journal of Global Optimization 12, 1–36 (1998)
13. Smith, E.: On the Optimal Design of Continuous Processes. PhD thesis, Imperial College of Science, Technology and Medicine, University of London (October 1996)
14. Cristianini, N., Shawe-Taylor, J.: An Introduction to Support Vector Machines and Other Kernel-based Learning Methods. Cambridge University Press, Cambridge (2000)
15. Schölkopf, B., Smola, A.J.: Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge (2002)
16. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Operations Research Letters 33(1), 42–54 (2005)
17. Chang, C.C., Lin, C.J.: LIBSVM: a library for support vector machines (2001), Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`
18. Bussieck, M.R., Drud, A.S., Meeraus, A.: MINLPLib — a collection of test models for Mixed-Integer Nonlinear Programming. INFORMS Journal on Computing 15(1) (2003)
19. CMU-IBM: Cyber-Infrastructure for MINLP, `http://www.minlp.org`
20. Koggalage, R., Halgamuge, S.: Reducing the number of training samples for fast support vector machine classification. Neural Information Processing - Letters 2(3), 57–65 (2004)

Recovering Indirect Solution Densities for Counting-Based Branching Heuristics

Gilles Pesant[1] and Alessandro Zanarini[2]

[1] École Polytechnique de Montréal, Canada
gilles.pesant@polymtl.ca
[2] Dynadec Europe, Belgium
alessandro.zanarini@dynadec.com

Abstract. Counting-based branching heuristics in CP have been very successful on a number of problems. Among the few remaining hurdles limiting their general applicability are the integration of counting information from auxiliary variables and the ability to handle combinatorial optimization problems. This paper proposes to answer these challenges by generalizing existing solution counting algorithms for constraints and by relaying counting information to the main branching variables through augmented element constraints. It offers more easily comparable solution counting information on variables and stronger back-propagation from the objective function in optimization problems. We provide supporting experimental results for the Capacitated Facility Location Problem.

1 Introduction

The recent development of generic branching heuristics based on counting the number of solutions of individual constraints has led to state-of-the-art results on several benchmark problems [5]. They are indeed generic in principle and even adapt to a problem since they are derived from the constraints present in the model. However in practice they are restricted to problems modeled using constraints for which solution counting algorithms have been designed, a limitation that is being pushed back as new algorithms are introduced. They are also oriented toward solving feasibility problems and not so much optimization problems. Finally a technical incovenience is that some constraints are not expressed on the main decision variables but instead on auxiliary variables dependent on the former: these different sets of variables will each have their own solution counting information, which will be difficult to mix or compare.

This paper proposes to answer the latter two challenges by generalizing existing solution counting algorithms for constraints and by relaying counting information to the main decision variables through augmented element constraints. Section 2 presents the technical contribution. Section 3 discusses its use for combinatorial optimization problems. Section 4 gives a first experimental evaluation of the ideas through the Capacitated Facility Location Problem.

2 Framework

The most interesting indicator arising from solution counting information has been the *solution density* of a variable-value assignment, denoted $\sigma(X, a, \gamma)$, that is the proportion of solutions to a given constraint γ that assign value a to variable X. One of the simplest and most effective branching heuristics built from this is maxSD, which branches on the assignment with the overall highest solution density (among all constraints) [5]. This section describes the adaptations to the existing counting framework in order to recover indirect solution densities from constraints on auxiliary variables.

2.1 Channelling Solution Densities through `element` Constraints

An `element`(X, f, Y) constraint, with f a fixed array of integers, states the functional relationship $Y = f(X)$, which maps every value in the domain of finite-domain variable X to a value in the domain of finite-domain variable Y. Its inverse, f^{-1}, is potentially a one-to-many mapping. Therefore a constraint expressed on Y will exhibit a solution density for some assignment $Y = b$ which, when transferred to X, should be "shared" between many values.

We define the *multiplicity* of a value $b \in D(Y)$ as $\mu(b) = |\{a \in X \mid f(a) = b\}|$. The implementation of an `element` constraint is augmented with these multiplicities and they are updated whenever the domain of X changes. Given branching variable X_i, we wish to compute the solution density of assignment $X_i = a$ with respect to some constraint $\gamma(Y_1, \ldots, Y_m)$ in whose scope X_i does not appear but to which it is connected through an `element`(X_i, f, Y_i) constraint:

$$\sigma(X_i, a, \gamma) = \frac{\sigma(Y_i, f(a), \gamma)}{\mu(f(a))}$$

Clearly $\sum_{a \in D(X_i)} \sigma(X_i, a, \gamma) = 1$ since the domain of Y_i is the image of $f(X_i)$ (i.e. the corresponding `element` is enforced) and $\sum_{b \in D(Y_i)} \sigma(Y_i, b, \gamma) = 1$. Also $\sigma(X_i, -, \gamma)$ is nonnegative so it does define a solution density function.

2.2 Generalizing the Counting Algorithm for `knapsack` Constraints

The solution counting algorithm of every family of constraints must be generalized to take into account multiplicities for values. We outline the adaptation for `knapsack` constraints, because these are used in the experimental section. That adaptation will also trivially apply to `regular` constraints since they maintain very similar data structures.

The domain consistency algorithm for `knapsack` builds and maintains a layered digraph that provides a compact encoding of its set of solutions: a path from the first to the last layer spells out a complete satisfying assignment of the variables in its scope, each of its arcs representing an individual assignment [3]. The solution density algorithm counts the number of paths with a certain property, e.g. assigning value a to variable X, through simple recursive formulas [2]. We generalize the algorithm by introducing multiplicities and adding them as labels on every arc in the graph. When counting paths, they are used as a multiplicative factor. For example, the formula for the number of incoming paths from the initial layer to vertex b in the i^{th} layer, $ip(i,b)$, is generalized as follows:

$$\begin{aligned} ip(0,0) &= 1 \\ ip(i,b) &= \sum_{(v_{i-1,b'},v_{i,b})\in Arcs} \mu((v_{i-1,b'},v_{i,b})) \times ip(i-1,b'), \quad 1 \le i \le m \end{aligned}$$

3 Stronger Back-Propagation from the Cost Variable

CP typically solves combinatorial optimization problems as a succession of feasibility problems with increasingly tighter bounds on the objective. Many optimization problems have a linear objective function. When such an objective is expressed as a weighted sum constraint in CP, back-propagation from the cost variable (or the bound on cost) to the decision variables is notoriously weak because inexpensive bounds consistency is typically enforced. Using a `knapsack` constraint can strengthen back-propagation but potentially at a prohibitive computational cost since its propagation algorithm is pseudo-polynomial. To make matters worse, individual cost values are often indexed by decision variables instead of being multiplied by them. For example an objective function such as $\sum_{ij} c_{ij}X_{ij}$ with 0-1 variables will instead be expressed as $\sum_i c_{i,S_i}$ with finite domain variables S_i such that $X_{ij} = 1 \Leftrightarrow S_i = j$. In practice one states `element` constraints linking each S_i to an auxiliary variable which represents its individual cost, the latter variable appearing in the sum constraint stating the objective.

The approach described in the previous section provides the means to handle combinatorial optimization problems more directly with counting-based heuristics. Solution densities from the "objective" constraint will be expressed directly on the main decision variables and put in the mix with other solution densities coming from feasibility considerations. They indicate which assignments often appear in "good enough" solutions, according to that constraint on the objective value.

4 Experiments: Capacitated Facility Location

We ran experiments on the proposed approach using the Capacitated Facility Location Problem (CSPLib problem 34). We are given a set of locations where we may open a facility, each with a maximum capacity c_i, to serve a set of customers, each with a demand d_j. The problem consists of assigning each customer to a single facility while respecting the capacity constraints and minimizing the total cost of the assignment. That cost is the sum of relevant service costs s_{ij}, defined on location-customer pairs and of set-up costs o_i for each open facility (i.e. a location with at least one customer assigned to it). We frame it as a feasibility problem by adding a constraint upper bounding that cost. This corresponds to a single step in the usual CP approach to optimization.

We use a set of 12 relatively small (10 facilities; 50 customers) instances from Holmberg [1]. As for branching heuristics, we compared maxSD to some of the best known generic heuristics in CP (dom/ddeg and IBM-Ilog Solver 6.3 IBS[1]).

$$\begin{aligned}
\texttt{knapsack}((X_{ij})_{1\le i\le m}, \mathbf{1}, 1, 1) \qquad & 1 \le j \le n && (1)\\
\texttt{knapsack}((X_{ij})_{1\le j\le n}, (d_j)_{1\le j\le n}, 0, c_i) \qquad & 1 \le i \le m && (2)\\
O_i \le \sum_{1\le j\le n} X_{ij} \le nO_i \qquad & 1 \le i \le m && (3)\\
\sum_{1\le i\le m, 1\le j\le n} s_{ij}X_{ij} + \sum_{1\le i\le m} o_iO_i \le \beta \qquad & && (4)\\
X_{ij} \in \{0,1\} \qquad & 1 \le i \le m, \quad 1 \le j \le n && (5)\\
O_i \in \{0,1\} \qquad & 1 \le i \le m && (6)
\end{aligned}$$

Consider a general instance with n customers and m locations for the facilities. The usual 0-1 model (1)-(6) does not behave well in CP (binary variables are usually avoided, with good reason) and even though it features knapsack constraints, only the packing constraints (2) provide non trivial solution densities to be used by counting-based heuristics. On this model, none of the branching heuristics produced any solution within the one-hour time limit.

$$\begin{aligned}
\texttt{alldiff}(C_{ij}) \qquad & && (7)\\
\texttt{knapsack}((D_{ij})_{1\le j\le k}, \mathbf{1}, 0, c_i) \qquad & 1 \le i \le m && (8)\\
\texttt{element}(C_{ij}, (d_\ell)_{1\le \ell\le mk}, D_{ij}) \qquad & 1 \le i \le m, \quad 1 \le j \le k && (9)\\
\texttt{element}(C_{ij}, (s_{i\ell})_{1\le \ell\le mk}, S_{ij}) \qquad & 1 \le i \le m, \quad 1 \le j \le k && (10)\\
C_{ij} < C_{i,j+1} \qquad & 1 \le i \le m, \quad 1 \le j \le k-1 && (11)\\
(O_i = 1) \Leftrightarrow (C_{i1} \le n) \qquad & 1 \le i \le m && (12)\\
\sum_{1\le i\le m, 1\le j\le k} S_{ij} + \sum_{1\le i\le m} o_iO_i \le \beta \qquad & && (13)\\
C_{ij} \in \{0, 1, \ldots, mk\} \qquad & 1 \le i \le m, \quad 1 \le j \le k && (14)\\
D_{ij} \in \{d_{j'} \mid {\scriptstyle 1\le j'\le n}\} \cup \{0\} \qquad & 1 \le i \le m, \quad 1 \le j \le k && (15)\\
S_{ij} \in \{s_{i'j'} \mid {\scriptstyle 1\le i'\le m, 1\le j'\le n}\} \cup \{0\} \qquad & 1 \le i \le m, \quad 1 \le j \le k && (16)\\
O_i \in \{0,1\} \qquad & 1 \le i \le m && (17)
\end{aligned}$$

A facility-centered model that assigns customers to k predefined slots for each location works much better, at least for counting-based heuristics. Since in general we define more slots than there are customers, we add dummy customers $n+1, n+2, \ldots, mk$ (in our experiments we use $k = 6$). Customer demands and service costs are correspondingly extended with 0's. Note that for instances

[1] Impacts are fully initialized at the root node, approximated impacts are used to preselect a subset of 5 variables, node impacts are computed on that subset and further ties are broken randomly.

Table 1. Performance of maxSD given different sources of solution densities

source of	time(s)		backtracks		obj value
solution density	avg	median	avg	median	avg
C_{ij}	55.6	27.6	40744.5	19680	16354
$C_{ij} + D_{ij}$	34.9	9.2	19191.3	5749	16354
C_{ij} + eltSD; 1.2	17.7	7.7	1243.9	4	14309
C_{ij} + eltSD; 1.0	22.7	5.8	2323.6	5	12856

whose optimal solutions open relatively few locations, such a model would needlessly create too many slots. Model (7)-(17) features an `alldiff` constraint spanning the main decision variables C_{ij} and `knapsack` packing constraints for each facility. However these latter ones are expressed on auxiliary variables D_{ij}. Constraints (11) break customer symmetries at a facility, (12) link the O_i variables, and (13) states a bound on the objective value.

Heuristic dom/ddeg still does not produce any solution within the one-hour time limit while IBS finds a solution to 4 out of 12 instances. Randomized restarts [4] help IBS solve more instances, though not all of them. maxSD performs much better, as indicated in Table 1 (and here randomized restarts generally worsened performance). In all four variants, the upper bound on cost was set to 1.8 times the optimal value. The first two do not use augmented `element` constraints (the second variant branches on both C_{ij} and D_{ij} variables) whereas the last two do use them, both for the knapsack packing constraints and for the service costs. For these service costs, individual knapsack constraints were stated for each facility and the upper bound was set to 1.2 (third variant) and 1.0 (fourth variant) times the optimal value divided by m, the number of facilities. We observe that the additional guidance provided by the indirect solution densities does decrease search effort and also contributes to improve the cost of the solutions found. Note that lowering the shared 1.8 bound or adding a bound per facility as above significantly increased search effort for the first two variants and caused several instances to time out.

5 Conclusion

In this paper, we proposed two generalizations of counting algorithms to overcome the limit of having solution density branching information on auxiliary variables. These algorithms applied to a benchmark problem showed promising results, underlining once more the efficiency of counting-based heuristics. Among the possible improvements, the approximated algorithm for computing the solution densities of the `knapsack` constraint [2] can help whenever the upper bound on the knapsack is quite large. We plan to extend this approach to other constraints such as `alldifferent` or `global-cardinality`; a possible avenue would be to use the multiplicities as entries of the adjacency matrix and compute the solution density with permanent upper bounds for generic nonnegative matrices [6].

References

1. Holmberg, K., Ronnqvist, M., Yuan, D.: An exact algorithm for the capacitated facility location problems with single sourcing. European Journal of Operational Research 113(3), 544–559 (1999)
2. Pesant, G., Quimper, C.-G.: Counting Solutions of Knapsack Constraints. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 203–217. Springer, Heidelberg (2008)
3. Trick, M.A.: A dynamic programming approach for consistency and propagation for knapsack constraints. Annals of Operations Research 118, 73–84 (2003)
4. Walsh, T.: Search in a small world. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 1999, pp. 1172–1177. Morgan Kaufmann Publishers Inc., San Francisco (1999)
5. Zanarini, A., Pesant, G.: Solution Counting Algorithms for Constraint-Centered Search Heuristics. Constraints 14, 392–413 (2009)
6. Zanarini, A., Pesant, G.: More Robust Counting-Based Search Heuristics with Alldifferent Constraints. In: Lodi, A., Milano, M., Toth, P. (eds.) CPAIOR 2010. LNCS, vol. 6140, pp. 354–368. Springer, Heidelberg (2010)

Using Hard Constraints for Representing Soft Constraints

Jean-Charles Régin

Université de Nice - Sophia Antipolis, I3S - CNRS
930 Route des Colles - BP 145
06903 Sophia Antipolis Cedex, France
jcregin@gmail.com

Abstract. Most of the current algorithms dedicated to the resolution of over-constrained problems, as PFC-MRDAC, are based on the search for a support for each value of each variable taken independently. The structure of soft constraints is only used to speed-up such a search, but not to globally deduce the existence or the absence of support. These algorithms do not use the filtering algorithms associated with the soft constraints.

In this paper we present a new schema where a soft constraint is represented by a hard constraint in order to automatically benefit from the pruning performance of the filtering algorithm associated with this constraint and from the incremental aspect of these filtering algorithms. In other words, thanks to this schema every filtering algorithm associated with a constraint can still be used when the constraint is soft. The PFC-MRDAC (via the Satisfiability Sum constraint) algorithm and the search for disjoint conflict sets are then adapted to this new schema.

1 Introduction

A constraint network (CN) consists of a set of variables, each of them associated with a domain of possible values, and a set of constraints linking the variables and defining the set of allowed combinations of values. The search for an assignment of values to all variables that satisfies all the constraints is called the Constraint Satisfaction Problem (CSP). Such an assignment is a solution of the CSP.

Unfortunately, the CSP is an NP-Hard problem. Thus, many works have been carried out in order to try to reduce the time needed to solve a CSP. Some of the suggested methods turn the original CSP into a new one, which has the same set of solutions, but which is easier to solve. The modifications are done through filtering algorithms, which remove from domains values that cannot belong to any solution of the current CSP. If the cost of such an algorithm is less than the time required by the backtrack algorithm to discover many times the same inconsistency, then the solving will be accelerated.

It often happens that a CSP has no solution. In this case we say that the problem is over-constrained, and the goal is then to find a good compromise. One of the most usual theoretical frameworks is called the Maximal Constraint Satisfaction Problem (Max-CSP). A solution of a Max-CSP is a total assignment that minimizes the number of constraint violations.

Almost all existing techniques for solving Max-CSP, as PFC-MRDAC [2], consider that the filtering algorithm associated with a constraint can be used only to speed up the search for, or the proof of absence of, a support. Since the constraints are soft (i.e. they can be violated) it is considered that it is not possible to directly use the filtering algorithm associated . Only two existing approaches exploit the structure of the constraints: the search for conflict sets (i.e. a set of soft constraints which leads to a failure if all these constraints are considered hard) [5] and the design of specific filtering algorithms for global soft constraints [3], for instance the alldiff constraint. Our goal is to explain how a filtering algorithm associated with a constraint can be used in an efficient way even if the constraint can be violated.

In this paper, we propose a general schema, called S2H (Soft to Hard), which is able to use every filtering algorithm associated with a constraint when the constraint is soft. The originality of our approach is to represent each soft constraint by a hard constraint and to manage the detection of failures. Each hard constraint is defined on new variables that are linked to the variables involved in a soft constraint by a specific hard constraint. This specific hard constraint will take into account the possible failure of the hard constraint representing the soft one. This approach has two advantages: first it can be easily used by any constraint programming solver system provided that the failure can be caught, second we immediately and automatically benefit from the pruning performance of the filtering algorithm associated with the soft constraint, because this is managed by the solver. Moreover, we will show how we can benefit from the incremental aspect of the filtering algorithms. The main interest of this approach is that it could lead to an improvement of the resolution of real world applications involving soft constraints similar as the improvement obtained with solvers when the filtering algorithms associated with constraints have been introduced.

Furthermore, we give two possible instantiations of this schema corresponding to two different filtering algorithms that have been proposed to improve the resolution of over-constrained problems: the partition based filtering and the conflict sets based filtering.

This paper is organized as follows. First we recall some notions about constraint programming, and the principle of the filtering algorithms associated with the Satisfiability sum constraint that deals with the soft constraints of a problem. Then, we emphasize on an example the usefulness of the structure of a constraint. Next, the S2H-Schema is presented in details. We explain how the S2H-Schema can be instantiated in order to efficiently implement the best filtering algorithm for over constrained problems. At last, we discuss about the implementation of the S2H-Schema in a solver.

2 Background

2.1 Constraint Network

A *constraint network* $\mathcal{N}$ is defined as a set of n *variables* $X = \{x_1, \ldots, x_n\}$, a set of *domains* $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible *values* for variable x_i, and a set $\mathcal{C}$ of *constraints* between variables. A *constraint* C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ that specifies the *allowed* combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D(x_{i_1}) \times \cdots \times D(x_{i_r})$ is called a *tuple on* $X(C)$. $|X(C)|$ is the *arity* of C. A value a for a variable x is often denoted by (x, a). A tuple τ on $X(C)$ is *valid* if $\forall (x, a) \in \tau, a \in D(x)$. C is *consistent* iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is *consistent with* C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ in which a is the value assigned to x.

2.2 Satisfiability Sum Constraint

Max-CSP can be expressed by a satisfiability sum constraint [5]:

Definition 1. *Let $\mathcal{C} = \{C_i, i \in \{1, \ldots, m\}\}$ be a set of constraints, and $S[\mathcal{C}] = \{s_i, i \in \{1, \ldots, m\}\}$ be a set of variables and unsat be a variable, such that a one-to-one mapping is defined between $\mathcal{C}$ and $S[\mathcal{C}]$. A* **Satisfiability Sum Constraint** *is the constraint $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ defined by:*

$$[unsat = \sum_{i=1}^{m} s_i] \wedge \bigwedge_{i=1}^{m} [(C_i \wedge (s_i = 0)) \vee (\neg C_i \wedge (s_i = 1))]$$

Notation 1. *Given a $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$, a variable x, a value $a \in D(x)$ and $\mathcal{K} \subseteq \mathcal{C}$:*
- *$max(D(unsat))$ is the highest value of current domain of unsat;*
- *$min(D(unsat))$ is the lowest value of current domain of unsat;*
- *$minUnsat(\mathcal{C}, S[\mathcal{C}])$ is the minimum value of unsat consistent with $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$;*
- *$S[\mathcal{K}]$ is the subset of $S[\mathcal{C}]$ equal to the projection of variables $S[\mathcal{C}]$ on $\mathcal{K}$;*
- *$X(\mathcal{C})$ is the union of $X(C_i), C_i \in \mathcal{C}$.*

The variables $S[\mathcal{C}]$ are used in order to express which constraints of $\mathcal{C}$ must be violated or satisfied: value 0 assigned to $s \in S[\mathcal{C}]$ expresses that its corresponding constraint C is satisfied, whereas 1 expresses that C is violated[1]. Variable *unsat* represents the objective, that is, the number of violations in $\mathcal{C}$, equal to the number of variables of $S[\mathcal{C}]$ whose value is 1. Note that no hypothesis is made on the arity of constraints $\mathcal{C}$. And if a value is assigned to $s_i \in S[\mathcal{C}]$, then a filtering algorithm associated with $C_i \in \mathcal{C}$ (resp. $\neg C_i$) can be used in a way similar to classical CSPs. Similarly if all values of a variable x are not consistent with C_i (resp. $\neg C_i$) then $s_i = 1$ (resp. 0).

Throughout this formulation, a solution of a Max-CSP is an assignment that satisfies the *ssc* with the minimal possible value of *unsat*. A lower bound of the objective of a Max-CSP corresponds to a necessary consistency condition of the ssc.
From the definition of $minUnsat(\mathcal{C}, S[\mathcal{C}])$ we have:

Property 1. *If $minUnsat(\mathcal{C}, S[\mathcal{C}]) > max(D(unsat))$ then $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ is not consistent.*

[1] An extension of the model can be performed [4], in order to deal with Valued CSPs [1]. Basically it consists of defining larger domains for variables in $S[\mathcal{C}]$.

Thus, any lower bound of $minUnsat(\mathcal{C}, S[\mathcal{C}])$ provides a necessary condition of consistency of a ssc.
The different domain reduction algorithms established for Max-CSP correspond to specific filtering algorithms associated with the ssc.

2.3 Ssc: Partition Based Filtering

A possible way for computing a lower bound is to perform a sum of independent lower bounds of violations, one per variable (See [2].) For each variable a lower bound can be defined by:

Definition 2. *Given x a variable, a a value of $D(x)$, $\mathcal{K}$ a set of constraints of $\mathcal{C}$, $\#inc((x,a),\mathcal{K}) = |\{C \in \mathcal{K}$ s.t. (x,a) is not consistent with $C\}|$.
$\#inc(x,\mathcal{K}) = min_{a \in D(x)}(\#inc((x,a),\mathcal{K}))$.*

The sum of these minima with $\mathcal{K} = \mathcal{C}$ cannot lead to a lower bound of the total number of violations, because some constraints can be taken into account more than once[2]. In this case, the lower bound can be overestimated, and an inconsistency could be detected while the *ssc* is consistent. Consequently, for each variable, an independent set of constraints must be considered.
Such a result is obtained by associating with each constraint C one and only one variable x involved in the constraint: C is then taken into account only for computing the $\#inc$ counter of x. Therefore, the constraints are *partitioned* w.r.t the variables that are associated with:

Definition 3. *Given a set of constraints $\mathcal{C}$, a* **var-partition** *of $\mathcal{C}$ is a partition $\mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$ of $\mathcal{C}$ in $|X(\mathcal{C})|$ sets such that $\forall P(x_i) \in \mathcal{P}(\mathcal{C}) : \forall C \in P(x_i), x_i \in X(C)$.*

Given a var partition $\mathcal{P}(\mathcal{C})$, the sum of all $\#inc(x_i, P(x_i))$ is a lower bound of the total number of violations, because all sets belonging to $\mathcal{P}(\mathcal{C})$ are disjoint; thus we have:

Definition 4. *$\forall \mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$,
$LB(\mathcal{P}(\mathcal{C})) = \sum_{x_i \in X(\mathcal{C})} \#inc(x_i, P(x_i))$.*

Property 2. *$\forall \mathcal{P}(\mathcal{C}) = \{P(x_1), ..., P(x_k)\}$, If $LB(\mathcal{P}(\mathcal{C})) > max(D(unsat))$ then $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ is not consistent.*

The quality of such a lower bound depends on the var-partition that is chosen.
The lower bound of Property 2 can also be used to detect some inconsistent values of a variable x:

Theorem 1. *$\forall \mathcal{P}(\mathcal{C})$ a var-partition of $\mathcal{C}, \forall x \in X(\mathcal{C}), \forall a \in D(x)$, if $\#inc((x,a), P(x)) + LB(\mathcal{P}(\mathcal{C} - P(x))) > max(D(unsat))$ then a can be removed from its domain.*

[2] For instance, given a constraint C and two variables x and y involved in C, C can be counted in $\#inc(x,\mathcal{C})$ and also in $\#inc(y,\mathcal{C})$.

2.4 Ssc: Conflict Set Based Filtering

Some inconsistencies are not taken into account by the previous filtering algorithm because it is based on counters of direct violations of constraints by values. Therefore another filtering algorithm based on successive computations of disjoint conflict sets were proposed in [5].

Definition 5. *A conflict set is a subset $\mathcal{K}$ of $\mathcal{C}$ which satisfies:* $minUnsat(\mathcal{K}, S[\mathcal{K}]) > 0$.

A conflict set leads to at least one violation in $\mathcal{C}$. Consequently, if there are q disjoint conflict sets of $\mathcal{C}$ then q is a lower bound of $minUnsat(\mathcal{C}, S[\mathcal{C}])$. They must be disjoint to guarantee that all violations are independent.

Property 3. *Let $\mathcal{Q}$ be a set of disjoint conflict sets of $\mathcal{C}$. If $|\mathcal{Q}| > max(D(unsat))$ then $ssc(\mathcal{C}, S[\mathcal{C}], unsat)$ is not consistent.*

3 The S2H Schema

The main issue of algorithms dedicated to the resolution of over constrained problems is the necessity to detect if a given value is consistent with a constraint. Indeed, it is necessary to know which values are violated by a soft constraint, for instance to update $\#inc$ counters.

In existing solvers, each constraint is associated with a filtering algorithm, which is able to remove some values that are not consistent with the constraint, and to perform this operation only when an event, which can lead to some removal, arises. Using the filtering algorithm to detect some inconsistent values is an efficient way, better than systematically and individually check for consistency. Moreover, without loss ogf generality we can consider that any CP solver (which is programmable) provides:

- a way to automatically notify a variable that one of its values has been removed when applying a filtering algorithm
- a way to call some filtering algorithms when some events on the domain of variables happen.

Thus, if a constraint is considered as hard, then we can use the solver in order to update data structures only when specific values are removed. Therefore, if we represent soft constraints by hard constraints we will be able to benefit from all these mechanisms.

Currently, solvers do not use complex mechanisms for dealing with soft constraints. They are usually limited to basic behaviors. Disjunctive constraints is a good example. Gecode, Comet or ILOG Solver do not implement the constructive disjunction [6]. They mainly implement disjunctive constraints by checking whether each part of the disjunction is satisfied or not. Consider for instance, the constraint (C_1 or C_2). The filtering algorithms associated with each constraint are not used. The solver just checks if the constraints C_1 or C_2 are violated. There is no filtering code which is used, so there is no need to catch some possible failures because the mechanism does not call any internal code that could fail. Such a mechanism is clearly weak and insufficient for implementing the constraints we mentioned in Section 2, because we want to use the existing algorithms associated with constraints even if there are soft. The constructive disjunction is complex to implement this is why it is rarely implemented. For some cases, it is possible to implement it in a specific way, because we have only one constraint that could fail. However, the problem we consider is much more general than constructive disjunction and we need a more general mechanism

The representation of soft constraints by hard constraints is not an easy task because a soft constraint should not necessarily be satisfied in all solutions whereas a hard constraint should have to. Thus, if we want to represent a soft constraint by a hard constraint, then we are faced to the following two main problems:

– The deductions made by a filtering algorithm are not necessary valid because the constraint is not obligatorily satisfied. Hence, these modifications cannot be effective on the variables on which the soft constraint is defined.
– The hard constraint corresponding to a soft constraint can fail and this failure is not a reason to backtrack.

The S2H (Soft to Hard) Schema deals with these problems. Consider a set $\mathcal{S}$ of soft constraints. Roughly, the principle of this schema is to copy the variables involved in constraints of $\mathcal{S}$ and then to add to the solver as hard constraints the constraints of $\mathcal{S}$ defined on the copied variables. Then some mechanisms are added in order to be able to:

- update the copied variables when the original variables are modified,
- use the modifications which occur on the domains of the copied variables
- catch some possible failure of a constraint of $\mathcal{S}$.

If such a failure happens, then the S2H schema is able to remove all the hard constraints corresponding to the soft ones that have been added and to continue the search as if these hard constraints have never been added.

The S2H schema is based on 2 operations:

1. creation: the hardening of soft constraints operation is applied
2. catch of a failure and deletion: if certain constraints fail then the solver does not consider that a global inconsistent state is detected. Then, some hard constraints must be removed from the solver in order to continue the search.

3.1 Hardening of Soft Constraints

Definition 6. *Given $\mathcal{S}$ a set of soft constraints involving the variables $X(\mathcal{S}) = \{x_1, ..., x_k\}$, $SoftManager$ a manager of soft constraints associated with 2 notification methods:* whenDomainReduction, whenFail. *The hardening of soft constraints is the operation that consists in:*

1. *for each variable $x \in X(\mathcal{S})$ creating a new variable $dcopy(x,\mathcal{S})$, called the directed copy of x.*
2. *for each constraint $C \in \mathcal{S}$ adding to the problem as hard constraint the constraint $hard(C)$ which is the constraint C defined on the directed copies of the variables of $X(C)$. $Hard(\mathcal{S})$ denotes the set of these constraints.*
3. *adding between each pair of variables $\{x, dcopy(x,\mathcal{S})\}$ a constraint* varToCopiedVarCt$(x, dcopy(x,\mathcal{S}))$ *stating that $D(x) \supseteq D(dcopy(x,\mathcal{S}))$. $DcopyCt(\mathcal{S})$ denotes the set of these constraints.*
4. *for each variable $dcopy(x,\mathcal{S})$ linking the notification method* whenDomainReduction *to $dcopy(x,\mathcal{S})$. This method is called each time the domain of $dcopy(x,\mathcal{S})$ is modified by a constraint $hard(C)$ of $Hard(\mathcal{S})$ and notifies $SoftManager$ of the modifications. The parameters of this method are $SoftManager$,x,a and the constraint C.*
5. *for each constraint $C_H \in Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ linking the notification method* whenFail *to C_H. This method is called when the constraint C_H fails. The parameters of this method are $SoftManager$, $\mathcal{S}$.*

An example of the application of the Hardening of soft constraints is given by Figure 1.

The constraint varToCopiedVarCt$(x, dcopy(x,\mathcal{S}))$ ensures that, when x is modified, $dcopy(x,\mathcal{S})$ is accordingly modified. When a directed copy $dcopy(x,\mathcal{S})$ is modified we cannot modify x and we use notification mechanism. The notification methods are used to update some data structures required to efficiently implement the filtering algorithms associated with the Satisfiability sum constraint. These data structures are encapsulated in the manager of soft constraints.

More precisely, each time a value a is removed from the domain of $dcopy(x,\mathcal{S})$, a directed copy of x, whenDomainReduction is called. It notifies the manager of soft constraints that the constraint C is violated by (x, a). Thus, this method establishes a link from the directed copy of variable to the variable. Since the hardening of a soft constraint is a hard constraint we benefit from its pruning performance and from its incremental mechanism of triggering of the filtering algorithm associated with it. Therefore, there is no need to ask for each value the constraints it violates. This result is automatically obtained by using the notification method.

Similarly, if a constraint in $Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ fails, then whenFail is called. This method notifies the manager of soft constraints that a constraint in $\mathcal{S}$ will be violated if $\mathcal{S}$ is considered as a set of hard constraints. In this case, some operations have to be done, because a failure is detected which is more complex than the deletion of one value of a domain.

3.2 Catch of the Failure and Deletion

Our problem is to manage a possible failure of the hard representation of a soft constraint, and to be able to continue the search as if these constraints had never been added. Consider $\mathcal{S}$ a set of constraints and suppose that the hardening operation has been applied on $\mathcal{S}$. Then, the solver must be able to perform two operations during the search for solutions:

Consider two variables x and y with $D(x) = [0..10]$ and $D(y) = [0..9]$, three hard constraints $(x > 5)$, $(y < 5)$ and $(x < 10)$, one soft constraint $(x < y)$, and $softManager$ a manager of soft constraints.

Suppose that WHENDOMAINREDUCTION prints the domain of the copied variable and that WHENFAIL prints the constraints that are defined soft; and that $SoftManager.addSoft(x < y)$ involves the hardening of the soft constraint $(x < y)$. Then, we can trace the behavior of the following pseudo-code:

Define x with $D(x) = [0..10]$ and y with $D(y) = [0..9]$
$SoftManager.addSoft(x < y)$

> **begin trace**: create x' with $D(x') = [0..10]$; y' with $D(y') = [0..9]$
> add VARTOCOPIEDVARCT(x, x') and VARTOCOPIEDVARCT(y, y')
> add $(x' < y')$; this constraint modifies the domains of x' and y'
> x' is modified then WHENDOMAINREDUCTION is called and prints $D(x') = [0..8]$; y' is modified then WHENDOMAINREDUCTION is called and prints $D(x') = [1..9]$. **end trace**

add$(x > 5)$

> **begin trace**: $D(x) = [6..10]$
> constraint VARTOCOPIEDVARCT(x, x') is triggered and $D(x') = [6..8]$; function WHENDOMAINREDUCTION is called and prints $D(x') = [6..8]$; constraint $(x' < y')$ is triggered and y' is modified; function WHENDOMAINREDUCTION is called and prints $D(y') = [7..9]$. **end trace**

add$(y < 5)$

> **begin trace**: $D(y) = [0..4]$
> constraint VARTOCOPIEDVARCT(y, y') is triggered and fails; function WHENFAIL is called and prints $(x < y)$; the constraints VARTOCOPIEDVARCT(x, x'), VARTOCOPIEDVARCT(y, y') and $(x' < y')$ are removed. **end trace**

add$(x < 10)$

> **begin trace**: $D(x) = [6..9]$; There is no other constraints to trigger and the program continues normally. **end trace**

Fig. 1. an Example of hardening of soft constraints

– to catch the failure of any constraint of $Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$. That is, a failure of any constraint of $Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ must not be considered as a global detection of an inconsistency.
– to delete the set of constraints $Hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ when one of them fails.

These operations can be implemented in different ways. This is the purpose of section 5.

4 Instantiation of the S2H Schema

An instantiation of the S2H Schema is defined by the notification methods WHENDOMAINREDUCTION and WHENFAIL and a specific instantiation of the manager of soft constraints. In the object language terminology, this means that we are provided with a class, for instance SoftManager, containing two virtual member functions (the two notification methods.) Then, an instantiation of S2H Schema is defined by a subclass of this class that implements these virtual functions.

All the other parts of this schema are handled by the solver.

4.1 Partition Based Filtering

The filtering algorithm is based on Theorem 1. Thus, its implementation is based on the computation of $\#inc$ counters and especially on the update of these counters.

The S2H method is instantiated for each soft constraint (i.e. $\mathcal{S} = \{C\}$) but all these instantiations share the same manager of soft constraints. So, the manager of soft constraints is unique. This object associates with each value a of each variable x on which a soft constraint is defined, the list of constraints that are violated by (x, a).

The notification methods are then defined as follows:

- WHENDOMAINREDUCTION method:
  when WHENDOMAINREDUCTION$(SoftManager, x, a, C)$ is called, constraint C is added to the list of constraints that are violated by (x, a). Then $\#inc$ counter associated with (x, a) and x are accordingly modified.

- WHENFAIL method: when WHENFAIL$(SoftManager, \{C\})$ is called, for each value a of each variable x involved in C, C is added to the list of constraints that are violated by (x, a)

Furthermore, the manager of soft constraints is also in charge of the $\#inc$ counters in regards to the current var-Partition. This object will notify the Satisfiability sum constraint of the modification of $\#inc$ counters. That constraint will then manage the possible domain reductions based on the application of Theorem 1.

4.2 Conflict Set Based Filtering

The S2H Schema can help us to efficiently implement the conflict sets based filtering algorithm, notably, to maintain incrementally the number of disjoint conflict sets detected.

First, we recall some principles on the computation of disjoint conflict sets, then we present the instantiation of S2H-Schema to improve the current implementation.

Consider that $\mathcal{Q} = \{CS_1, .., CS_k\}$ is a set of disjoint conflict sets of $\mathcal{C}$. Our goal is to find a set of disjoint conflict sets of greatest size (cf property 3). It is possible that, during the search for solutions, some new conflict sets can be found and thus the lower bound of the number of constraints that will be violated can be increased.

There are several ways to try to improve the number of disjoints conflict sets detected:

1. by recomputing the conflict sets from scratch,
2. by studying the set of constraints of $\mathcal{C}$ which do not belong to any set contained in $\mathcal{Q}$. This set of constraints can form some new conflict sets,
3. by refining the detection of conflict sets within the conflict sets of $\mathcal{Q}$.

The first possibility does not seem realistic. In fact, the computation of disjoint conflict sets is costly. Determining if a set of constraints $\mathcal{S}$ satisfies the condition of definition 5 is a NP-complete problem. Indeed, it consists of checking the global consistency of the constraint network $\mathcal{N}[\mathcal{S}]$ defined by $\mathcal{S}$ and by the set of variables involved in the constraints of $\mathcal{S}$. However, the identification of some conflict sets is sufficient. Instead of performing global consistency, we can easily identify a subset of constraints of a set $\mathcal{S}$ which forms a conflict set. The idea is to successively add the constraints of $\mathcal{S}$ into a solver until a failure occurs. All the constraint that have been added until this failure form a conflict set. A set of disjoint conflict sets is then obtained by repeating the previous algorithm on the constraints that are not yet member of a computed conflict set (each computation starts from scratch). It is also possible to refine this detection of conflict as mentioned in [5].

The problem of the approach 1 is that the mechanism is not obviously incremental. Nevertheless, we can use the previous computations to try to improve point 2 and 3 mentioned before.

The set of constraints $\mathcal{C}$ can be split into different parts: one for each conflict set of $\mathcal{Q}$ and one for the constraints of $\mathcal{C}$ which are not involved in any conflict set of $\mathcal{Q}$. We will denote by $NDCS$ (not detected conflict set) this latter set of constraint. More formally, $\mathcal{C}$ can be written: $\mathcal{C} = CS_1 \cup ... \cup CS_k \cup NDCS$.

First, consider the $NDCS$ set. The S2H-Schema can be used to efficiently detect during the search for solutions if this set contains a conflict set. The S2H method is instantiated as follows:

• $\mathcal{S} = NDCS$
• whenDomainReduction method is not used
• whenFail method: when whenFail$(SoftManager, \mathcal{S})$ is called, the manager of soft constraints notifies the Satisfiability sum constraint that a new conflict set has been detected. Then, we search whether the set of constraints S contains some disjoint conflict sets by using the algorithm presented before, and the set of disjoint conflict sets is accordingly updated.

The constraints which do not belong to a conflict set form the new $NDCS$ set. The S2H-Schema is then applied to this new set.

The Satisfiability sum constraint will then manage the possible domain reductions based on the application of property 3 or the filtering algorithm given in [5].

Now, consider a conflict set $CS = \{C_1, ..., C_k\}$. If this conflict set has been computed by using the algorithm we mentioned before, then we know that the set of constraints $\{C_1, ..., C_{k-1}\}$ is not detected as a conflict set by the solver. This means that this set is an $NDCS$ and we can apply the previous method on it. Therefore, for each conflict set an instantiation of S2H-Schema will be used in order to detect some subsets of conflict sets that also form a conflict set. Moreover, if a failure is detected in a set $\{C_1, ..., C_{k-1}\}$, then the constraint C_k is released. That is, C_k is no longer a member of a conflict set, and C_k is added to the $NDCS$ set of $\mathcal{C}$. Then, this addition may lead to a failure of the $NDCS$ set. In this case, the previous algorithm is applied. The current instantiation of the S2H-Schema for $NDCS$ is accordingly modified in order to take into account C_k[3]. Furthermore, the S2H-Schema is used to maintain the detection of subset of the conflict sets that have been newly detected.

5 Implementation of the S2H Schema

Notification methods are usually easy to implement. In fact, most of the solvers provide the user with methods that are called when certain events on the domain of a variable occur. For instance, in ILOG Solver, IlcDemon instances are especially well suited to be used for this purpose. In this case, we just have to define one IlcDemon per variable, and to link this demon to each modification of the directed copy of the variable (IlcWhenDomain event in ILOG Solver.) Then, each IlcDemon will be triggered when the domain of the corresponding variable will be modified.

The hardening of soft constraints, the catch of the failure and the deletion of some hard constraints is often a difficult task. We propose to give some details on the implementation problems and to give some possible general solutions that concern most of the solvers.

Generally, a solver works as follows. At the top level, the constraints are added, and the filtering algorithms associated with them are called. If a filtering removes some values of some variables, then a propagation is triggered, that is the filtering algorithms associated with the constraints involving a modified variables are called again and so on. Then, the search for a solution starts. This search creates choice points (i.e. nodes of a tree search). In other words, a decision is made and the corresponding constraint, which is usually an assignment constraint, is added to the solver. The propagation mechanism is then triggered. When there is nothing to propagate (the current choice point is a success), a new choice point is made. On the other hand, if a failure occurs the current choice is abandoned. The consequences are:

[3] Either the current instantiation of the S2H-Schema for $NDCS$ is modified, or it is deleted and a new one is created.

– Everything that has been allocated since the choice point is destroyed.
– All filtering algorithms must be immediately stopped.
– All the propagation queues are emptied.
– A backtrack/undo is done.

Thus, every solver is able to perform these kinds of operations.

In order to manage the operations required by any instantiation of S2H-Schema: catch of the failure, deletion of constraints; we propose to study three kinds of possibilities:

1. Creation of a new choice point
2. Use of an independent solver in parallel.
3. Internal addition and catch of the failure

In next paragraphs we discuss these solutions, through the example of a set of constraint $\mathcal{S}$ containing the constraint C: $x < y$, on which the hardening of soft constraint operation is applied.

5.1 Creation of a New Choice Point

The directed copy of variables and the constraint $hard(\mathcal{S}) \cup DcopyCt(\mathcal{S})$ are defined within the very same solver but they are encapsulated inside a new choice point. For instance, this operation can be easily done in ILOG Solver by using function IloSolver::solve.

The catch of a failure and the deletion of constraints are easily managed because when a failure occurs the solver has just to abandon the choice point. The main drawback of this method is that the constraints added inside the new choice point are added from scratch. Therefore, no incremental mechanism can be used. Moreover, it is necessary to create a choice point for each instantiation of S2H-Schema. This fact can be prohibitive to implement the partition based filtering algorithm. This is not the kind of result we aim to obtain.

5.2 Use of an Independent Solver in Parallel

The principle is to define the directed copy of variables and $hard(\mathcal{S})$ into another solver. The constraints between a variable and its directed copy are defined in the initial solver.

The advantage is that there is no problem due to failure of $dcopy(x, \mathcal{S}) < dcopy(y, \mathcal{S})$ because this constraint is defined in a specific solver. There is also no problem of continuation of the main solver. The deletion $hard(\mathcal{S})$ is simply done by stopping to call the other solver.

However, a main difficulty is the necessity to ensure a simultaneous backtracking of the two solvers and also to implement the notification methods that are defined on variables of different solvers. For instance, assume that initial domains are $D(x) = [0, 10], D(dcopy(x, \mathcal{S}) = [0, 10]$. Assume that domains are reduced as follows: $D(x) = [3, 7]$ and $D(dcopy(x, \mathcal{S})) = [3, 7]$. If a backtrack occurs, then we will have $D(x) = [0, 10]$. The problem is then to backtrack also the second solver, that is, to update $D(dcopy(x, \mathcal{S})) = [0, 10]$.

Moreover, this solution requires having one solver for each instantiation of S2H-Schema.

5.3 Internal Addition and Catch of Failure

The addition of constraints performed by the hardening of soft constraints is made in the same solver. If $(dcopy(x, \mathcal{S}) < dcopy(y, \mathcal{S}))$ or a constraint of $DcopyCt(\{C\})$ fails , then this failure has to be caught, and all the constraints added by the hardening of soft constraints operation must be removed. This means that:

– The current code in which the failure occurs has to be abandoned.
– The parent constraint of the failing constraint and all the descendant constraints of the parent constraint has to be abandoned.
– A global failure must not be triggered.
– Some hard constraints have to be removed.

The last point is fundamental. The implementation of this solution depends on the management of the failure by the solver and on the management of the addition/removal of constraints.

This solution can be particularly difficult to implement in a constraint solver, but it is clearly the most promising with respect to efficiency and memory consumption. It also benefits from all the advantages of a solver.

5.4 Summary

The following table recapitulates the advantages and the drawbacks of each method.

Method	Advantages	Drawbacks
Independent Solver	No problem of failure	Requires simultaneous backtracks and synchronization Hard to implement No incrementality Requires One Solver per instantiation
Creation of a New Choice Point	No problem of failure	Hard Constraint always added from scratch No incrementality Requires One Choice Point per instantiation
Internal Addition And Catch of Failure	Incremental. Simple. Easy to use	Requires to change classical behavior of a solver

We implemented the S2H method in ILOG Solver. This was available as a beta functionality.

We tried to implement the use of independent solvers, but we encounter problems with the memory management and the fact that ILOG Solver was not designed for having several solvers at the same time. However, with some other solvers this method could be certainly efficient and competitive with teh catch of the failure.

We also considered the encapsulation into choice point. First, it is really slow in regards to the other approaches (10 times slower in general). In fact, the creation of a choice point is not a simple task in ILOG Solver. However, the main issue is the lost of the incrementality. After each instantiation the same constraints are posted and reposted... In addition we encounter some problems because ILOG Solver is not reentrant. In conclusion, we think that this method is not really good.

At last, we modified the internal code of ILOG Solver in order to be able to catch the failures. This method performs well.

6 Conclusion

In this paper, we have proposed a general schema, called S2H (Soft to Hard), which is able to exploit the filtering algorithm associated with a constraint even if this constraint is soft. The advantage of this approach is double: first it can be easily used by any constraint programming solver system provided that the failure can be caught, second we immediately and automatically benefit from the pruning performance of the filtering algorithm associated with the soft constraint because this is managed by the solver.

Furthermore, two instantiations of this schema have been presented, corresponding to the two different filtering algorithms that have been proposed to improve the resolution of over-constrained problems: the partition based filtering and the conflict sets based filtering. And, efficient implementation of these algorithms is now available.

The implementation of the S2H-Schema in a solver is also discussed.

Acknowledgements

We would like to thank T. Petit and J-F. Puget for their useful comments.

References

1. Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., Fargier, H.: Semiring-based csps and valued csps: Frameworks, properties, and comparison. Constraints 4, 199–240 (1999)
2. Larrosa, J., Meseguer, P., Schiex, T.: Maintaining reversible DAC for Max-CSP. Artificial Intelligence 107, 149–163 (1999)
3. Petit, T., Régin, J.-C., Bessière, C.: Specific filtering algorithms for over-constrained problems. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 451–465. Springer, Heidelberg (2001)
4. Petit, T., Régin, J.-C., Bessière, C.: Meta constraints on violations for over-constrained problems. In: Proceedings ICTAI 2000, pp. 358–365 (2000)
5. Régin, J.-C., Petit, T., Bessière, C., Puget, J.-F.: New lower bounds of constraint violations for over-constrained problems. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 332–345. Springer, Heidelberg (2001)
6. van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(fd). Journal of Logic Programming 37(1-3), 139–164 (1998)

The Objective Sum Constraint

Jean-Charles Régin[1] and Thierry Petit[2]

[1] Université de Nice-Sophia Antipolis, I3S, CNRS
jcregin@gmail.com
[2] École des Mines de Nantes, LINA, CNRS
thierry.petit@mines-nantes.fr

Abstract. Constraint toolkits generally propose a sum constraint where a global objective variable should be equal to a sum of local objective variables, on which bound-consistency is achieved. To solve optimization problems this propagation is poor. Therefore, ad-hoc techniques are designed for pruning the global objective variable by taking account of the interactions between constraints defined on local objective variables. Each technique is specific to each (class of) practical problem. In this paper, we propose a new global constraint which deals with this issue in a generic way. We propose a sum constraint which exploits the propagation of a set of constraints defined on local objective variables.

1 Introduction

A lot of optimization problems aim to minimize (or maximize) the sum (or a scalar product) of some variables. The variable equals to that sum is usually called the *objective variable* and denoted by obj, and the sum is called the *objective constraint*. For convenience, we will call *sub-objective variables* the variables involved in the sum while the other variables but obj will be called *problem variables*. The sub-objective variables are denoted by $sobj_i$ and the problem variables by x_i.

Lower bounds of the objective variable are computed from the sum constraint by considering minimum values of the sub-objective variables. Thus, we usually state that $\sum \underline{sobj_i} = \underline{obj}$ where $\underline{y}$ denotes the minimum value in the domain of the variable y. The minimum value of the sub-objective variables are computed from the constraints (different from the objective constraint) which involve them.

This model has three main drawbacks:

- The filtering algorithm associated with the objective constraint is weak although it should deserve more attention because it is often the most important one.
- The fact that some problem variables may occur in several constraints, each involving a different sub-objective variable, is ignored.
- The fact that some sub-objective variables may be involved in several constraints (different from the objective constraint) is ignored while it is important [3,4].

This paper tries to remedy to these drawbacks. The main idea is to change the way the sub-objective constraints are propagated. Consider, for instance, that we have the following problem to solve: $obj = sobj_1 + sobj_2$ has to be minimized while respecting the constraints $sobj_1 = 2x + y$ and $sobj_2 = z - x$ with x taking its value in $[0, 10]$, y in $[0, 10]$ and z in $[10, 20]$. The classical filtering algorithm will lead to $\underline{sobj_1} = 2*0+0 = 0$ and $\underline{sobj_2} = 10 - 10 = 0$, therefore $\underline{obj} = 0$. However, if we look at problem from the point of view of the variable x, this means that we need to find a value for x such that $sobj_1 + sobj_2$ is minimized with $sobj_1 = 2x + y$ and $sobj_2 = z - x$. Clearly, $x = 0$ minimizes the value of $sobj_1$ but not at all the value of $sobj_2$ and $x = 10$ minimizes the value of $sobj_2$ but not the value of $sobj_1$. If we try all values for x then we will discover that $x = 0$ minimizes the value of $sobj_1 + sobj_2$ which is equal to $y + z$ whereas any other value v will lead to $y + z + v$. Thus, by applying a stronger form of consistency we deduce that obj is greater than or equal to 10.

The main idea of this paper is to count for each problem variable involved in a constraint involving a sub-objective variable a lower bound of its contribution to the final objective. We can easily figure out how we can use the result obtained from one problem variable but it is unclear to see how the values obtained for all the problem variables can be sum up. In this paper, we propose a generic answer to this question, inspired from the PFC-MRDAC algorithm [2]: we select a problem variable x and compute a lower bound of the sum of the sub-objective variables involved in a constraint with x. Then we remove these sub-objective variables and we repeat the process for another variable. At the end we can sum up the different lower bounds of the subsums because there are disjoint, and we obtain a lower bound for the objective variable.

The paper is organized as follows. First, we propose a new lower bound of the objective variable. Then, we introduce a new filtering algorithm. At last, we give some related work and we conclude.

2 Objective Sum Constraint

Definition 1. *Given*

- *obj an objective variable*
- *$SO = \{sobj_1, ..., sobj_p\}$ a set of sub-objective variables,*
- *$PX = \{x_1, ..., x_q\}$ a set of problem variables disjoint from SO,*
- *C_{SO} be a set of constraints, each of them involving at least one sub-objective variable.*

The **objective-sum constraint** *is equivalent to the constraint network defined by the variables $(obj \cup SO \cup PX)$ and the constraints C_{SO} and the objective constraint $obj = \sum_{sobj_i \in SO} sobj_i$*

We will consider that we are provided with `minsobj`$((x, a), sobj_i, C)$, a function returning the minimum value of $sobj_i$ compatible with (x, a) on the constraint C. It returns the minimum value of $sobj_i$ consistent with C if x is not involved in C. This function can be implemented using the filtering algorithm of C and it does not have to be exact (a lower bound can be used).

3 Lower Bounds

3.1 Multiple Constraints Involving a Sub-objective Variable

Consider a variable x involved in several constraints with the same sub-objective variable $sobj_i$ and a value a of $D(x)$. A lower bound of $sobj_i$ from the point of view of the variable x is the maximum value of $\texttt{minsobj}((x,a), sobj_i, C)$ among all constraints involving $sobj_i$:

Property 1. *We define*

- $C_{SO}(sobj)$*, the set of constraints involving* $sobj$.
- $\underline{xsobj_i}(x,a) = \max_{C \in C_{SO}(sobj_i)}(\mathtt{minsobj}((x,a), sobj_i, C))$
- $\underline{xsobj_i}(x) = \min_{a \in D(x)}(\underline{xsobj_i}(x,a))$

Then, $sobj_i \geq \underline{xsobj_i}(x)$

If we denote by $minseparate(sobj_i)$ the minimum value of $sobj_i$ consistent with each constraint of $C_{SO}(sobj_i)$ taken separately then we have:

Property 2. $\underline{xsobj_i}(x) \geq minseparate(sobj_i)$

3.2 Multiple Sub-objective Constraints Involving a Variable

The computation of the minimum value of the objective variable is usually made by independent computations of the minimum value of all the sub-objective variables. We propose here to take into account simultaneously some other constraints, that is, establishing a stronger form of consistency.

Consider a variable x involved in a constraint C_1 involving the sub-objective variable $sobj_i$ and in a constraint C_2 involving the sub-objective variable $sobj_j$. Considering the constraints separately means that the value of x consistent with the minimum value of $sobj_i$ on C_1 may be different than the one consistent with $sobj_j$ on C_2. So, for the sum $sobj_i + sobj_j$, we can compute a lower bound of the minimum value of this sum by considering different values of x, which is clearly an underestimation because in any solution the value of x will be the same. Instead, for each value a of x, we propose to compute a lower bound of $sobj_i + sobj_j$. There is a value (x,a) for which $l = sobj_i + sobj_j$ is minimum among all the values in the domain of x. Then, l is a new lower bound of $sobj_i + sobj_j$. Note that this value can never be less than the one computed for each sub-objective variable separately. Hence, we have:

Property 3. *Let* $S \subseteq SO$ *be a set of sub-objective variables. We define:*

- $L(S,(x,a)) = \sum_{sobj_i \in S} \underline{xsobj_i}(x,a)$
- $\underline{sumobj}(x,S) = \min_{a \in D(x)}(L(S(x,a)))$ *Then,*

$$\sum_{sobj_i \in S} sobj_i \geq \underline{sumobj}(x,S)$$

Property 4. *If S is equal to all the sub-objective variables involved in a constraint with x we have:*

$$\underline{sumobj}(x,S) \geq \sum_{sobj_i \in S} minseparate(sobj_i)$$

It is not relevant to consider sub-objective variables $sobj_i$ such that there is no constraint in $C_{SO}(sobj_i)$ involving x. Next section shows how SO can be partitioned so as to sum up efficiently several lower-bounds computed thanks to Property 3, in order to obtain a lower-bound of obj.

3.3 A New Lower Bound of the Objective Variable

Property 3 shows how we can improve the computation of some sums of sub-objective variables. We will obtain a new lower bound of the objective if the addition of that sub-sums is equal to the whole sum at the end. That is, if no sub-objective variable is added twice. By partionning the set of sub-objective variables SO we avoid counting twice the same variable. In addition we can apply for each part Property 3. Therefore, we have the following proposition:

Proposition 1. *Let SO be the set of sub-objective variables, and $\mathcal{P}(SO) = \{S_1, ..., S_k\}$ a partition of SO, and $\{x_1, ..., x_k\}$ a set of variables. Then,*

$$obj = \sum_{sobj_i \in SO} sobj_i \geq \sum_{S_i \in \mathcal{P}(SO)} \underline{sumobj}(x_i, S_i)$$

The main issue is to determine how the partition is defined and which variable is associated with each set. In fact, these questions are strongly related. We can imagine several techniques for computing this partition. For instance, here is a possible algorithm:

1. For each $x \in PX$ define $sobjvar(x)$, the set of sub-objective variables which are involved in a constraint involving x
2. Define O equal to SO and $lobj$ equals to 0
3. Select the variable x having the largest $sobjvar(x) \cap O$ cardinality.
4. Compute $\underline{sumobj}(x, sobjvar(x) \cap O)$ and add the result to $lobj$
5. Remove $sobjvar(x)$ from O
6. Repeat from step 3) until there is no more variable in O

At the end of this algorithm $lobj$ is a new lower bound of obj.

4 Filtering Algorithm

From the lower bound presented in Section 3 we can design a filtering algorithm whose goal is to reduce the domain of some problem variables and not only the domains of the objective or sub-objective variables. We will consider two types of problem variables: those that are associated with a part of $\mathcal{P}(SO)$ and those that are not.

Consider a variable x and S be the part of $\mathcal{P}(SO)$ associated with it. If x is assigned to $b \neq a$ then the lower bound is increased by $L(S,(x,b)) - \underline{sumobj}(x,S)$. If this increment is too much for the objective variable (i.e. the lower bound is greater than $\overline{obj}$) then the value (x,b) is not consistent with the constraint. Thus we have:

Property 5. *For any value b of x we define:*
- $slack((x,b),S) = L(S,(x,b)) - \underline{sumobj}(x,S)$
- $K = \overline{obj} - [\sum_{S_i \in \mathcal{P}(SO)} \underline{sumobj}(x_i,S_i)]$

Then, each value (x,b) such that $slack((x,b),S) > K$ **is not consistent** *with the objective constraint.*

Once the new lower bound has been computed this filtering algorithm does not cost anything more because all the sums $\sum_{sobj_i \in S} \underline{xsobj_i}(x,b)$ have been computed for all the values of x.

With respect to variables which are not associated with a part, we can either ignore them (or compute a new partition and apply the filtering algorithm).

5 Application

The objective sum constraint arises frequently in problem with multi-objectives. For instance, it occurs in multidimensional bin packing problems (in which an item has several dimensions), like the one appearing in the cloud computing management. In these problems, we need to fill in servers (bins) with virtual machine (items) while respecting all the sums of the capacities, one for each dimension. In the same time, we associate with each dimension and with each server a cost representing the assignment of the virtual machine to the server. For instance, a dimension may be the disk access time and the cost represents a penalty depending of the time access given by the server. The objective is to minimize the sum of all the penalties and penalties may be involved in several side constraints.

6 Related Work

The idea of the partition comes from PFC-MRDAC, an algorithm for solving over-constrained problems. The original version of the algorithm dealing only with binary constraints is given in [2]. A simpler presentation, not restricted to binary constraints, is given in [5]. Our generic technique is an alternative to some dedicated algorithms for propagating sum constraints, which are specific to particular classes of optimization problems (e.g., see [1] for constraint-based scheduling).

7 Conclusion

This paper presented a preliminary work about a new filtering algorithm for the objective sum constraint, which is useful in the resolution of multi-objective problems.

References

1. Kovács, A., Beck, J.C.: A global constraint for total weighted completion time for cumulative resources. Constraints (2010) (in print)
2. Larrosa, J., Meseguer, P., Schiex, T., Verfaillie, G.: Reversible DAC and other improvements for solving Max-CSP. Proceedings AAAI, pp. 347–352 (1998)

3. Larsson, T., Patriksson, M.: On side constrained models of traffic equilibria. In: Proc. of the 19th Course of the International School of Mathematics, pp. 169–178. Plenum Press, New York (1995)
4. Petit, T., Poder, E.: Global propagation of side constraints for solving over-constrained problems. Annals of Operations Research 184, 295–315 (2011)
5. 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, pp. 543–548. Springer, Heidelberg (2000)

Almost Square Packing

Helmut Simonis and Barry O'Sullivan

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

Abstract. The almost square rectangle packing problem involves packing all rectangles with sizes 1×2 to $n \times (n+1)$ (almost squares) into an enclosing rectangle of minimal area. This extends the previously studied square packing problem by adding an additional degree of freedom for each rectangle, deciding in which orientation the item should be packed. We show how to extend the model and search strategy that worked well for square packing to solve the new problem. Some adapted versions of known redundant constraints improve overall search times. Based on a visualization of the search tree, we derive a decomposition method that initially only looks at the subproblem given by one of the cumulative constraints. This decomposition leads to further modest improvements in execution times. We find a solution for problem size 26 for the first time and dramatically improve best known times for finding solutions for smaller problem sizes by up to three orders of magnitude.

1 Introduction

The almost square rectangle packing problem [9, 12, 13] involves packing all rectangles with sizes 1×2 to $n \times (n+1)$ into an enclosing rectangle of minimum area. The orientation of the rectangles can be freely chosen, adding an additional degree of freedom compared to the previously studied square packing problem [8, 10, 11, 15, 18]. General rectangle packing is an important problem in a variety of real-world settings. For example, in electronic design automation the packing of blocks into a circuit layout is essentially a rectangle packing problem [14, 16]. Rectangle packing problems are also motivated by applications in scheduling [10, 11, 15]. Rectangle packing is an important application domain for constraint programming, with significant research into improved constraint propagation methods reported in the literature [1–7, 19].

2 Constraint Programming Model

We initially use the established constraint model [2, 6, 18] for the rectangle packing problem. Each item to be placed is defined by domain variables X and Y for the origin in the x and y dimension respectively, and two domain variables W and H for the width and the height of the rectangle, respectively. In the particular case of packing almost squares, W and H can take only two possible values

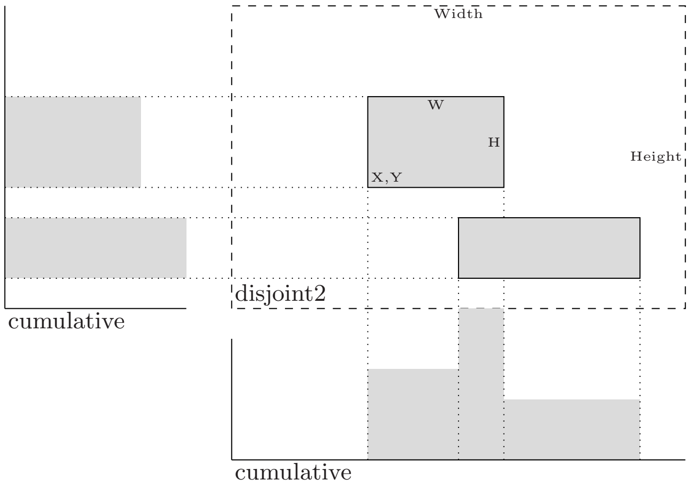


Fig. 1. The basic constraint programming model

(n and $n+1$), and must be different from each other. The constraints are expressed by a non-overlapping constraint in two dimensions and two (redundant) Cumulative constraints that work on the projection of the packing problem in the x and y direction. This is illustrated by Figure 1. We use SICStus Prolog 4.0.4 (on a 3GHz Intel Xeon 5450 with 3.25GB of memory), which provides both Cumulative [1] and Disjoint2 [3] constraints.

2.1 Generating Candidate Enclosing Rectangles

To find the enclosing rectangle of smallest area, we need a decomposition strategy that generates sub-problems with fixed enclosing rectangle sizes. We enumerate on demand all pairs *Width*, *Height* in order of increasing area $\texttt{Width} \times \texttt{Height}$ that satisfy

$$[\texttt{Width}, \texttt{Height}] :: n..\infty, \texttt{Width} \geq \texttt{Height}$$

$$\sum_{i=1}^{n} i \times (i+1) \leq \texttt{Width} * \texttt{Height}$$

$$k = \left\lfloor \frac{\texttt{Height}+1}{2} \right\rfloor, \texttt{Width} \geq \sum_{j=k}^{n} j \tag{1}$$

Equation 1 provides a simple bound on the required area, considering all large items that cannot be stacked on top of each other, which, thus, must fit horizontally. For candidates with the same area, we try them by increasing *Height*, i.e. for two subproblems with the same surface we try the "less square-like" solution first. We then solve the rectangle packing problem for each such candidate enclosing rectangle in turn, until we find the first feasible solution. By construction, this is an optimal solution. The number of candidates seems to grow linearly with the amount of slack (empty space) allowed. In comparison with the square packing problem, we find that the optimal solution in many cases does not use any slack at all, and the number of candidates to be tested remains quite small.

2.2 Symmetry Removal

The model so far contains a number of symmetries, which we need to remove as we may have to explore the complete search space. We restrict the domain of the largest square of size $n \times (n+1)$ to be placed in an enclosing rectangle of size $Width \times Height$ to

$$X :: 1..1 + \left\lfloor \frac{Width - n}{2} \right\rfloor, Y :: 1..1 + \left\lfloor \frac{Height - n}{2} \right\rfloor.$$

Other symmetries are discussed below, but are not yet handled as part of the constraint model.

3 Search

We studied a number of different search strategies for square packing in [18]. The best method found used an interval labeling approach, first assigning the X variables to intervals, small enough to create obligatory parts, then fixing the X variables to values, and then repeating the process for the Y variables. For the problem sizes studied (up to 27) this provided the best solutions, when fixing the interval size to a fraction between 0.2 and 0.3 of the square width.

In the almost square packing problem, we have to assign W and H variables in addition to the X and Y variables. As the W and H variables of one rectangle are linked by a disequality, and can only take two possible values, it is enough to assign W, this will force the assignment of the H variable.

When should we assign the W variables in the search process? We have studied three cases:

eager. Assign all W variables before assigning any X variables, leading to multiple problems with oriented rectangles;
lazy. Assign the W variables once all X variables have been assigned to intervals, but before assigning fixed values for X;
mixed. For each rectangle, ordered by decreasing size, first assign the W variable, then fix the X variable to an interval. Repeat this process for all rectangles, before assigning the X variables to values.

Not surprisingly, the mixed method clearly outperforms the two other methods. In Figures 2, 3, and 4 we show the node distribution of the search for the first solution, considering problem size 17. The display shows the number of TRY and FAIL nodes at each level of the search tree. A TRY node is generated, when we try to assign an interval or value to a variable and the resulting propagation succeeds. A FAIL node is generated when the assignment leads to a failure and backtracking. The displays are generated with CP-Viz [17], a generic visualization tool for finite domain constraint solvers.

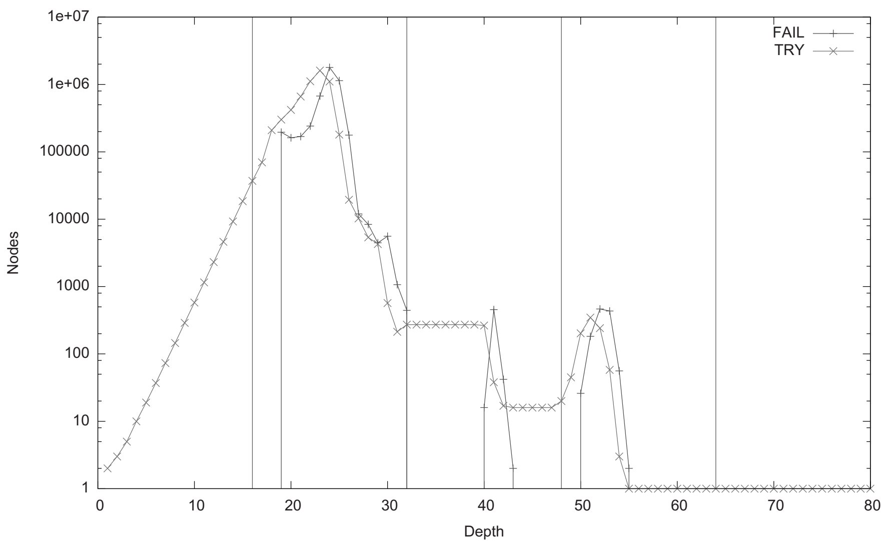


Fig. 2. Eager Orientation (N=17)

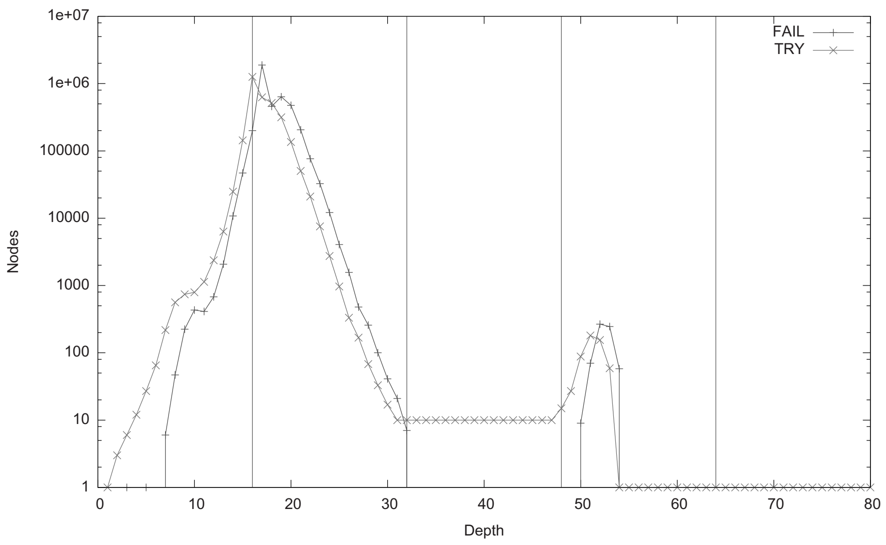


Fig. 3. Lazy Orientation (N=17)

For the eager method (Figure 2) we see that failures only start once we begin to assign the X variables to intervals. The initial fixing of the rectangle orientation leads to an exponential growth of the search tree (straight line on the left side of the graph due to the log-scale), peaking at over a million nodes at level 23. Note that after the assignment of the X variables only 20 possible solutions remain. Starting with the assignment of the Y variables, the search tree expands again, but only to a few hundred nodes.

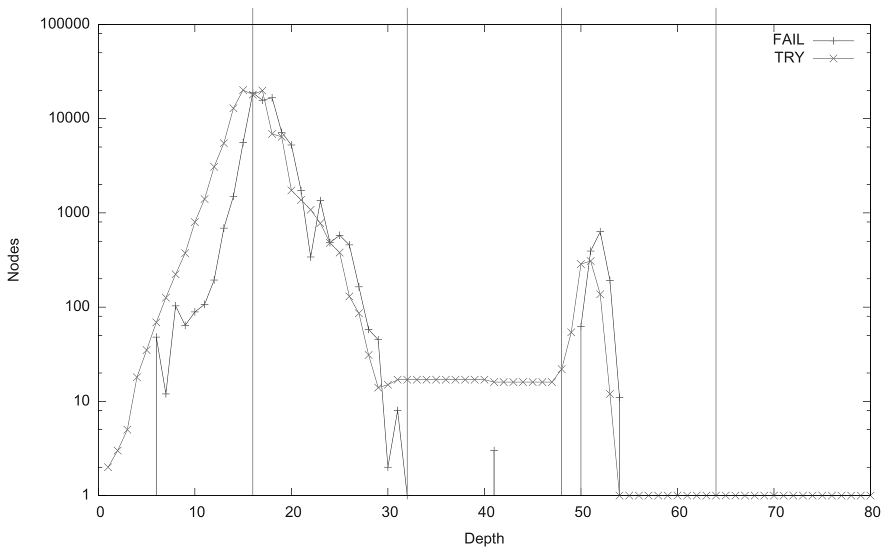


Fig. 4. Interleaved Orientation (N=17)

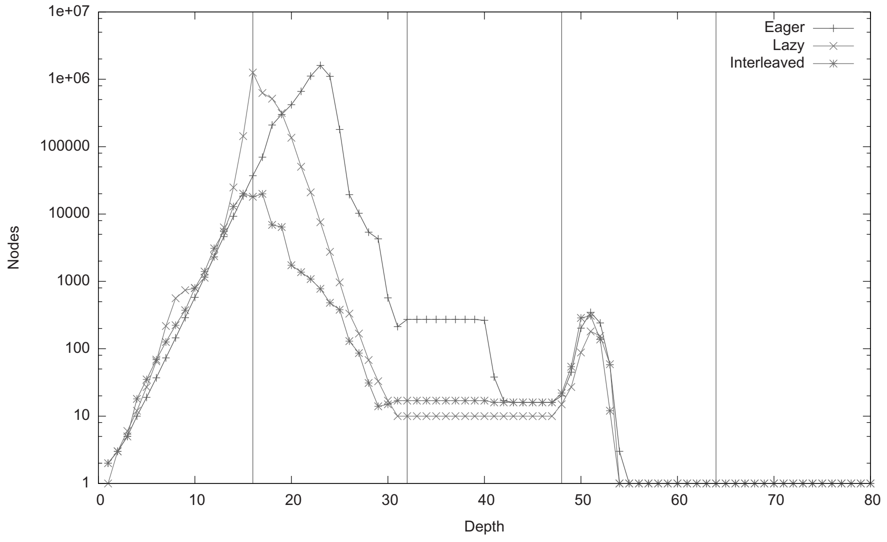


Fig. 5. Assignment Strategies Compared (N=17)

For the lazy method (Figure 3), the overall structure of the graph is similar, although the maximal width of the search tree (again, over a million nodes) is reached earlier, at the end of the X variable interval assignment. Forcing the orientation of the rectangles then leads to a rapid elimination of candidate solutions. Although failures occur earlier in the search, the propagation is not

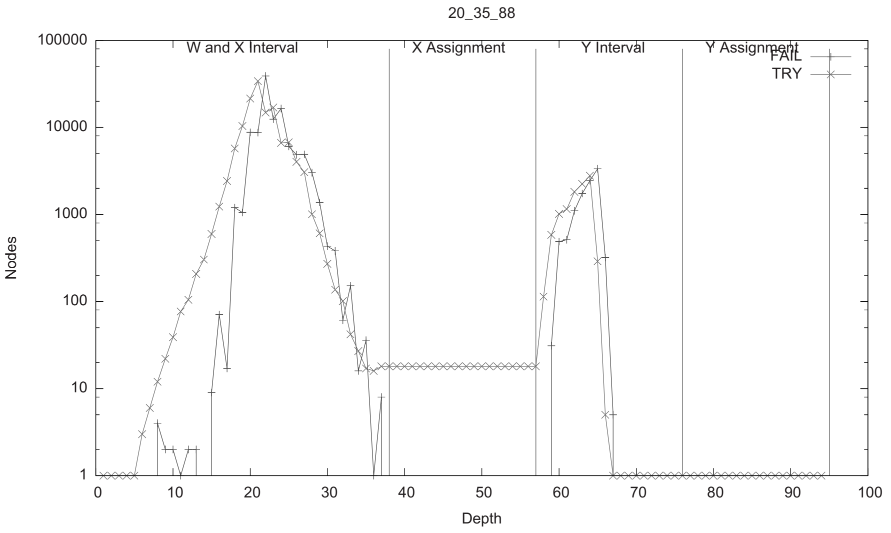


Fig. 6. Node Distribution (N=20)

powerful enough to eliminate unfeasible candidates without knowing the orientation of the rectangles.

In the mixed method (Figure 4), the propagation can eliminate more partial assignments early in the search, so that the maximal width of the tree is around 20000 nodes. Figure 5 compares the three methods considering only the TRY nodes. We see that the search for the last X variable assignments and for finding the Y variables is quite similar, but that the mixed method clearly outperforms the two other methods early in the search.

The overall structure of the search tree is remarkably similar for most problem sizes: Figure 6 shows the node distribution for problem size 20. An exception is problem size 21, shown in Figure 7. This shows the node distribution for the 46×77 candidate rectangle with no slack. Even after the orientation and X interval assignment of all rectangles a large number of partial assignments remains, which are only reduced by the assignment of the X variables to particular values. But there is no solution to this problem, therefore all possible assignments must be enumerated.

The optimal solution for size 26 is shown in Figure 8. This result has not been previously published. Previous work only obtained solutions for problem sizes up to 25 [12].

The results for the basic model are shown in Table 1. It shows the problem size N, the total *Surface* of the rectangles to be placed, the number of candidate enclosing rectangles studied (K), the *Width* and *Height* of the optimal enclosing rectangle, its *Area* and the amount of lost space (*Lost*). It then counts the number of backtracking steps and the time required to find the first solution, the total number of solutions for the given enclosing rectangle, and the number of backtracking steps and time required to enumerate all such solutions. Note

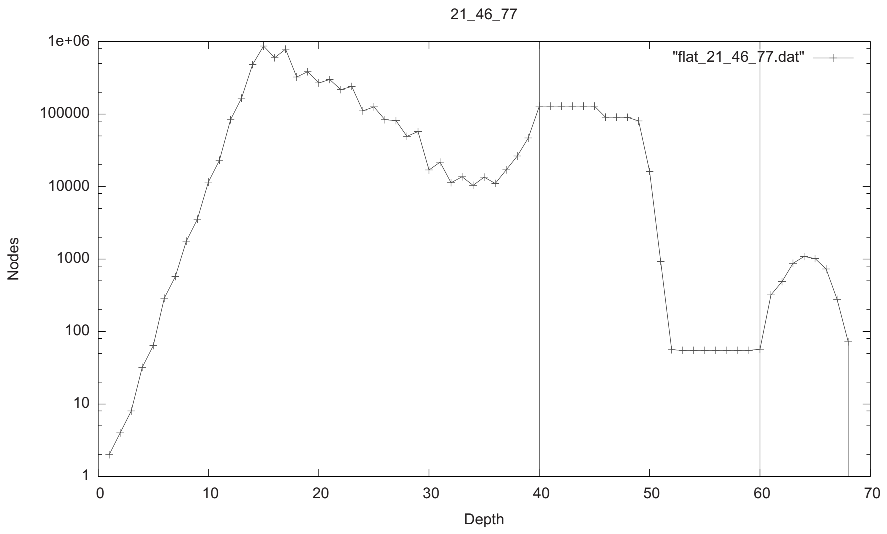


Fig. 7. Infeasible Problem Instance (N=21)

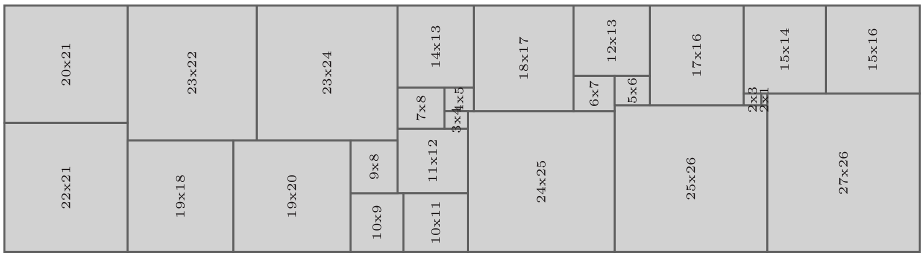


Fig. 8. Optimal Solution Size 26; The X axis is along the *shorter* side

that the total number of all optimal solutions can be higher, as there can be candidate rectangles with the same optimal area which are not explored by our algorithm, which stops at the first feasible candidate.

The total number of solutions varies widely with the problem size. For the problem sizes (6, 9, 10, 12, 21) where the optimal solution is not perfect (i.e. requiring some slack), the number of solutions increases as the 1×2 rectangle can be placed in many of the empty spaces.

In general, if a solution contains two (consecutive) rectangles which share a common edge, then we can exchange these rectangles creating a new solution. In Figure 8 for example, the rectangles 22×21 and 20×21 (on the left) can be exchanged. Indeed, in Figure 8 there are 5 such pairs of rectangles, which can be flipped independently, leading to 32 symmetrical solutions.

3.1 Redundant Constraints

We have previously described [18] two methods which were quite effective in reducing problem complexity:

- The first was to ignore the 1×1 square when setting up the constraints, while still reserving space for it in the enclosing rectangle. This both reduced the amount of unnecessary work inside the constraints dealing with this small square, and avoided symmetries in the search when the 1×1 square was placed in all possible empty places.
- The second idea was to eliminate certain X and Y values, when squares were placed close to the border of the enclosing space. If a large object is placed near a border, then it might be impossible to fill the gap between the border and the object with the few available, smaller items and the slack allowed (empty space). These gap limits can be precomputed and domain values can be removed a priori, reducing the search space.

For the almost square packing problem, the smallest item is the 1×2 rectangle. If we remove it from the problem, we might find an infeasible solution, if an assignment exists where all empty space is allocated to non-connected 1×1

Table 1. Basic Model Results

							First Solution		All Solutions		
N	Surface	K	Width	Height	Area	Loss	Back	Time	Sols	Back	Time
4	40	1	4	10	40	0.00	2	00:00	8	6	00:00
5	70	1	5	14	70	0.00	4	00:00	16	14	00:00
6	112	3	6	19	114	1.79	16	00:00	216	24	00:00
7	168	3	12	14	168	0.00	19	00:00	65	76	00:00
8	240	4	15	16	240	0.00	6	00:00	12	83	00:00
9	330	6	14	24	336	1.82	54	00:00	9170	3137	00:00
10	440	6	17	26	442	0.45	323	00:00	1854	1379	00:00
11	572	3	22	26	572	0.00	99	00:00	4	268	00:00
12	728	8	21	35	735	0.96	546	00:00	25180	13795	00:02
13	910	3	26	35	910	0.00	1900	00:00	42	6197	00:00
14	1120	4	28	40	1120	0.00	2937	00:00	4	9604	00:00
15	1360	4	34	40	1360	0.00	14440	00:00	4	50592	00:03
16	1632	4	32	51	1632	0.00	15967	00:01	544	48711	00:03
17	1938	3	34	57	1938	0.00	210878	00:14	16	398759	00:27
18	2280	4	30	76	2280	0.00	9734	00:00	110288	152032	00:24
19	2660	4	35	76	2660	0.00	102235	00:08	526	3240741	04:26
20	3080	4	35	88	3080	0.00	351659	00:34	1988	3612859	05:52
21	3542	5	39	91	3549	0.20	14036353	21:38	3250117	720146935	25:13:20
22	4048	3	44	92	4048	0.00	58206362	01:37:30	688	122563947	03:23:19
23	4600	3	40	115	4600	0.00	14490682	30:12	6784	136039535	04:38:40
24	5200	3	40	130	5200	0.00	27475258	55:05	96	99731414	03:20:37
25	5850	5	45	130	5850	0.00	35282646	01:23:12	1007780		
26	6552	5	42	156	6552	0.00	92228265	03:28:20	1056		

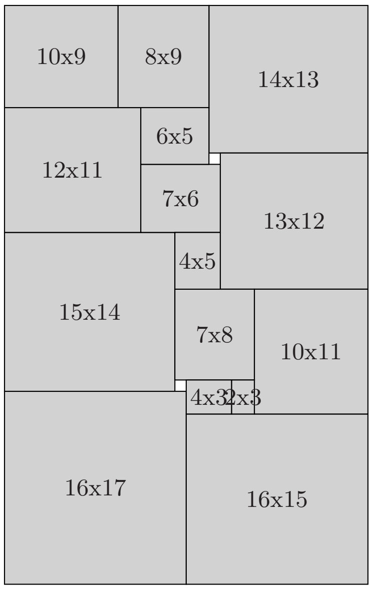


Fig. 9. Pseudo Solution N=16 Width=32 Height=51 with 1×2 item removed; This can not be extended to a complete solution

pieces. Fortunately, that situation rarely occurs; Figure 9 shows a case for size 16. We correct this by enforcing an additional non-overlapping constraint at the end of the search, where we add the 1×2 piece back to the problem. If there is no room to place that item, the constraint will fail and we backtrack to find another candidate for the relaxed problem, until a valid solution is generated.

The precomputation of infeasible gap values can also be done for the almost square case, although the domain restrictions are somewhat weaker.

The effect of the redundant constraints are shown in Table 2. Ignoring the 1×2 rectangle leads to a small, but consistent improvement (Not One Column) compared to the Basic Model. Removing values close to the border of the placement area (Gap Column) has a more significant effect, while combining both leads to the best results.

3.2 Impact of Interval Size

In [18], we also studied the impact of the chosen interval size on the performance of the algorithm. We repeated these tests for the almost square packing problem, which lead to a similar conclusion. Setting the interval to 0.3 times the size of the item leads to the best performance, both in number of search nodes and execution time. As Figure 10 shows, the effect is rather restricted, with an obvious effect visible only for problem size 21, which is the only large instance which requires some slack.

Table 2. Redundant Constraint Model Results

	Basic Model		Not One		Gap		Both	
N	Back	Time	Back	Time	Back	Time	Back	Time
4	2	00:00	2	00:00	2	00:00	2	00:00
5	4	00:00	3	00:00	2	00:00	1	00:00
6	16	00:00	16	00:00	6	00:00	6	00:00
7	19	00:00	18	00:00	10	00:00	9	00:00
8	6	00:00	5	00:00	17	00:00	10	00:00
9	54	00:00	54	00:00	27	00:00	27	00:00
10	323	00:00	323	00:00	159	00:00	159	00:00
11	99	00:00	99	00:00	54	00:00	54	00:00
12	546	00:00	546	00:00	274	00:00	274	00:00
13	1900	00:00	1900	00:00	1040	00:00	1040	00:00
14	2937	00:00	2936	00:00	1505	00:00	1501	00:00
15	14440	00:00	14425	00:00	7632	00:00	7617	00:00
16	15967	00:01	9338	00:00	7264	00:00	3989	00:00
17	210878	00:14	210850	00:13	107639	00:07	107611	00:07
18	9734	00:00	9734	00:00	5550	00:00	5550	00:00
19	102235	00:08	102235	00:08	13694	00:01	13690	00:01
20	351659	00:34	355964	00:33	157312	00:14	161410	00:14
21	14036353	21:38	10859861	16:01	9499957	14:14	6524396	09:13
22	58206362	01:37:30	58214183	01:33:03	17312971	24:37	17319946	23:54
23	14490682	30:12	14490682	29:16	6400629	11:01	6400629	10:33
24	27475258	55:05	27475258	53:11	9801577	16:39	9801577	16:10
25	35282646	01:23:12	35502799	01:21:25	13030167	25:16	13232221	25:15
26	92228265	03:28:20	92228259	03:22:33	29432477	55:38	29432467	54:08

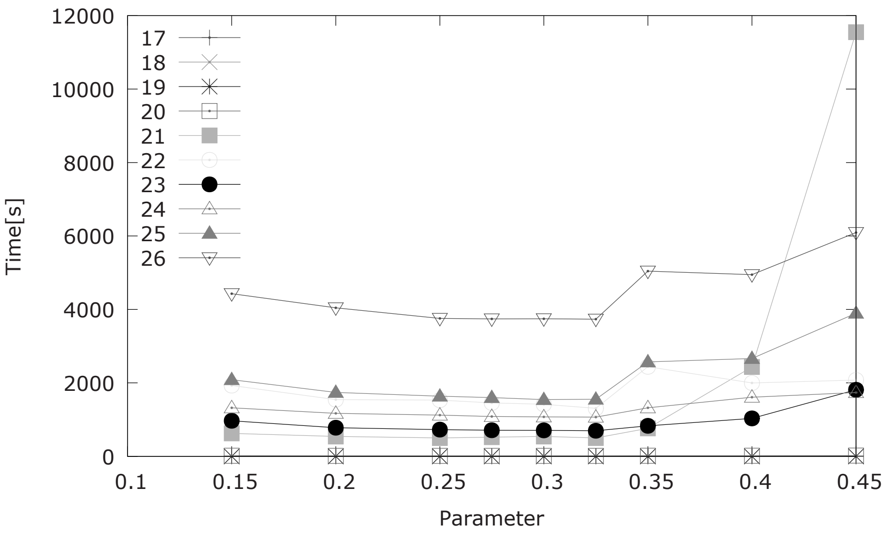


Fig. 10. Impact of Interval Size

4 Decomposition

We saw in Figure 4 that only rather few complete assignments of the X variables have to be tested to find an optimal solution for the problem. This suggests a further decomposition where we solve the first part of the problem, the orientation of the rectangles and the assignment of the X variables, without considering the Y variables at all. For this we only need the cumulative constraint for the X variables, the second cumulative and the non-overlapping constraints are stated only once the first subproblem has been solved, before we start the assignment of the Y variables. This will avoid waking these constraints repeatedly as the X variables are assigned. Given the number of nodes in the search tree, this can lead to significant savings. At the same time, we may loose important propagation due to these constraints, and therefore increase the size of the search tree of the subproblem. Experiments shows that this is not the case. Table 3 compares backtracking steps and execution times for the basic model without and with the redundant constraints and the decomposed model, also without and with the redundant constraints. The number of backtracks is the same for all problem instances except 10 and 12. This is a clear indication that the non-overlapping constraint and the second cumulative are not contributing anything to the search in the initial phase. The difference in execution times are solely due to avoiding unnecessary calls to these constraints in the first phase of the search. The savings are limited, but still worthwhile. In the last two columns (Decomposed Reified) we show results for a model where we replace the disjoint2 constraint of SICStus with reified sets of inequalities for each pair of rectangles.

Table 3. Decomposed Model Results

	Without Redundant Constraints				With Redundant Constraints					
	Basic Model		Decomposed Model		Basic Model		Decomposed Model		Decomposed Reified	
N	Back	Time	Back	Time	Back	Time	Back	Time	Back	Time
4	2	00:00	2	00:00	2	00:00	2	00:00	2	00:00
5	4	00:00	4	00:00	1	00:00	1	00:00	1	00:00
6	16	00:00	16	00:00	6	00:00	6	00:00	6	00:00
7	19	00:00	19	00:00	9	00:00	9	00:00	9	00:00
8	6	00:00	6	00:00	10	00:00	10	00:00	10	00:00
9	54	00:00	54	00:00	27	00:00	27	00:00	27	00:00
10	323	00:00	323	00:00	159	00:00	176	00:00	176	00:00
11	99	00:00	99	00:00	54	00:00	54	00:00	54	00:00
12	546	00:00	546	00:00	274	00:00	301	00:00	301	00:00
13	1900	00:00	1900	00:00	1040	00:00	1040	00:00	1040	00:00
14	2937	00:00	2937	00:00	1501	00:00	1501	00:00	1501	00:00
15	14440	00:00	14440	00:00	7617	00:00	7617	00:00	7617	00:00
16	15967	00:01	15967	00:00	3989	00:00	3989	00:00	3989	00:00
17	210878	00:14	210878	00:11	107611	00:07	107611	00:05	107611	00:06
18	9734	00:00	9734	00:00	5550	00:00	5550	00:00	5550	00:00
19	102235	00:08	102235	00:06	13690	00:01	13690	00:00	13690	00:00
20	351659	00:34	351659	00:28	161410	00:14	161410	00:11	161410	00:16
21	14036353	21:38	14036353	18:26	6524396	09:13	6524396	07:10	6524396	08:05
22	58206362	01:37:30	58206362	01:21:36	17319946	23:54	17319946	19:13	17319946	21:50
23	14490682	30:12	14490682	24:45	6400629	10:33	6400629	08:04	6400629	08:58
24	27475258	55:05	27475258	44:23	9801577	16:10	9801577	12:11	9801577	13:07
25	35282646	01:23:12	35282646	01:10:17	13232221	25:15	13232221	20:07	13232773	23:34
26	92228265	03:28:20	92228265	02:51:27	29432467	54:08	29432467	40:26	29432467	43:51

Table 4. Comparison with [12]

		Korf, Moffitt and Pollack		Pure		Redundant		Decomposition	
Size	Area	Nodes	Times	Back	Times	Back	Times	Back	Times
17	34×57	6,889,973	:07	210878	00:14	107611	00:07	107611	00:05
18	30×76	22,393,428	:26	9734	00:00	5550	00:00	5550	00:00
19	35×76	11,918,834	:11	102235	00:08	13690	00:01	13690	00:00
20	35×88	608,635,198	12:50	351659	00:34	161410	00:14	161410	00:11
21	39×91	792,197,287	23:21	14036353	21:38	6524396	09:13	6524396	07:10
22	44×92	4,544,585,807	1:49:32	58206362	01:37:30	17319946	23:54	17319946	19:13
23	40×115	32,222,677,089	15:06:56	14490682	30:12	6400629	10:33	6400629	08:04
24	40×130	41,976,042,836	18:39:34	27475258	55:05	9801577	16:10	9801577	12:11
25	45×130	557,540,262,189	12:11:30:32	35282646	01:23:12	13232221	25:15	13232221	20:07
26	42×156	-	-	92228265	03:28:20	29432467	54:08	29432467	40:26

This is a much weaker form of the non-overlapping constraint, but the results for the decomposed model are quite similar. Clearly, the non-overlapping constraint affects the performance only in a minor way.

Do we need the non-overlapping constraint at all? In [1] perfect placement problems were solved by creating all solutions for the cumulative projections in the x and y directions, and then combining them with a checker for the non-overlapping constraint. This will not be competitive for the almost square packing problem. We have seen above (Table 1) that some problem instances have millions of solutions. There will be a similar number of solutions for solving the x cumulative alone. Testing each of those solutions against all solutions of the y cumulative will be too expensive.

We can try to push the non-overlapping constraint to the overall end of the search, and use it only as a checker. This will mean that in the second part of the search we only use a single cumulative constraint in the y direction. Experiments indicate that this is not a competitive approach.

5 Comparison

In Table 4, we compare our results to those reported in [12]. Note that we only count backtracking steps, not the total number of nodes as in [12]. We can see that even our basic model dramatically outperforms Korf et al. for large problem sizes, and the difference increases when our further improvements are taken into account. But the differences are not uniform with the problem size, e.g. the differences for instances 21 and 22 are much smaller.

6 Conclusion

In this paper we have extended our previous results [18] for packing squares into the smallest enclosing rectangle to packing "almost squares", rectangles of sizes $n \times (n+1)$. For problem size N, this adds 2^N additional choices. Using the existing constraint model and carefully interleaving the assignment of X intervals and the orientation of the rectangles, we can solve the problem to optimality up to size 26, extending the previously best results [12] by one instance and obtaining a large reduction in execution time. For this problem type, a further decomposition of the problem into two phases is suggested by a visualization of the search tree. We first solve the problem in x direction with a single cumulative constraint, interleaving the orientation of the rectangles with the assignment of intervals to the X variables, before fixing the X values. Only then do we state the second cumulative constraint and the non-overlapping constraint. Together with some redundant constraints, this leads to a further reduction of the search space required.

Acknowledgment

The work reported here was supported by Science Foundation Ireland (Grant Number 05/IN/I886). The authors wish to thank Mats Carlsson, who provided the SICStus Prolog 4.0.4 used for the experiments.

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling problems. Journal of Mathematical and Computer Modelling 17(7), 57–73 (1993)
2. Beldiceanu, N., Bourreau, E., Simonis, H.: A note on perfect square placement, Prob009 in CSPLIB (1999)
3. Beldiceanu, N., Carlsson, M.: Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In: Walsh [20], pp. 377–391.
4. Beldiceanu, N., Carlsson, M., Poder, E.: New filtering for the cumulative constraint in the context of non-overlapping. In: CP-AI-OR 2008, Paris (May 2008)
5. Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C.: A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 180–194. Springer, Heidelberg (2007)
6. Beldiceanu, N., Contejean, E.: Introducing global constraints in CHIP. Journal of Mathematical and Computer Modelling 20(12), 97–123 (1994)
7. Beldiceanu, N., Guo, Q., Thiel, S.: Non-overlapping constraints between convex polytopes. In: Walsh [20], pp. 392–407
8. Huang, E., Korf, R.E.: New improvements in optimal rectangle packing. In: Boutilier, C. (ed.) IJCAI, pp. 511–516 (2009)
9. Huang, E., Korf, R.E.: Optimal rectangle packing on non-square benchmarks. In: Fox, M., Poole, D. (eds.) AAAI. AAAI Press, Menlo Park (2010)
10. Korf, R.E.: Optimal rectangle packing: Initial results. In: Giunchiglia, E., Muscettola, N., Nau, D.S. (eds.) ICAPS, pp. 287–295. AAAI, Menlo Park (2003)
11. Korf, R.E.: Optimal rectangle packing: New results. In: Zilberstein, S., Koehler, J., Koenig, S. (eds.) ICAPS, pp. 142–149. AAAI, Menlo Park (2004)
12. Korf, R., Moffitt, M., Pollack, M.: Optimal rectangle packing. Annals of Operations Research 179, 261–295 (2010), 10.1007/s10479-008-0463-6
13. MacHale, D.: The almost square problem. Personal Communication (2008)
14. Moffitt, M.D., Ng, A.N., Markov, I.L., Pollack, M.E.: Constraint-driven floorplan repair. In: Sentovich, E. (ed.) DAC, pp. 1103–1108. ACM, New York (2006)

15. Moffitt, M.D., Pollack, M.E.: Optimal rectangle packing: A meta-CSP approach. In: Long, D., Smith, S.F., Borrajo, D., McCluskey, L. (eds.) ICAPS, pp. 93–102. AAAI, Menlo Park (2006)
16. Roy, J.A., Markov, I.L.: Eco-system: Embracing the change in placement. In: ASP-DAC, pp. 147–152. IEEE, Los Alamitos (2007)
17. Simonis, H., Davern, P., Feldman, J., Mehta, D., Quesada, L., Carlsson, M.: A generic visualization platform for CP. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 460–474. Springer, Heidelberg (2010)
18. Simonis, H., O'Sullivan, B.: Search Strategies for Rectangle Packing. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 52–66. Springer, Heidelberg (2008)
19. Van Hentenryck, P.: Scheduling and packing in the constraint language cc(FD). In: Zweben, M., Fox, M. (eds.) Intelligent Scheduling. Morgan Kaufmann Publishers, San Francisco (1994)
20. Walsh, T. (ed.): CP 2001. LNCS, vol. 2239. Springer, Heidelberg (2001)

Efficient Planning of Substation Automation System Cables

Thanikesavan Sivanthi and Jan Poland

ABB Switzerland Ltd, Corporate Research,
Segelhofstrasse 1K, 5405, Baden-Dättwil, Aargau, Switzerland

Abstract. The manual selection and assignment of appropriate cables to the interconnections between the devices of a substation automation system is a major cost factor in substation automation system design. This paper discusses about the modeling of the substation automation system cable planning as an integer linear optimization problem to generate an efficient cable plan for substation automation systems.

1 Introduction

Cabling between different devices of a substation automation system (SAS) [1] is a major cost factor in the SAS design process. Usually computer aided design software is used to create the design templates of SAS devices and their interconnections. The design templates are then instantiated in a SAS project and the cables are manually assigned to the connections. The selection and assignment of cables to connections must follow certain engineering rules. This engineering process is usually time consuming and can cause engineering errors, thereby increasing the engineering cost. Apparently, the SAS cable planning is related to the well known bin packing problem. The SAS cable planning can be formulated as an integer linear optimization problem with the cable engineering rules expressed as a set of linear constraints and a cost objective for minimizing the total cable cost. This paper describes the formulation of SAS cable planning problem as an integer linear optimization problem and presents the results for some representative test cases. To the best of the authors' knowledge the work is the first of the kind to study SAS cable planning.

The paper is organized as follows. Section 2 presents an overview of the SAS cable planning process. Section 3 expresses the SAS cable planning problem as an integer linear optimization problem. The results obtained by solving the optimization problem using some solvers is presented in Section 4. Section 5 draws some conclusions of this work.

2 SAS Cable Planning

The SAS cable planning begins after the system design phase of a SAS project. The SAS cable planning is at present done manually by computer aided design

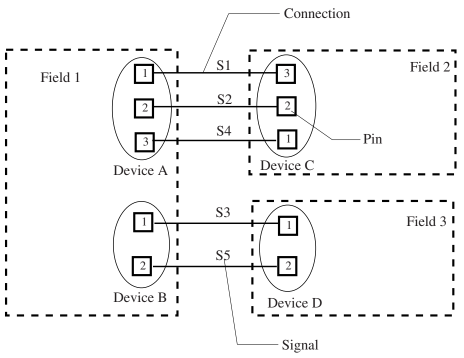


Fig. 1. Fields, devices and their interconnections

(CAD) engineers. The different design templates corresponding to the actual devices of a SAS are instantiated in one or more CAD jobs. Each job consists of one or more sheets and each sheet has fields which are logical groups of devices as shown in Figure 1. Moreover, a field corresponds to a physical assembly interface class e.g. Metering box, Protection cubicle etc. Each device has pins which are the physical interconnection interfaces of the device. A valid connection is a unique path between exactly two pins and every connection carries a physical signal. A signal can traverse over one or more connections. Each connection is assigned to exactly one of the conductors of a cable. The type of cables to which the connections are assigned is based on cable engineering rules. The cable engineering rules can be classified into two types, namely the cable rules and the signal rules. The *cable rules* specify the allowed cable types for a set of connections. It can also specify the number of spare conductors which must be left free in each instance of the allowed cable types. The *signal rules* specify restrictions on allocation of connections which carry signals that should not be allocated to the same cable or preferably allocated to the same cable. The current practice is to manually select and assign cables to connections according to the cable engineering rules. This procedure is time consuming and can cause engineering errors thereby increasing the engineering cost. In what follows is the formulation of the SAS system cable planning as an integer linear optimization problem with which a more efficient cable plan for SAS can be generated.

3 Integer Linear Program Formulation

The SAS cable planning problem is divided into sub problems where each sub problem considers connections between distinct set of field pairs within a given set of CAD jobs. The rationale behind this decomposition is that the cable plan should consider the physical assembly interface classes and should not mix connections between two different source or destination physical assembly interface classes in one cable. This is ensured by deriving a cable plan for each distinct field pairs.

Let $\mathcal{C} = \{1, 2, 3, \ldots, N\}$ represent the set of all connections between two field pairs, where N is the total number of connections, and $\mathcal{K} = \{1, 2, 3, \ldots, M\}$

represent the set of all cable types, where M is the total number of cable types in a sub problem. In a cable instance, there can be one or more connections and we refer to the connection with lowest index among all connections in the cable instance as the *leader* and the other connections as the *followers*. This implies that all connections except the first connection in $\mathcal{C}$ can either be a leader or follower. Moreover, based on the signal rules a set of connection pairs $\mathcal{X}$ can be derived where each $(i,\hat{\imath}) \in \mathcal{X}$ represents the connections i and $\hat{\imath}$ that must not be assigned to the same cable. Let $\bar{C}$ be the set of connection pairs $(i,\hat{\imath})$ where $i,\hat{\imath} \in \mathcal{C}, i > \hat{\imath}, (i,\hat{\imath}) \notin \mathcal{X}$. We introduce the following binary variable $X_{i,\hat{\imath}}$, where $(i,\hat{\imath}) \in \bar{C}$, which when true implies that connection i is a follower of a leader $\hat{\imath}$.

$$X_{i,\hat{\imath}} = 0 \text{ or } 1, \text{ where } (i,\hat{\imath}) \in \bar{C} \; . \tag{1}$$

Similarly, based on the cable rules a set of connection cable pairs $\mathcal{Y}$ can be derived where each $(i,j) \in \mathcal{Y}$ implies that cable type j is not allowed for connection i. Let $\bar{K}$ be the set of connection cable pairs (i,j), where $i \in \mathcal{C}, j \in \mathcal{K}, (i,j) \notin \mathcal{Y}$. We introduce the following binary variable $Y_{i,j}$, where $(i,j) \in \bar{K}$, which when true implies that the leader i is assigned to an instance of cable type j.

$$Y_{i,j} = 0 \text{ or } 1, \text{ where } (i,j) \in \bar{K} \; . \tag{2}$$

Table 1 illustrates all binary variables corresponding to the example shown in Figure 1 for the case with two cable types $K1$ and $K2$. It is assumed that connections $C1$ and $C3$ cannot be assigned to the same cable and $K1$ is not an allowed cable type for connection $C3$. As mentioned before all connections except the first connection, which must be a leader, can either be a leader or follower. This is ensured by the following constraint.

$$\sum_{(i,\hat{\imath}) \in \bar{C}} X_{i,\hat{\imath}} + \sum_{\substack{j \in \mathcal{K} \\ (i,j) \in \bar{K}}} Y_{i,j} = 1, \forall i \in \mathcal{C} \; . \tag{3}$$

A connection which is a leader in a cable cannot be a follower of a leader in another cable. This is expressed by the following constraint.

$$X_{i,\hat{\imath}} + \sum_{(\hat{\imath},i^*) \in \bar{C}} X_{\hat{\imath},i^*} \leq 1, \forall (i,\hat{\imath}) \in \bar{C} \; . \tag{4}$$

An implicit constraint of the cable planning problem is the capacity constraint which implies that the number of connections assigned to a cable must be less

Table 1. Binary variables corresponding to Figure 1 example

	$C1$	$C2$	$C3$	$C4$	$C5$	$K1$	$K2$
$C1$	-	-	-	-	-	$Y_{1,1}$	$Y_{1,2}$
$C2$	$X_{2,1}$	-	-	-	-	$Y_{2,1}$	$Y_{2,2}$
$C3$	-	$X_{3,2}$	-	-	-	-	$Y_{3,2}$
$C4$	$X_{4,1}$	$X_{4,2}$	$X_{4,3}$	-	-	$Y_{4,1}$	$Y_{4,2}$
$C5$	$X_{5,1}$	$X_{5,2}$	$X_{5,3}$	$X_{5,4}$	-	$Y_{5,1}$	$Y_{5,2}$

than the capacity requirement i.e. the total number of conductors in the cable minus the spare core requirement of the cable. Let U_j and S_j be the total number of conductors and the required spare core in cable type j, then the following equation expresses the capacity constraint. In this equation, if the connection $\hat{i}$ is a leader then the sum of all connections including the connection $\hat{i}$ and its followers is less than the capacity requirement of the cable type j to which $\hat{i}$ is assigned, otherwise the equation is by default satisfied.

$$1 + \sum_{(i,\hat{i}) \in \bar{C}} X_{i,\hat{i}} - \sum_{(\hat{i},i^*) \in \bar{C}} X_{\hat{i},i^*} \leq \sum_{(\hat{i},j) \in \bar{K}} (U_j - S_j) \cdot Y_{\hat{i},j}, \forall \hat{i} \in \mathcal{C} \, . \tag{5}$$

In addition the problem formulation needs the following constraint to avoid indirect pairing of connections i and i^* which have the same leader $\hat{i}$ but (i, i^*) is in $\mathcal{X}$.

$$X_{i,\hat{i}} + X_{i^*,\hat{i}} \leq 1, \forall (i,\hat{i}), (i^*,\hat{i}) \in \bar{C} \text{ where } i > i^*, (i,i^*) \notin \bar{C} \, . \tag{6}$$

Similarly, the following constraint prohibits a follower to choose a leader whose selected cable type is not one of the allowed cable types of the follower.

$$X_{i,\hat{i}} + Y_{\hat{i},j} \leq 1, \forall (i,\hat{i}) \in \bar{C}, j \in \mathcal{K} \text{ where } (\hat{i},j) \in \bar{K}, (i,j) \notin \bar{K} \, . \tag{7}$$

Finally, the sub problem may include a set of preferred allocation rules which specify that all connections carrying certain signals should preferably be assigned to the same cable. This is achieved by introducing a penalty cost in the objective function. The penalty cost will increase when not all connections of any preferred allocation rule have the same leader or when there exists more than one leader among the connections within any preferred allocation rule. The constraints related to preferred allocation rules are not expressed due to space limitation. The objective of the cable planning problem is then specified as

$$\textbf{minimize:} \sum_{(i,j) \in \bar{K}} M_j \cdot Y_{i,j} \, . \tag{8}$$

where M_j is the cost of cable type j. The optimization of the above problem results in a SAS cable plan with minimal total cable cost.

4 Results

In order to conduct a meaningful experiment, due to the lack of sufficient real sub-problem instances, we generated random sub problem instances with nine cable types. The number of connections N in each sub problem instance is varied from 10 to 50. Each cable type has a cable cost which is discrete uniformly distributed between 1 and 2 and has a total number of conductors which is discrete uniformly distributed between 1 and 5. Each connection is allowed to be assigned to M out of the nine cable types, where M is discrete uniformly distributed between 3 and 6. Furthermore, the number of connection pairs which

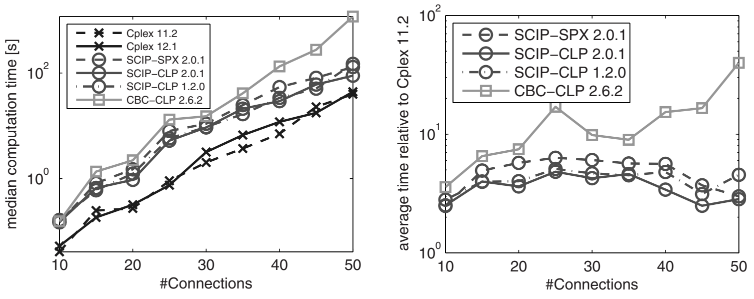


Fig. 2. Performance obtained with different solvers

cannot be assigned to the same cable is on average equal to $(N \cdot 2)/3$. It should be noted that the sub problem instances generated are harder than typical SAS cable planning sub problems. The instances are solved using different solvers and the results obtained are shown in Figure 2. The left plot shows the median computation time to obtain the optimal solution with some non-commercial solvers SCIP-SOPLEX [2] [3] [4], CBC [5] and commercial solver CPLEX [6]. The right plot shows the performance of the non-commercial solvers relative to CPLEX. It is observed that SCIP-CLP 2.0.1 which is on average 3.6 times slower than CPLEX scales well with increasing problem size unlike CBC-CLP 2.6.2 which scales poorly and is on average 13.9 times slower than CPLEX. The salient result of our experiment is that even the harder than typical instances are fairly easily solved, therefore the integer linear optimization formulation clearly offers time and cost efficient solution for SAS cable planning.

5 Conclusion

This paper presented the modeling of substation automation system cable planning as an integer optimization problem to generate a more efficient cable plan for substation automation systems. The results obtained for typical test cases show that the integer linear optimization formulation clearly offers time and cost efficient solution for the substation automation system cable planning.

References

1. Brand, K.-P., et al.: Substation Automation Handbook. Utility Automation Consulting (2003)
2. Achterberg, T.: SCIP: Solving Constraint Integer Programs. J. Math. Prog. Comp. 1(1), 1–41 (2009)
3. Achterberg, T.: Constraint Integer Programming, Technische Universität Berlin (2007)
4. SCIP Mixed Integer Programming Solver, `http://zibopt.zib.de`
5. CLP Linear Programming Solver, `https://projects.coin-or.org/Clp`
6. CPLEX Optimizer, `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer`

A New Algorithm for Linear and Integer Feasibility in Horn Constraints*

K. Subramani and James Worthington

LDCSEE,
West Virginia University,
Morgantown, WV
{ksmani,jworthing}@csee.wvu.edu

Abstract. In this paper, we detail a new algorithm for the problem of checking linear and integer feasibility of a system of Horn constraints. For certain special cases, the new algorithm is faster than the "Lifting Algorithm" described in [1]. Moreover, the new approach is based on different ideas and in fact exploits several properties of Horn constraint systems (HCS) which are not known to be part of the literature. In the case of constraints of bounded width (corresponding to "loosely coupled" systems), our algorithm can be modified to run in $O(n^3+m\cdot n+\frac{m\cdot n^2}{\log(\max(m,n))})$ time. Our main result establishes that checking the feasibility of an HCS can be reduced to three subproblems: negative-cost cycle detection in networks (NCCD), matrix-vector multiplication (MV), and the conversion of an HCS to a non-redundant set of difference constraints (H2D). The MV and NCCD problems have been extremely well-studied, and specialized, fast algorithms exist for relevant special cases. We have identified a new problem, H2D, which warrants future research, since improved algorithms for H2D could be implemented in our algorithm to decrease the running time.

1 Introduction

In this paper, we introduce a new algorithm for the problem of checking the feasibility of a conjunction of linear Horn constraints. This work builds upon our work in [1], wherein the first combinatorial algorithm for this problem was proposed. The algorithm we propose is based on substantially different ideas than those in [1] and in fact exploits a number of properties of Horn constraint systems (HCS) which are not known to be part of the literature. Essentially, we Turing-reduce the Horn constraint feasibility problem to three problems: negative-cost cycle detection (NCCD), matrix-vector multiplication (MV), and the conversion of a system of Horn constraints to a non-redundant system of difference constraints (H2D). The new algorithm may be either combinatorial or non-combinatorial, depending on the algorithms chosen for NCCD, MV, and H2D. Furthermore,

* This research was supported in part by the National Science Foundation through Award CCF-0827397.

the complexity of our algorithm is expressed in terms of the complexities of the NCCD, MV, and H2D problems. Accordingly, an improvement in the running times of algorithms for these three problems leads to an improvement in the running time of our algorithm. In the case of bounded-width constraints, the algorithm can be modified to run in time $O(n^3 + m \cdot n + \frac{m \cdot n^2}{\log(\max(m,n))})$. To the best of our knowledge, our algorithm is the fastest known in this case.

The main contributions of this paper are as follows:

(i) Design and analysis of a new algorithm for checking feasibility in Horn constraint systems, and
(ii) Extending the analysis to handle the case of Extended Horn constraints.

The rest of this paper is organized as follows: Section 2 formally specifies the problem under consideration. In Section 3, we discuss the motivation for our work as well as related approaches in the literature. Section 4 describes a technique by which an HCS can be converted into a difference constraint system (DCS), such that the infeasibility of the DCS guarantees the infeasibility of the HCS. In Section 5, we prove a lemma essential for proving the correctness of our algorithm. Section 6 presents the new algorithm for checking feasibility in an HCS. Section 7 examines the complexity of our algorithm. The techniques detailed in Section 6 are used in Section 8 to develop a new algorithm for a larger class of constraints called Extended Horn constraints. We conclude in Section 9 by summarizing our contributions and outlining avenues for future research.

2 Statement of Problem

Let

$$\begin{aligned} \mathbf{A} \cdot \vec{\mathbf{x}} &\geq \vec{\mathbf{b}} \\ \vec{\mathbf{x}} &\geq \vec{\mathbf{0}} \end{aligned} \tag{1}$$

denote a *polyhedral system* in which $\mathbf{A}$ is an $m \times n$ integral matrix, $\vec{\mathbf{b}}$ is an integral m-vector, and $\vec{\mathbf{x}} = [x_1, \ x_2, \ \ldots x_n]^{\mathrm{T}}$ is an rational n-vector.

Definition 1. *A polyhedral system is said to be a* Difference Constraint System *(DCS) if each row of* $\mathbf{A}$ *contains at most two non-zero entries with one of these entries being* 1 *and the other being* -1.

For instance, $x_1 - x_2 \geq 3$ is a difference constraint.

Definition 2. *A polyhedral system is said to be a* Horn Constraint System *(HCS) or a* Horn polyhedron *if*

(i) the entries of $\mathbf{A}$ *belong to the set* $\{0, 1, -1\}$,
(ii) each row of $\mathbf{A}$ *contains at most one positive entry.*

The matrix $\mathbf{A}$ *is said to satisfy the Horn structure.*

For instance, $x_1 - x_5 - x_7 \geq -3$ is a Horn constraint. A Horn system could include *absolute* constraints (also called *unary* constraints), i.e., constraints of the form

$x_1 \geq 5$ or $-x_2 \geq -6$. Note that a constraint such as $x_1 \geq 5$ can be replaced by the constraint $x'_1 \geq 0$, where $x'_1 = x_1 - 5$, and the resultant constraint system is feasible if and only if the original system is. Moreover, constraints of the form $-x_2 \geq -6$ can be ignored until after Algorithm 6.1 has executed; see Remark 1 below.

Definition 3. *A Horn system is said to be* standardized *if every row and every column of the defining matrix* $\mathbf{A}$ *has at least one positive entry and at least one negative entry.*

Assumption 1. *In the sequel, all Horn constraint systems are standardized (this assumption is justified in [1]).*

Horn constraints subsume difference constraints syntactically. An important distinction between the two constraint systems can be observed in their respective dual polyhedra. For instance, consider the linear program:

$$\begin{aligned} \text{minimize} \sum_{i=1}^{n} x_i & \\ \mathbf{A} \cdot \vec{\mathbf{x}} \geq \vec{\mathbf{b}} & \\ \vec{\mathbf{x}} \geq \vec{\mathbf{0}}. & \end{aligned} \tag{2}$$

Suppose $\vec{\mathbf{b}} = [1, 1, 1, 1]^{\mathrm{T}}$. If System (2) is a DCS, it is well-known that it can be represented by a constraint network $\mathbf{G}$ such that $\mathbf{G}$ has a simple, negative-cost cycle if and only if the DCS is infeasible [2]. Moreover, if System (2) is feasible, then the shortest path distances from a specified source comprise the optimal solution. In either case, the dual variables are binary, i.e., 0 or 1 [3].

However, suppose in System (2) that the matrix $\mathbf{A}$ is as follows:

$$\begin{bmatrix} -1 & 1 & 0 & 0 & 0 \\ -1 & -1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 1 & 0 \\ -1 & -1 & -1 & -1 & 1 \end{bmatrix}.$$

It can be verified that the optimal primal solution is $\vec{\mathbf{x}} = [0, 1, 2, 4, 8]^{\mathrm{T}}$, and the corresponding optimal (non-binary) dual solution is $\vec{\mathbf{y}} = [8, 4, 2, 1]^{\mathrm{T}}$.

With respect to System (1), we are interested in the following two questions:

(i) Is System (1) feasible? Observe that System (1) represents a set, viz., the set of points that satisfy the constraints defining System (1). This problem is called the Linear Feasibility (LF) problem.
(ii) Does System (1) enclose a lattice point? This problem is called the Integer Feasibility (IF) problem.

We make the following observations:

(i) For an HCS (DCS), the LF and IF problems coincide, as a consequence of the Lifting Algorithm discussed in [1].

(ii) The non-negativity constraints are clearly not required for a DCS. In [1], we showed that these constraints are not needed for an HCS either. In other words, given a constraint system of the form $\mathbf{A} \cdot \vec{\mathbf{x}} \geq \vec{\mathbf{b}}$, in which the matrix $\mathbf{A}$ satisfies the Horn structure, we can derive a new constraint system $\mathbf{A}' \cdot \vec{\mathbf{x}} \geq \vec{\mathbf{b}'}$, $\vec{\mathbf{x}} \geq \vec{\mathbf{0}}$, with $\mathbf{A}'$ satisfying the Horn structure. Therefore, we can assume without loss of generality that the HCS we consider has explicit non-negativity constraints.

As mentioned above, our algorithm may or may not be combinatorial, depending on how it is implemented. For the convenience of the reader, we recall the following definitions.

Definition 4. *An algorithm is said to run in* strongly polynomial time *if it satisfies the following three conditions:*

(i) Arithmetic operations are applied exclusively to integers.
(ii) The number of arithmetic operations is bounded by a polynomial in the number of integers in the input.
(iii) The space required by the algorithm is bounded by a polynomial in the size of the input.

Definition 5. *An algorithm which runs in strongly polynomial time is said to be* combinatorial *if the only arithmetic operations used are addition, subtraction, and multiplication.*

3 Motivation and Related Work

Horn constraint systems arise in a number of problem domains including constraint logic programming [4], econometrics [5], and program verification [6]. Our interest in Horn constraints arose from their application to *Abstract Interpretation*, which is a technique in program verification. Abstract Interpretation was introduced in [7] and remains a highly effective methodology to approximate the semantics of programs.

Associated with any computer program is its concrete semantics, i.e., the set of execution traces it may produce. In general, the concrete semantics is non-recursive. In fact, Rice's theorem implies that all non-trivial questions about the semantics of arbitrary programs are undecidable [8]. An abstract interpreter constructs conservative approximations of the semantics over a properly chosen domain. Static analysis over this chosen domain reveals interesting properties of the concrete semantics. The domains that are chosen can either be relational (where the relationships between program variables are taken into account) or non-relational (where the relationships between program variables are essentially ignored). The domain of intervals is a non-relational domain, whereas difference-bound matrices (DBMs) and convex polyhedra are examples of relational domains. Non-relational domains can be analyzed more efficiently, whereas the relational domains provide more precise information about the concrete semantics.

An abstract domain is characterized by a set of constraints belonging to a constraint class. For example, DBMs are represented by conjunctions of difference constraints; i.e., constraints of the form $x_i - x_j \le c_{ij}$. Polyhedral convex domains are represented by conjunctions of arbitrary linear constraints. Solution techniques for such constraint systems form the core of Satisfiability Modulo Theories (SMT) solvers, which are useful in certain program verification procedures [9,10]. These solvers are also part of procedures for bounded model checking of infinite state systems and test-case generation [11]. The *Octagon Domain* is represented by Unit Two Variable per Inequality (UTVPI) constraints, i.e., constraints of the form: $ax_i + bx_j \le c_{ij}; a, b \in \{-1, 1\}$. The literature documents a fair amount of work in the field of UTVPI constraints [6,12]. Horn constraints are more expressive than difference constraints; it can also be shown that a feasible UTVPI system can be expressed as a Horn system [13]. Thus, Horn systems can be thought of as an intermediate step between simple abstract domains represented by intervals and DBMs and the more complicated abstract domain represented by convex polyhedra.

Moreover, some uses of the techniques of Abstract Interpretation utilize linear constraints involving the integer variables in a program [14]. In general, only a few variables will occur in each constraint. If, given a system of constraints, the maximum number of variables occurring is k (and the system is Horn), then we are in the case of *bounded width* constraints. We can exploit this structure to speed up the algorithm in this paper (see Section 7.1 below).

Our algorithm makes essential use of the fact that any feasible HCS can be standardized into a system such that the corresponding polyhedron has a least element. Thus our constraints belong to the class of *min-closed constraints* studied in [15] and [16]. In [15], the constraints considered are very general and in fact can be subsets of D^n for an arbitrary set D. Our approach is much less general and exploits specific aspects of HCSs. In [16], the constraints are in fact linear constraints, but additional linear programs must be introduced because of the *recognition problem*, which asks whether a given system can be represented by certain types of constraints. We avoid this by first standardizing the input.

In [1], we proposed the *Lifting Algorithm*, the first combinatorial algorithm to check the feasibility of a HCS. This algorithm runs in time $O(m \cdot n^2)$ and is based on the following inference rule:

$$\frac{x_1 - x_2 - x_3 - \cdots - x_k \ge c \qquad \vec{\mathbf{x}} \ge \vec{\mathbf{0}}}{x_1 \ge c}$$

where n is the number of variables and m is the number of constraints. We organized the implications in a greedy fashion into a sequence of $n-1$ rounds and showed that in every round, one variable achieves its final value. Since each round can be implemented in $O(m \cdot n)$ time, the algorithm runs in $O(m \cdot n^2)$ time. Algorithm 6.1 below is based on an entirely different insight, viz., the reduction of feasibility checking for an HCS to the NCCD, MV, and H2D problems.

A linear program is said to be *combinatorial* if the entries in the constraint matrix $\mathbf{A}$ are polynomially bounded in the dimensions of $\mathbf{A}$. A strongly polynomial time algorithm for combinatorial linear programs is given in [17]. The running time is $O(n^5)$ for HCSs. We note that the algorithm in [17] is not combinatorial. Furthermore, while the algorithm runs in strongly polynomial time for Horn Systems, it is not strongly polynomial for Extended Horn Systems (Definition 8 below).

Depending on the implementations of NCCD, MV, and H2D used, our algorithm may or may not be combinatorial. Our reduction is advantageous from the standpoint of implementation. Well-tested libraries implementing algorithms for the NCCD and MV problems are widely available. This not only simplifies the implementation process, but also allows a certain degree of trust in the resulting code.

4 The Implied DCS of a HCS

Let $l_1\colon x_1 - x_2 - x_3 \geq 4$ denote a constraint of an HCS. Since all the variables are required to be non-negative, we must have

$$x_1 - x_2 \geq 4$$
$$x_1 - x_3 \geq 4.$$

In a similar fashion, every constraint of the HCS $P_1\colon \mathbf{A}\cdot\vec{\mathbf{x}} \geq \vec{\mathbf{b}},\ \vec{\mathbf{x}} \geq \vec{\mathbf{0}}$ implies a sequence of difference constraints. The conjunction of these constraints is a DCS denoted by $P_2\colon \mathbf{A}'\cdot\vec{\mathbf{x}} \geq \vec{\mathbf{b}}',\ \vec{\mathbf{x}} \geq \vec{\mathbf{0}}$. We say that P_2 is the *implied DCS* corresponding to the HCS P_1.

By the discussion above,

$$\{\vec{\mathbf{x}} : \mathbf{A}\cdot\vec{\mathbf{x}} \geq \vec{\mathbf{b}},\ \vec{\mathbf{x}} \geq \vec{\mathbf{0}}\} \subseteq \{\vec{\mathbf{x}} : \mathbf{A}'\cdot\vec{\mathbf{x}} \geq \vec{\mathbf{b}}',\ \vec{\mathbf{x}} \geq \vec{\mathbf{0}}\}.$$

Therefore, if P_2 is infeasible, then so is P_1. However, there exist simple examples showing that the feasibility of P_2 does not imply the feasibility of P_1.

We note that:

(i) The DCS P_2 has m' constraints and n variables and can be solved in $O(\mathcal{N})$ time, where $\mathcal{N}$ is the running time of an NCCD algorithm.
(ii) We can assume that $m' = O(n^2)$, since a DCS cannot have more than $O(n^2)$ non-redundant constraints. Moreover, m' is at most the number of (-1) entries in the matrix $\mathbf{A}$.

5 Least Elements of Horn Constraint Systems

For this section, we require a few concepts from [1].

Definition 6. *A* least element *of a polyhedron defined by* $\mathbf{A}\cdot\vec{\mathbf{x}} \geq \vec{\mathbf{b}}, \vec{\mathbf{x}} \geq \vec{\mathbf{0}}$ *is a vector* $\vec{\mathbf{z}} \geq \vec{\mathbf{0}}$ *such that*

(i) $\mathbf{A}\cdot\vec{\mathbf{z}} \geq \vec{\mathbf{b}}$ *and*
(ii) $(\forall \vec{\mathbf{x}} \geq \vec{\mathbf{0}})\ \mathbf{A}\cdot\vec{\mathbf{x}} \geq \vec{\mathbf{b}} \Rightarrow \vec{\mathbf{z}} \leq \vec{\mathbf{x}}$.

Observe that a polyhedron can have at most one least element.

Lemma 1 ([1]). *Let* $P_1 \colon \mathbf{A} \cdot \vec{\mathbf{x}} \geq \vec{\mathbf{b}}, \vec{\mathbf{x}} \geq \vec{\mathbf{0}}$ *be a feasible HCS. Then* P_1 *has a least element.*

Note that if P_1 has a least element, then this element can be obtained by minimizing the linear function $\vec{\mathbf{p}} \cdot \vec{\mathbf{x}}$ over P_1, where $\vec{\mathbf{p}} > \vec{\mathbf{0}}$ is an arbitrary vector. For simplicity, we set $\vec{\mathbf{p}} = \vec{\mathbf{1}}$. The least element can then be obtained by solving the following linear program:

$$\begin{aligned} \text{minimize} \sum_{i=1}^{n} x_i \\ \mathbf{A} \cdot \vec{\mathbf{x}} \geq \vec{\mathbf{b}} \\ \vec{\mathbf{x}} \geq \vec{\mathbf{0}}. \end{aligned} \tag{3}$$

The following lemma gives some information about least elements.

Lemma 2. *Let* $P_1 \colon \mathbf{A} \cdot \vec{\mathbf{x}} \geq \vec{\mathbf{b}}, \vec{\mathbf{x}} \geq \vec{\mathbf{0}}$ *be a feasible HCS. In the optimal solution,* $x_i = 0$ *for at least one* i.

Proof. Let $\vec{\mathbf{z}}$, an $n \times 1$ vector, denote the solution to the linear program (3) constructed from P_1. Suppose all entries of $\vec{\mathbf{z}}$ are strictly positive. Let Ψ denote the objective function $\sum_{i=1}^{n} x_i$. Let $k = \min\{z_1, z_2, \cdots, z_n\}$. Observe that $\vec{\mathbf{u}} = (\vec{\mathbf{z}} - k\vec{\mathbf{1}})$ is also feasible for P_1 (here we are using the Horn structure of $\mathbf{A}$), and $\vec{\mathbf{u}} < \vec{\mathbf{z}}$. Hence $\Psi(\vec{\mathbf{u}}) < \Psi(\vec{\mathbf{z}})$, which contradicts the optimality of $\vec{\mathbf{z}}$.

Lemma 3. *Let* P_1 *be a feasible HCS and* P_2 *its implied DCS. Let* $\vec{\mathbf{x_1}}$ *and* $\vec{\mathbf{x_2}}$ *be the least elements of* P_1 *and* P_2*, respectively. Then* $\vec{\mathbf{x_2}} \leq \vec{\mathbf{x_1}}$.

Proof. By the discussion in Section 4, the set of feasible solutions to P_1 is a subset of the set of feasible solutions to P_2.

For what follows, we assume familiarity with the Lifting Algorithm given in [1]. Given a feasible (standardized) HCS, the Lifting Algorithm returns its least element. Lemma 4 below gives some information about how variables attain their final values as the Lifting Algorithm executes.

Lemma 4 ([1]). *Let* P_1 *be a feasible HCS. If we apply one round of the Lifting Algorithm, i.e., lift each variable once, at least one variable attains its final value (its value in the least element).*

The following lemma is a refinement of Lemma 4.

Lemma 5. *Let*

$$\mathbf{A} \cdot \vec{\mathbf{x}} \geq \vec{\mathbf{b}} \tag{4}$$

$$\vec{\mathbf{x}} \geq \vec{\mathbf{0}} \tag{5}$$

be a feasible HCS whose least element contains at least one non-zero component. There exists a variable x_i *such that*

1. x_i achieves its final value in the first round of lifting, and
2. the final value of x_i is equal to a component of $\vec{\mathbf{b}}$, say b_k. The value b_k is the RHS of a constraint in which x_i occurs positively.

Proof. Let $x_1, x_2, \ldots, x_n$ be the variables of the HCS. We assume that the Lifting Algorithm processes the variables in this order. Let x_i be the least-numbered variable which achieves its final value in the first round. We claim that the final value of x_i is b_k, where b_k is the RHS of a constraint in which x_i occurs positively.

Let Y be the set of all variables which occur negatively in a constraint with RHS b_k and in which x_i occurs positively. Note that there may be multiple such constraints. We claim that no variable in Y is ever lifted. Let $y \in Y$. Once x_i is lifted for the first and only time, the HCS will contain a constraint of the form

$$x_i - \cdots - y - \cdots \geq 0.$$

If y is lifted after x_i is, the RHS of this constraint will become positive, and x_i will be lifted again, a contradiction. The only other possibility is that y is lifted before x_i. If this were the case, then since y can't be lifted again, this lifting must take y to its final value. This contradicts the fact that x_i is the least-numbered variable to be lifted to its final value.

Note that a variable which is lifted in the first round might be lifted by an amount which is not a component of $\vec{\mathbf{b}}$, even if that variable achieves its final value in the first round. The reason is that some other variable may have been lifted first, and every lifting of a variable changes $\vec{\mathbf{b}}$. Lemma 5 shows that this can not be the case for all of the variables which are lifted to their final values in the first round. Also observe that there could be a variable x_j which occurs negatively in a constraint in which x_i occurs positively such that x_j is lifted in later rounds. In this case, the right hand side of the constraint must remain non-positive (otherwise x_i would be lifted again).

We now prove a relationship between the least element of an HCS and the least element of its implied DCS. This is the crux of the correctness proof of Algorithm 6.1 in Section 6.1.

Lemma 6. *Let*

$$P_1 \colon \mathbf{A} \cdot \vec{\mathbf{x}} \geq \vec{\mathbf{b}} \tag{6}$$
$$\vec{\mathbf{x}} \geq \vec{\mathbf{0}} \tag{7}$$

be a standardized, feasible HCS whose least element contains at least one non-zero component. Let x_i be the variable guaranteed to exist by Lemma 5, and let b_k be the final value of x_i. Let P_2 be the implied DCS of P_1. In the least element of P_2, $x_i = b_k$.

Proof. In P_1, x_i must occur positively in a constraint with RHS b_k. Since x_i is lifted by b_k and the Lifting Algorithm is greedy, b_k must be the largest RHS of the constraints in which x_i occurs positively. Therefore b_k will be the largest RHS of the constraints in the implied DCS P_2 in which x_i occurs positively.

Therefore, when applying one round of the Lifting Algorithm to P_2, x_i will be lifted by at least b_k. In fact, it must be lifted by exactly b_k and never lifted again, since the least element of P_2 is less than or equal to the least element of P_1 by Lemma 3. The lemma follows by the correctness of the Lifting Algorithm.

6 The New Algorithm

In this section, we give an algorithm (Algorithm 6.1) for determining whether an HCS is feasible. If the HCS is feasible, Algorithm 6.1 returns the least element of the corresponding polyhedron. Algorithm 6.1 proceeds in stages. At each stage, it converts an HCS into its implied DCS. We refer to this conversion as H2D. Using an NCCD algorithm, it computes the least element of the implied DCS and then uses this element to compute a new HCS. Upon completion, Algorithm 6.1 will have either computed the least element of the original HCS or determined that it is infeasible.

6.1 Correctness

Lemma 7. *Let P_1 be a feasible HCS and let $\vec{\mathbf{x}}_\mathbf{1}$ be its least element. Let $\vec{\mathbf{y}}$ be a vector that satisfies the relation $\vec{\mathbf{y}} \le \vec{\mathbf{x}}_\mathbf{1}$. Consider the system:*

$$P_2\colon \mathbf{A}\cdot\vec{\mathbf{x}} \ge (\vec{\mathbf{b}} - \mathbf{A}\cdot\vec{\mathbf{y}})$$
$$\vec{\mathbf{x}} \ge \vec{\mathbf{0}}.$$

```
Function Horn-Check ($P_1\colon \mathbf{A}, \vec{\mathbf{b}}$)
 1: if ($\vec{\mathbf{b}} \le \vec{\mathbf{0}}$) then
 2:     assert("System is feasible").
 3:     return ($\vec{\mathbf{0}}$).
 4: end if
 5: Initialize $\vec{\mathbf{o}} = \vec{\mathbf{0}}$.
 6: Preprocess $\mathbf{A}$ as necessary (see discussion in Section 7.3).
 7: for ($i = 1$ to $(n-1)$) do
 8:     Replace $P_1$ with its implied DCS $P_2\colon \mathbf{A}' \cdot \vec{\mathbf{x}} \ge \vec{\mathbf{b}}',\ \mathbf{x} \ge \mathbf{0}$. (H2D)
 9:     Obtain the least element $\vec{\mathbf{z}}$ of $P_2$ using an NCCD algorithm.
10:     Set $\vec{\mathbf{o}} = \vec{\mathbf{o}} + \vec{\mathbf{z}}$.
11:     if ($\mathbf{A}\cdot\vec{\mathbf{z}} \ge \vec{\mathbf{b}}$) (MV) then
12:         assert("System is feasible")
13:         return ($\vec{\mathbf{o}}$).
14:     else
15:         Set $P_1$ to the HCS obtained by replacing $\vec{\mathbf{b}}$ with $\vec{\mathbf{b}} - \mathbf{A}\cdot\vec{\mathbf{z}}$.
            {Note that this changes $\vec{\mathbf{b}}$ and not $\mathbf{A}$. In essence, we have shifted the origin
            to the point $\vec{\mathbf{z}}$.}
16:     end if
17: end for
18: assert("System is infeasible").
```

Algorithm 6.1: The Algorithm

The system P_2 is feasible and its least element is $\vec{\mathbf{x}}_{\mathbf{1}} - \vec{\mathbf{y}}$.

Proof. Clearly $\vec{\mathbf{x}}_{\mathbf{1}} - \vec{\mathbf{y}}$ is a feasible solution to P_2. If $\vec{\mathbf{z}} < \vec{\mathbf{x}}_{\mathbf{1}} - \vec{\mathbf{y}}$ were the least element in P_2, then $\vec{\mathbf{z}}+\vec{\mathbf{y}} < \vec{\mathbf{x}}_{\mathbf{1}}$ would be feasible for P_1, contradicting minimality of $\vec{\mathbf{x}}_{\mathbf{1}}$ for P_1.

Theorem 1. *Algorithm 6.1 is correct.*

Proof. The algorithm terminates and returns either "feasible (with minimal element $\vec{\mathbf{o}}$)" or "infeasible" on any input. Observe that P_1 is feasible with least element $\vec{\mathbf{0}}$ if and only if all components of $\vec{\mathbf{b}}$ are non-positive. Hence the conditional in Line 1 guarantees that the algorithm returns "feasible" with least element $\vec{\mathbf{0}}$ if and only if P_1 is feasible with least element $\vec{\mathbf{0}}$.

If the algorithm returns "feasible" with non-zero least element $\vec{\mathbf{o}}$, then the conditional in Line 11 must be true after some number of iterations. Note that when the algorithm terminates, $\vec{\mathbf{z}}$ is the least element of the HCS for the current value of $\vec{\mathbf{b}}$: although $\vec{\mathbf{z}}$ is computed as the least element of the implied DCS, it must be feasible for the corresponding HCS in order for the conditional in Line 11 to be true. Moreover, the least element of the implied DCS is less than or equal to the least element of the HCS by Lemma 3. Replacing $\vec{\mathbf{b}}$ by $\vec{\mathbf{b}} - \mathbf{A} \cdot \vec{\mathbf{z}}$ is equivalent to performing the substitutions $x_i = x_i - z_i$ on the constraints in $\vec{\mathbf{A}} \cdot \vec{\mathbf{x}} \geq \vec{\mathbf{b}}$. These substitutions are not performed on the non-negativity constraints. Since $\vec{\mathbf{z}}$ must be less than the least element of the HCS in order for the substitutions to be performed, each substitution defines a bijection between feasible points of the HCS before the substitutions and feasible points of the HCS after the substitutions (no component of a feasible point becomes negative). Therefore, if the algorithm finds a feasible point of the HCS with an updated value of $\vec{\mathbf{b}}$, the original HCS P_1 is feasible. It remains to show that the returned value of $\vec{\mathbf{o}}$ is the least element of P_1. If not, the least element of P_1 could be mapped according to the (monotonic) substitutions performed and contradict the fact that when the conditional in Line 11 is true, the algorithm has found the least element to some HCS.

Finally, we show that Algorithm 6.1 returns the least element of P_1 if P_1 is feasible with at least one non-zero component in its least element. In this case, the algorithm will execute the body of the **for** loop on Line 7 at least once. At every iteration, the least element of the current HCS, $\vec{\mathbf{x}}_{\mathbf{1}}$, and the least element of its implied DCS, $\vec{\mathbf{y}}$, are equal in at least one component by Lemma 6. By Lemma 3, $\vec{\mathbf{y}} \leq \vec{\mathbf{x}}_{\mathbf{1}}$. Therefore we can apply Lemma 7 to the new system in which $\vec{\mathbf{b}}$ is replaced by $\vec{\mathbf{b}} - \mathbf{A} \cdot \vec{\mathbf{y}}$. Every iteration decreases the number of variables which only occur positively in constraints with positive RHSs (and which remain so for the rest of the execution). By Lemma 2, this can happen at most $n-1$ times. Therefore, if Algorithm 6.1 has not found a feasible solution after $n-1$ iterations, there is no feasible solution. The theorem follows.

Note that even if we do not compute the solution of the implied DCS (i.e., find the least element of P_2), but only do one cycle of the Lifting Algorithm on the implied DCS at each step, the same result holds by Lemma 6.

Remark 1. *Since Algorithm 6.1 finds the least element of the polyhedron corresponding to an HCS, we can ignore constraints of the form* $-x_2 \geq -6$ *until the algorithm has finished executing.*

7 Analysis

Let m be the number of constraints and n be the number of variables, so $\mathbf{A}$ is $m \times n$ and $\vec{\mathbf{b}}$ is $m \times 1$. Note that we can ignore Lines 1 – 6 when calculating the complexity of the Algorithm 6.1: Lines 1 – 5 require only $O(m)$ time and Line 6 is not worth doing if it will take more time to execute than the **for** loop.

The **for** loop on Lines 7 – 17 executes $O(n)$ times in the worst case. Lines 8, 9, and 11 are discussed below. The modifications to P_1 in Line 15 require $O(m)$ time since $\mathbf{A} \cdot \vec{\mathbf{z}}$ has already been computed in Line 11. In summary, and ignoring operations of low complexity, the time required for Algorithm 6.1 is:

$$O(n(\text{H2D} + \text{NCCD} + \text{MV})).$$

We have thus expressed the running time of Algorithm 6.1 in terms of the running times of three subproblems.

7.1 Analysis of H2D

As mentioned in Section 4, there are at most $O(n^2)$ many non-redundant difference constraints in the implied DCS of an HCS. The difference constraints can be stored in an $n \times n$ matrix $\mathbf{D}$, where $\mathbf{D}_{i,j}$ is equal to the largest entry of $\vec{\mathbf{b}}$ occurring to the right of $x_i - x_j$ in the implied DCS (i.e., the strongest difference constraint involving $x_i - x_j$).

In the worst case, each entry of $\vec{\mathbf{b}}$ bounds $n-1$ many constraints in the implied DCS, so we must examine each entry of $\mathbf{A}$ when computing $\mathbf{D}$. Therefore the complexity of computing $\mathbf{D}$ is $O(m \cdot n)$. Unfortunately, it is unclear how to avoid recomputing $\mathbf{D}$ at each iteration of the **for** loop. Note that $\vec{\mathbf{b}}$ changes during each iteration of the **for** loop. Suppose that

$$x_i - x_j \geq 4 (= b_k)$$

is the strongest difference constraint on $x_i - x_j$ at some iteration. At the next iteration, suppose b_k becomes 5. There is no guarantee that

$$x_i - x_j \geq 5$$

is now the strongest constraint on $x_i - x_j$ since their difference could be bounded by other entries of $\vec{\mathbf{b}}$ which have also changed since the last iteration.

However, for certain special HCSs, some preprocessing will allow us to do better than $O(m \cdot n)$, even though we will still compute $\mathbf{D}$ at each iteration.

Definition 7. *An HCS* P_1 *is said to be of* bounded width k *if every constraint contains at most* k *non-zero entries.*

For example, a DCS is an HCS of bounded width 2. For an HCS of bounded width k, we can create an $n \times n$ matrix $\mathbf{D}'$ to aid in the construction of $\mathbf{D}$. Each entry of $\mathbf{D}'$ will contain a list of pointers to the entries of $\vec{\mathbf{b}}$ which appear to the right of $x_i - x_j$ in the implied DCS. The matrix $\mathbf{D}'$ can be constructed in time $O(m \cdot n)$ and need only be constructed once. To construct $\mathbf{D}$, the algorithm examines each entry of $\mathbf{D}'$ and finds the smallest referenced entry of $\vec{\mathbf{b}}$. Since each entry of $\vec{\mathbf{b}}$ can appear in at most $k-1$ entries of $\mathbf{D}'$, $\mathbf{D}$ can be constructed from $\mathbf{D}'$ in $O(n^2 + (k-1) \cdot m) = O(n^2 + m)$, when k is a constant.

7.2 Analysis of NCCD

We must make a few modifications to ensure that an NCCD algorithm returns the least element of the DCS. These operations do not significantly affect the time required. Let m' denote the number of non-redundant constraints in the implied DCS of P_1. Note that $m' \le n^2$ (see Section 4) and that m' is constant over iterations of Algorithm 6.1.

We have three options for this subproblem:

(i) Using the Bellman-Ford algorithm for NCCD would result in this step being $O(m' \cdot n)$.
(ii) An $O(\sqrt{n} \cdot m' \cdot \log C)$ algorithm for the NCCD problem is given in [18]. Here C is the absolute value of the negative entry in $\vec{\mathbf{b}}$ of greatest magnitude.
(iii) By Lemma 6, we could also perform one round of the Lifting Algorithm on the DCS (i.e., lift each variable once). This requires $O(n^2)(= O(m'))$ time.

7.3 Analysis of MV

The naive algorithm for MV is $O(m \cdot n)$. However, we can do better by using the "Mailman Algorithm" given in [19].

Theorem 2 ([19]). *Let* $\mathbf{A}$ *be an* $m \times n$ *matrix. After an initial* $O(m \cdot n)$ *preprocessing step, subsequent multiplications of* $\mathbf{A}$ *with arbitrary* $m \times 1$ *vectors can be performed in* $O((\log |\Sigma|)\frac{m \cdot n}{\log(\max(m,n))})$, *where* $|\Sigma| \ge 2$ *is the number of distinct entries in* $\mathbf{A}$.

Since $\mathbf{A}$ is a standardized Horn matrix, the number of distinct entries is $|\{-1, 1, 0\}| = 3$. As mentioned above, the preprocessing does not affect the time bound of Algorithm 6.1.

7.4 Analysis of Algorithm 6.1

The above discussion implies that by judiciously choosing which algorithms to use for the three subproblems, we can achieve the following time bound for Algorithm 6.1:

$$O(n(m \cdot n + \min\{m' \cdot n, \sqrt{n} \cdot m' \cdot \log C, m'\} + \frac{m \cdot n}{\log(\max(m,n))})).$$

Note that a better algorithm for the H2D problem immediately yields an $o(m \cdot n^2)$ algorithm for checking the feasibility of an HCS (for various special cases). In the bounded width case, using the matrix $\mathbf{D}'$ described in Section 7.1 yields a time bound of $O(n^3 + m \cdot n + \frac{m \cdot n^2}{\log(\max(m,n))})$.

8 Extended Horn Constraints

Definition 8. *A polyhedral system*

$$\begin{aligned} \mathbf{A} \cdot \vec{\mathbf{x}} &\geq \vec{\mathbf{b}} \\ \vec{\mathbf{x}} &\geq \vec{\mathbf{0}} \end{aligned}$$

is said to be an Extended Difference Constraint System *(EDCS) if each row of* $\mathbf{A}$ *contains at most two non-zero entries with one of these entries being* 1 *and the other being an arbitrary negative integer.*

For instance, $x_1 - 2 \cdot x_2 \geq 3$ is an extended difference constraint.

Definition 9. *A polyhedral system*

$$\begin{aligned} \mathbf{A} \cdot \vec{\mathbf{x}} &\geq \vec{\mathbf{b}} \\ \vec{\mathbf{x}} &\geq \vec{\mathbf{0}} \end{aligned}$$

is said to be an Extended Horn System *(EHS) if each row of* $\mathbf{A}$ *contains at most one positive entry with the entry being* 1 *and the other entries being arbitrary non-positive integers.*

For instance, $x_1 - x_2 - 3 \cdot x_5 - 7 \cdot x_9 \geq -3$ is an extended Horn constraint.

We now argue that with a slight modification, Algorithm 6.1 works correctly even for Extended Horn Systems. We must make the following modifications to Algorithm 6.1:

(i) Line 8 must be modified to transform an EHS into a DCS. This can be accomplished by first changing all negative coefficients to -1, which transforms the EHS to an HCS, and then proceeding with the H2D procedure. For example, the extended Horn constraint

$$x_1 - x_2 - 3 \cdot x_5 - 7 \cdot x_9 \geq -3$$

becomes the following conjunction of difference constraints:

$$\begin{aligned} x_1 - x_2 &\geq -3 \\ x_1 - x_5 &\geq -3 \\ x_1 - x_7 &\geq -3. \end{aligned}$$

We also assume that the EHS is standardized, implying that each variable occurs at least once positively and at least once negatively.

(ii) In general, the Mailman Algorithm can no longer be used to improve the running time of MV, because we are not guaranteed a useful bound on the number of distinct entries in **A**. The running time of the modified algorithm is:

$$O(n(m \cdot n + \min\{m' \cdot n, \sqrt{n} \cdot m' \cdot \log C, n^2\} + m \cdot n)).$$

Note that the inability to use the Mailman Algorithm implies that the running time is at least $O(m \cdot n^2)$ even for bounded width EHS.

The correctness of this procedure follows from the discussion of the Lifting Algorithm applied to EHSs in [1].

9 Conclusion

In this paper, we introduced a new algorithm for checking the feasibility of Horn constraint systems. For systems of bounded width constraints, a natural subclass corresponding to "loosely coupled" systems, the running time can be improved to $O(n^3 + m \cdot n + \frac{m \cdot n^2}{\log(\max(m,n))})$. The algorithm is a Turing-reduction of HCS feasibility checking to the NCCD, MV, and H2D problems. The theory connecting HCS feasibility and the three aforementioned problems is developed in this paper and is, to the best of our knowledge, novel. The complexity of the new algorithm is expressed in terms of the complexities of these three problems. Advancement in techniques for the (slowest of the) latter problems results directly in an improvement to our algorithm. We also extended our analysis to include the case of Extended Horn constraints.

From our perspective, the following issues remain to be studied:

(i) A lower bound for checking the feasibility of an arbitrary HCS — All known algorithms for feasibility checking in a DCS run in $\Omega(m \cdot n)$ time. It would be interesting to obtain a bound of $\Omega(n \cdot \mathcal{N})$ for feasibility checking in an HCS, where $\mathcal{N}$ is the running time of a DCS algorithm. Such a lower bound would imply that the bound obtained in this paper is the best that we can hope for without a new asymptotic lower bound for feasibility checking of a DCS.

(ii) Arbitrary boolean combinations of Horn constraints — From the program verification perspective, satisfiability checking in formulas in which the atoms are Horn constraints is of enormous importance. Although this problem is trivially and strongly `NP-complete`, enumeration bounds have been obtained for the cases in which the atoms are either difference constraints or UTVPI constraints (see [20] for details).

Acknowledgements

This research was conducted primarily in the Carnegie Mellon University School of Computer Science, where the first author was an Invited Professor. We are grateful to R. Chandrasekaran for helpful discussions.

References

1. Chandrasekaran, R., Subramani, K.: A combinatorial algorithm for horn programs. In: Dong, Y., Du, D.-Z., Ibarra, O. (eds.) ISAAC 2009. LNCS, vol. 5878, pp. 1114–1123. Springer, Heidelberg (2009)
2. Nemhauser, G.L., Wolsey, L.A.: Integer and Combinatorial Optimization. John Wiley & Sons, New York (1999)
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press, Cambridge (2001)
4. Lever, J., Wallace, M., Richards, B.: Constraint logic programming for scheduling and planning. British Telecom Technology Journal 13, 73–80 (1995)
5. Truemper, K.: Personal communication (2003)
6. Miné, A.: The Octagon Abstract Domain. Higher-Order and Symbolic Computation 19, 31–100 (2006)
7. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
8. Homer, S., Selman, A.L.: Computability and Complexity Theory. Springer, Heidelberg (2001)
9. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N.: The ICS decision procedures for embedded deduction. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNCS (LNAI), vol. 3097, pp. 218–222. Springer, Heidelberg (2004)
10. Ford, J., Shankar, N.: Formal verification of a combination decision procedure. In: CADE, pp. 347–362 (2002)
11. Duterre, B., de Moura, L.: The yices smt solver. Technical report, SRI International (2006)
12. Harvey, W., Stuckey, P.J.: A unit two variable per inequality integer constraint solver for constraint logic programming. In: Proceedings of the 20th Australasian Computer Science Conference, pp. 102–111 (1997)
13. Lewis, H.R., Papadimitriou, C.H.: Symmetric space-bounded computation. Theor. Comput. Sci. 19, 161–187 (1982)
14. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1978, pp. 84–96. ACM, New York (1978)
15. Jeavons, P.G., Cooper, M.C.: Tractable constraints on ordered domains. Artif. Intell. 79, 327–339 (1995)
16. van Maaren, H., Dang, C.: Simplicial pivoting algorithms for a tractable class of integer programs. J. Comb. Optim. 6, 133–142 (2002)
17. Tardos, E.: A strongly polynomial algorithm to solve combinatorial linear programs. Oper. Res. 34, 250–256 (1986)
18. Goldberg, A.V.: Scaling algorithms for the shortest paths problem. SIAM Journal on Computing 24, 494–504 (1995)
19. Liberty, E., Zucker, S.W.: The mailman algorithm: A note on matrix–vector multiplication. Inf. Process. Lett. 109, 179–182 (2009)
20. Seshia, S.A., Subramani, K., Bryant, R.E.: On solving boolean combinations of UTVPI constraints. Journal on Satisfiability, Boolean Modeling and Computation 3, 67–90 (2007)

Timetable Edge Finding Filtering Algorithm for Discrete Cumulative Resources

Petr Vilím

IBM, V Parku 2294/4
148 00 Praha 4 - Chodov, Czech Republic
petr_vilim@cz.ibm.com

Abstract. Edge Finding filtering algorithm is one of the reasons why Constraint Programming is a successful approach in the scheduling domain. However edge finding for cumulative resources was never as successful as edge finding for disjunctive resources. This paper presents a new variant of the edge finding algorithm which improves filtering by taking into account minimum capacity profile - a data structure known from timetabling algorithm. In comparison with standard and extended edge finding algorithms the new algorithm is stronger but it may need more iterations in order to reach the fixpoint. Time complexity of the algorithm is $\mathcal{O}(n^2)$ where n is number of activities on the resource. We also propose further improvement of the filtering by incorporating some ideas from not-first/not-last and energetic reasoning algorithms. The filtering power of the algorithm is tested on computation of destructive lower bounds for 438 open RCPSP problems. For 169 of them we improve current best lower bound, in 9 cases backtrack free.

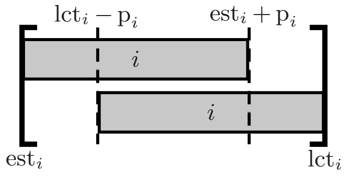
keywords: Constraint Programming, Scheduling, Discrete Cumulative Resource, Propagation.

1 Introduction

This paper focuses on discrete cumulative resource – an abstraction of manpower, electricity, machinery or any other (renewable) resource which is used to perform activities (tasks to be scheduled). Although the resource can be used by several activities simultaneously, total resource capacity used at any time cannot exceed capacity limit C. In a constraint programming framework we usually associate a constraint with each resource. The task of this resource constraint is to remove inconsistent values from temporal variables associated with activities. For the rest of the paper we will concentrate on propagation for a single resource.

A discrete cumulative resource is characterized by maximum capacity of the resource $C \in \mathbb{N}$ and a set T of n activities, $n = |T|$. Each activity has the following attributes:

– the earliest start time $\mathrm{est}_i \in \mathbb{N}$,
– the latest completion time (deadline) $\mathrm{lct}_i \in \mathbb{N}$,
– the processing time (duration) $\mathrm{p}_i \in \mathbb{N}$ (a constant),



$\mathrm{lct}_i - \mathrm{p}_i$
$\mathrm{est}_i + \mathrm{p}_i$
i
i
est_i
lct_i

Fig. 1. Two extreme positions of activity i. Regardless of its position, activity i always uses the resource during $[\mathrm{lct}_i - \mathrm{p}_i, \mathrm{est}_i + \mathrm{p}_i]$.

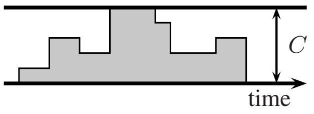


Fig. 2. Accumulated minimum capacity profile

– the required capacity $\mathrm{c}_i \in \mathbb{N}$ (a constant),
– and energy of the activity $\mathrm{e}_i = \mathrm{c}_i\,\mathrm{p}_i$ (a constant).

Activities are assumed to be non-preemptive: once the processing of activity i starts at time t then it must continue without preemption until $t + \mathrm{p}_i$. For each activity we maintain a decision variable for the start time of the activity with domain $[\mathrm{est}_i, \mathrm{lct}_i - \mathrm{p}_i]$. The aim of the resource constraint is to remove inconsistent values from this domain by increasing est_i and decreasing lct_i.

1.1 Related Works

This section reviews some of the existing techniques to propagate cumulative resource constraint.

Timetabling. The idea of timetabling is to look for activities i such that $\mathrm{lct}_i - \mathrm{p}_i < \mathrm{est}_i + \mathrm{p}_i$, see Figure 1. Such activities necessarily use the resource during interval $[\mathrm{lct}_i - \mathrm{p}_i,\ \mathrm{est}_i + \mathrm{p}_i]$. By aggregating these intervals we compute a minimum capacity profile (a timetable) which shows minimum resource usage over time (Figure 2). Typically, the minimum capacity profile is maintained during the search and used to detect infeasibility and also to update time bounds of activities. For more information see [6, chapters 3.3.1 and 2.1.1].

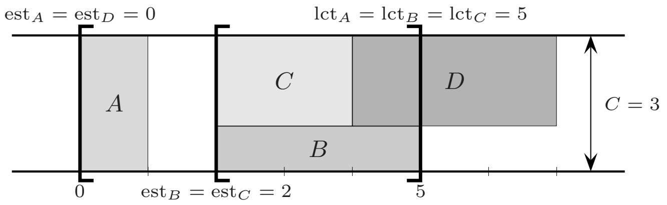
Edge Finding and Extended Edge Finding. Unlike timetabling, edge finding propagation is based on reasoning about a set of activities. Let us consider a set of activities $\Omega \subseteq T$. For Ω we define earliest starting time est_Ω, latest completion time lct_Ω and energy e_Ω as:

$$\mathrm{est}_\Omega = \min\{\mathrm{est}_i,\ i \in \Omega\} \qquad \mathrm{e}_\Omega = \sum_{i \in \Omega} \mathrm{e}_i$$
$$\mathrm{lct}_\Omega = \max\{\mathrm{lct}_i,\ i \in \Omega\}$$

The total energy available for Ω is $C\,(\mathrm{lct}_\Omega - \mathrm{est}_\Omega)$. If Ω requires more energy then there is no solution (this is called overload rule):

$$\forall \Omega \subseteq T : (\ \ C\,(\mathrm{lct}_\Omega - \mathrm{est}_\Omega) < \mathrm{e}_\Omega \ \ \Rightarrow \ \ \text{fail}\ \ )$$

Edge finding is also able to update temporal bounds of activities. Informally speaking, it checks whether scheduling an activity i at its earliest start time



$\text{est}_A = \text{est}_D = 0$
$\text{lct}_A = \text{lct}_B = \text{lct}_C = 5$
A
C
D
B
$C = 3$
0
$\text{est}_B = \text{est}_C = 2$
5

act.	est	lct	p	c
A	0	5	1	3
B	2	5	3	1
C	2	5	2	2
D	0	99	4	2

Fig. 3. An example from [11]: resource with capacity $C = 3$ and activities $\{A, B, C, D\}$. The table on the right summarizes attributes of these activities. Edge finding updates est_D from 0 to 4 because scheduling D at 0 would lead to overflow in interval $[0, 5]$.

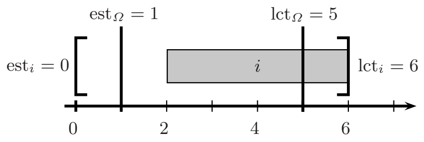
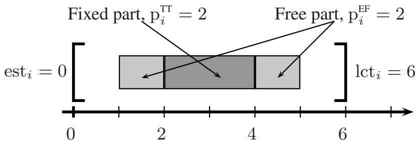


Fig. 4. Activity i with $\text{est}_i = 0$, $\text{lct}_i = 6$, $\text{p}_i = 4$ and $\text{c}_i = 1$. Activity i requires at least 3 energy units during $[1, 5]$. However edge finding does not take them into account.



Fixed part, $\text{p}_i^{\text{TT}} = 2$
Free part, $\text{p}_i^{\text{EF}} = 2$
$\text{est}_i = 0$
$\text{lct}_i = 6$
0
2
4
6

Fig. 5. Fixed and free parts of activity i with $\text{est}_i = 0$, $\text{lct}_i = 6$, $\text{p}_i = 4$

est_i would lead to overload as described above. If it is the case then est_i is updated. For an example see Figure 3, details are described in Section 4. There is also a symmetrical algorithm to update lct_i. Paper [5] provides algorithms for both standard and extended edge finding algorithms with time complexity $\mathcal{O}(kn^2)$ (k is number of distinct capacity demands $k = |\{c_i,\ i \in T\}|$). There is also a standard (not extended) edge finding algorithm with time complexity $\mathcal{O}(kn \log n)$ in [11]. There are also independent attempts to design edge finding algorithms with better time complexities by Roger Kameugne.

Energetic Reasoning. Edge finding is not perfectly accurate in computation of energy requirement during the interval $[\text{est}_\Omega, \text{lct}_\Omega]$. In particular, e_Ω does not take into account activities that only partially overlap with $[\text{est}_\Omega, \text{lct}_\Omega]$, see Figure 4. Energetic reasoning (also called CNP-ER) is able to take such partial overlaps into account. It also considers more intervals than only $[\text{est}_\Omega, \text{lct}_\Omega]$. Chapter 3.3.6 in [6] presents an algorithms to detect infeasibility and to update temporal bounds, their time complexities are $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$.

Dominance Relations. Chapter 4.2.4 in [6] contains a detailed analysis of dominance relations between different filtering algorithms for cumulative resources. The conclusion is that energetic reasoning is stronger than both timetabling and edge finding. However edge finding does not dominate timetabling and vice versa. Therefore edge finding can be improved by taking into account the timetable as we suggest in this paper. Energetic reasoning is still stronger than the proposed algorithm but slower (time complexity $\mathcal{O}(n^3)$ versus $\mathcal{O}(n^2)$). What we propose is a good trade off between filtering power and speed.

Note that for practical reasons the proposed algorithm is not designed to subsume the timetable algorithm: it is faster to do some propagation by the timetable algorithm.

2 TimeTable Edge Finding

The idea is to improve computation of the energy consumed during $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ using information from the timetable. Let us consider again the example on Figure 4. From timetabling point of view, activity i contributes to the minimum capacity profile by 2 energy units on interval $[2, 4]$. By using timetable in the edge finding we will be able to take the 2 energy units into account and therefore improve filtering. Note that it is still less than the 3 energy units detected by energetic reasoning but often the values are the same.

In order to use timetable we split each activity into two parts: *fixed* part (counted in the timetable) and remaining *free* part. See Figures 1 and 5. The split is done in the following way. For each activity i we compute its fixed duration $\mathrm{p}_i^{\mathrm{TT}}$ and free duration $\mathrm{p}_i^{\mathrm{EF}}$ as:

$$\mathrm{p}_i^{\mathrm{TT}} = \max(0,\ \mathrm{est}_i + \mathrm{p}_i - (\mathrm{lct}_i - \mathrm{p}_i)) \qquad\qquad \mathrm{p}_i^{\mathrm{EF}} = \mathrm{p}_i - \mathrm{p}_i^{\mathrm{TT}}$$

Fixed and free parts are handled differently:

Fixed Part. If $\mathrm{p}^{\mathrm{TT}} = 0$ then the fixed part is empty and we ignore it. Otherwise we increase the minimum consumption level in the timetable on interval $[\mathrm{lct}_i - \mathrm{p}_i, \mathrm{est}_i + \mathrm{p}_i]$ by c_i.

Free Part. Empty free parts ($\mathrm{p}_i^{\mathrm{EF}} = 0$) are ignored. Activities with non-empty free parts form a set $T^{\mathrm{EF}} = \{i,\ i \in T\ \&\ \mathrm{p}^{\mathrm{EF}} > 0\}$. We also define the energy of the free part as $\mathrm{e}_i^{\mathrm{EF}} = \mathrm{c}_i\,\mathrm{p}_i^{\mathrm{EF}}$. We can consider a free part of activity i as a separate activity with the same earliest start time est_i and latest completion time lct_i as the original activity i. However, unlike the original activity, the free part is preemptive: it can (and must) be suspended during $[\mathrm{lct}_i - \mathrm{p}_i, \mathrm{est}_i + \mathrm{p}_i]$.

2.1 Timetable

Timetable records the minimum capacity consumption at each time point t. In other words this data structure represents a function $\mathrm{TT}(t)$ such that $\mathrm{TT}(t)$ is the sum of capacities of all fixed parts which overlap time t (Figure 2).

For the presented algorithm we need to know how much energy is stored in the timetable for interval $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$. To compute it we introduce functions $\mathrm{ttAfterEst}(i)$ and $\mathrm{ttAfterLct}(i)$, they compute the total energy stored in the timetable after est_i and after lct_i:

$$\mathrm{ttAfterEst}(i) = \sum_{t \in \mathbb{N} \wedge t \geq \mathrm{est}_i} \mathrm{TT}(t) \qquad \mathrm{ttAfterLct}(i) = \sum_{t \in \mathbb{N} \wedge t \geq \mathrm{lct}_i} \mathrm{TT}(t)$$

Algorithm 1. Overload Checking

```
for $b \in T^{EF}$ do begin
  $eEF := 0$;
  for $a \in T^{EF}$ in non-increasing order by $\mathrm{est}_a$ do
    if $\mathrm{lct}_a \le \mathrm{lct}_b$ then begin
      $eEF := eEF + e_a^{EF}$;
      if $C(\mathrm{lct}_b - \mathrm{est}_a) < eEF + $ttAfterEst$[a] - $ttAfterLct$[b]$ then
        fail;
    end;
end;
```

Let us consider set $\Omega \subseteq T$ and activities $a, b \in \Omega$ such that $\mathrm{est}_a = \mathrm{est}_\Omega$ and $\mathrm{lct}_b = \mathrm{lct}_\Omega$. Then the energy stored in the timetable for interval $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ is $\mathrm{ttAfterEst}(a) - \mathrm{ttAfterLct}(b)$. For simplicity we define also:

$$\mathrm{ttAfterEst}(\Omega) = \mathrm{ttAfterEst}(a) \;\text{ where } a \in \Omega \text{ and } \mathrm{est}_a = \mathrm{est}_\Omega$$
$$\mathrm{ttAfterLct}(\Omega) = \mathrm{ttAfterLct}(b) \;\text{ where } b \in \Omega \text{ and } \mathrm{lct}_b = \mathrm{lct}_\Omega$$

Functions $\mathrm{ttAfterEst}(i)$ and $\mathrm{ttAfterLct}(i)$ can be computed from function $\mathrm{TT}(t)$ in $\mathcal{O}(n \log n)$ as follows. We sort activities into four lists: according to $\mathrm{est}_i + \mathrm{p}_i$, $\mathrm{lct}_i - \mathrm{p}_i$, est_i and lct_i. Then we sweep over all these events in anti-chronological order. During the sweep we maintain total energy stored in the timetable after the current event. The result of the computation is stored in arrays `ttAfterEst` and `ttAfterLct`.

2.2 Overload Checking

We start by the algorithm for checking infeasibility. Let's consider set $\Omega \subseteq T^{EF}$. Energy of free parts of activities in Ω is $e_\Omega^{EF} = \sum_{i \in \Omega} e_i^{EF}$. Therefore minimum energy consumption by both fixed and free parts during $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ is $e_\Omega^{EF} + \mathrm{ttAfterEst}(\Omega) - \mathrm{ttAfterLct}(\Omega)$. However energy available in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ is $C(\mathrm{lct}_\Omega - \mathrm{est}_\Omega)$. Therefore remaining energy "reserve" in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ is:

$$\mathrm{reserve}(\Omega) = C(\mathrm{lct}_\Omega - \mathrm{est}_\Omega) - \left(e_\Omega^{EF} + \mathrm{ttAfterEst}(\Omega) - \mathrm{ttAfterLct}(\Omega)\right)$$

If the reserve is negative then the problem is infeasible:

$$\forall \Omega \subseteq T^{EF} : \; (\,\mathrm{reserve}(\Omega) < 0 \;\; \Rightarrow \mathrm{fail})$$

Clearly it is not necessary to check all sets $\Omega \subseteq T^{EF}$. Let us consider two sets Ω_1 and Ω_2 such that:

$$\mathrm{est}_{\Omega_1} = \mathrm{est}_{\Omega_2} \qquad \mathrm{lct}_{\Omega_1} = \mathrm{lct}_{\Omega_2} \qquad e_{\Omega_1}^{EF} > e_{\Omega_2}^{EF}$$

Then it is enough to check only set Ω_1. Therefore the rule must be checked only for sets Ω with the following property:

$$\forall i \in T^{EF} : \mathrm{est}_i \ge \mathrm{est}_\Omega \;\&\; \mathrm{lct}_i \le \mathrm{lct}_\Omega \;\Rightarrow\; i \in \Omega$$

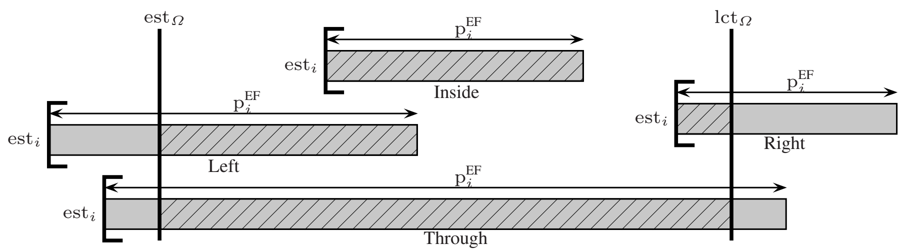


Fig. 6. Different relative positions of activity $i \in T^{\mathrm{EF}}$ and set Ω such that scheduling i on est_i increases energy consumption in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$. The increase is marked by hatched lines.

Table 1. Additional consumption by activity i in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ when scheduled on est_i

Position	**Characterization**	**Additional consumption**
Inside	$\mathrm{est}_\Omega \le \mathrm{est}_i$ & $\mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}} \le \mathrm{lct}_\Omega$	$\mathrm{e}_i^{\mathrm{EF}}$
Right	$\mathrm{est}_\Omega < \mathrm{est}_i$ & $\mathrm{lct}_\Omega < \mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}}$	$\mathrm{c}_i(\mathrm{lct}_\Omega - \mathrm{est}_i)$
Left	$\mathrm{est}_i < \mathrm{est}_\Omega < \mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}} < \mathrm{lct}_\Omega$	$\mathrm{c}_i(\mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}} - \mathrm{est}_\Omega)$
Through	$\mathrm{est}_i \le \mathrm{est}_\Omega$ & $\mathrm{lct}_\Omega \le \mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}}$	$\mathrm{c}_i(\mathrm{lct}_\Omega - \mathrm{est}_\Omega)$
Out	Otherwise	0

Such sets Ω are traditionally called task intervals.

Algorithm 1 checks infeasibility by the rule above. The idea is to pick a task $b \in T^{\mathrm{EF}}$ and iterate over sets Ω such that $\mathrm{lct}_\Omega = \mathrm{lct}_b$. Time complexity of the algorithm is $\mathcal{O}(n^2)$.

2.3 Time Bound Adjustment Rule

Value est_i is invalid if scheduling activity i on est_i would cause overload as described above. In this case est_i can be updated. To check such potential overloads, we need to compute how much additional energy activity $i \in T^{\mathrm{EF}}$ would require during $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ if i is scheduled on est_i. If $i \in \Omega$ then the whole energy of i is already counted so we concentrate only on the case $i \notin \Omega$. For $i \notin \Omega$ we distinguish four relative positions of Ω and free part of i such that scheduling the free part of i on est_i increases energy consumption in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$. See Figure 6 and Table 1.

Let function $\mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i)$ denote additional energy consumption by activity i in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ as defined by Table 1. Scheduling activity i on est_i causes overload with set Ω if:

$$\mathrm{reserve}(\Omega) < \mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i)$$

In this case current est_i can be updated. In the following we will show how to compute this update.

We start by computation of the maximum duration that activity i can spend inside $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$. This duration has two parts:

1. **Mandatory part:** the part of i which is in the timetable and which overlaps with $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$. It cannot move and therefore it must stay inside $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$.
2. **Optional part:** this is the maximum of free duration $\mathrm{p}_i^{\mathrm{EF}}$ which can still fit inside $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ considering $\mathrm{reserve}(\Omega)$.

The mandatory part is intersection of intervals $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ and $[\mathrm{lct}_i - \mathrm{p}_i, \mathrm{est}_i + \mathrm{p}_i]$. Its length is:

$$\begin{aligned}&\mathrm{mandatoryIn}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i) = \\ &\qquad = \max\left(0, \min\left(\mathrm{lct}_\Omega, \mathrm{est}_i + \mathrm{p}_i\right) - \max\left(\mathrm{est}_\Omega, \mathrm{lct}_i - \mathrm{p}_i\right)\right)\end{aligned}$$

Maximum length of the optional part is:

$$\mathrm{maxAddIn}(\Omega, i) = \left\lfloor \frac{\mathrm{reserve}(\Omega)}{c_i} \right\rfloor$$

The remaining duration of i must be spent after lct_Ω. Therefore est_i can be updated to the following new value:

$$\mathrm{est}_i := \mathrm{lct}_\Omega - \mathrm{mandatoryIn}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i) - \mathrm{maxAddIn}(\Omega, i)$$

The full propagation rule is:

$$\begin{aligned}&\forall \Omega \subset T^{\mathrm{EF}},\ \forall i \in T^{\mathrm{EF}} \setminus \Omega : \mathrm{reserve}(\Omega) < \mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i) \quad \Rightarrow \\ &\qquad \mathrm{est}_i := \mathrm{lct}_\Omega - \mathrm{mandatoryIn}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i) - \mathrm{maxAddIn}(\Omega, i)\end{aligned}$$

There is quite a big difference between the rule above and standard edge finding (or extended edge finding) propagation rule. Standard edge finding tries to find best subset $\Omega' \subseteq \Omega$ to immediately achieve the best update of est_i. The rule above fixes only potential overflow with Ω. Unlike standard edge finding it doesn't notice that the new est_i can be still invalid with respect to some $\Omega' \subset \Omega$. The proposed rule simply assumes that the algorithm will run again and if there is still a problem with the new est_i then it will be updated again. In this respect it may take more iterations for this algorithm to reach the fixpoint. In practice it is not a big problem as discussed later in section 2.5.

2.4 Time Bound Adjustment Algorithm

First of all, notice that to achieve maximum propagation it is enough to concentrate on sets Ω in the form of task intervals. The reason is the same as for the overload rule.

The idea of the algorithm is to iterate over sets Ω in the form of task intervals in the same way as the overload algorithm does. For each set Ω we also compute maximum value of $\mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i)$ over all $i \in T^{\mathrm{EF}} \setminus \Omega$. Let ι be activity i such that $\mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i)$ is maximal. We distinguish two cases:

1. If $\mathrm{reserve}(\Omega) \geq \mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, \iota)$ then activity ι cannot be updated by Ω. And because $\mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, \iota)$ is maximal, Ω cannot update any other activity $i \in T^{\mathrm{EF}} \setminus \Omega$ neither. Thus we can continue with the next set Ω.

2. If reserve(Ω) $<$ add($\text{est}_\Omega, \text{lct}_\Omega, \iota$) then the algorithm updates est_ι. There could be more activities than only ι which could be updated by Ω. However the algorithm does not update them, they will be updated in the next iteration of the algorithm. This is the second reason why the algorithm needs more iterations to reach the fixpoint. This issue is discussed in Section 2.5.

It remains to show how to maintain ι during the algorithm. The value of function add($\text{est}_\Omega, \text{lct}_\Omega, i$) depends on the relative position of i and Ω as described in Table 1. To maintain the maximum of add($\text{est}_\Omega, \text{lct}_\Omega, i$) the algorithm is split into three phases, each phase deals only with some relative positions. In the following we describe each phase separately. For simplicity we will assume in the description that there are no duplicates in est_i and lct_i. However the algorithm is sound even in case of duplicates.

Inside and Right. For Inside and Right positions we iterate over activities a and b the same way as we do in the Algorithm 1 for overload checking (lines 3–7 in Algorithm 2). That is, in outer loop we iterate over activity b (in arbitrary order) and in the inner loop we iterate over activity a in non-increasing order by est_a. If $\text{lct}_a \leq \text{lct}_b$ then activities a and b define set $\Omega = \{j,\ j \in T^{\text{EF}}\ \&\ \text{est}_a \leq \text{est}_j\ \&\ \text{lct}_j \leq \text{lct}_b\}$, its energy $\text{e}^{\text{EF}}_\Omega$ is stored in variable eEF.

The set of activities in Inside or Right position with Ω is $I = \{i,\ i \in T^{\text{EF}}\ \&\ \text{est}_\Omega \leq \text{est}_i\ \&\ \text{lct}_\Omega < \text{lct}_i\}$. Therefore as we iterate over activities a, each activity a is either put into Ω (if $\text{lct}_a \leq \text{lct}_b$, line 9) or put into I (if $\text{lct}_a > \text{lct}_b$). Each time when i is added into I we have to recompute ι. According to Table 1 value add($\text{est}_\Omega, \text{lct}_\Omega, i$) for positions Inside and Right can be computed as $\min\left(\text{e}^{\text{EF}}_i, \text{c}_i\left(\text{lct}_b - \text{est}_i\right)\right)$. Notice that this value does not depend at all on activity a. This justifies computation of ι on lines 10–11.

Finally, if reserve(Ω) is less then add($\text{est}_\Omega, \text{lct}_\Omega, \iota$) then est_ι is updated (lines 13 and 14).

Through. Again there are two nested cycles over activities a and b which define set $\Omega = \{j,\ j \in T^{\text{EF}}\ \&\ \text{est}_a \leq \text{est}_j\ \&\ \text{lct}_j \leq \text{lct}_b\}$. Energy $\text{e}^{\text{EF}}_\Omega$ is stored in variable eEF. However in comparison with the previous phase we iterate over activities a in reverse order. That is, we gradually remove activities a from Ω.

As we iterate over activities a we first check whether $a \in \Omega$ (line 21). If $a \in \Omega$ then we remove it from Ω (line 25) but just before the removal we check whether ι can be updated by Ω (lines 22–24). Furthermore if $\text{est}_a + \text{p}^{\text{EF}}_a \geq \text{lct}_b$ (line 27) then activity a is in Through position with all following sets Ω. In this case ι needs to be recomputed. According to Table 1 value add($\text{est}_\Omega, \text{lct}_\Omega, i$) for case Through is $\text{c}_i(\text{lct}_\Omega - \text{est}_\Omega)$. Therefore the maximum value of add($\text{est}_\Omega, \text{lct}_\Omega, i$) over all activities in Through position with Ω is achieved by the activity with the maximum capacity (lines 27–28).

Left. One more time, activities a and b define set Ω such that $\text{est}_\Omega = \text{est}_a$ and $\text{lct}_\Omega = \text{lct}_b$; energy $\text{e}^{\text{EF}}_\Omega$ is stored in variable eEF. However this time the outer cycle is over variable a in arbitrary order (line 32) and inner cycle is over variable b in non-decreasing order of est_b (line 36). For each a we start with the

Algorithm 2. Adjustments of est_i

```
for $i \in T^{\mathrm{EF}}$ do
   $\mathrm{est}'_i := \mathrm{est}_i$;
for $b \in T^{\mathrm{EF}}$ do begin
   // Cases “Inside” and “Right”
   $\mathrm{eEF} := 0$;
   $\iota := -1$;
   for $a \in T^{\mathrm{EF}}$ in non-increasing order by $\mathrm{est}_a$ do begin
      if $\mathrm{lct}_a \le \mathrm{lct}_b$ then
         $\mathrm{eEF} := \mathrm{eEF} + \mathrm{e}^{\mathrm{EF}}_a$;
      else if $\iota = -1$ or $\min(\mathrm{e}^{\mathrm{EF}}_a, \mathrm{c}_a(\mathrm{lct}_b - \mathrm{est}_a)) > \min(\mathrm{e}^{\mathrm{EF}}_\iota, \mathrm{c}_\iota(\mathrm{lct}_b - \mathrm{est}_\iota))$
         then $\iota := a$;
      $\mathrm{reserve} := C(\mathrm{lct}_b - \mathrm{est}_a) - \mathrm{eEF} - (\mathtt{ttAfterEst}[a] - \mathtt{ttAfterLct}[b])$;
      if $\iota \ne -1$ and $\mathrm{reserve} < \min(\mathrm{e}^{\mathrm{EF}}_\iota, \mathrm{c}_\iota(\mathrm{lct}_b - \mathrm{est}_\iota))$ then
         $\mathrm{est}'_\iota := \max(\mathrm{est}'_\iota, \mathrm{lct}_b - \mathrm{mandatoryIn}(\mathrm{est}_a, \mathrm{lct}_b, \iota) - \lfloor \mathrm{reserve}/\mathrm{c}_\iota \rfloor)$;
   end;
   // Case “Through”
   $\iota := -1$;
   for $a \in T^{\mathrm{EF}}$ in non-decreasing order by $\mathrm{est}_a$,
         break ties by non-increasing $\mathrm{est}_a + \mathrm{p}^{\mathrm{EF}}_a$
   do begin
      if $\mathrm{lct}_a \le \mathrm{lct}_b$ then begin
         $\mathrm{reserve} := C(\mathrm{lct}_b - \mathrm{est}_a) - \mathrm{eEF} - (\mathtt{ttAfterEst}[a] - \mathtt{ttAfterLct}[b])$;
         if $\iota \ne -1$ and $\mathrm{reserve} < \mathrm{c}_\iota(\mathrm{lct}_b - \mathrm{est}_a)$ then
            $\mathrm{est}'_\iota := \max(\mathrm{est}'_\iota, \mathrm{lct}_b - \mathrm{mandatoryIn}(\mathrm{est}_a, \mathrm{lct}_b, \iota) - \lfloor \mathrm{reserve}/\mathrm{c}_\iota \rfloor)$;
         $\mathrm{eEF} := \mathrm{eEF} - \mathrm{e}^{\mathrm{EF}}_a$;
      end;
      if $\mathrm{est}_a + \mathrm{p}^{\mathrm{EF}}_a \ge \mathrm{lct}_b$ and $(\iota = -1$ or $\mathrm{c}_a > \mathrm{c}_\iota)$ then
         $\iota := a$;
   end;
end;
// Case “Left”
for $a \in T^{\mathrm{EF}}$ do begin
   $\mathrm{eEF} := 0$;
   $\iota := -1$;
   $\mathrm{Q} :=$ queue of activities $i \in T^{\mathrm{EF}}$ sorted by non-decreasing $\mathrm{est}_i + \mathrm{p}^{\mathrm{EF}}_i$;
   for $b \in T^{\mathrm{EF}}$ in non-decreasing order by $\mathrm{est}_b$ do
      if $\mathrm{est}_a \le \mathrm{est}_b$ then begin
         $\mathrm{eEF} := \mathrm{eEF} + \mathrm{e}^{\mathrm{EF}}_b$;
         while $\mathrm{est}_{\mathrm{Q.top}} + \mathrm{p}^{\mathrm{EF}}_{\mathrm{Q.top}} < \mathrm{lct}_b$ do begin
            $i := \mathrm{Q.dequeue}$;
            if $\mathrm{est}_i < \mathrm{est}_a$ and $\mathrm{est}_a < \mathrm{est}_i + \mathrm{p}^{\mathrm{EF}}_i$ and
                  ( $\iota = -1$ or $\mathrm{c}_i(\mathrm{est}_i + \mathrm{p}^{\mathrm{EF}}_i - \mathrm{est}_a) > \mathrm{c}_\iota(\mathrm{est}_\iota + \mathrm{p}^{\mathrm{EF}}_\iota - \mathrm{est}_a)$ )
               then $\iota := i$;
         end;
         $\mathrm{reserve} := C(\mathrm{lct}_b - \mathrm{est}_a) - \mathrm{eEF} - (\mathtt{ttAfterEst}[a] - \mathtt{ttAfterLct}[b])$;
         if $\iota \ne -1$ and $\mathrm{reserve} < \mathrm{c}_\iota(\mathrm{est}_\iota + \mathrm{p}^{\mathrm{EF}}_\iota - \mathrm{est}_a)$ then
            $\mathrm{est}'_\iota := \max(\mathrm{est}'_\iota, \mathrm{lct}_b - \mathrm{mandatoryIn}(\mathrm{est}_a, \mathrm{lct}_b, \iota) - \lfloor \mathrm{reserve}/\mathrm{c}_\iota \rfloor)$;
      end;
end;
for $i \in T^{\mathrm{EF}}$ do
   $\mathrm{est}_i := \mathrm{est}'_i$;
```

empty set Ω (line 33) and we gradually add into Ω activities b (line 38) such that $\mathrm{est}_a \leq \mathrm{est}_b$.

Activities which are in the Left position with Ω are $I = \{i,\ i \in T^{\mathrm{EF}}\ \&\ \mathrm{est}_i < \mathrm{est}_\Omega < \mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}} < \mathrm{lct}_\Omega\}$. As we iterate over b, value $\mathrm{lct}_\Omega = \mathrm{lct}_b$ is increasing and set I is growing: there are more activities i fulfilling condition $\mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}} < \mathrm{lct}_\Omega$. To enumerate all activities i as they are added into I we create a queue Q of all activities sorted by $\mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}}$ (line 35). Each time we change lct_Ω by adding b into Ω we also enumerate all activities i which newly fulfill condition $\mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}} < \mathrm{lct}_\Omega$ (lines 39–40). These activities are candidates to be added into I. If they really belong to I (line 41) then we check whether i is better than ι (line 42). Note that for position Left $\mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i) = \mathrm{c}_i(\mathrm{est}_i + \mathrm{p}_i^{\mathrm{EF}} - \mathrm{est}_\Omega)$, see Table 1. Finally, after each addition of b into Ω we check whether ι can be updated by Ω (lines 45–47).

2.5 Time Complexity

Time complexity of Algorithm 2 is $\mathcal{O}(n^2)$: there are nested cycles over variables a and b with max n iterations each. The cycle on lines 39–44 is executed at most n times for each a because each time i is removed from queue Q.

As explained before, the Algorithm 2 does not make all updates by the propagation rule in one run because it updates only activity ι with the maximum potential overload. The remaining activities are updated in the next run(s). Furthermore it does not look over all subsets $\Omega' \subseteq \Omega$ to find the best possible update as standard edge finding algorithm does. Therefore it is not possible to directly compare time complexity $\mathcal{O}(n^2)$ of Algorithm 2 with time complexities $\mathcal{O}(kn^2)$ and $\mathcal{O}(kn \log n)$ of algorithms [5, 11].

Nevertheless, we believe that additional iterations needed to reach the fixpoint are not such a big disadvantage. There are several important aspects:

1. Even if the algorithm would fix all potential overflows with the current bounds, new bounds may generate new potential overflows (especially after applying symmetrical algorithm to update lct_i). Therefore such an algorithm would not be idempotent anyway – it would be still necessary to repeat the algorithm until the fixpoint is found.
2. In practice most of the time the edge finding algorithm runs without changing any bound. For benchmarks in section 5, the algorithm changes some est_i only in circa 30% of cases. Therefore it pays off to tune the algorithm for the case where it does not propagate. Probability of two updates in two consecutive runs is only $30\% \times 30\% = 9\%$. Maybe we can save some of these 9% of runs, but it could slow down of the remaining 91% of runs.
3. For future research, it may be interesting to look for an algorithm with output-sensitive time complexity such as $\mathcal{O}(n^2 + ln)$ where l is the number of changes done.
4. This is not the first propagation algorithm which using this “lazy” approach, see for example the not-first/not-last algorithm in [10].

2.6 Symmetry

Algorithm 2 updates values est_i. An algorithm to update lct_i is symmetrical. The symmetry is to consider time going backwards. It is possible to use the algorithm for est_i to do updates also on lct_i, the idea is to feed the algorithm with symmetrical data about activities:

$$\mathrm{inputEst}_i = -\mathrm{lct}_i \qquad\qquad \mathrm{inputLct}_i = -\mathrm{est}_i$$

The resulting bounds have to be also interpreted symmetrically.

3 Improvements

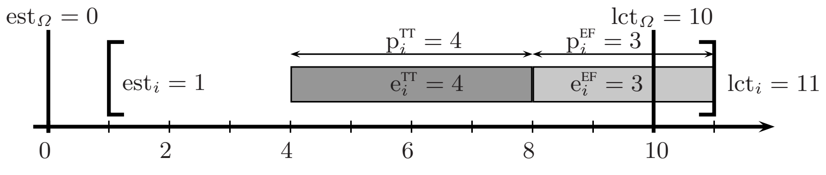
This section describes further improvements of algorithms 1 and 2 by incorporating some ideas from energetic reasoning and not-first/not-last.

3.1 Improved Time Bound Adjustments

As explained before, est_i can be updated if scheduling i at current est_i would cause overload in some interval $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$. In this case there are some activities which must be finished in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ before i can start. In particular, at least one of these activities must end before i can start. So the new est_i must be bigger than minimum $\mathrm{est}_j + \mathrm{p}_j$ over all activities j which contributes to reserve(Ω). This new lower bound for est_i can be sometimes better than the one computed by the Algorithm 2. This idea is similar to the not-first propagation rule [7].

Note that the activity which ends before i in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ does not have to be from set Ω. It could be also one of the activities which contributes to timetable in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$. Furthermore, in the following section we will consider even more activities in the computation of reserve(Ω).

As activities j come from different sources, computation of minimum $\mathrm{est}_j + \mathrm{p}_j$ can slow down the algorithm. Simpler but less accurate alternative is to precompute for each activity a minimum $\mathrm{est}_j + \mathrm{p}_j$ over all $j \in T$ such that $\mathrm{est}_a < \mathrm{est}_j + \mathrm{p}_j$. Then each time we compute est'_i we can use this precomputed value (for current a) as another lower bound. This approach is used for experimental results reported at the end of this paper.



$\mathrm{est}_\Omega = 0$
$\mathrm{lct}_\Omega = 10$
$\mathrm{p}_i^{\mathrm{TT}} = 4$
$\mathrm{p}_i^{\mathrm{EF}} = 3$
$\mathrm{est}_i = 1$
$\mathrm{e}_i^{\mathrm{TT}} = 4$
$\mathrm{e}_i^{\mathrm{EF}} = 3$
$\mathrm{lct}_i = 11$
0
2
4
6
8
10

Fig. 7. An example for better counting of energy consumption in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$

Algorithm 3. Improved Overload Checking

```
1  for $b \in T^{EF}$ do begin
2    eEF := 0;
3    for $a \in T^{EF}$ in non-increasing order by $\mathrm{est}_a$ do begin
4      if $\mathrm{lct}_a \le \mathrm{lct}_b$ then
5        eEF := $\mathrm{eEF} + \mathrm{e}_a^{EF}$;
6      else
7        eEF := $\mathrm{eEF} + \mathrm{c}_a \max(0, \mathrm{lct}_b - (\mathrm{lct}_a - \mathrm{p}_a^{EF}))$;
8      if $C(\mathrm{lct}_b - \mathrm{est}_a) < \mathrm{eEF} + \texttt{ttAfterEst}[a] - \texttt{ttAfterLct}[b]$ then
9        fail;
10   end;
11 end;
```

3.2 Improved Computation of Energy Consumption

Let us consider an example from Figure 7. What we see there is set Ω with $\mathrm{est}_\Omega = 0, \mathrm{lct}_\Omega = 10$ and activity $i \notin \Omega$ with $\mathrm{est}_i = 1, \mathrm{lct}_i = 11, \mathrm{p}_i = 7$ and $\mathrm{c}_i = 1$. Activity i has both fixed and free parts: $\mathrm{p}_i^{TT} = 4$, $\mathrm{e}_i^{TT} = 4$, $\mathrm{p}_i^{EF} = 3$, $\mathrm{e}_i^{EF} = 3$. Timetable edge finding knows from timetable that at least $\mathrm{e}_i^{TT} = 4$ energy units of i must be executed during $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$. However as we can see in Figure 7 there are at least 6 energy units of i which must be executed during $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$. So in this example timetable edge finding does better energy computation than standard edge finding (4 energy units versus 0) however it is still less accurate than energetic reasoning (6 energy units). In this section we show how to improve energy computation in timetable edge finding to get also 6 energy units. It is a step towards energetic reasoning, however only in limited cases.

Let's generalize the situation from Figure 7. If i is in Right or Inside position with Ω then we can further increase energy consumption in $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ by:

$$\mathrm{sureIn}(\Omega, i) = \mathrm{c}_i \max(0, \mathrm{lct}_\Omega - (\mathrm{lct}_i - \mathrm{p}_i^{EF}))$$

Algorithm 3 shows how to take $\mathrm{sureIn}(\Omega, i)$ into account in the overload checking algorithm. When activity a is not added into Ω then it is in Inside or Right position with all future sets Ω with $\mathrm{lct}_\Omega = \mathrm{lct}_b$. As $\mathrm{sureIn}(\Omega, i)$ does not depend on est_Ω we can add $\mathrm{sureIn}(\Omega, a)$ to variable eEF (line 7).

Improved energy consumption can be also taken into account in Algorithm 2. We do not present the modified algorithm in this paper. The general idea is to first store improved values eEF for all pairs of activities a and b in a two dimensional array. Then we can use this array in the update algorithm instead of eEF variable (be careful about duplicates in est_a and lct_b). To avoid counting energy $\mathrm{sureIn}(\Omega, i)$ twice for positions Inside and Right, it is necessary to decrease $\mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i)$ by $\mathrm{sureIn}(\Omega, i)$ (lines 10 and 13) and also increase $\mathrm{mandatoryIn}(\mathrm{est}_a, \mathrm{lct}_b, \iota)$ by $\mathrm{sureIn}(\Omega, i)$ at line 14.

4 Comparison with Standard and Extended Edge Finding

This section compares propagation power of timetable edge finding (without improvements from Section 3) with standard and extended edge finding algorithms.

Standard and extended edge finding propagation rules have two parts. First part considers a set $\Omega \subset T$ and an activity $i \in T \setminus \Omega$. If one of the following two conditions holds:

edge finding (EF):

$$C\left(\mathrm{lct}_\Omega - \mathrm{est}_{\Omega\cup\{i\}}\right) < \mathrm{e}_\Omega + \mathrm{e}_i$$

extended edge finding (EEF):

$$\mathrm{est}_i < \mathrm{est}_\Omega < \mathrm{est}_i + \mathrm{p}_i \quad \& \quad C\left(\mathrm{lct}_\Omega - \mathrm{est}_\Omega\right) < \mathrm{e}_\Omega + \mathrm{c}_i\left(\mathrm{est}_i + \mathrm{p}_i - \mathrm{est}_\Omega\right)$$

then i must end after lct_Ω (i.e. $\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_\Omega$) otherwise there is no solution. However both algorithms search for a better update of est_i. In particular they enumerate all subsets Ω' of Ω to find the best possible update:

$$\mathrm{est}_i := \max\left(\mathrm{est}_i, \max_{\substack{\Omega'\subseteq\Omega \\ \mathrm{rest}(\Omega',\mathrm{c}_i)>0}} \mathrm{est}_{\Omega'} + \left\lceil \frac{\mathrm{rest}\left(\Omega', \mathrm{c}_i\right)}{\mathrm{c}_i} \right\rceil\right) \tag{UPD}$$

where $\mathrm{rest}\left(\Omega', \mathrm{c}_i\right) = \mathrm{e}_{\Omega'} - (C - \mathrm{c}_i)\left(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'}\right)$

To compare the filtering power of the rules above with timetable edge finding it is best to concentrate on fixpoints[1]. In the following we prove that when timetable edge finding reaches a fixpoint then neither standard nor extended edge finding can propagate anything. We begin by the following lemma:

Lemma 1. *If (EF) or (EEF) holds and timetable edge finding reached a fixpoint (i.e. it cannot propagate any more) then* $\mathrm{est}_i + \mathrm{p}_i > \mathrm{lct}_\Omega$.

Proof. By contradiction: we will assume that $\mathrm{est}_i + \mathrm{p}_i \leq \mathrm{lct}_\Omega$. There are 3 cases:

1. **(EEF) holds.** In this case $\mathrm{est}_i < \mathrm{est}_\Omega < \mathrm{est}_i + \mathrm{p}_i \leq \mathrm{lct}_\Omega$. Let us consider total energy used by i and Ω in interval $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ if i is scheduled on est_i. Without timetable we can estimate it as:

$$\mathrm{e}_\Omega + \mathrm{c}_i(\mathrm{est}_i + \mathrm{p}_i - \mathrm{est}_\Omega)$$

With timetable we can make a better estimation:

$$\mathrm{e}_\Omega^{\mathrm{EF}} + \mathrm{ttAfterEst}(\Omega) - \mathrm{ttAfterLct}(\Omega) + \mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i)$$

Computation with timetable is more precise therefore:

$$\begin{aligned} &\mathrm{e}_\Omega + \mathrm{c}_i(\mathrm{est}_i + \mathrm{p}_i - \mathrm{est}_\Omega) \leq \\ &\qquad \leq \mathrm{e}_\Omega^{\mathrm{EF}} + \mathrm{ttAfterEst}(\Omega) - \mathrm{ttAfterLct}(\Omega) + \mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i) \end{aligned}$$

[1] Note that rules (EF) and (EEF) together are monotonic, therefore by Domain Reduction Theorem [2] their repetitive application (in arbitrary order) leads to a unique fixpoint. Similarly, propagation rule behind timetable edge finding algorithm (without improvements from Section 3) is also monotonic, therefore its repetitive application also leads to a unique fixpoint.

This together with (EEF) results in $\mathrm{reserve}(\Omega) < \mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i)$. That contradicts the assumption that timetable edge finding cannot propagate.

2. **(EF) holds and $\mathbf{est_i \geq est_\Omega}$.** In this case if i is scheduled on est_i then its energy contribution to $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ is e_i (remember that we are assuming for contradiction that $\mathrm{est}_i + \mathrm{p}_i \leq \mathrm{lct}_\Omega$). Again, this contribution is counted in timetable and in $\mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i)$. Therefore:

$$\mathrm{e}_i \leq \mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i) + \mathrm{ttAfterEst}(\Omega) - \mathrm{ttAfterLct}(\Omega)$$

   Together with (EF) it gives the contradiction $\mathrm{reserve}(\Omega) < \mathrm{add}(\mathrm{est}_\Omega, \mathrm{lct}_\Omega, i)$.

3. **(EF) holds and $\mathbf{est_i < est_\Omega}$.** Then condition (EF) can be rewritten as:

$$\begin{aligned}
C\,(\mathrm{lct}_\Omega - \mathrm{est}_i) &< \mathrm{e}_\Omega + \mathrm{e}_i && \text{because } \mathrm{est}_i < \mathrm{est}_\Omega\\
C\,(\mathrm{lct}_\Omega - \mathrm{est}_\Omega) &< \mathrm{e}_\Omega + \mathrm{e}_i - C\,(\mathrm{est}_\Omega - \mathrm{est}_i) &&\\
C\,(\mathrm{lct}_\Omega - \mathrm{est}_\Omega) &< \mathrm{e}_\Omega + \mathrm{e}_i - \mathrm{c}_i\,(\mathrm{est}_\Omega - \mathrm{est}_i) && \text{because } C > \mathrm{c}_i\\
C\,(\mathrm{lct}_\Omega - \mathrm{est}_\Omega) &< \mathrm{e}_\Omega + \mathrm{c}_i\,(\mathrm{est}_i + \mathrm{p}_i - \mathrm{est}_\Omega) && \text{because } \mathrm{e}_i = \mathrm{c}_i\,\mathrm{p}_i
\end{aligned}$$

   When i is scheduled on est_i then its energy contribution to $[\mathrm{est}_\Omega, \mathrm{lct}_\Omega]$ is $\mathrm{c}_i\,(\mathrm{est}_i + \mathrm{p}_i - \mathrm{est}_\Omega)$. Together with the inequality above it means that timetable edge finding propagates what is a contradiction. □

Proposition 1. *When timetable edge finding reaches a fixpoint then both standard and extended edge finding cannot propagate anything.*

Proof. By contradiction: we will assume that timetable edge finding reached a fixpoint, i.e. there is no potential overload, however standard or extended edge finding can propagate, i.e. there is i, Ω and Ω', $\Omega' \subseteq \Omega$, such that (EF) or (EEF) holds and (UPD) improves est_i using Ω'.

We are going to prove (for the contradiction) that timetable edge finding propagates for Ω' and i. By Lemma 1 we know that $\mathrm{lct}_\Omega < \mathrm{est}_i + \mathrm{p}_i$. And because Ω' is a subset of Ω we conclude $\mathrm{lct}_{\Omega'} \leq \mathrm{lct}_\Omega < \mathrm{est}_i + \mathrm{p}_i$. We distinguish two cases:

1. $\mathbf{est_i \leq est_{\Omega'}}$**.** In this case when i is scheduled on est_i then its energy contribution to $[\mathrm{est}_{\Omega'}, \mathrm{lct}_{\Omega'}]$ is $\mathrm{c}_i\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'})$. However because Ω' is used by (UPD) to improve est_i it must hold:

$$\begin{aligned}
0 &< \mathrm{rest}(\Omega', \mathrm{c}_i)\\
0 &< \mathrm{e}_{\Omega'} - (C - \mathrm{c}_i)\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'})\\
C\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'}) &< \mathrm{e}_{\Omega'} + \mathrm{c}_i\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'})
\end{aligned}$$

   Therefore timetable edge finding propagates what is a contradiction.

2. $\mathbf{est_i > est_{\Omega'}}$**.** In this case if i is scheduled on est_i then its energy contribution to $[\mathrm{est}_{\Omega'}, \mathrm{lct}_{\Omega'}]$ is $\mathrm{c}_i\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_i)$. However because (UPD) improves est_i:

$$\begin{aligned}
\mathrm{est}_i &< \mathrm{est}_{\Omega'} + \left\lceil \frac{\mathrm{rest}(\Omega', \mathrm{c}_i)}{\mathrm{c}_i} \right\rceil \\
\mathrm{est}_i &< \mathrm{est}_{\Omega'} + \left\lceil \frac{\mathrm{e}_{\Omega'} - (C - \mathrm{c}_i)\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'})}{\mathrm{c}_i} \right\rceil \\
0 &< \left\lceil \frac{\mathrm{c}_i(\mathrm{est}_{\Omega'} - \mathrm{est}_i) + \mathrm{e}_{\Omega'} - (C - \mathrm{c}_i)\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'})}{\mathrm{c}_i} \right\rceil \\
0 &< \mathrm{c}_i(\mathrm{est}_{\Omega'} - \mathrm{est}_i) + \mathrm{e}_{\Omega'} - (C - \mathrm{c}_i)\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'}) \\
C\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_{\Omega'}) &< \mathrm{e}_{\Omega'} + \mathrm{c}_i\,(\mathrm{lct}_{\Omega'} - \mathrm{est}_i)
\end{aligned}$$

Therefore timetable edge finding propagates what is a contradiction. □

5 Experimental Results

The presented algorithm (including the improvements described in Section 3) was tested on 438 open instances of the RCPSP problem from PSPLIB [1]. The RCPSP problem is to find the shortest possible schedule for a set of activities while fulfilling precedence constraints and cumulative resource constraints. For the open instances the minimum possible length l of the schedule is still not known, but PSPLIB keeps track of the best published lower and upper bounds (LB and UB such that $\mathrm{LB} \leq l \leq \mathrm{UB}$), including recent results of Schutt et al. [8, 9]. PSPLIB does not contain results of Laborie [4], however these are with a single exception (lower bound 83 for instance j90_25_5) overcome by results of Schutt et al.

We improved current best known lower bounds using destructive lower bounds: first we try to find a solution with length equal to the current best known LB. We used simple SetTimes search as recapped in [3] and after each decision we propagate all constraints to fixpoint (using timetable edge finding for cumulative resources). If there is no solution then we continue by trying to find a solution with length $\mathrm{LB} + 1$. If that also fails then we try $\mathrm{LB} + 2$ and so on. The time limit for each improvement step is 60 seconds. Experiments were done on Intel(R) Core(TM)2 Duo CPU T9400 on 2.53GHz.

Table 2 summarizes the results. Open instances are split into 3 groups by size. For each group there is the number of open instances in the group, number of instances with LB improved by one (column LB+1), by two etc. For 9 instances we were able to improve LB without starting SetTimes search, that is,

Table 2. Experimental results

Size	# Instances	LB+1	LB+2	LB+3	LB+4
60	49	5	3	-	-
90	78	24	10	-	-
120	311	82	32	9	4

propagation itself found the problem infeasible. Detailed results can be found at `http://vilim.eu/petr/cpaior2011-results.txt`.

Acknowledgment. I would like to thank Philippe Laborie and Jerôme Rogerie for their help with the algorithm and with this paper. I also thank to anonymous referees for providing very useful feedback.

References

[1] Project scheduling problem library, `http://webserver.wi.tum.de/psplib`
[2] Apt, K.R.: The essence of constraint propagation. Theoretical Computer Science 221(1-2), 179–210 (1999)
[3] Godard, D., Laborie, P., Nuijten, W.: Randomized large neighborhood search for cumulative scheduling. In: Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), pp. 81–89. AAAI, Menlo Park (2005)
[4] Laborie, P.: Complete MCS-based search: Application to resource constrained project scheduling. In: Kaelbling, L.P., Saffiotti, A. (eds.) IJCAI, pp. 181–186. Professional Book Center (2005)
[5] Mercier, L., Van Hentenryck, P.: Edge finding for cumulative scheduling. Informs Journal of Computing 20, 143–153 (2008)
[6] Pape, C.L., Baptiste, P., Nuijten, W.: Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems. Kluwer Academic Publishers, Dordrecht (2001)
[7] Schutt, A., Wolf, A., Schrader, G.: Not-first and not-last detection for cumulative scheduling in $O(n^3 \log n)$. In: 16th International Conference on Applications of Declarative Programming and Knowledge Management, INAP 2005, pp. 66–80. Springer, Heidelberg (2005)
[8] Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: 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)
[9] Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Explaining the cumulative propagator. Constraints, 1–33–33 (2010), doi:10.1007/s10601-010-9103-2, ISSN 1383-7133
[10] Torres, P., Lopez, P.: On not-first/not-last conditions in disjunctive scheduling. European Journal of Operational Research 127(2), 332–343 (1999)
[11] Vilím, P.: Edge finding filtering algorithm for discrete cumulative resources in $\mathcal{O}(kn \log n)$. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 802–816. Springer, Heidelberg (2009)

Identifying Patterns in Sequences of Variables

Alessandro Zanarini[1] and Pascal Van Hentenryck[2]

[1] Dynadec Europe, Louvain La Neuve, Belgium
`alessandro.zanarini@dynadec.com`
[2] Brown University, Box 1910, Providence, RI 02912
`pvh@cs.brown.edu`

Abstract. Complex rostering problems often require to recognize and count some patterns in the employees' schedules. The number of occurrences of such patterns is then constrained to comply union rules, business requirements, and other workflow constraints. A common approach to deal with these constraints is to model them as `cost-regular` constraints but the resulting automata are not trivial to encode manually. This paper proposes a new constraint, the `pattern` constraint, whose goal is to recognize sets of patterns and constrains their occurrences. The pattern constraint is implemented in two different ways, relying respectively on a modified version of the `regular` constraint and on the `cost-regular` constraint. Both approaches employ an algorithm that automatically encodes the underlying automaton. As a result, the pattern constraint provides a high-level modeling abstraction, removing the burden of encoding automata for pattern recognition and allowing to automate the creation of complex models for rostering problems.

1 Introduction

Rostering problems often present complex constraints on the employees' schedules derived from union rules, business processes, and other work-flow rules [1] [8]. Typical constraints include: "No day shift after two consecutive night shifts" or "After four working shifts, there must be a day off at least three times in a month". Such constraints require the ability to recognize in the schedule some sequence of shifts (or patterns) and to constrain their occurrences. In the literature (e.g., [7]), these constraints have been modelled with the `cost-regular` constraint, where the automaton encodes the pattern recognition logic and the cost variable represents the number of pattern occurrences identified. Despite promising results, this approach has the major drawback of requiring a manual encoding of the automaton for each pattern. This task is time-consuming, error-prone, and induces manual intervention for each new problem/constraint. The study in [6] addresses this issue proposing a 6-stage algorithm using automaton operations including concatenation and minimization.

This paper proposes a simpler approach, which exploits a well-known pattern-matching algorithm [2] and reduces the algorithmic complexity compared to [6]. Our approach is embedded in a new constraint – called `pattern` – with two alternative implementations based respectively on the `cost-regular` constraint [4] and on a variation of the `regular` constraint [9].

2 Background

This section briefly presents finite state machines and the pattern-matching algorithm of Aho and Corasick in 1975. The presentation substantially differs from [2] to ease reading and bridge the gap with Constraint Programming.

A *finite state machine*[1] is formally defined as 6-tuple $(Q, q_0, \Sigma, \Omega, \delta, \tau)$ consisting of: a set of states Q; an initial state $q_0 \in Q$; a finite set of input symbols Σ; a finite set of output symbols Ω; a transition function $\delta : Q \times \Sigma \to Q$ that maps a state and an input symbol to a corresponding next state; and an output function $\tau : Q \times \Sigma \to \Omega$ that maps a state and an input symbol to a corresponding output state.

The pattern-matching algorithm proposed by Aho-Corasick employs a finite state machine that receives as input the string to match against the set of patterns and it outputs the identifiers of the patterns recognized if any. For space reasons, we only illustrate the algorithm through an example. A formal description of the algorithm can be found in [2]. The algorithm consists of two main phases. The first phase unfolds the patterns in consecutive connected states, while the second phase connects different paths of the finite state machines using suffixes as a guide. In the following, the state labels represent the last symbols encountered by the finite state machine. For example, state $s = [1, 1, 3]$ indicates that the last three input symbols have been respectively 1, 1, and 3.

Consider the three patterns: $p_1 = [1, 2]$, $p_2 = [1, 3]$ and $p_3 = [1, 2, 1]$. In the first phase, each pattern is unfolded in separate states starting from the initial state $[]$. Pattern p_1 starts from state $[]$, then proceeds with value 1 to state $[1]$, and then with value 2 to $[1, 2]$. Pattern p_3 shares the same states as p_1 but adds an additional transition from $[1, 2]$ to $[1, 2, 1]$ with value 1. Figure 1(a) shows the automaton built after executing the first phase.

The second phase considers how to extend each state s by concatenating it with each possible domain value v. Let s' be the state resulting from the concatenation: The algorithm searches the longest suffix of the s' label that represents a valid state. A transition is then inserted from s to the appropriate s' suffix with the value v. If no such state is found, a transition from state s to the initial state $[]$ is inserted.

Figure 1(b) shows the resulting state machine with the transitions going back to the initial state omitted. Consider the state $[1, 2, 1]$ when an additional value 3 is concatenated to it: The resulting string is $[1, 2, 1, 3]$. Among all the suffixes $[1, 2, 1, 3]$, $[2, 1, 3]$, $[1, 3]$, $[3]$ and $[]$, $[1, 3]$ is the longest one and a transition starting from $[1, 2, 1]$ with value 3 and ending to $[1, 3]$ is added to the finite state machine.

The output function is created from the set of patterns and the set of transitions created in the previous step. For each transition, the algorithm checks whether any suffix of any length of the destination state represents a possible pattern to recognize. If that is the case then it outputs the identifier of the identified pattern. If no pattern is recognized, then the output is empty. In the general case, the created automaton is not minimal. Figure 1(b) shows the

[1] We consider the Mealy machine model in this paper.

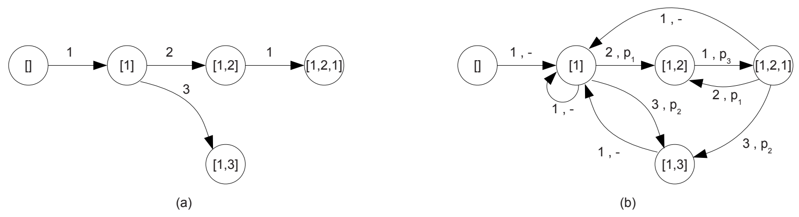


Fig. 1. (a) The automaton after the first phase of the transition creation. The arc labels represents the input symbols of the corresponding transitions. (b) The finite state machine for pattern recognition. The arc labels represents respectively the input symbol and the output symbol of the corresponding transitions.

output function as the second argument of the arc labels. Consider the input string $[3, 1, 2, 1, 3]$, after the first symbol the finite state machine is still in the initial state [] with a *null* ouput; the second input symbol 1 triggers a transition that ends up in state [1]. Afterwards, the symbol 2 brings the state machine to the state $[1, 2]$ with corresponding output p_1. The following 1 brings the state machine to state $[1, 2, 1]$ and it outputs p_3. Finally, the last symbol 3 triggers the transition to state $[1, 3]$ with output p_2.

The Aho-Corasick algorithm creates the finite state machine in linear time with respect to the sum of the lengths of the patterns to be identified.

3 Pattern Constraint

Definition 1 (The `pattern` Constraint). *Let p_i be a pattern defined as a sequence of values $p_i = (v_1, \ldots, v_k)$, $\mathcal{P}$ a set of patterns, $x = (x_1, \ldots, x_n)$ a sequence of finite domain variables, and z be a finite domain variable representing the number of occurrences of any pattern $p_i \in \mathcal{P}$ in x. The `pattern` constraint holds iff the sum of the number of occurrences of each $p_i \in \mathcal{P}$ in the sequence x is equal to z.*

The `pattern` constraint is a specialization of the `cardinality-path` constraint [3]. In the following, we introduce two possible alternatives to implement the `pattern` constraint.

The Cost-Regular Implementation. The first possibility is to employ the `cost-regular` constraint [4], where the automaton would be extracted from the finite state machine built by the Aho-Corasick algorithm. This is similar to the approach of Métivier et al. in [7] except that we employ the Aho-Corasick algorithm to derive the automaton automatically. The costs on the automaton arcs are defined as follows: if the corresponding arc in the finite state machine has an empty output, then the arc cost is equal to 0; otherwise its cost is 1.[2]

[2] In the most general case, a transition may trigger more than one output symbol; in this case the cost would be the number of output symbols.

Therefore, the total cost of the constraint represents the number of occurrences of the patterns. This solution allows to exploit the cost-based filtering of the `cost-regular` and to propagate back to the variables x any change of the bounds of z. Nonetheless, it does not achieve domain consistency as the `cost-regular` constraint only performs bound consistency on z. The time complexity is $O(ndQ)$ where n is the number of variables, d is the cardinality of the input alphabet, and Q is the number of automaton states which is bounded from above by the sum of the length of the patterns to recognize. As a comparison, the complexity of the `cardinality-path` is $O(n^2d^k + n^3d)$ where k is the length of the longest pattern to recognize.

The Finite-State-Machine Implementation. The `finite-state-machine` constraint (in the following `FSM` constraint) is a slight modification of the `regular` constraint; it shares with the latter the automaton abstraction but it has some differences: all the states are considered finals and it constrains two sequences of variables (the input and output strings).

Definition 2 (Finite State Machine Constraint).
Let $\mathcal{FSM} = (Q, q_0, \Sigma, \Omega, \delta, \tau)$ be a finite state machine, $(x_1, \ldots, x_n)$ a sequence of input variables with domains $D_{x_i} \in \Sigma$ and $(y_1, \ldots, y_n)$ a sequence of output variables with domains $D_{y_i} \in \Omega$. The finite state machine constraint holds iff for each input symbol the corresponding output symbol is consistent with $\mathcal{FSM}$, i.e. given q_{i-1} the state reached by the $\mathcal{FSM}$ after processing $i-1$ input symbols, than δ is defined for (q_{i-1}, x_i) and $\tau(q_{i-1}, x_i) = y_i$.

To simplify understanding, we only represent the output variables as finite integer variables although, in the most general case, they may be set variables. A possible decomposition of the `fsm` constraint follows closely the one proposed by Quimper and Walsh in [10]:

$$q_i = \delta(q_{i-1}, x_i) \tag{1}$$
$$y_i = \tau(q_{i-1}, x_i) \tag{2}$$

The `pattern` constraint can then be modelled as a `fsm` constraint where the finite state machine is generated by the Aho-Corasick algorithm; then with a `global-cardinality-constraint` (in the following `gcc` constraint) defined on the y variables that forces the appropriate pattern occurrences.

Theorem 1. *The `pattern` constraint based on the `fsm` constraint does not dominate in term of pruning the one based on the `cost-regular` constraint.*

Proof. Consider a constraint with $x = (x_1, \ldots, x_4)$ with domains $D_1 = 1$ and $D_2 = D_3 = D_4 = \{1, 2\}$, a pattern $p_1 = [1, 2]$ that must appear exactly twice in x. The only feasible solution is $x = (1, 2, 1, 2)$. Since the propagation of the x variables is equivalent in the `cost-regular` and `fsm` constraints, we only analyze the back-propagation respectively from the cost variable and from the output variables to the x.

Thanks to cost-based filtering the `cost-regular` is able to detect that $x_2 = 1$ is not feasible as its related shortest and longest path to a sink node in the layered graph is respectively equal to 0 and 1. Since the cost is bounded to be equal to 2 then the value 1 is removed from variable x_2. With similar arguments, the constraint is able to detect also that x_3 must be equal to 1 and x_4 equal to 2.

In case of the decomposition based on the `fsm` and `gcc` constraints, the variable y_1 is equal to 0, whereas y_2, y_3 and y_4 are free to take values 0 or 1. The *gcc* forces two y variables out of the four to be equal to 1 but it cannot perform any filtering on y_2, y_3 and y_4. Similarly, the `fsm` constraint has supports for both value 0 and 1 for the same variables, therefore it does not detect $x_2 \neq 1$.

Despite the weaker propagation, the finite state machine approach can be valuable when dealing with some over-constrained rostering applications. In these cases, `pattern` constraints are soft and may have different violation costs. In general, intersecting different finite state machines entailing different `pattern` constraints on the same variables is typically useful to increase the propagation (despite the fact that the size of the finite state machine may increase exponentially). Suppose that a problem requires to detect patterns p_1 and p_2 with a violation cost on the number of occurrences of respectively c_1 and c_2 on the same set of variables; one solution would be to have two separated `pattern` constraints and consider them independently. An alternative approach would be to intersect the two finite state machines and have a single `pattern` constraint counting the occurrences of both p_1 and p_2. The `pattern` constraint based on `fsm` would allow to extract the occurrences of each individual pattern and possibly apply different violation costs. In contrast, the `pattern` constraint based on `cost-regular` counts indistinguishably p_1 and p_2 occurrences, therefore different violation costs cannot be applied; `multicost-regular` [5] can be employed however to overcome this limitation as suggested in [6].

4 Conclusions

We introduced a new constraint - the `pattern` - that, along with the Aho-Corasick algorithm, is an important step towards an automated modelling of rostering problems. The burden of manually encoding the automata for pattern-matching is no more on the final user. Future researches should look into the intersection of several `pattern` constraints on the same set of variables and on handling wildcards.

References

1. Patat 2010 - Nurse Rostering Competition, `http://www.kuleuven-kortrijk.be/nrpcompetition` (Online; accessed March 11, 2011)
2. Aho, A.V., Corasick, M.J.: Efficient String Matching: an Aid to Bibliographic Search. Communication of ACM 18, 333–340 (1975)

3. Beldiceanu, N., Carlsson, M.: Revisiting the *Cardinality* Operator and Introducing the $Cardinality - Path$ Constraint Family. In: Codognet, P. (ed.) ICLP 2001. LNCS, vol. 2237, pp. 59–73. Springer, Heidelberg (2001)
4. Demassey, S., Pesant, G., Rousseau, L.-M.: A Cost-Regular Based Hybrid Column Generation Approach. Constraints 11(4), 315–333 (2006)
5. 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)
6. Menana, J., Demassey, S., Jussien, N.: Modélisation et Optimisation des Préférences en Planification de Personnel. Research report 10-01-INFO, École des Mines de Nantes (2010)
7. 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)
8. University of Nottingham. Staff Rostering Benchmark, `http://www.cs.nott.ac.uk/~tec/NRP/` (Online; accessed March 11, 2011)
9. 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)
10. Quimper, C.-G., Walsh, T.: Global grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 751–755. Springer, Heidelberg (2006)

Author Index