Pascal Van Hentenryck
Laurence Wolsey (Eds.)

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

LNCS 4510

4th International Conference, CPAIOR 2007
Brussels, Belgium, May 2007
Proceedings

Springer

Lecture Notes in Computer Science 4510

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Pascal Van Hentenryck Laurence Wolsey (Eds.)

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

4th International Conference, CPAIOR 2007
Brussels, Belgium, May 23-26, 2007
Proceedings

Springer

Volume Editors

Pascal Van Hentenryck
Brown University
Box 1910, Providence, RI 02912, USA
E-mail: pvh@cs.brown.edu

Laurence Wolsey
Université catholique de Louvain
34, Voie du Roman Pays, 1348 Louvain-la-Neuve, Belgium
E-mail: wolsey@core.ucl.ac.be

Library of Congress Control Number: 2007926618

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

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

ISSN 0302-9743
ISBN-10 3-540-72396-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-72396-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12061068 06/3142 5 4 3 2 1 0

Preface

This volume contains the papers presented at CP-AI-OR 2007: The Fourth International Conference on Integration of Artificial Intelligence, Constraint Programming, and Operations Research Techniques for Combinatorial Optimization Problems held May 23–26, 2007 in Brussels, Belgium. More information about the conference can be found at the Web site:

`http://www.cs.brown.edu/sites/cpaior07/Welcome.html`.

There were 80 submissions and each submission was reviewed by at least three Program Committee members. After careful consideration and discussion, the committee decided to accept 28 papers (one of which was withdrawn just before the publication of the proceedings). The papers submitted this year were of high quality and representative of a vibrant, multi-disciplinary community at the intersection of artificial intelligence, constraint programming, and operations research. The program also included three invited talks and two tutorials. We were extremely pleased that Claude Le Pape, George Nemhauser, and Bart Selman accepted our invitation to be invited speakers at the conference. Similarly, we were very fortunate to attract Thierry Benoist and John Chinneck to give tutorials at CP-AI-OR 2007. The conference was also preceded by a Master Class on constraint-based scheduling organized by Amedeo Cesta.

We would like to thank the Program Committee members who worked hard to produce high-quality reviews for the papers under tight deadlines, as well as the reviewers involved in the paper selection. We also would like to acknowledge the contributions of Laurent Michel (Publicity Chair), Barry O'Sullivan (Sponsorship Chair), Susanne Heipcke and Michael Juenger (Academic and Industrial Liaison Chairs), and Etienne Loute (Conference Co-chair) to the success of CP-AI-OR 2007. It is also a great pleasure to thank Fabienne Henry for her tremendous help in organizing the conference. The submissions, reviews, discussions, and the preparation of the proceedings were all handled by the EasyChair system. Finally, we would also like to thank the sponsors of the conference: The Association for Constraint Programming, the Cork Constraint Computation Centre (Ireland), ILOG S.A. (France), the Intelligent Information Systems Institute at Cornell University (USA), the European Commission (through the Marie Curie Research Training Network: ADONET), the Fonds National de Recherche Scientifique in Belgium, the Belgian Science Policy, ORBEL (Master Class), as well as Brown University (USA), the Facultés St. Louis (Brussels, Belgium) and the Université catholique de Louvain (Louvain-la-Neuve, Belgium).

March 2007

Pascal Van Hentenryck
Laurence Wolsey

Conference Organization

Program Chairs

Pascal Van Hentenryck
Laurence Wolsey

Local Organization

Etienne Loute, Facultés Universitaires de Saint-Louis, Brussels, Belgium
Laurence Wolsey, Université catholique de Louvain, Louvain-La-Neuve, Belgium

Program Committee

Philippe Baptiste
Chris Beck
Frederic Benhamou
Mats Carlsson
Amedeo Cesta
John Chinneck
Andrew Davenport
Rina Dechter
Yves Deville
Hani El Sakkout
Bernard Gendron
Carla Gomes
John Hooker
Stefan Karish
Andrea Lodi
Laurent Michel
Michela Milano
Yehuda Naveh
Barry O'Sullivan
Jean-Francois Puget
Jean-Charles Régin
Andrea Roli
Louis-Martin Rousseau
Michel Rueher
Martin Savelsbergh
Meinolf Sellmann
David Shmoys
Helmut Simonis

Barbara Smith
Stephen Smith
Peter Stuckey
Michael Trick
Mark Wallace
Willem-Jan van Hoeve

External Reviewers

Ionut Aron
Christian Artigues
Gregory Barlow
Marco Benedetti
John Betts
Eyal Bin
Pierre Bonami
Sebastian Brand
Marie-Claude Côté
Emilie Danna
Sophie Demassey
Bistra Dilkina
Nizar El Hachemi
Dominique Feillet
Felix Geller
Carmen Gervet
Vibhav Gogate
Tarik Hadzic
Jin-Kao Hao
Shlomo Hoory
Ayoub Insea Correa
Manuel Iori
Kalev Kask
Serge Kruk
Yahia Lebbah
Bernd Meyer
Claude Michel
Jean-Noël Monette
Bertrand Neveu
Angelo Oddi
Gilles Pesant
Federico Pecora
Lam Phuong
Nicola Policella
Jakob Puchinger
Luis Queseda
Claude-Guy Quimper
Yossi Richter
Ashish Sabharwal
Frederic Saubion
Pierre Schaus
Stefano Smriglio
Christine Solnon
Peter Stuckey
Daria Terekhov
Andrea Tramontani
Gilles Trombettoni
Charlotte Truchet
Jeremie Vautard
Petr Vilim
Toby Walsh
Stéphane Zampelli
Terry Zimmerman

Table of Contents

Minimum Cardinality Matrix Decomposition into Consecutive-Ones Matrices: CP and IP Approaches

Davaatseren Baatar[1], Natashia Boland[1], Sebastian Brand[2], and Peter J. Stuckey[2]

[1] Department of Mathematics, University of Melbourne, Australia
[2] NICTA Victoria Research Lab, Department of Comp. Sci. and Soft. Eng. University of Melbourne, Australia

Abstract. We consider the problem of decomposing an integer matrix into a positively weighted sum of binary matrices that have the consecutive-ones property. This problem is well-known and of practical relevance. It has an important application in cancer radiation therapy treatment planning: the sequencing of multileaf collimators to deliver a given radiation intensity matrix, representing (a component of) the treatment plan.

Two criteria characterise the efficacy of a decomposition: the *beam-on time* (length of time the radiation source is switched on during the treatment), and the *cardinality* (the number of machine set-ups required to deliver the planned treatment).

Minimising the former is known to be easy. However finding a decomposition of minimal cardinality is NP-hard. Progress so far has largely been restricted to heuristic algorithms, mostly using linear programming, integer programming and combinatorial enumerative methods as the solving technologies. We present a novel model, with corresponding constraint programming and integer programming formulations. We compare these computationally with previous formulations, and we show that constraint programming performs very well by comparison.

1 Introduction

The problem of decomposing an integer matrix into a weighted sum of binary matrices has received much attention in recent years, largely due to its application in radiation treatment for cancer.

Intensity-modulated radiation therapy (IMRT) has been increasingly used for the treatment of a variety of cancers [17]. This treatment approach employs two devices that allow higher doses of radiation to be administered to the tumour, while decreasing the exposure of sensitive organs (Fig. 1). The first is that the source of radiation can be rotated about the body of the patient: by positioning the tumour at a "focal point", and aiming the radiation beam at this point from various angles, the tumour receives a high dose from all angles, while the surrounding tissue only gets high exposure from some angles.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 1–15, 2007.

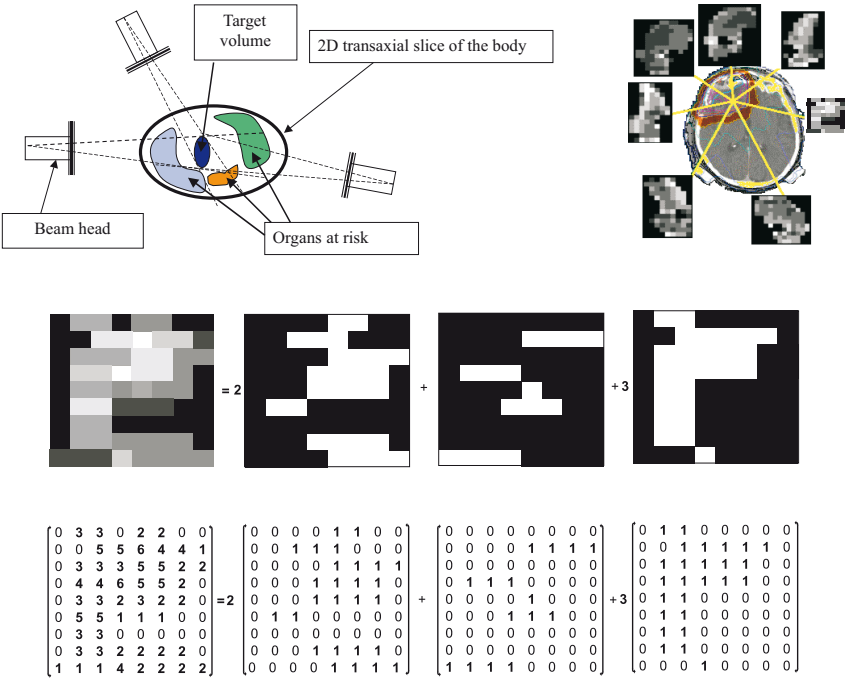


Fig. 1. Intensity-modulated radiotherapy

The second is more subtle, and involves repeated exposures from the same angle, where the uniform-intensity rectangular field of radiation delivered by the radiation source is "shaped" in a different way for each exposure, and each exposure can be for a different length of time. This process builds up a complex profile of received radiation in the patient's body, effectively converting the uniform radiation field delivered by the machine to an intensity-modulated field. The latter is usually described by discretising the 2-dimensional rectangular field, and specifying a radiation intensity level in each discrete element, representing the total length of time for which that element should be exposed to radiation.

A *treatment plan* for a single IMRT treatment session with a patient thus typically consists of a set of angles, together with a matrix for each angle, known as the *intensity matrix*, which represents the modulated field to be delivered at that angle. Typically the intensity is scaled so that the entries in the intensity matrix are integer. Indeed, they are usually quite small integers. Finding a good treatment plan is a challenging problem in its own right, and has been the subject of a great deal of research. We recommend the reader refer to the papers [13,9,15] and references therein.

In this paper, we assume a treatment plan is given, and focus on the delivery of the modulated field (intensity matrix) at a given angle. IMRT can be

delivered by a variety of technologies: here we focus on its delivery via a machine known as a multileaf collimator, operating in "step-and-shoot" mode [7]. This machine delivers a rectangular field of radiation, of uniform intensity, that can be shaped through partial occlusion of the field by lead rods, or "leaves". These are positioned horizontally on the left and right side of the field, and can slide laterally across the field to block the radiation, and so shape the field. The discretisation giving rise to the intensity matrix is taken to be compatible with the leaf widths. In step-and-shoot mode, the leaves are moved into a specified position, the radiation source switched on for a specified length of time and then switched off, the leaves moved to a new position, and so on (Fig. 1).

The shaped radiation field delivered by the leaves in each position can be represented as a binary matrix, with 1's in elements exposed in that position, and 0's in elements covered by the leaves (Fig. 1). The structure of the machinery ensures that all 1's in any row occur in a consecutive sequence: the matrix has the consecutive-ones property. The length of time radiation is applied to the shaped field is called its beam-on time. To correctly deliver the required intensity matrix, the matrices corresponding to the shaped fields, weighted with their beam-on times, must sum to the intensity matrix.

This motivates the following problem specification.

2 Problem Specification and Related Work

Let I be an $m \times n$ matrix of non-negative integers (the intensity matrix). The problem is to find a decomposition of I into a positive linear combination of binary matrices that have the consecutive-ones property. Often the radiation delivery technology imposes other constraints on the matrices, but here we focus on the simplest form, in which only the consecutive-ones property is required. For convenience, we use the abbreviation C1 for a binary consecutive-ones matrix. We also refer to a shaped field, represented by a C1 matrix, as a *pattern*.

Formally, we seek positive integer coefficients b_k (the beam-on times) and C1 matrices X_k (the patterns), such that

$$I = \sum_{k \in \Omega} b_k X_k \tag{1}$$

where Ω is the index set of the binary matrices X_k, and for $k \in \Omega$:

$$X_{k,i,j_L} = 1 \wedge X_{k,i,j_R} = 1 \;\rightarrow\; X_{k,i,j_M} = 1 \tag{2}$$

for all $1 \leqslant j_L < j_M < j_R \leqslant n$ and all $i = 1, \ldots, m$.

Example 1. Consider the matrix

$$I = \begin{pmatrix} 2 & 5 & 3 \\ 3 & 5 & 2 \end{pmatrix}.$$

Two decompositions are

$$D_1 = 1\begin{pmatrix}0&1&1\\1&1&0\end{pmatrix} + 2\begin{pmatrix}1&1&1\\1&1&1\end{pmatrix} + 2\begin{pmatrix}0&1&0\\0&1&0\end{pmatrix} \text{ and}$$
$$D_2 = 2\begin{pmatrix}1&1&0\\0&1&1\end{pmatrix} + 3\begin{pmatrix}0&1&1\\1&1&0\end{pmatrix},$$

that is, we have $I = D_1 = D_2$. ◊

We denote by B and K the sum of coefficients b_k and the number of matrices X_k used in the decomposition (1), respectively, i.e.

$$B = \sum_{k\in\Omega} b_k \qquad \text{and} \qquad K = |\{\, b_k \mid \ k \in \Omega \,\}|\,.$$

We call B the *total beam-on time* and K the *cardinality* of the decomposition. The efficacy of a decomposition is characterised by its values for B and K: the smaller these values the better. In Example 1, the decompositions have the values $B_1 = B_2 = 5$, $K_1 = 3$ and $K_2 = 2$; so D_2 is preferred.

The problem of finding a decomposition that minimises B can be solved in polynomial time using linear programming or combinatorial algorithms [1,8,10,3]. However, it is possible for a decomposition to have minimal B but large K; indeed algorithms for minimising B tend to produce solutions with much larger K values than is necessary.

In radiation therapy, clinical practitioners would prefer solutions that minimise B, while ensuring K is as small as possible, i.e. they would prefer a lexicographically minimum solution, minimising B first and then K, written as $lex\_min(B, K)$. Since minimising B is easy, its minimal value, which we denote by B^*, is readily computable. The problem then becomes one of minimising K subject to the constraint that $B = B^*$. Although this problem, too, is NP-hard (it follows directly from the proof of NP-hardness of the problem of minimising K alone, given in [3]), it is hoped that solution methods effective in practice can be developed.

In the last decade, dozens of heuristic algorithms have indeed been developed, for example [1,8,10,6,3]; approximation algorithms are studied in [4]. Some of these attempt to find solutions in which both B and K are "small", while some seek low cardinality solutions while ensuring $B = B^*$ is fixed. An exact algorithm for the $lex\_min(B, K)$ problem has also been developed [10]: it is a highly complex, specialised enumerative algorithm that appears to carry out similar steps to those that might be expected in a constraint programming approach.

However the development of tractable exact formulations has lagged behind. Several exact integer programming models were introduced in [2] and [11] in order to solve the $lex\_min(B, K)$ problem, but these were either not tested computationally or were able to solve only small problems in reasonable CPU time. In this paper we develop a new model, that we refer to as the *Counter Model*. We derive both an integer programming formulation and a constraint programming method, and test both of these computationally against previous integer programming models. Our integer programming formulation performs substantially

better than existing formulations, and the constraint programming approach provides the best computational results overall.

In the remainder of this paper, we first briefly review the existing integer programming formulations, and then present the Counter Model, our new integer programming formulation, and our constraint programming method. We then provide a computational comparison of these, and make our conclusions.

3 Existing Integer Programming Formulations

The central issue in modelling the $lex\_min(B, K)$ problem is that K, the cardinality of the decomposition, is unknown, and yet the natural variable indices depend on it. The two integer linear programming models in the current literature that can be used for the $lex\_min(B, K)$ problem take different approaches in tackling this issue. [11] overcome it by indexing according to radiation units; [2] instead calculates an upper bound on K. Here we give descriptions of these models and some additional symmetry breaking constraints.

Notation. The range expression $[a..b]$ with integers a, b denotes the integer set $\{\, e \mid a \leqslant e \leqslant b \,\}$.

3.1 The Unit Radiation Model

The model of [11] focuses on individual units of radiation. It is based on the assumption that the total beam-on time is fixed, in our case to B^*. What is not known is: for each of the B^* units of radiation, what pattern should be used for the delivery of that unit? In the model, binary variables $d_{t,i,j}$ are used to indicate whether the element (i, j) is exposed in the tth pattern corresponding to the tth unit of radiation, for $t \in [1..B^*]$. They are linked to the intensity matrix by

$$I_{ij} = \sum_{t=1}^{B^*} d_{t,i,j}, \qquad \text{for all } i \in [1..m], j \in [1..n]. \tag{3}$$

The leaf structure in the pattern is captured by binary variables:

$$p_{t,i,j} = \begin{cases} 1 & \text{if the right leaf in row } i \text{ of pattern } t \text{ covers column } j, \\ 0 & \text{otherwise,} \end{cases}$$

$$\ell_{t,i,j} = \begin{cases} 1 & \text{if the left leaf in row } i \text{ of pattern } t \text{ covers column } j, \\ 0 & \text{otherwise,} \end{cases}$$

for all $t \in [1..B^*]$, $i \in [1..m]$, $j \in [1..n]$. The relationship between these three sets of binary variables is given by

$$p_{t,i,j} + \ell_{t,i,j} = 1 - d_{t,i,j} \qquad \text{for all } t \in [1..B^*], i \in [1..m], j \in [1..n], \tag{4}$$

and

$$\begin{aligned} p_{t,i,j} &\leqslant p_{t,i,j+1}, \\ \ell_{t,i,j+1} &\leqslant \ell_{t,i,j} \end{aligned} \qquad \text{for all } t \in [1..B^*], i \in [1..m], j \in [1..n-1]. \tag{5}$$

These constraints ensure that the d_t induce a C1 matrix.

Under these constraints, the indices t can be permuted to create equivalent solutions. Thus, the model is "free" to order the patterns so that identical patterns appear consecutively. To minimise the number of different patterns in a solution, the number of times adjacent patterns are different can be minimised. That patterns t and $t+1$ differ is reflected in the binary variable g_t, and the sum of these variables corresponds to the number of patterns by

$$K = 1 + \sum_{t=1}^{B^*-1} g_t. \tag{6}$$

Minimising this sum ensures that identical patterns appear consecutively. Each unique pattern yields a C1 matrix for the decomposition; the associated beam-on time is given by the number of copies of this pattern among the d_t. [11] tally values for g using binary additional variables:

$$c_{t,i,j} = \begin{cases} 1 & \text{if } d_{t,i,j} = 1 \text{ and } d_{t+1,i,j} = 0, \\ 0 & \text{otherwise,} \end{cases}$$

$$u_{t,i,j} = \begin{cases} 1 & \text{if } d_{t,i,j} = 0 \text{ and } d_{t+1,i,j} = 1, \\ 0 & \text{otherwise,} \end{cases}$$

$$s_{t,i,j} = \begin{cases} 1 & \text{if } d_{t,i,j} \neq d_{t+1,i,j}, \\ 0 & \text{otherwise,} \end{cases}$$

for all $t \in [1 \,..\, B^* - 1]$, $i \in [1..m]$, $j \in [1..n]$. The relationship of these variables is established by the linear constraints

$$\begin{aligned} & -c_{t,i,j} \leqslant d_{t+1,i,j} - d_{t,i,j} \leqslant u_{t,i,j}, \\ & u_{t,i,j} + c_{t,i,j} = s_{t,i,j}, \\ & \sum_{i=1}^{m} \sum_{j=1}^{n} s_{t,i,j} \leqslant mng_t, \\ & \text{for all } t \in [1 \,..\, B^* - 1], i \in [1..m], j \in [1..n]. \end{aligned} \tag{7}$$

The original model in [11] does not contain symmetry-breaking constraints. We added symmetry breaking constraints as follows. We wish to enforce that the matrices appear in order of non-increasing beam-on time. This means the pattern groups should appear in order of non-increasing size, which is to say that no (possibly empty) sequence of 0's enclosed by 1's and followed by a longer

sequence of 0's occurs in the g vector extended by $g_0 = 1$. This can be enforced by

$$\sum_{v=r}^{r+t} g_v - 1 \leqslant \sum_{v=r+t+1}^{r+2t} g_v \quad \text{for all } r \in [0 \,..\, B^* - 3], t \in [1 \,..\, \lfloor (B^* - r - 1)/2 \rfloor], \quad (8)$$

for which we define $g_0 = 1$.

The constraints (3)-(8) with the objective of minimising K constitute the Unit Radiation model.

3.2 The Leaf-Implicit Model

This model of [2] is based on calculating an upper bound on K, denoted by $\bar{K}$. A value for $\bar{K}$ is not difficult to compute; the cardinality of any solution to the (polynomially solvable) minimum beam-on time problem will do. For each $k \in [1..\bar{K}]$, a C1 matrix X_k and associated beam-on time b_k need to be found. If X_k is the zero matrix then pattern k is not needed; minimising decomposition cardinality is minimising the number of non-zero matrices in the decomposition.

The model of [2] uses a characterisation of matrix decomposition into C1 matrices derived in [3]. In this model, the structure of the solution is encoded by recording beam-on time against each leaf position. It uses integer variables $x_{k,i,j}$ to represent the beam-on time for pattern k if the left leaf in row i of that pattern covers exactly the columns $[0 \,..\, j-1]$; otherwise $x_{k,i,j}$ is zero. The left leaf being in position 0 means it is fully retracted. Similarly, the integer variable $y_{k,i,j}$ represents the beam-on time for pattern k if the right leaf in row i of that pattern covers exactly the columns $[j \,..\, n+1]$, and is zero otherwise. The right leaf "covering" only column $n+1$ means it is fully retracted. For convenience, we define the function *inc* to compute the non-negative difference between two values,

$$inc(x, y) = \max(y - x, 0),$$

and the matrices Δ^+, Δ^- are defined by

$$\Delta^+_{i,j} = inc(I_{i,j-1}, I_{i,j}),$$
$$\Delta^-_{i,j} = inc(I_{i,j}, I_{i,j-1}),$$

for all $j \in [1..n+1]$, $i \in [1..m]$, where we take $I_{i,0} = I_{i,n+1} = 0$. Delivering the intensity matrix I is equivalent to asking that

$$\sum_{k=1}^{\bar{K}} x_{k,i,j} - w_{i,j} = \Delta^+_{i,j} \quad \text{and} \quad \sum_{k=1}^{\bar{K}} y_{k,i,j} - w_{i,j} = \Delta^-_{i,j},$$
$$\text{for all } i \in [1..m], j \in [1 \,..\, n+1], \quad (9)$$

where the $w_{i,j}$ are non-negative integer variables. That the total beam-on time is B^* is ensured using integer variables b_k constrained by

$$\sum_{j=1}^{n+1} x_{k,i,j} = b_k \quad \text{and} \quad \sum_{j=1}^{n+1} y_{k,i,j} = b_k, \qquad \text{for all } k \in [1..\bar{K}], i \in [1..m], \tag{10}$$

and

$$\sum_{k=1}^{\bar{K}} b_k = B^*. \tag{11}$$

Counting the number of patterns is similarly encoded against leaf positions. The model uses binary variables $\ell_{k,i,j}$ to represent whether the left leaf in row i of pattern k covers exactly columns $[0\,..\,j-1]$, and $r_{k,i,j}$ to represent whether the right leaf in row i of pattern k covers exactly columns $[j\,..\,n+1]$. Further, binary variables β_k indicate whether pattern k is used at all. The pattern structure is enforced by the constraints

$$\sum_{j=1}^{n+1} \ell_{k,i,j} = \beta_k \quad \text{and} \quad \sum_{j=1}^{n+1} r_{k,i,j} = \beta_k, \qquad \text{for all } k \in [1..\bar{K}], i \in [1..m], \tag{12}$$

and, ensuring that the left leaf is indeed to the left of the right leaf,

$$\sum_{j=1}^{s} \ell_{k,i,j} - \sum_{j=1}^{s} r_{k,i,j} \geqslant 0, \quad \text{for all } s \in [1\,..\,n+1], k \in [1..\bar{K}], i \in [1..m]. \tag{13}$$

If pattern k is not used, then it cannot supply any radiation. This logic is encoded via the constraints

$$x_{k,i,j} \leqslant M^+_{k,i,j}\ell_{k,i,j} \quad \text{and} \quad y_{k,i,j} \leqslant M^-_{k,i,j} r_{k,i,j}, \qquad \text{for all } s \in [1..\bar{K}], i \in [1..m], j \in [1\,..\,n+1], \tag{14}$$

where $M^+_{k,i,j}$, $M^-_{k,i,j}$ are any appropriate upper bounds. We use

$$M^\circ_{k,i,j} = B^* - \sum_{s=1}^{n+1} \Delta^+_{i,s} + \Delta^\circ_{i,j}, \quad \text{for } \circ \in \{+,-\}.$$

The decomposition cardinality is found by

$$K = \sum_{k=1}^{\bar{K}} \beta_k. \tag{15}$$

The description of the original model in [2] does not discuss symmetry breaking. To make the comparison with the other models fairer, we add the following symmetry breaking constraints. In a closed row, the leaves can meet anywhere. We choose the point at which the left leaf is fully retracted, by requiring

$$\ell_{k,i,j} + r_{k,i,j} \leqslant 1, \qquad \text{for all } i \in [1..m], j \in [2\,..\,n+1]. \tag{16}$$

Furthermore, we order the beam-times associated to the patterns,

$$b_1 \leqslant \ldots \leqslant b_{\bar{K}}. \tag{17}$$

Unfortunately, symmetry-breaking constraint (17) does not remove symmetries arising when the coefficients of two of the X_k matrices are equal. Sometimes, values can be swapped between matrices without breaking their consecutive-ones properties. Consider a fragment of D_2 from Example 1:

$$2\begin{pmatrix}1&1&1\\1&1&\mathbf{1}\end{pmatrix}+2\begin{pmatrix}0&1&0\\0&1&\mathbf{0}\end{pmatrix}=2\begin{pmatrix}1&1&1\\1&1&\mathbf{0}\end{pmatrix}+2\begin{pmatrix}0&1&0\\0&1&\mathbf{1}\end{pmatrix}.$$

Note that the matrices of left-hand side and right-hand side are both lexicographically ordered (row-wise as well as column-wise).

In summary, the Leaf-Implicit model consists of the constraints (9)-(17) with the objective of minimising K.

4 New Constraint Programming and Integer Programming Approaches

The problem specification gives rise to a number of interesting models. Our interest is to compare models in combination with solving techniques. Although we try to model solver-independently, we need to get specific eventually, so we target integer linear programming (IP) solvers on the one hand, and constraint programming (CP) solvers (allowing arbitrary constraints) on the other.

We first discuss the most direct model that could be derived from the formulation, as a CP model. Clearly this has a number of drawbacks, such as a great deal of symmetry, so we go on to develop a compact model, useful for both IP and CP, in which much of the symmetry is eliminated.

4.1 The Direct CP Model

The problem specification can almost directly be interpreted as a CP model. The decision variables are the binary variables $X_{k,i,j}$ and the positive integer variables b_k. Requirement (1) is a linear equality constraint. Requirement (2) corresponds to the contiguity constraint studied in [12]. The critical point is that the number of variables depends on K. Hence, as in the Leaf-Implicit model, we need to make use of an upper bound $\bar{K}$ on K and program the search to try increasing values of K.

The great deal of symmetries permitted by the Direct model is a drawback. We can add (17) to remove some of the symmetries, and indeed some CP systems provide support for the combination of the constraints in (11) and (17), yielding stronger constraint propagation, e.g. the ordered_sum constraint of ECL^{i}PS^e [16]. Still, as we have seen, many symmetries remain.

4.2 The Counter Model

This novel model is based on counting the patterns according to their beam-on times. We use non-negative integer variables $Q_{b,i,j}$ to represent the *number* of patterns that have associated beam-on time b and expose the element (i, j). An upper bound $\bar{b}$ on the beam-on times is thus needed. It is easy to see that the maximum intensity, i.e. the largest value in I, is such a bound. The link between the $Q_{b,i,j}$ variables and the intensity matrix is

$$\sum_{b=1}^{\bar{b}} bQ_{b,i,j} = I_{i,j}, \qquad \text{for all } i \in [1..m], j \in [1..n]. \tag{18}$$

To derive a C1 decomposition of I from Q satisfying the above constraint, we must take a C1 decomposition of Q_b for each b. The C1 matrices in the decompositions of Q_b each have a *multiplicity* given by the number of times they occur in the decomposition of Q_b.

We claim that we can restrict our attention to decompositions of Q_b in which the multiplicities are all precisely 1. Imagine to the contrary a decomposition of I into C1 matrices $X_1, \ldots, X_K$ with respective weights $b_1, \ldots, b_K$ such that $X_i = X_j$ for some i, j with $1 \leqslant i < j \leqslant K$. Then we can construct a smaller cardinality solution, by replacing $b_i X_i + b_j X_j$ by a single C1 matrix X_i with weight $b_i + b_j$. This results in a decomposition of I with strictly smaller cardinality. Hence in any minimal cardinality decomposition of I there are no repeated C1 matrices. Hence in any minimal cardinality decomposition of I all matrices in the decompositions of Q_b have unit multiplicity.

For example, consider the following decomposition of $I = \left(\begin{smallmatrix} 2 & 4 & 3 \\ 3 & 4 & 2 \end{smallmatrix}\right)$, which has non-unit multiplicity in the decomposition of Q_1:

$$1\left\{\overset{X_1}{\begin{pmatrix} 0&1&0 \\ 0&1&0 \end{pmatrix}} + \overset{X_2}{\begin{pmatrix} 0&1&0 \\ 0&1&0 \end{pmatrix}} + \overset{X_3}{\begin{pmatrix} 0&0&1 \\ 1&0&0 \end{pmatrix}}\right\} + 2\overset{X_4}{\begin{pmatrix} 1&1&1 \\ 1&1&1 \end{pmatrix}} = 1\overset{Q_1}{\begin{pmatrix} 0&2&1 \\ 1&2&0 \end{pmatrix}} + 2\overset{Q_2}{\begin{pmatrix} 1&1&1 \\ 1&1&1 \end{pmatrix}}.$$

Here $X_1 = X_2$, so we can replace these by a single matrix in the decomposition with weight $b_1 + b_2 = 2$. The new decomposition and the resulting Q_b matrices, Q'_1 and Q'_2, are:

$$1\overset{X_1}{\begin{pmatrix} 0&0&1 \\ 1&0&0 \end{pmatrix}} + 2\left\{\overset{X_2}{\begin{pmatrix} 1&1&1 \\ 1&1&1 \end{pmatrix}} + \overset{X_3}{\begin{pmatrix} 0&1&0 \\ 0&1&0 \end{pmatrix}}\right\} = 1\overset{Q'_1}{\begin{pmatrix} 0&0&1 \\ 1&0&0 \end{pmatrix}} + 2\overset{Q'_2}{\begin{pmatrix} 1&2&1 \\ 1&2&1 \end{pmatrix}}.$$

From the above reasoning, we can assume that decompositions of Q_b into C1 matrices, each with unit weight, exist. So we have created a simpler form of the original problem. Instead of looking for a weighted decomposition of I, we seek an unweighted decomposition of each Q_b. We introduce non-negative integer variables N_b to represent the cardinality of the decomposition of Q_b, for each $b \in [1..\bar{b}]$. As we have argued, we may assume N_b is also the sum of weights for the minimum sum of weights decomposition of Q_b.

A convenient formula for the minimum sum of weights of a C1 decomposition is given in [8,3]. The idea is as follows. The decomposition of any general non-negative integer row vector V with m elements into positive integer-weighted C1 matrices (row vectors) must satisfy the property that for all $j \in [1..m]$, the sum of weights applied to C1 matrices that expose element j but not $j-1$ must be exactly $V_j - V_{j-1}$ in the case that this is non-negative, and zero otherwise, i.e. must be exactly $inc(V_{j-1}, V_j)$, where we define $V_0 = 0$. Each nonzero C1 matrix in the decomposition must have a first element equal to one; i.e. there must be some element $j \in [1..n]$ with $j-1$ not exposed. So the sum of weights applied to nonzero C1 matrices in the decomposition must be $\sum_{j=1}^{n} inc(V_{j-1}, V_j)$. This observation extends to an $m \times n$ non-negative integer matrix G. It is straightforward to show that any decomposition of G into non-zero C1 matrices has a sum of weights equal to the maximum over $i \in [1..m]$ of

$$\sum_{j=1}^{n} inc(G_{i,j-1}, G_{i,j}) \qquad (\star)$$

where we define $G_{i,0} = 0$ for all $i \in [1..m]$. Indeed, this quantity minimises the sum of weights over all decompositions of G into $C1$ matrices.

Thus, we can calculate N_b by finding the smallest N_b satisfying

$$N_b \geqslant \sum_{j=1}^{n} inc(Q_{b,i,j-1}, Q_{b,i,j}), \qquad \text{for all } i \in [1..m], \tag{19}$$

where we define $Q_{b,i,0} = 0$, for all b and i.

To summarise, the variables N_b represent the number of patterns that have associated beam-on time of b, and the matrix Q_b encodes the C1 matrices in the decomposition of I that should be given weight b. In other words, the matrix Q_b should itself decompose into (a sum of) N_b C1 matrices, each of which appears in the decomposition of I with weight b. Since we can restrict our attention to the decompositions of Q_b with unit multiplicities, the cardinality of the decomposition of Q_b is precisely the sum of the multiplicities. Furthermore, since we seek to minimise the cardinality of the solution, we can take N_b to be the minimum sum of multiplicities over C1 decompositions of Q_b, i.e. N_b can be related to Q_b via (19). The cardinality of a decomposition corresponding to N and Q is given by

$$K = \sum_{b=1}^{\bar{b}} N_b, \tag{20}$$

and for the total beam-on time we find

$$B^* = \sum_{b=1}^{\bar{b}} bN_b. \tag{21}$$

The Counter model thus consists of the constraints (18)-(21) with the objective of minimising K.

The Counter Model with Integer Programming. To express the Counter Model as an IP, the nonlinear constraint (19), involving max expressions, needs to be linearised. We do this by replacing the *inc* expressions in (19), that is, $\max(Q_{b,i,j} - Q_{b,i,j-1}, 0)$, by new variables $S_{b,i,j}$ constrained by

$$\begin{aligned} &S_{b,i,j} \geqslant Q_{b,i,j} - Q_{b,i,j-1}, \\ &S_{b,i,j} \geqslant 0, \end{aligned} \qquad \text{for all } b \in [1..\bar{b}], i \in [1..m], j \in [1 \,..\, n+1]. \tag{22}$$

This transformation is correct since K and hence the (non-negative) N_b and $S_{b,i,j}$ are minimised.

The Counter Model with Constraint Programming. The Counter Model is directly implementable in CP systems that provide linear arithmetic constraints and the max constraint. The constraints (19) will usually be decomposed into linear inequalities over new variables representing the max expression. Our implementation uses bounds($\mathcal{R}$)-consistency for all linear arithmetic, and decomposes (19) as explained. An important part of a CP solution is the strategy used to search for a solution, which we choose as follows:

minimise K by branch-and-bound search
  for $b := 1$ to $\bar{b}$
    instantiate N_b by lower half first bisection
  $S := [1..n]$
  while $S \neq \varnothing$
    choose the row $i \in S$ with greatest row hardness
    $S := S - \{i\}$
    for $j := 1$ to m
      for $b := 1$ to $\bar{b}$
        instantiate $Q_{b,i,j}$ by lower half first bisection
    on failure break (return to the last choice on N_b)

After the N_b variables are fixed, rows are investigated in order of hardness. The hardness of row i is defined as the value of the expression (★) with $G_{i,j} = I_{i,j}$. It captures the minimal sum of weights required to build a solution to that row.

The search strategy uses a simple form of intelligent backtracking based on the constraint graph. $Q_{b,i,j}$ and $Q_{b',i',j'}$ where $i \neq i'$ do not appear directly in any constraint together, and once the N_b are fixed the remaining constraints are effectively partitioned into independent problems on i. Hence failure for any row i indicates we must try a different solution to N_b.

While the ordering of the rows can make an order of magnitude improvement in performance, the independent solving of the subproblems is vital for tackling the larger problems.

5 Benchmarks

We tested several model/solver combinations on random intensity matrices. The parameters were their dimension, ranging from 3×3 to 10×10, and their maxi-

Table 1. Benchmark results

	Unit Radiation		**Leaf-Implicit**		**Counter/IP**		**Counter/CP**	
	CPU time (s)		CPU time (s)		CPU time (s)		CPU time (s)	
Max. val.	avg.	max.	avg.	max.	avg.	max.	avg.	max.
5 × 5								
3	33.02	844.05	2.17	55.23	0.01	0.01	0.00	0.02
4	64.98	1479.85	2.32	47.45	0.01	0.05	0.01	0.02
5	120.41	(1) 1800	2.48	7.52	0.01	0.03	0.01	0.06
6	509.14	(7) 1800	78.76	180.01	0.02	0.05	0.04	0.07
7	609.33	(9) 1800	84.89	610.70	0.02	0.07	0.05	0.07
8	845.41	(11) 1800	639.67	(8) 1800	0.05	0.26	0.06	0.08
9	728.39	(9) 1800	614.61	(10) 1800	0.06	0.28	0.07	0.09
10	1183.06	(15) 1800	797.61	(13) 1800	0.08	0.39	0.07	0.09
11	1416.51	(21) 1800	712.34	(10) 1800	0.10	0.21	0.08	0.11
12	1369.00	(19) 1800	989.09	(16) 1800	0.22	1.84	0.08	0.11
13	1596.02	(21) 1800	1341.48	(22) 1800	0.28	1.73	0.09	0.16
14	—	—	—	—	0.41	2.75	0.11	0.20
15	—	—	—	—	0.54	1.52	0.12	0.25
8 × 8								
3	1085.75	(17) 1800	731.85	(10) 1800	0.01	0.02	0.05	0.12
4	1484.24	(23) 1800	950.58	(11) 1800	0.03	0.05	0.06	0.06
5	1553.29	(23) 1800	1586.38	(22) 1800	0.06	0.09	0.06	0.08
6			—	—	3.87	45.19	0.08	0.09
7			—	—	0.51	3.96	0.09	0.11
8			—	—	133.74	(1) 1800	0.12	0.19
9			—	—	74.56	(1) 1800	0.15	0.24
10			—	—	372.53	(5) 1800	0.26	0.55
11			—	—	232.80	(2) 1800	0.39	2.07
12			—	—	507.40	(8) 1800	0.73	5.28
13			—	—	743.32	(11) 1800	0.87	2.14
14			—	—	—	—	1.36	4.19
15			—	—	—	—	2.45	6.37
10 × 10								
3					0.02	0.04	0.07	0.12
4					0.05	0.25	0.06	0.08
5					0.17	1.64	0.07	0.09
6					1.69	15.16	0.09	0.14
7					108.95	(1) 1800	0.12	0.21
8					215.97	(3) 1800	0.20	0.39
9					807.67	(12) 1800	0.46	4.51
10					1120.93	(18) 1800	0.87	4.75
11					1068.42	(14) 1800	0.97	2.82
12					1447.72	(23) 1800	1.79	7.86
13					—	—	6.84	46.89
14					—	—	15.41	133.22
15					—	—	21.17	118.51

mum value, ranging from 3 to 15. For each parameter combination we considered 30 instances. We set a time limit of 30 minutes per instance. All benchmarks were run on the same hardware, a PC with a 2.0 GHz Intel Pentium M Processor and 2.0 GB RAM. The IP solver was CPlex version 9.13. As the CP platform we used the prototype currently being developed on top of the Mercury system [5].

We compare the Unit Radiation model, the Leaf-Implicit model, the Counter model, all with IP, and the Counter model with CP. A subset of the results are shown in Table 1. We show the average CPU times (of all times including time outs) and maximum CPU time in seconds, and in parentheses the number of instances that timed out for a parameter combination. A '—' represents that all instances timed out, and a blank entry indicates we did not run any instances since the approach was unable to effectively solve smaller instances.

Clearly the Unit Radiation model is bettered by the Leaf-Implicit model which is again substantially bettered by the Counter model. The CP solution is substantially better than the IP solution to the Counter model because of the ability to decompose the problem into independent sub-problems after the N_b are fixed.

We also experimented with some other models. The Unit Radiation and Leaf-Implicit models without symmetry breaking performed significantly worse than the models with symmetry breaking, as expected. The direct CP model described in Section 4.1 worked for very small dimensions (4,5) but did not scale; therefore, no benchmark results are reported. Finally, we experimented with a CP/IP hybrid of the Counter model, where the linear relaxation of the IP model is used as a propagator on the objective function and to check relaxed global satisfiability inside the CP search (see e.g. [14]). While the hybrid decreased the search space, and sometimes substantially so, the overhead of running the LP solver meant the resulting times were many times the pure CP solving time.

6 Concluding Remarks

We have defined the Counter model for minimal cardinality decomposition of integer matrices with the consecutive-ones property. The model significantly improves upon earlier models for the same problem, in both an integer programming and constraint programming formulation. Its critical feature is an indexing that avoids introducing symmetries.

A drawback of the Counter model is that, as in the Unit Radiation model, the number of variables depends on the maximum intensity. For the practically interesting cases in cancer radiation therapy, this may not be an issue: in instances available to us, the maximum intensity does not exceed 20. It would be interesting to see if there are other problems where the approach of indexing on number of patterns can lead to good models.

Finally, practical problem instances may have larger dimensions: current multileaf collimators allow up to 40 rows (although the outer ones may largely be empty), making further efficiency improvements useful. For example, the Counter model in a CP solver might benefit from a special constraint for (19) to avoid decomposing it into parts, where propagation strength is lost.

References

1. R.K. Ahuja and H.W. Hamacher. Linear time network flow algorithm to minimize beam-on-time for unconstrained multileaf collimator problems in cancer radiation therapy. *Networks*, 45(1):36–41, 2004.
2. D. Baatar. *Matrix decomposition with time and cardinality objectives: Theory, Algorithms and Application to Multileaf collimator sequencing*. PhD thesis, University of Kaiserslautern, Germany, 2005.
3. D. Baatar, H. W. Hamacher, M. Ehrgott, and G. J. Woeginger. Decomposition of integer matrices and multileaf collimator sequencing. *Discrete Applied Mathematics*, 152(1-3):6–34, 2005.
4. N. Bansal, D. Coppersmith, and B. Schieber. Minimizing setup and beam-on times in radiation therapy. In J. Díaz, K. Jansen, J. D. P. Rolim, and U. Zwick, editors, *Proc. 9th Int. WS on Approximation Algorithms for Combinatorial Optimization Problems (APPROX'06)*, volume 4110 of *LNCS*, pages 27–38. Springer, 2006.
5. R. Becket, M. J. Garcia de la Banda, K. Marriott, Z. Somogyi, P. J. Stuckey, and M. Wallace. Adding constraint solving to Mercury. In P. Van Hentenryck, editor, *Proc. 8th Int. Symposium of Practical Aspects of Declarative Languages (PADL'06)*, volume 3819 of *LNCS*, pages 118–133. Springer, 2006.
6. D. Z. Chen, X.S. Hu, C. Wang, and X.R. Wu. Mountain reduction, block matching, and applications in intensity-modulated radiation therapy. In *Proc. 21st Annual Symposium on Computational Geometry*, pages 35–44, 2005.
7. M. Dirkx. *Static and dynamic intensity modulation in radiotherapy using a multileaf collimator*. PhD thesis, Daniel de Hoed Cancer Centre, University Hospital Rotterdam, The Netherlands, 2000.
8. K. Engel. A new algorithm for optimal multileaf collimator leaf segmentation. *Discrete Applied Mathematics*, 152(1-3):35–51, 2005.
9. H. W. Hamacher and K.-H. Kuefer. Inverse radiation therapy planning: A multiple objective optimisation approach. *Berichte des ITWM*, 12, 1999.
10. T. Kalinowski. *Optimal multileaf collimator field segmentation*. PhD thesis, University of Rostock, Germany, 2005.
11. M. Langer, V. Thai, and L. Papiez. Improved leaf sequencing reduces segments of monitor units needed to deliver IMRT using MLC. *Medical Physics*, 28:2450–58, 2001.
12. M. J. Maher. Analysis of a global contiguity constraint. In *Proc. 4th Workshop on Rule-based Constraint Reasoning and Programming (RCoRP'02)*, 2002.
13. H. E. Romeijn, R. K. Ahuja, J. F. Dempsey, A. Kumar, and J. G. Li. A novel linear programming approach to fluence map optimization for intensity modulated radiation therapy planning. *Phys. Med. Biol.*, 48:3521–3542, 2003.
14. K. Shen and J. Schimpf. Eplex: Harnessing mathematical programming solvers for constraint logic programming. In P. van Beek, editor, *Proc. 11th Int. Conference on Principles and Practice of Constraint Programming*, pages 622–636, 2005.
15. J. E. Tepper and T. R. Mackie. Radiation therapy treatment optimization. In *Seminars in Radiation Oncology*, volume 9 of *1*, pages 1–117, 1999.
16. M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, 1997.
17. S. Webb. Intensity modulated radiation therapy. In *Institute of Physics Publishing Bristol and Philadelphia*, 2001.

Connections in Networks: Hardness of Feasibility Versus Optimality*

Jon Conrad, Carla P. Gomes,
Willem-Jan van Hoeve, Ashish Sabharwal, and Jordan Suter

Cornell University, Ithaca, NY 14853, USA
{jmc16,jfs24}@cornell.edu, {gomes,vanhoeve,sabhar}@cs.cornell.edu

Abstract. We study the complexity of combinatorial problems that consist of competing infeasibility and optimization components. In particular, we investigate the complexity of the *connection subgraph problem*, which occurs, e.g., in resource environment economics and social networks. We present results on its worst-case hardness and approximability. We then provide a typical-case analysis by means of a detailed computational study. First, we identify an easy-hard-easy pattern, coinciding with the feasibility phase transition of the problem. Second, our experimental results reveal an interesting interplay between feasibility and optimization. They surprisingly show that proving optimality of the solution of the feasible instances can be substantially easier than proving infeasibility of the infeasible instances in a computationally hard region of the problem space. We also observe an intriguing easy-hard-easy profile for the optimization component itself.

1 Introduction

There is a large body of research studying typical-case complexity of decision problems. This work has provided us with a deeper understanding of such problems: we now have a finer characterization of their hardness beyond the standard worst-case notion underlying NP-completeness results, which in turn has led to the design of new algorithmic strategies for combinatorial problems. Nevertheless, while pure decision problems play a prominent role in computer science, most practical combinatorial problems, as they arise in fields like economics, operations research, and engineering, contain a clear optimization objective in addition to a set of feasibility constraints. In our research agenda we are interested in understanding the interplay between feasibility and optimality. We note that there has been some work on the study of the typical-case complexity of pure optimization problems [6, 10], but not concerning problems that naturally combine a feasibility and an optimization component.

As a study case we consider the typical-case complexity of a problem motivated from resource environment economics and social networks, containing

* Research supported by the Intelligent Information Systems Institute (IISI), Cornell University (AFOSR grant F49620-01-1-0076).

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 16–28, 2007.

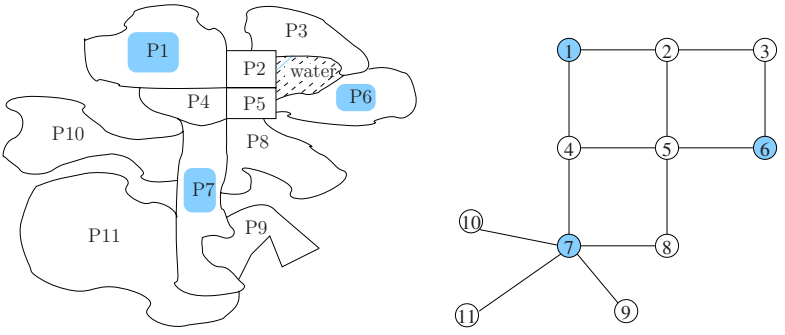


Fig. 1. The "corridor" problem and the corresponding graph representation. The reserves (P1, P6, and P7) and their corresponding vertices are shaded.

competing feasibility and optimization components. Our experimental results show that the complexity profile of this problem introduces several intriguing aspects that do not occur in pure decision problems. A good understanding of these issues will allow researchers to design better algorithms for a range of applications in a variety of domains.

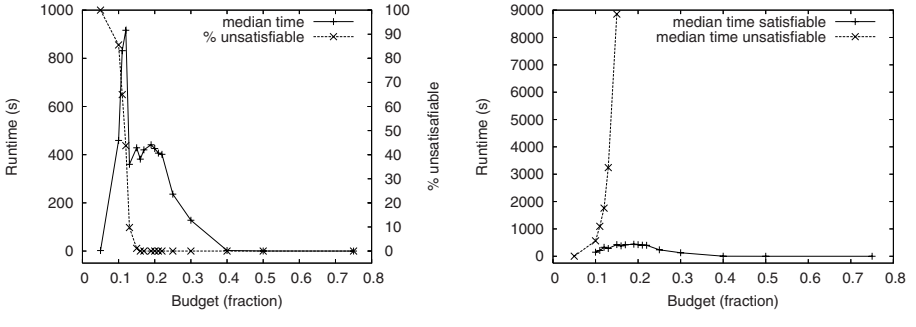
In the context of resource environment economics, our problem is an abstraction of an application that arises in the design of wildlife preserves (see e.g. [1], [3]). In many parts of the world, land development has resulted in a reduction and fragmentation of natural habitat. Wildlife populations living in a fragmented landscape are more vulnerable to local extinction due to stochastic events and are also prone to inbreeding depression. One method for alleviating the negative impact of land fragmentation is the creation of *conservation corridors* (alternatively referred to as wildlife-, habitat-, environmental-, or movement-corridors). Conservation corridors are continuous areas of protected land that link zones of biological significance [9] (see Figure 1). In designing conservation corridors, land use planners generally operate with a limited budget with which to secure the land to make up the corridor. The most environmentally beneficial conservation corridor would entail protecting every piece of land that exists between the areas of biological significance, hereafter referred to as *natural areas* or *reserves*. In most cases, however, purchasing (the development rights to) every piece of available land would be exceedingly expensive for a land trust or government that is operating with a limited budget. The objective is therefore to design corridors that are made up of the land parcels that yield the highest possible level of environmental benefits (the "utility") within the limited budget available.

In the context of social networks, a similar problem has been investigated by Faloutsos, McCurley, and Tomkins [5]. Here, one is interested, for example, in identifying the few people most likely to have been infected with a disease, or individuals with unexpected ties to any members of a list of other individuals. This relationship is captured through links in an associated social network graph with people forming the nodes. Faloutsos et al. consider networks containing two special nodes (the "terminals") and explore practically useful utility functions

that capture the connection between these two terminal nodes. Our interest, on the other hand, is in studying this problem with the sum-of-weights utility function but with several terminals. In either case, the problem has a bounded-cost aspect that competes with a utility one is trying to maximize.

We formalize the above problems as the *connection subgraph problem*. Somewhat informally, given a graph G on a set of vertices with corresponding utilities, costs, and reserve labels (i.e., whether or not a vertex is a reserve), a set of edges connecting the vertices, and a cost bound (the "budget"), our problem consists of finding a connected subgraph of G that includes all the vertices labeled as reserves and maximizes the total utility, while not exceeding the cost bound. In terms of worst-case complexity, we show that the optimization task associated with the connection subgraph problem is NP-hard, by relating it to the Steiner tree problem. Unlike the original Steiner tree problem, the NP-hardness result here holds even when the problem contains no reserves. We also show that the dual cost minimization problem is NP-hard to approximate within a certain constant factor.

In order to investigate the typical-case complexity of the connection subgraph problem, we perform a series of experiments on semi-structured graphs with randomly placed terminals and randomly generated cost and utility functions. To this end, we introduce a mixed integer linear programming formulation of the problem, which is applied to solve the instances to optimality using Cplex [7]. Figure 2 shows a preview of our results; we defer the details of the experimental setup to Section 5.



a. Hardness and SAT/UNSAT profile

b. Infeasibility vs. optimization

Fig. 2. Aggregated and separated hardness profiles for connection subgraph

The empirical complexity profile of this problem reveals an interesting interplay between the computational hardness of feasibility testing and optimization. In particular, for very low cost bounds (budgets below fraction 0.05 in Figure 2.a), almost all instances are infeasible, which is relatively easy to determine. With increasing cost bounds, one reaches the now standard phase transition point in the feasibility profile, where instances switch from being

mostly infeasible to mostly feasible (at roughly budget fraction 0.13 in the plot). At this transition, we see a sharp increase in the complexity of determining feasibility. More interestingly, however, at this phase transition boundary, we have a mixture of feasible and infeasible instances. For the feasible instances, we still need to solve the optimization problem to find (and prove) the maximum utility given the budget constraints. Quite surprisingly, proving such optimality of the solution of these feasible instances can be substantially easier than showing the infeasibility of the other instances in this region (see Figure 2.b). In other words, we have a region in our problem space where the feasible vs. infeasible decision is computationally much harder than proving optimality of the feasible instances. This is surprising because showing optimality also involves a notion of infeasibility: one has to show that there is no solution with a higher utility for the given budget. Intuitively, it appears that the purely combinatorial task of not being able to satisfy the hard constraints of the problem is harder than optimizing solutions in the feasible region.

The second part of the complexity profile of the connection subgraph problem, shown as the lower curve in Figure 2.b, concerns what happens in the feasible region when we further increase the budget beyond the satisfiability phase transition. Almost all instances are now easily shown to be feasible. However, the complexity of finding a solution with the maximum utility and proving its optimality first increases (till budget fraction roughly 0.2 in the plot) and, subsequently, for larger and larger budgets, decreases. Therefore, we have an easy-hard-easy profile in the computational cost of showing optimality, whose peak lies to the right of the feasible to infeasible transition (which, as we saw earlier, is at budget fraction roughly 0.13). In the combined plot of the median runtime of all instances (Figure 2.a), we obtain a curve that peaks around the feasible to infeasible transition because the high cost of proving infeasibility dominates the median cost in the phase transition area.

We note that such easy-hard-easy patterns have been observed in some pure optimization problems before, albeit under a-typical circumstances. For instance, Zhang and Korf [10] identify a similar pattern for the Traveling Salesperson Problem, using a log-normal distribution of the distance function. In our case, the pattern appears to emerge naturally from the model.

These aspects are quite intriguing and require further study. Of course, we do not claim that these observations will hold for all optimization problems that involve a feasibility component. In fact, quite often the feasibility part of optimization tasks is relatively easy provided one has sufficient resources (including budget). However, our study suggests that there may be classes of models or even problems where the feasibility component in and of itself is surprisingly hard, even compared to the optimization aspect. One issue that requires further research is the extent to which the mixed integer programming (MIP) formulation and the Cplex algorithm are well suited to capture the combinatorial nature of the feasibility problem. In Section 6, we mention two alternative problem formulation/solution methods that might initially appear to be more promising than using Cplex on a pure MIP formulation, but are unlikely to change the overall

picture. Lastly, we note that from a practical point of view, this interplay between feasibility and optimization can be quite important. For example, under tight budget constraints, one may want to spend significant computational resources to ensure that no feasible solution exists, before deciding on an increased budget or another relaxation of the problem constraints.

The rest of the paper is organized as follows. In Section 2 we present the connection subgraph problem. We discuss the theoretical complexity of this problem in Section 3. Section 4 describes our Mixed Integer Linear Programming model of the connection subgraph problem. The empirical results are presented in Section 5. Finally, we conclude with a discussion in Section 6.

2 Connection Subgraph Problem

Let $\mathbb{Z}^+$ denote the set $\{0, 1, 2, \ldots\}$ of non-negative integers. The decision version of the connection subgraph problem is defined on an undirected graph as follows:

Definition 1 (Connection Subgraph Problem). *Given an undirected graph $G = (V, E)$ with terminal vertices $T \subseteq V$, vertex costs $c : V \to \mathbb{Z}^+$, vertex utilities $u : V \to \mathbb{Z}^+$, a cost bound $C \in \mathbb{Z}^+$, and a desired utility $U \in \mathbb{Z}^+$, does there exist a vertex-induced subgraph H of G such that*

1. *H is connected,*
2. *$T \subseteq V(H)$, i.e., H contains all terminal vertices,*
3. *$\sum_{v \in V(H)} c(v) \leq C$, i.e., H has cost at most C, and*
4. *$\sum_{v \in V(H)} u(v) \geq U$, i.e., H has utility at least U?*

In this decision problem, we can relax one of the last two conditions to obtain two natural optimization problems:

- **Utility Optimization:** given a cost bound C, maximize the utility of H,
- **Cost Optimization:** given a desired utility U, minimize the cost of H.

3 NP-Completeness and Hardness of Approximation

The connection subgraph problem is a generalized variant of the Steiner tree problem on graphs, with costs on vertices rather than on edges and with utilities in addition to costs. The utilities add a new dimension of hardness to the problem. In fact, while the Steiner tree problem is polynomial time solvable when $|T|$ is any fixed constant [cf. 8], we will show that the connection subgraph problems remains NP-complete even when $|T| = 0$. We prove this by a reduction from the Steiner tree problem. This reduction also applies to planar graphs, for which the Steiner tree problem is still NP-complete [cf. 8].

Theorem 1 (NP-Completeness). *The decision version of the connection subgraph problem, even on planar graphs and without any terminals, is NP-complete.*

Proof. The problem is clearly in NP, because a certificate subgraph H can be easily verified to have the desired properties, namely, connectedness, low enough cost, and high enough utility. For NP-hardness, consider the Steiner tree problem on a graph $\widehat{G} = (\widehat{V}, \widehat{E})$ with terminal set $\widehat{T} \subseteq \widehat{V}$, edge cost function $\widehat{c} : \widehat{E} \to \mathbb{Z}^+$, and cost bound $\widehat{C}$.

An instance of the connection subgraph problem can be constructed from this as follows. Construct a graph $G = (V, E)$ with $V = \widehat{V} \cup \widehat{E}$ and edges defined as follows. For every edge $e = \{v, w\} \in \widehat{E}$, create edges $\{v, e\}, \{w, e\} \in E$. The terminal set remains the same: $T = \widehat{T}$. Overall, $|V| = |\widehat{V}| + |\widehat{E}|$, $|E| = 2|\widehat{E}|$, and $|T| = |\widehat{T}|$. For costs, set $c(v) = 0$ for $v \in \widehat{V}$ and $c(e) = \widehat{c}(e)$. For utilities, set $u(v) = 1$ for $v \in T$ and $u(v) = 0$ for $v \notin T$. Finally, the cost bound for the connection subgraph is $C = \widehat{C}$ and the utility bound is $U = |E|$.

It is easy to verify that the Steiner tree problem on $\widehat{G}$ and $\widehat{T}$ has a solution with cost at most C iff the connection subgraph problem on G and T has a solution with cost at most C and utility at least U. This completes the reduction.

Note that if $\widehat{G}$ is planar, then so is G. Further, the reduction is oblivious to the number of terminals in G. Hence, NP-completeness holds even on planar graphs and without any terminals. □

This immediately implies the following:

Corollary 1 (NP-Hardness of Optimization). *The cost and utility optimization versions of the connection subgraph problem, even on planar graphs and without any terminals, are both NP-hard.*

Observe that in the reduction used in the proof of Theorem 1, $\widehat{G}$ has a Steiner tree with cost C' iff G has a connection subgraph with cost C'. Consequently, if the cost optimization version of the connection subgraph instance (i.e., cost minimization) can be approximated within some factor $\alpha \geq 1$ (i.e., if one can find a solution of cost at most α times the optimal), then the original Steiner tree problem can also be approximated within factor α. It is, however, known that there exists a factor α_0 such that the Steiner tree problem *cannot* be approximated within factor α_0, unless P=NP. This immediately gives us a hardness of approximation result for the utility optimization version of the connection subgraph problem. Unfortunately, the best known value of α_0 is roughly $1 + 10^{-7}$ [cf. 8].

We now describe a different reduction — from the NP-complete Vertex Cover problem — which will enable us to derive as a corollary a much stronger approximation hardness result.

Lemma 1. *There is a polynomial time reduction from Vertex Cover to the connection subgraph problem, even without any terminals, such that the size of the vertex cover in a solution to the former equals the cost of the subgraph in a solution to the latter.*

Proof. We give a reduction along the lines of the one given by Bern and Plassmann [2] for the Steiner tree problem. The reduction is oblivious to the number of terminals, and holds in particular even when there are no terminals.

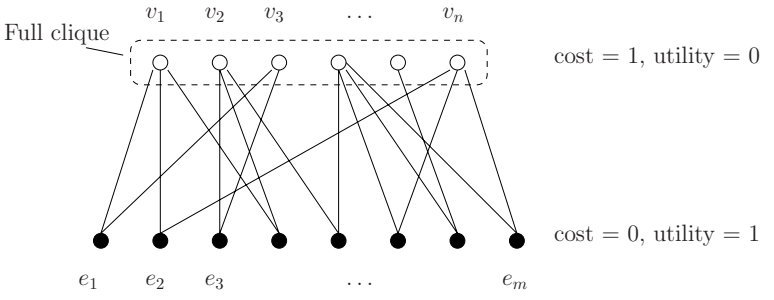


Fig. 3. Reduction from Vertex Cover

Recall that a vertex cover of a graph $\widehat{G} = (\widehat{V}, \widehat{E})$ is a set of vertices $V' \subseteq \widehat{V}$ such that for every edge $\{v, w\} \in \widehat{E}$, at least one of v and w is in V'. The vertex cover problem is to determine whether, given $\widehat{G}$ and $C \geq 0$, there exists a vertex cover V' of $\widehat{G}$ with $|V'| \leq C$. We convert this into an instance of the connection subgraph problem. An example of such a graph is depicted in Fig. 3.

Create a graph $G = (V, E)$ with $V = \widehat{V} \cup \widehat{E}$ and edges defined as follows. For every $v, w \in \widehat{V}, v \neq w$, create edge $\{v, w\} \in E$; for every $e = \{v, w\} \in \widehat{E}$, create edges $\{v, e\}, \{w, e\} \in E$. Overall, G has $|\widehat{V}| + |\widehat{E}|$ vertices and $\binom{\widehat{V}}{2} + 2\widehat{E}$ edges. For costs, set $c(v)$ to be 1 if $v \in \widehat{V}$, and 0 otherwise. For utilities, set $u(e)$ to be 1 if $e \in \widehat{E}$, and 0 otherwise. Finally, fix the set of terminals to be an arbitrary subset of $\widehat{E}$.

We prove that solutions to the connection subgraph problem on G with costs and utilities as above, cost bound C, and desired utility $U = |\widehat{E}|$ are in one-to-one correspondence with vertex covers of $\widehat{G}$ of size at most C.

First, let vertex-induced subgraph H of G be a solution to the connection subgraph instance. Let $V' = V(H) \cap \widehat{V}$. We claim that V' is a vertex cover of $\widehat{G}$ of size at most C. Clearly, $|V'| \leq C$ because of the cost constraint on H. To see that V' is indeed a vertex cover of $\widehat{G}$, note that (A) because of the utility constraint, V' must contain *all* of the vertices from $\widehat{E}$, and (B) because of the connectedness constraint, every such vertex must have at least one edge in $E(H)$, i.e., for each $e = \{v, w\} \in \widehat{E}$, V' must include at least one of v and w.

Conversely, let V' be a vertex cover of $\widehat{G}$ with at most C vertices. This directly yields a solution H of the connection subgraph problem: let H be the subgraph of G induced by vertices $V' \cup \widehat{E}$. By construction, H has the same cost as V' (in particular, at most C) and has utility exactly U. Since V' is a vertex cover, for every edge $e = \{v, w\} \in \widehat{E}$, at least one of v and w must be in V', which implies that H must have at least one edge involving e and a vertex in V'. From this,

and the fact that all vertices of V' already form a clique in H, it follows that H itself is connected.

This settles our claim that solutions to the two problem instances are in one-to-one correspondence, and finishes the proof. □

Combining Lemma 1 with the fact that the vertex cover problem is NP-hard to approximate within a factor of 1.36 [4] immediately gives us the following:

Theorem 2 (APX-Hardness of Cost Optimization). *The cost optimization version of the connection subgraph problem, even without any terminals, is NP-hard to approximate within a factor of* 1.36.

4 Mixed Integer Linear Programming Model

Next we present the Mixed Integer Linear Programming Model (MIP model) for the connection subgraph problem, that we used in our experiments. Let $G = (V, E)$ be the graph under consideration, with $V = \{1, \ldots, n\}$.

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

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

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

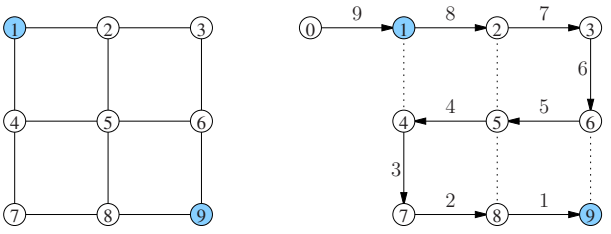
$$x_i \in \{0, 1\}, \quad \forall i \in V. \tag{3}$$

To ensure the connectivity of the subgraph, we apply a particular network flow model, where the network is obtained by replacing all undirected edges $\{i, j\} \in E$ by two directed edge $\{i, j\}$ and $\{j, i\}$. First, we introduce a source vertex 0, with maximum total outgoing flow n. We arbitrarily choose one terminal vertex $t \in T$, and define a directed edge $\{0, t\}$ to insert the flow into the network, assuming that there exists at least one such vertex.[1] Then, by demanding that the flow reaches all terminal vertices, the edges carrying flow (together with the corresponding vertices) represent a connected subgraph. To this end, each of the vertices with a positive incoming flow will act as a 'sink', by 'consuming' one unit of flow. In addition, flow conservation holds: for every vertex the amount of incoming flow equals the amount of outgoing flow.

More formally, for each edge $\{i, j\} \in E$, we introduce a nonnegative variable y_{ij} to indicate the amount of flow from i to j. For the source, we introduce a variable $x_0 \in [0, n]$, representing the eventual residual flow. The insertion of the flow into the network is then stated as:

$$x_0 + y_{0t} = n, \tag{4}$$

[1] If there are no terminal vertices specified, we add edges from the source to all vertices in the graph, and demand that at most one of these edges is used to carry flow.



a. Original graph b. Feasible flow

Fig. 4. Flow representation of the connection subgraph problem on a graph with 9 vertices. The terminal vertices 1 and 9 are shaded.

where $t \in T$ is arbitrarily chosen. Each of the vertices with positive incoming flow retains one unit of flow, i.e., $(y_{ij} > 0) \Rightarrow (x_j = 1), \forall \{i, j\} \in E$. We convert this relation into a linear constraint:

$$y_{ij} < nx_j, \ \forall \{i, j\} \in E. \tag{5}$$

The flow conservation is modeled as:

$$\sum_{i:\{i,j\}\in E} y_{ij} = x_j + \sum_{i:\{j,i\}\in E} y_{ij}, \ \forall j \in V. \tag{6}$$

Finally, terminal vertices retain one unit of flow:

$$x_t = 1, \ \forall t \in T. \tag{7}$$

In Figure 4 we give an example of our flow representation, where we omit the costs for clarity. Figure 4a presents a graph on 9 vertices with terminal vertices 1 and 9. In Figure 4b, a feasible flow for this graph is depicted, originating from the source 0, with value 9. It visits all vertices, while each visited vertex consumes one unit of flow. The thus connected subgraph contains all vertices in this case, including all terminal vertices.

5 Computational Hardness Profiles

We next perform a detailed empirical study of the connection subgraph problem. In this study, our parameter is the feasibility component of the problem, i.e., the cost bound (or budget). For a varying budget, we investigate the satisfiability of the problem, as well as its computational hardness with respect to proving infeasibility or optimality.

In our experiments, we make use of semi-structured graphs, with uniform random utility and cost functions. The graphs are composed of an $m \times m$ rectangular lattice or grid, where the order m is either 6, 8, or 10. This lattice graph is motivated by the structure of the original conservation corridors problem. In

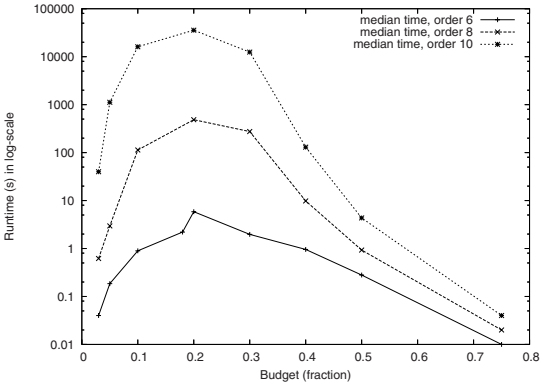


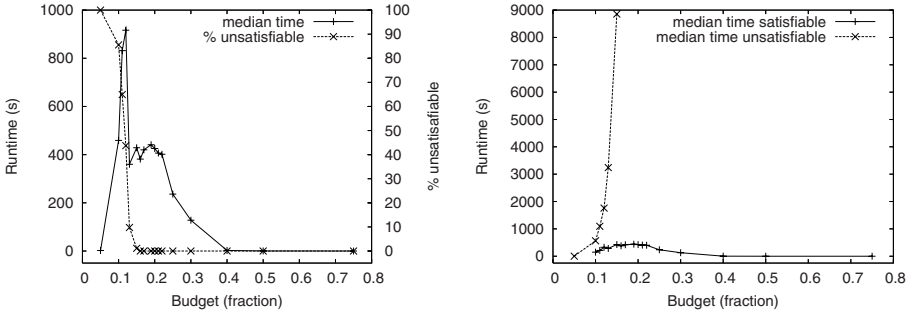
Fig. 5. Hardness profile for lattices of order 6, 8, and 10, without terminal vertices

this lattice, we place k terminal vertices, where k is 0, 3, 10, or 20. When $k \geq 2$, we place two terminal vertices in the 'upper left' and 'lower right' corners of the lattice, so as to maximize the distance between them and "cover" most of the graph. This is done to avoid the occurrence of too many pathological cases, where most of the graph does not play any role in constructing an optimal connection subgraph. The remaining $k-2$ terminal vertices are placed uniformly at random in the graph. To define the utility and cost functions, we assign uniformly at random a utility and a cost from the set $\{1, 2, \ldots, 10\}$ to each vertex in the graph. The cost and utility functions are uncorrelated.

In the figures below, each data point is based on 100 random instances or more, at a given budget. For the figures comparing infeasible and feasible instances, this means that the sum of the feasible and infeasible instances at each budget is at least 100. The hardness curves are represented by median running times over all instances per data point, while for the feasibility curves we take the average. As the scale for the budget (on the x-axis), we use the following procedure. For every instance, we compute the total cost of all vertices. The budget is calculated as a fraction of this total cost. We plot this fraction on the x-axis. All our experiments were conducted on a 3.8 GHz Intel Xeon machine with 2 GB memory running Linux 2.6.9-22.ELsmp. We used Cplex 10.1 [7] to solve the MIP problems.

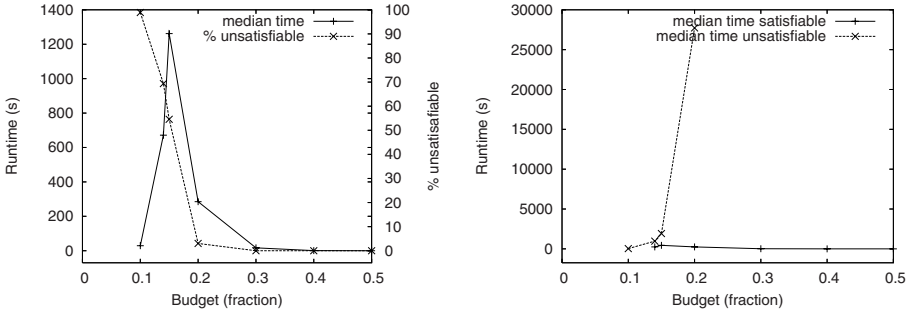
First, we present computational results on graphs without terminal vertices. These problems are always satisfiable, and can thus be seen as pure optimization problems. Figure 5 shows the hardness profile (i.e., the running time) on lattices of order 6, 8, and 10. Notice that the median time is plotted in log-scale in this figure. The plots clearly indicate an easy-hard-easy pattern for these instances, even though they are all feasible with respect to the budget. As remarked earlier, such patterns have been observed earlier in some pure optimization problems, but only under specific random distributions.

Second, we turn our attention to graphs with terminal vertices. In Figure 5.a, we show the hardness profile of lattices of order 10, with 3 terminals. In addition,



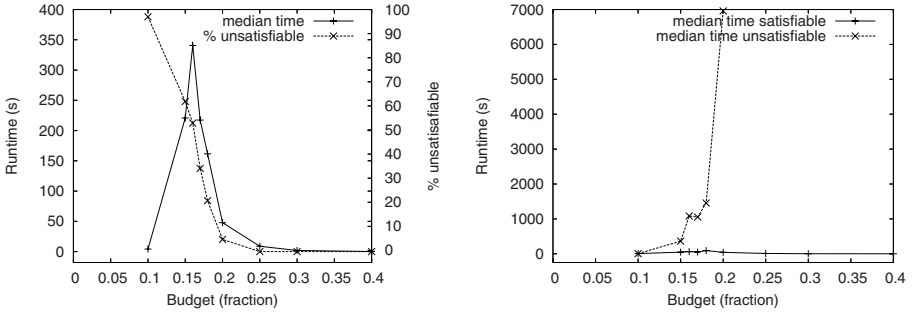
a. Hardness and SAT/UNSAT profile b. Infeasibility vs. optimization

Fig. 6. Hardness and satisfiability profiles for lattices of order 10 with 3 terminals



a. Hardness and SAT/UNSAT profile b. Infeasibility vs. optimization

Fig. 7. Hardness and satisfiability profiles for lattices of order 10 with 10 terminals



a. Hardness and SAT/UNSAT profile b. Infeasibility vs. optimization

Fig. 8. Hardness and satisfiability profiles for lattices of order 10 with 20 terminals

the satisfiability profile is shown in this figure: we plot the percentage of unsatisfiable instances with respect to a varying budget. Figures 7.a and 8.a present similar graphs for lattices of order 10, with 10 and 20 terminals, respectively. In all figures, we see a sharp phase transition from a region in which almost all instances are unsatisfiable, to a region in which almost all instances are satisfiable (when the budget fraction is around 0.15). Furthermore, again these problems exhibit an easy-hard-easy pattern, the peak of which coincides with the satisfiability phase transition with respect to the budget. Similar relations between the peak of computational hardness and feasibility phase transitions have been demonstrated often before for pure satisfiability problems. However, we are unaware of such results for problems combining both a feasibility and an optimality aspect.

Our experiments also indicate an easy-hard-easy pattern for the hardness of the problem, depending number of terminals in the graph. For 10 terminals, the problems are considerably more difficult than for 3 or 20 terminals. Intuitively, this can be explained by two rivaling aspects: the difficulty of connecting k terminals, and the complexity on $n - k$ free variables. As k increases, it is more difficult to connect the terminals. However, when k is large, the resulting problem on $n - k$ variables becomes easy.

Finally, we compare the hardness of optimization to the hardness of proving infeasibility. To this end, we separate the hardness profiles for satisfiable and unsatisfiable problem instances. The resulting plots for lattices of order 10 with 3, 10, and 20 terminals are depicted in Figure 5.b, Figure 7.b, and Figure 8.b, respectively. In these figures, the curve for unsatisfiable instances represents the hardness of proving infeasibility, while the curve for satisfiable instances represents the hardness of proving optimality. Clearly, proving infeasibility becomes increasingly more difficult when the budget increases, especially inside the phase transition region. At the same time, the difficulty of proving optimality does not exhibit this extreme behavior. In fact, when the budget fraction is around 0.15, we observe that proving infeasibility takes up to 150 times longer than proving optimality.

6 Summary and Discussion

In this work, we investigated the interplay between the computational tasks of feasibility testing and optimization. We studied in detail the connection subgraph problem, for which we presented theoretical worst-case complexity results, as well as empirical typical-case results. Our experiments reveal interesting trade-offs between feasibility testing and optimization. One of our main observations is that proving infeasibility can be considerably more difficult than proving optimality in a computationally hard region of the problem space. In addition to this, we identified a satisfiability phase transition coinciding with the complexity peak of the problem. Somewhat more surprisingly, for the optimization component itself, we discovered an easy-hard-easy pattern based on the feasibility parameter, even when the underlying problems are always satisfiable.

In our experimental results, we have applied a mixed integer linear programming model in conjunction with the solver Cplex. Naturally, one could argue

that a different solver or even a different model could have produced different results. For example, one might propose to check separately the feasibility of the cost constraint before applying a complete solver. Indeed, checking feasibility of the cost constraint is equivalent to the metric Steiner tree problem. Although this latter problem is solvable in polynomial time for a constant number of terminals, it is likely not to be fixed parameter tractable [8]. Hence, it appears unrealistic to apply such a separate feasibility check as a pre-processor before using a complete solution technique.

Another direction is to apply a constraint programming (CP) model, which could perhaps better tackle the feasibility aspect of the problem. However, a good CP model should ideally capture the cost constraint as a whole, for example as a global constraint. For the same reason as above, it is unlikely that an efficient and effective filtering algorithm exists for such a constraint. Moreover, a CP model by itself is not particularly suitable for the optimization component. More specifically, for the connection subgraph problem the objective is a weighted sum, which is known to be difficult to handle by constraint solvers. Nevertheless, a hybrid constraint programming and mixed integer programming approach might be effective for this problem, which we leave open as future work.

References

[1] A. Ando, J. Camm, S. Polasky, and A. Solow. Special distributions, land values, and efficient conservation. *Science*, 279(5359):2126–2128, 1998.
[2] M. W. Bern and P. E. Plassmann. The Steiner tree problem with edge lengths 1 and 2. *Information Processing Letters*, 32(4):171–176, 1989.
[3] J. D. Camm, S. K. Norman, S. Polasky, and A. R. Solow. Nature reserve site selection to maximize expected species covered. *Operations Research*, 50(6):946–955, 2002.
[4] I. Dinur and S. Safra. On the hardness of approximating minimum vertex cover. *Annals of Mathematics*, 162(1):439–486, 2005.
[5] C. Faloutsos, K. McCurley, and A. Tomkins. Fast discovery of connection subgraphs. In *Proceedings of the 2004 ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 118–127. ACM Press, 2004.
[6] I. Gent and T. Walsh. The TSP Phase Transition. *Artificial Intelligence*, 88(1–2): 349–358, 1996.
[7] ILOG, SA. CPLEX 10.1 Reference Manual, 2006.
[8] H. J. Prömel and A. Steger. *The Steiner Tree Problem: A Tour Through Graphs, Algorithms, and Complexity.* Vieweg, 2002.
[9] D. Simberloff, J. Farr, J. Cox, and D. Mehlman. Movement corridors: Conservation bargains or poor invesments? *Conservation Biology*, 6:493–504, 1997.
[10] W. Zhang and R. Korf. A Study of Complexity Transitions on the Asymmetric Traveling Salesman Problem. *Artificial Intelligence*, 81:223–239, 1996.

Modeling the Regular Constraint with Integer Programming

Marie-Claude Côté[1,3], Bernard Gendron[2,3], and Louis-Martin Rousseau[1,3]

[1] Département de mathématiques appliquées et génie industriel, École Polytechnique de Montréal, Montréal, Canada
[2] Département d'informatique et de recherche opérationnelle, Université de Montréal, Montréal, Canada
[3] Interuniversity Research Center on Enterprise Networks, Logistics and Transportation, Montréal, Canada
{macote,bernard,louism}@crt.umontreal.ca

Abstract. Many optimisation problems contain substructures involving constraints on sequences of decision variables. Such constraints can be very complex to express with mixed integer programming (MIP), while in constraint programming (CP), the global constraint `regular` easily represents this kind of substructure with deterministic finite automata (DFA). In this paper, we use DFAs and the associated layered graph structure built for the `regular` constraint consistency algorithm to develop a MIP version of the constraint. We present computational results on an employee timetabling problem, showing that this new modeling approach can significantly decrease computational times in comparison with a classical MIP formulation.

1 Introduction

Many optimisation problems contain substructures involving constraints on sequences of decision variables. Such substructures are often present, for example, in rostering and car sequencing problems. This paper uses constraint programming (CP) global constraint `regular` [1] and integrates it to mixed integer programming (MIP) models to express constraints over sequences of decision variables. Specifically, we use the layered graph built by the `regular` constraint consistency algorithm to formulate a network flow problem that represents feasible sequences of values. This modeling approach allows MIP formulations to use the expressiveness of deterministic finite automata (DFA), in a way similar to the `regular` constraint. We present an application of this approach with an employee timetabling problem.

The paper is organized as follows. Section 2 presents a review of the literature on modeling constrained sequences of variables with MIP and CP and provides background material about the CP `regular` constraint and network flow theory. Section 3 gives details about the MIP version of the `regular` constraint. Finally, Section 4 describes an employee timetabling problem and compares computational results obtained with a classical MIP model of the problem and with the alternative formulation using the `MIP regular` constraint.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 29–43, 2007.

2 Literature Review

2.1 MIP Formulations

The literature on staff scheduling and rostering problems offers many different approaches to model constrained sequences of decision variables (see [2,3] for recent surveys on staff scheduling and rostering).

Mathematical programming approaches to model sequences of variables can be divided in three categories: set covering with explicit formulations (Dantzig [4], Segal [5]), set covering with implicit formulations (Moondra [6], Bechtolds and Jacobs [7,8], Thompson [9], Aykin [10], Brusco and Jacobs [11]) and compact formulations (Laporte et al. [12], Balakrishnan and Wong [13], Isken [14]).

Çezik et al. [15] propose a MIP formulation for the Weekly Tour Scheduling Problem. It handles the weekly horizon by combining seven daily shift-timetabling models in a network flow framework, which handles the weekly requirements. The network flow framework is similar to the structure we propose here. Millar et Kiragu [16] and Ernst et al. [17] use a layered network to represent allowed transitions between shift patterns to develop rosters. The shift patterns are fixed a priori. Sodhi [18] combines weekly patterns to create an entire cyclic roster by forming a directed graph with nodes representing allowed weekly patterns and arcs representing allowed week-to-week transitions between these patterns. A MIP model is then used to find an optimal cyclic path to cover all the weeks of the roster. None of these works suggest a way to generate the shift patterns automatically. We present a way to implicitly express a large set of rules and represent all possible patterns for a given planning horizon.

2.2 CP Formulations

CP global constraints allow to formulate some rules over sequences of decision variables in an expressive way. In particular, the `pattern` constraint and the `stretch` constraint are well adapted to these types of problems. The `regular` constraint, introduced in [1], can express several existing global constraints. This constraint enables to formulate constrained sequences of variables effectively with CP. Before we define the `regular` constraint, we briefly introduce important definitions related to automata theory and regular languages (for more details on the subject, see [19]).

A deterministic finite automaton (DFA) is described by a 5-tuple $\Pi = (Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states;
- Σ is an alphabet;
- $\delta : Q \times \Sigma \rightarrow Q$ is a transition function;
- $q_0 \in Q$ is the initial state;
- $F \subseteq Q$ is a set of final states.

An alphabet is a finite set of symbols. A language is a set of words, formed by symbols over a given alphabet. Regular languages are languages recognized by a DFA.

The `regular` constraint is defined as follows:

Definition 1. ***(Regular language membership constraint).*** *Let $\Pi = (Q, \Sigma, \delta, q_0, F)$ denote a DFA, let $L(\Pi)$ be the associated regular language and let $X = x_1, x_2, \ldots, x_n$ be a finite sequence of variables with respective finite domains $D_1, D_2, \ldots, D_n \subseteq \Sigma$. Then*

$$\texttt{regular}(X, \Pi) = \{(d_1, \ldots, d_n) | d_i \in D_i, d_1 d_2 \ldots d_n \in L(\Pi)\}$$

Note, that other variants of the `regular` constraint have been studied, in particular, a `soft-regular` constraint for over constrained problems [20] and a `cost-regular` approach [21].

For the rest of this work, we are interested in the layered graph representation of the `regular` constraint used for the consistency algorithm. Let X be a sequence of variables constrained by a `regular` constraint, n be the sequence length and k be the number of states in the DFA Π of the constraint. Let G be the associated layered graph. G has $n + 1$ layers $(N^1, N^2, \ldots, N^{n+1})$. Each layer counts at most k nodes, associated with the DFA states. We note q_k^i the node representing state k of Π in layer i of G. The `initialize`() procedure in [1] builds G. At the end of the procedure, every arc out of a layer i of G represents a consistent value for domain D_i and each path from initial state q_0 to a final state q_f^{n+1}, $f \in F$ represents an admissible instantiation for the variables X.

Example 1. Let $\Sigma = \{a, b, c\}$ be an alphabet and Π be a DFA, represented in Figure 1, recognizing a regular language over this alphabet. The constraint `regular`(X, Π), where $X = x_1, x_2, \ldots, x_5$ and $D_1 = D_2 = \ldots = D_5 = \{a, b, c\}$, would give the layered graph in Figure 2 after the `initialize`() procedure [1].

The layered graph suggested in [1] is not the only encoding available for the `regular` constraint. Notably, Quimper and Walsh [22] proposed to encode it using a simple sequence of ternary constraints. They use the sequence of decision variables X and introduce the sequence of variables Q_0 to Q_n to represent the

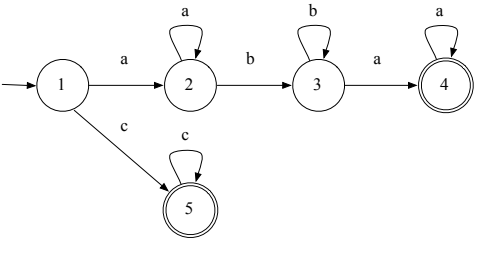


Fig. 1. DFA Π with each state shown as a circle, each final state as double circle, and each transition as an arc

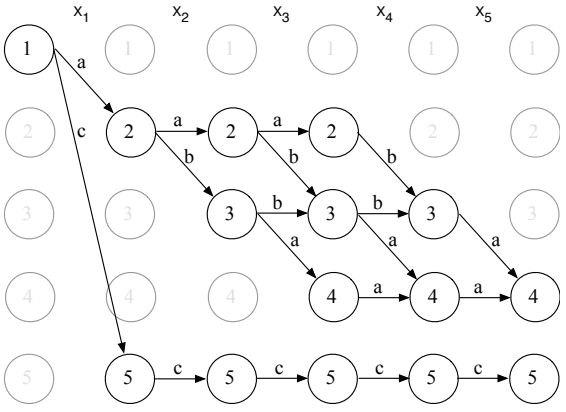


Fig. 2. Layered graph builded by the `initialize()` procedure for `regular` $(x_1, x_2, \ldots, x_5, \Pi)$

states of the DFA. Then, transition constraints $C(x_{i+1}, Q_i, Q_{i+1})$ are posted for $0 \leq i < n$, which hold iff $Q_{i+1} = \delta(x_{i+1}, Q_i)$, where δ is the transition function. To complete the encoding, unary constraints $Q_0 = q_0$ and $Q_n \in F$ are also posted. Basically, after a filtering of the domains on such an encoding, each consistent triplet (x_{i+1}, Q_i, Q_{i+1}) is equivalent to an arc in the layered graph described above and each triplet are linked together by the the state variables (Q_0 to Q_n).

2.3 Network Flow Theory

Let $G = (V, A)$ be a directed graph with V, the vertex set, A, the arc set, and nonnegative capacities μ_{ij} associated with each arc $(i, j) \in A$. We distinguish two special nodes in G: the source node s and the sink node t. A flow from s to t in G is defined by the flow variables f_{ij} on each arc (i, j) that represent the amount of flow on this arc. The flow is subject to the following constraints, where w is the value of the flow from s to t:

$$\sum_{\{j:(i,j)\in A\}} f_{ij} - \sum_{\{j:(j,i)\in A\}} f_{ji} = \begin{cases} w, & \text{for } i = s, \\ 0, & \text{for all } i \in V - \{s, t\}, \\ -w, & \text{for } i = t, \end{cases} \tag{1}$$

$$0 \leq f_{ij} \leq \mu_{ij}, \qquad \forall (i, j) \in A. \tag{2}$$

Equations (1) ensure that for any node $q \neq s, t$, the amount of flow entering q is equal to the amount of flow leaving q. They are called flow conservation equations. Constraints (2) ensure that the amount of flow on an arc is nonnegative and does not exceed its capacity.

For more information on network flow theory, we refer to [23].

3 MIP Regular Constraint

The use of DFAs to express constraints on values taken by sequences of variables is very useful in CP. Equivalent constraints can be very complex to formulate in a MIP 0-1 compact model. The aim of our work is precisely to propose a way to formulate MIP 0-1 compact models by using DFAs. Our approach is based on the CP `regular` constraint. Before presenting the `MIP regular` constraint, we recall a well-known correspondence between CP and MIP 0-1 variables.

Let X_i be a CP variable in a sequence of n variables with domain D_i, $i = 1, 2, \ldots, n$. We define a corresponding set of MIP 0-1 variables as follows. For all i, for all $j \in D_i$, we associate a 0-1 variable x_{ij}. We also need the following constraints: $\sum_{j \in D_i} x_{ij} = 1$ for all i. These constraints ensure that only one value is assigned to a position of the sequence. A solution with $x_{ij} = 1$ means "position i of the sequence is assigned to value j".

The idea behind the `MIP regular` constraint is to use the layered graph built and filtered by the `initialize`() procedure for the consistency algorithm of the CP `regular` constraint [1]. We create a similar graph, but, instead of having, between each layer of the graph, arcs related to values of a domain, we have a MIP 0-1 variable for each arc of the graph. With little adjustment of this structure, we can approach the problem as a network flow problem. With one unit of flow entering and leaving the graph and capacities equal to one on each arc, each "position" is assigned to exactly one value. This property captures the $\sum_{j \in D_i} x_{ij} = 1, \forall i$ constraints.

To complete the transformation, we have to define a source node and a sink node in this graph. The source node is simply the node related to the initial state of the associated DFA on the first layer. We introduce a sink node t connected to each node representing a final state of the DFA on the last layer. A feasible flow in this graph represents an instantiation in the CP version.

To write the flow conservation equations correctly, we must consider that, in the CP layered graph, multiple arcs representing the same value may be related to the same variable. For instance, in the graph of Figure 2, x_{3b} would correspond to the arc from node 2 of layer 3 to node 3 of layer 4, and, to the arc from node 3 of layer 3 to node 3 of layer 4.

To distinguish the arcs we introduce the $s_{ijq_k} \in \{0, 1\}$ *flow variables*, where s_{ijq_k} is the amount of flow on the arc leaving node representing state q_k of layer i with value j. These variables are unique due to the determinism of the automaton. We also introduce the variables $sf_{q_k} \in \{0, 1\}$ to represent the arcs leaving the final states of the last layer. The variable $w \in \{0, 1\}$ is the amount of flow entering and leaving the graph.

An arc from a to c with label b is defined as a triplet (a, b, c). We note $inArcs[i][k]$ and $outArcs[i][k]$ the sets of arcs entering and leaving the node q_k of layer i. The MIP formulation of the `regular` constraint is then written as follows:

$$\sum_{j \in \Omega_{10}} s_{1jq_0} = w, \tag{3}$$

$$\sum_{(j,q'_k) \in \Delta_{ik}} s_{(i-1)jq'_k} = \sum_{j \in \Omega_{ik}} s_{ijq_k}, \quad \forall i \in \{2, \ldots, n\}, q_k \in N^i, \tag{4}$$

$$\sum_{(j,q'_k) \in \Delta_{(n+1)k}} s_{njq'_k} = sf_{q_k}, \quad \forall q_k \in N^{n+1}, \tag{5}$$

$$\sum_{q_k \in N^{n+1}} sf_{q_k} = w, \tag{6}$$

$$x_{ij} = \sum_{q_k \in \Sigma_{ij}} s_{ijq_k}, \quad \forall i \in \{1, \ldots, n\}, j \in D_i, \tag{7}$$

$$s_{ijq_k} \in \{0,1\} \quad i \in \{1, \ldots, n\}, q_k \in N^i, j \in \Omega_{ik}, \tag{8}$$

$$sf_{q_k} \in \{0,1\} \quad q_k \in N^{n+1}, \tag{9}$$

where $\Delta_{ik} = \{(j, q'_k) | (q'_k, j, q_k) \in inArcs[i][k]\}$, $\Omega_{ik} = \{j | (q_k, j, \_) \in outArcs[i][k]\}$, $\Sigma_{ij} = \{q_k | (q_k, j, \_) \in outArcs[i][k]\}$.

Constraints (7) link the *decision variables* x with the *flow variables*. Note that in a case where the `MIP regular` constraint is the only constraint in the MIP Model, the *decision variables* and constraints (7) are not needed in the model. Without them, the resulting model is a network flow formulation that can be solved by a specialized algorithm [23].

Thus, introducing a `MIP regular` constraint to a MIP model induces the addition of a set of flow conservation linear constraints (3)-(6) and linking constraints (7) to the model. We use a procedure with the following signature:

$$\texttt{AddMIPRegular}(\Pi(Q,\Sigma,\delta,q_0,F), n, x, w, M),$$

to add the linear constraints associated with a `MIP regular` constraint to a model M, given a DFA Π, the *decision variables* x subject to the constraint, the length of the sequence n formed by these variables and the amount of flow w entering the graph.

Example 2. Figure 3 is the graph for `MIP regular`$(x_1, x_2, \ldots, x_5, \Pi)$, where Π is the DFA presented in Figure 1. With the domains of Example 1, the constraint would add all the flow conservation constraints related to this graph and the following "linking" constraints to the MIP model:

$x_{1a} = s_{1a1}$	$x_{2a} = s_{2a2}$	$x_{3a} = s_{3a2} + s_{3a3}$	$x_{4a} = s_{4a3} + s_{4a4}$	$x_{5a} = s_{5a3} + s_{5a4}$
$x_{1b} = 0$	$x_{2b} = s_{2b2}$	$x_{3b} = s_{3b2} + s_{3b3}$	$x_{4b} = s_{4b2} + s_{4b3}$	$x_{5b} = 0$
$x_{1c} = s_{1c1}$	$x_{2c} = s_{2c5}$	$x_{3c} = s_{3c5}$	$x_{4c} = s_{4c5}$	$x_{5c} = s_{5c5}$

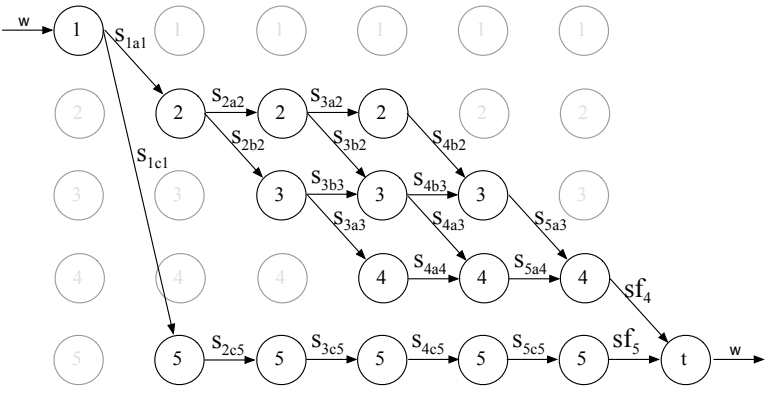


Fig. 3. Layered graph for `MIP regular`($x_1, x_2, \ldots, x_5$, Π)

The `MIP regular` constraint is easily extended to a cost version by adding costs on the decision variables in the objective function of the model.

4 Case Study

To evaluate the quality of our modeling approach for constrained sequences of decision variables, we present computational results on employee timetabling problems described in [24]. The benchmarks are randomly generated but are based on data rules from a real-world timetabling problem. The demand curves come from a retail store. The objective is to create an optimal employee timetable for one day that satisfies the demands for each work activity and the work regulation rules. For our experiments, we used datasets with 1 or 2 work activities. For each of these datasets, 10 instances are available.

4.1 Problem Definition

The one day planning horizon is decomposed into 96 periods of 15 minutes each. Let us introduce the following notations before we define the problem:

- I : set of available employees.
- W : set of work activities.
- A : set of all activities ($A = W \cup \{l, p, o\}$)
  where l = lunch, p = break, o = rest.
- $T = \{1, 2, \ldots, n\}$: set of periods. $T' = T - \{1\}$.
- $F_t \subseteq W$: set of activities that are not allowed to be performed at period $t \in T$.
- c_{at}: cost for an employee to cover an activity $a \in W - F_t$ at period $t \in T$.

Work Regulation Rules

1. Activities $a \in F_t$ are not allowed to be performed at period $t \in T$.
2. If an employee is working, he must cover between 3 hours and 8 hours of work activities.
3. If a working employee covers at least 6 hours of work activities, he must have two 15 minute-breaks and a lunch break of 1 hour.
4. If a working employee covers less than 6 hours of work activities, he must have a 15 minute break, but no lunch.
5. If performed, the duration of any activity $a \in W$ is at least 1 hour (4 consecutive periods).
6. A break (or lunch) is necessary between two different work activities.
7. Work activities must be inserted between breaks, lunch and rest stretches.
8. Rest shifts have to be assigned at the beginning and at the end of the day.

Demand Covering

1. The required number of employees for activity $a \in W - F_t$ at period $t \in T$ is d_{at}. Undercovering and overcovering are allowed. The cost of undercovering activity $a \in W - F_t$ at period $t \in T$ is c^-_{at} and the cost of overcovering activity $a \in W - F_t$ at period $t \in T$ is c^+_{at}.

The following sections present two ways of modeling this problem. The first model is a classical MIP formulation that does not exploit the `MIP regular` constraint. The second model uses the `MIP regular` constraint.

4.2 A Classical MIP Model

Decision Variables

$$x_{iat} = \begin{cases} 1, \text{ if employee } i \in I \text{ covers activity } a \in A \text{ at period } t \in T, \\ 0, \text{ otherwise.} \end{cases}$$

Work Regulation Rules

Rule 1:

$$x_{iat} = 0, \qquad i \in I, t \in T, a \in F_t. \tag{10}$$

Rule 2:

$$w_i = \begin{cases} 1, \text{ if employee } i \in I \text{ is working,} \\ 0, \text{ otherwise.} \end{cases}$$

$$\sum_{a \in A} x_{iat} = w_i, \qquad i \in I, t \in T, \tag{11}$$

$$12w_i \leq \sum_{t \in T} \sum_{a \in W - F_t} x_{iat} \leq 32w_i, \qquad i \in I. \tag{12}$$

Rules 3 and 4:

$$u_i = \begin{cases} 1, \text{ if employee } i \text{ covers at least 6 hours of work activities,} \\ 0, \text{ otherwise.} \end{cases}$$

$$\sum_{t \in T} \sum_{a \in W - F_t} x_{iat} - 8u_i \leq 24, \qquad i \in I, \tag{13}$$

$$\sum_{t \in T} \sum_{a \in W - F_t} x_{iat} \geq 23u_i, \qquad i \in I, \tag{14}$$

$$\sum_{t \in T} x_{ipt} = u_i + w_i, \qquad i \in I, \tag{15}$$

$$\sum_{t \in T} x_{ilt} = 4u_i, \qquad i \in I. \tag{16}$$

Rule 5:

$$v_{iat} = \begin{cases} 1, \text{ if employee } i \in I \text{ starts activity } a \in A \text{ at period } t \in T, \\ 0, \text{ otherwise.} \end{cases}$$

$$v_{iat} \geq x_{iat} - x_{ia(t-1)}, \qquad i \in I, t \in T, a \in W - F_t \cup \{o\}, \tag{17}$$

$$v_{iat} \leq x_{iat}, \qquad i \in I, t \in T, a \in W - F_t \cup \{o\}, \tag{18}$$

$$v_{iat} \leq 1 - x_{ia(t-1)}, \qquad i \in I, t \in T, a \in W - F_t \cup \{o\}, \tag{19}$$

$$x_{ilt'} \geq v_{ilt}, \qquad i \in I, t \in T, t' = t, t+1, t+2, t+3, \tag{20}$$

$$x_{iat'} \geq v_{iat}, \qquad i \in I, t \in T, t' = t, t+1, t+2, t+3, a \in W - F_t. \tag{21}$$

Rule 6:

$$v_{iat} \leq 1 - \sum_{a' \in W - F_{t-1}} x_{ia'(t-1)}, \qquad i \in I, t \in T', a \in W - F_t. \tag{22}$$

Rule 7:

$$x_{ipt} \leq 1 - x_{il(t-1)}, \qquad i \in I, t \in T', \tag{23}$$

$$x_{ipt} \leq \sum_{a \in W - F_{t-1}} x_{ia(t-1)}, \qquad i \in I, t \in T', \tag{24}$$

$$x_{ipt} \leq \sum_{a \in W - F_{t+1}} x_{ia(t+1)}, \qquad i \in I, t \in T', \tag{25}$$

$$v_{ilt} \leq 1 - x_{ip(t-1)} \qquad , i \in I, t \in T', \tag{26}$$

$$v_{ilt} \leq \sum_{a \in W - F_{t-1}} x_{ia(t-1)}, \qquad i \in I, t \in T', \tag{27}$$

$$v_{ilt} \leq \sum_{a \in W - F_{t+1}} x_{ia(t+1)}, \qquad i \in I, t \in T'. \tag{28}$$

Rule 8:

$$v_{it}^{-} = \begin{cases} 1, \text{ if employee } i \in I \text{ covers at least one working activity} \\ \quad \text{beginning before period } t \in T; \\ 0, \text{ otherwise.} \end{cases}$$

$$v_{it}^{+} = \begin{cases} 1, \text{ if employee } i \in I \text{ covers at least one working activity} \\ \quad \text{beginning after period } t \in T; \\ 0, \text{ otherwise.} \end{cases}$$

$$v_{it}^{-} \leq \sum_{t^{-}<t} \sum_{a \in W - F_{t^{-}}} v_{iat^{-}}, \qquad i \in I, t \in T', \tag{29}$$

$$v_{it}^{-} \geq \sum_{a \in W - F_{t^{-}}} v_{iat^{-}}, \qquad i \in I, t \in T, t^{-} < t, \tag{30}$$

$$v_{it}^{+} \leq \sum_{t^{+}>t} \sum_{a \in W - F_{t^{+}}} v_{iat^{+}}, \qquad i \in I, t \in T', \tag{31}$$

$$v_{it}^{+} \geq \sum_{a \in W - F_{t^{+}}} v_{iat^{+}}, \qquad i \in I, t \in T, t^{+} > t, \tag{32}$$

$$x_{iot} \leq (1 - v_{it}^{-}) + (1 - v_{it}^{+}), \qquad i \in I, t \in T. \tag{33}$$

Demand Covering

$$\sum_{i \in I} x_{iat} - s_{at}^{+} + s_{at}^{-} = d_{at}, \qquad t \in T, a \in W - F_t. \tag{34}$$

Objective Function

$$\min \qquad \sum_{t \in T} \sum_{a \in W - F_t} (\sum_{i \in I} c_{at} x_{iat} + c_{at}^{+} s_{at}^{+} + c_{at}^{-} s_{at}^{-}). \tag{35}$$

4.3 A MIP Regular Model

To observe the impact of modeling with the `MIP regular` constraint, we include several rules of the problem in a DFA and we formulate the other rules and the objective function as stated in the previous section. Work regulation rules 1 to 4 and demand covering constraints are formulated as in the classical model, and work regulation rules 5 to 8 are included in the DFA. We use the DFA suggested in [21] for the same problem. The DFA presented in Figure 4 is for the problem with two work activities (a and b on the Figure). It is easily generalized for any number of work activities. Let us denote Π_n the DFA for the problem with n work activities.

We insert a `MIP regular` constraint for each employee $i \in I$ to the model. This constraint ensures that the covering of the activities $a \in A$ for each $t \in T$ for any employee i is a word recognized by the DFA $\Pi_{|W|}$.

$$\textbf{AddMIPRegular}(\Pi_{|W|}, |T|, x_i, w_i, M), \qquad \forall i \in I. \tag{36}$$

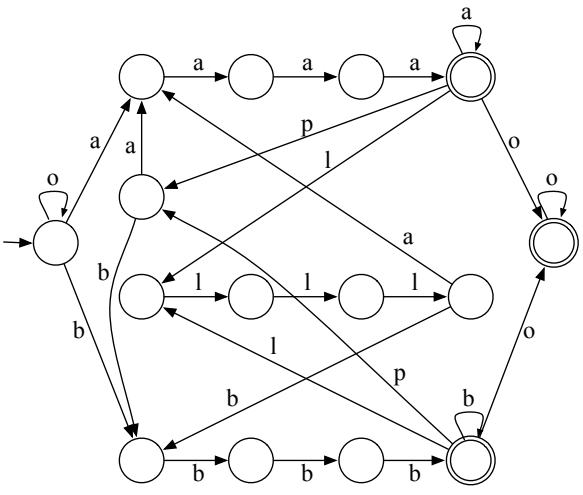


Fig. 4. DFA Π_2 for 2 activities

where M is the model presented in the previous section without work regulation constraints 5 to 8. Specifically, the output of the procedure gives the model M with the following variables and constraints:

$s^i_{taq_k} =$ amount of flow on the arc leaving the node $q_k \in N^t$ of layer $t \in T$ with activity value $a \in A$ for employee $i \in I$.
$sf^i_{q_k} =$ amount of flow leaving the node $q_k \in N^{n+1}$ for employee $i \in I$.

$$\sum_{a \in \Omega_{10}} s^i_{1jq_0} = w_i, \qquad \forall i \in I, \tag{37}$$

$$\sum_{(a,q'_k) \in \Delta_{tk}} s^i_{(t-1)aq'_k} = \sum_{a \in \Omega_{tk}} s^i_{taq_k}, \qquad \forall i \in I, t \in T', q_k \in N^t, \tag{38}$$

$$\sum_{(a,q'_k) \in \Delta_{(n+1)k}} s^i_{naq'_k} = sf^i_{q_k}, \qquad \forall i \in I, q_k \in N^{n+1}, \tag{39}$$

$$\sum_{q_k \in N^{n+1}} sf^i_{q_k} = w_i, \qquad i \in I, \tag{40}$$

$$x_{iat} = \sum_{q_k \in \Sigma_{ta}} s^i_{taq_k}, \qquad \forall i \in I, t \in T, a \in A, \tag{41}$$

$$s^i_{taq_k} \in \{0,1\} \qquad i \in I, t \in T, q_k \in N^t, a \in \Omega_{tk}, \tag{42}$$

$$sf^i_{q_k} \in \{0,1\} \qquad i \in I, q_k \in N^{n+1}, \tag{43}$$

where Δ_{tk}=$\{(a,q'_k)|(q'_k,a,q_k) \in inArcs[t][k]\}$, Ω_{tk}=$\{a|(q_k,a,\_) \in outArcs[t][k]\}$, $\Sigma_{ta} = \{q_k|(q_k,a,\_) \in outArcs[t][k]\}$.

4.4 Computational Results

Experiments were run on a 3.20 GHz Pentium 4 using the MIP solver CPLEX 10.0. All results obtained within the 3600 seconds elapsed time limit are within 1% of optimality. Table 1 presents the results for the problem described before with one work activity. For the two work-activity problems, we only show results for the MIP regular model in Table 2 as experiments on the classical model did not give any integer results within the time limit. In the tables, $|C|$ and $|V|$ are the number of constraints and the number of variables after CPLEX presolve algorithm, *Time* is the elapsed execution time in seconds, *Cost* is the solution cost and *Gap* is the gap between the initial LP lower bound, Z_{LP}, and the value of the best known solution to the MIP model, Z_{MIP}:

$$Gap = \left(\frac{Z_{MIP} - Z_{LP}}{Z_{LP}}\right) \times 100$$

The symbol ">" in the *Time* column means that no integer solution was found within the time limit.

Table 1. Results for the classical MIP model and the MIP regular model for 1 work activity

	Classical MIP model					MIP regular model				
Id	$\|C\|$	$\|V\|$	**Time**	**Cost**	**Gap**	$\|C\|$	$\|V\|$	**Time**	**Cost**	**Gap**
1	29871	4040	1706,36	172,67	24,31	1491	1856	1,50	172,67	24,31
2	56743	5104	>			2719	3976	1029,01	164,58	1,01
3	56743	5104	>			2719	3976	393,96	169,44	0,36
4	45983	4704	3603,98	149,42	13,49	2183	3144	39,89	133,45	1,36
5	40447	4360	3603,99	152,47	6,07	1915	2728	10,41	145,67	0,87
6	40447	4360	3603,70	135,92	5,28	1915	2728	20,36	135,06	1,49
7	45551	4608	>			2183	3144	6,25	150,36	0,73
8	56743	5104	>			2719	3976	274,92	148,05	0,58
9	36175	4156	3603,84	182,54	28,11	1759	2416	15,47	182,54	28,11
10	45983	4704	3603,91	153,38	5,09	2183	3144	5,50	147,63	0,95

The experiments on the one work-activity instances show that the `MIP_regular` modeling approach significantly decreases computational time in comparison with the classical MIP formulation for the given employee timetabling problem. It is also interesting to observe that the use of the `MIP regular` constraint reduces the model size.

With our approach, within the one hour time limit, we solved all the instances to optimality for the one work-activity problems with an average execution time of 180 seconds. For the two work-activity problems, six instances were solved to optimality within one hour with an average execution time of 550 seconds. These results are competitive with those of the branch-and-price approach described in [21]. Indeed, as reported in [21], for the one work-activity problems, 8 instances are solved to optimality by the branch-and-price with an average of 144 seconds

Table 2. Results for the MIP regular model for 2 work activities

Id	$\|C\|$	$\|V\|$	**Time**	**Cost**	**Gap**
1	3111	5084	623	201,78	0,00
2	3779	6192	3600,64	214,42	1,37
3	3991	6412	748,94	260,8	1,00
4	4475	7512	1128,73	245,87	1,00
5	3559	5668	3600,53	412,31	10,77
6	3879	6116	107,75	288,88	0,91
7	3199	4972	45,44	232,14	12,29
8	5799	9740	643,73	536,83	0,59
9	4583	7680	3600,61	309,69	3,92
10	4667	7584	3600,55	265,67	2,12

of computation time, while for the two work-activity problems, 8 instances are solved to optimality by the same approach with an average execution time of 394 seconds.

An asset of our method over the approach suggested in [21] is the very little development time needed. Once the model using `MIP regular` is automatically translated into a MIP model, any existing MIP solver can be used to solve it.

5 Conclusion

We presented a new MIP modeling approach for sequences of decision variables inspired by the CP `regular` constraint. As for the CP constraint, the `MIP regular` constraint uses a DFA to express rules over constrained sequences of variables. It then gives a flow formulation of the layered graph built for the consistency algorithm of the `regular` constraint. We compared our modeling approach to a classical MIP formulation on an employee timetabling problem. Computational results on instances of this problem showed that the use of the `MIP regular` constraint to model the subset of constraints on sequences of variables, significantly improves solution times.

The idea of using global constraints in MIP has been considered before. In particular, Achterberg [25] present a programming system (SCIP) to integrate CP and MIP based on a branch-and-cut-and-price framework that features global constraints (see also the references therein for other similar contributions). The `MIP regular` constraint formulation is a new contribution in this direction to allow MIP to benefit from CP. A natural next step in this trend would be to formulate a MIP version of the `grammar` constraints introduced in [26] and [22].

Acknowledgments. We thank Marc Brisson for his help with the implementation and experiments.

References

1. Pesant, G.: A regular membership constraint for finite sequences of variables. Proc. of CP'04, Springer-Verlag LNCS **3258** (2004) 482–495
2. 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** (2004) 3–27
3. Ernst, A., Jiang, H., Krishnamoorthy, M., Owens, B., Sier, D.: An annotated bibliography of personnel scheduling and rostering. Annals of Operations Research **127** (2004) 21–144
4. Dantzig, G.: A comment on Edie's traffic delay at toll booths. Operations Research **2** (1954) 339–341
5. Segal, M.: The operator-scheduling problem: A network-flow approach. Operations Research **22** (1974) 808–823
6. Moondra, B.: An LP model for work force scheduling for banks. Journal of Bank Research **7** (1976) 299–301
7. Bechtolds, S., Jacobs, L.: Implicit optimal modeling of flexible break assigments. Management Science **36** (1990) 1339–1351
8. Bechtolds, S., Jacobs, L.: The equivalence of general set-covering and implicit integer programming formulations for shift scheduling. Naval Research Logistics **43** (1996) 223–249
9. Thompson, G.: Improved implicit modeling of the labor shift scheduling problem. Management Science **41** (1995) 595–607
10. Aykin, T.: Optimal shift scheduling with multiple break windows. Management Science **42** (1996) 591–602
11. Brusco, M., Jacobs, L.: Personnel tour scheduling when starting-time restrictions are present. Management Science **44** (1998) 534–547
12. Laporte, G., Nobert, Y., Biron, J.: Rotating schedules. European Journal of Operational Research **4**(1) (1980) 24–30
13. Balakrishan, A., Wong, R.: Model for the rotating workforce scheduling problem. Networks **20** (1990) 25–42
14. Isken, M.: An implicit tour scheduling model with applications in healthcare. Annals of Operations Research, Special Issue on Staff Scheduling and Rostering **128** (2004) 91–109
15. Çezik, T., Günlük, O., Luss., H.: An integer programming model for the weekly tour scheduling problem. Naval Research Logistic **48**(7) (1999)
16. Millar, H., Kiragu, M.: Cyclic and non-cyclic scheduling of 12 h shift nurses by network programming. European Journal of Operational Research **104**(3) (1998) 582–592
17. Ernst, A., Hourigan, P., Krishnamoorthy, M., Mills, G., Nott, h., Sier, D.: Rostering ambulance officers. Proceedings of the 15th National Conference of the Australian Society for Operations Research, Gold Coast (1999) 470–481
18. Sodhi, M.: A flexible, fast, and optimal modeling approach applied to crew rostering at London Underground. Annals of Operations Research **127** (2003) 259–281
19. Hopcroft, J.E., Ullman, J.D.: Introduction to automata theory, languages and computation. Addison Wesley (1979)
20. van Hoeve, W.J., Pesant, G., Rousseau, L.M.: On global warming: Flow-based soft global constraints. Journal of Heuristics **12**(4-5) (2006) 347–373
21. Demassey, S., Pesant, G., Rousseau, L.M.: A cost-regular based hybrid column generation approach. Constraints **11**(4) (2006) 315–333

22. Quimper, C.G., Walsh, T.: Global grammar constraints. Proc. of CP'06, Springer-Verlag LNCS **4204** (2006) 751–755
23. Ahuja, R., Magnanti, T., Orlin, J.: Network Flows. Prentice Hall (1993)
24. Demassey, S., Pesant, G., Rousseau, L.M.: Constraint programming based column generation for employee timetabling. Proc. 2th Int. Conf. on Intergretion of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems - CPAIOR'05, Springer-Verlag LNCS **3524** (2005) 140–154
25. Achterberg, T.: SCIP - a framework to integrate constraint and mixed integer programming. Technical Report 04-19, Zuse Institute Berlin (2004) http://www.zib.de/Publications/abstracts/ZR-04-19/.
26. Sellmann, M.: The theory of grammar constraints. Proc. of CP'06, Springer-Verlag LNCS **4204** (2006) 530–544

Hybrid Local Search for Constrained Financial Portfolio Selection Problems

Luca Di Gaspero[1], Giacomo di Tollo[2], Andrea Roli[3], and Andrea Schaerf[1]

[1] DIEGM, Università degli Studi di Udine, via delle Scienze 208, I-33100, Udine, Italy
{l.digaspero,schaerf}@uniud.it
[2] Dipartimento di Scienze, Università "G.D'Annunzio", viale Pindaro 42, I-65127, Pescara, Italy
ditollo@sci.unich.it
[3] DEIS, *Alma Mater Studiorum* Università di Bologna, via Venezia 52, I-47023 Cesena, Italy
andrea.roli@unibo.it

Abstract. Portfolio selection is a relevant problem arising in finance and economics. While its basic formulations can be efficiently solved through linear or quadratic programming, its more practical and realistic variants, which include various kinds of constraints and objectives, have in many cases to be tackled by approximate algorithms. In this work, we present a hybrid technique that combines a local search, as *master* solver, with a quadratic programming procedure, as *slave* solver. Experimental results show that the approach is very promising and achieves results comparable with, or superior to, the state of the art solvers.

1 Introduction

The *portfolio selection* problem consists in selecting a portfolio of *assets* that provides the investor a given expected return and minimises the *risk*. One of the main contributions in this problem is the seminal work by Markowitz [25], who introduced the so-called *mean-variance* model, which takes the variance of the portfolio as the measure of investor's risk. According to Markowitz, the portfolio selection problem can be formulated as an optimisation problem over real-valued variables with a quadratic objective function and linear constraints.

In this paper we consider the basic objective function introduced by Markowitz, and we take into account two additional constraints: the *cardinality* constraint, which limits the number of assets, and the *quantity* constraint, which fixes minimal and maximal shares of each asset included in the portfolio. For an overview of the formulations presented in the literature we forward the interested reader to [7].

We devise a hybrid solution based on a local search metaheuristic (see, e.g., [13]) for selecting the assets to be included in the portfolio, which at each step resorts to a quadratic programing (QP) solver for computing the best allocation for the chosen assets. The QP procedure implements the Goldfarb-Idnani dual algorithm [11] for strictly convex quadratic programs.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 44–58, 2007.

The use of a hybrid solver has been (independently) proposed also by Moral-Escudero et al. [26], who make use of genetic algorithms instead of local search for the determination of the discrete variables.

The paper is organised as follows: In Section 2 we introduce the problem formulation and in the following section (3) we succinctly review the most relevant works that describe metaheuristic techniques applied to formulations closely related to the one discussed in this paper. In Section 4 we present our hybrid solver detailing its components and Section 5 collects the results of the experimental analysis we performed. Finally, in Section 6, we draw some conclusions and point out our plans for further work.

2 Problem Definition

Following Markowitz [25], we are given a set of n *assets*, $A = \{a_1, \ldots, a_n\}$. Each asset a_i has an associated real-valued *expected return* (per period) r_i, and each pair of assets $\langle a_i, a_j \rangle$ has a real-valued *covariance* σ_{ij}. The matrix $\sigma_{n \times n}$ is symmetric and the diagonal elements σ_{ii} represent the *variance* of assets a_i. A positive value R represents the desired expected return. The values r_i and σ_{ij} are usually estimated from past data and are relative [to] one fixed period of time.

A portfolio is a vector of real values $X = \{x_1, \ldots, x_n\}$ such that each x_i represents the fraction invested in the asset a_i. The value $\sum_{i=1}^{n} \sum_{j=1}^{n} \sigma_{ij} x_i x_j$ represents the variance of the portfolio, and is considered as the measure of the risk associated with the portfolio. Whilst the initial formulation by Markowitz [25] was a bi-objective optimisation problem, in many contexts financial operators prefer to tackle a single-objective version, in which the problem is to minimise the overall variance, ensuring the expected return R. The formulation of the basic (unconstrained) problem is thus the following.

$$\min f(X) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sigma_{ij} x_i x_j$$

$$s.t. \quad \sum_{i=1}^{n} r_i x_i \geq R \tag{1}$$

$$\sum_{i=1}^{n} x_i = 1 \tag{2}$$

$$0 \leq x_i \leq 1 \quad (i = 1, \ldots, n) \tag{3}$$

This is a quadratic programming problem, and nowadays it can be solved optimally using available tools despite the NP-completeness of the underlying decision problem [20].

Since R can be considered a parameter of the problem, solvers are usually compared over a set of instances, each with a specific value of minimum required expected return. By solving the problem as a function of R, ranging over a finite

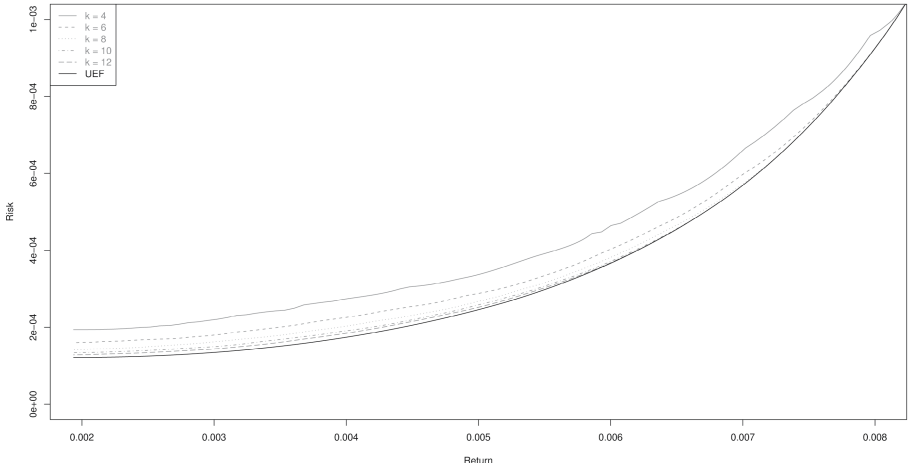


Fig. 1. Unconstrained and constrained efficient frontier

and discrete domain, we obtain the so-called *unconstrained (Pareto) efficient frontier* (UEF), that gives for each expected return the minimum associated risk. The UEF for one of the benchmark instances employed in this study is provided in Figure 1 (the lowest black solid line).

Although the classical Markowitz's model is extremely useful from the theoretical point of view, dealing with real-world financial markets imposes some additional constraints that are going to be considered in this work. In order to model them correctly, we need to add to the formulation a vector of n binary decision variables Z such that $z_i = 1$ if and only if asset i is in the solution (i.e., $x_i > 0$).

Cardinality constraint: The number of assets that compose the portfolio is bounded: we give two values k_{min} and k_{max} (with $1 \leq k_{min} \leq k_{max} \leq n$) such that:

$$k_{min} \leq \sum_{i=1}^{n} z_i \leq k_{max} \tag{4}$$

Quantity constraints: The quantity of each asset i that is included in the portfolio is limited within a given interval: we give a minimum ϵ_i and a maximum δ_i for each asset i, such that:

$$x_i = 0 \quad \vee \quad \epsilon_i \leq x_i \leq \delta_i \qquad (i = 1, \ldots, n) \tag{5}$$

Notice that the minimum cardinality constraints are especially meaningful in presence of constraints on the minimum quantity, otherwise they can be satisfied by infinitesimal quantities.

We call CEF the analogous of the UEF for the constrained problem. In Figure 1 we plot the CEF found by our solver for the values $\epsilon_i = 0.01$, $\delta_i = 1$ (for

$i = 1, \ldots, n$), $k_{min} = 1$, and k_{max} varying from 4 to 12. For higher values of k_{max} the cardinality constraint reduces its effect and the curve is almost indistinguishable from the UEF, indeed the distance among the CEF and the UEF[1] becomes smaller than 10^{-3} for the instance at hand.

Constraints 4 and 5 make it intractable to solve real-world instances of the problem with proof of optimality [14]. Therefore, either simplified models are considered, such as formulations with linear objective function [21, 22], or approximate methods are applied.

3 Related Work

Local search approaches have been widely applied to portfolio selection problems under many different formulations. The first work on this subject appearing in the literature is due to Rolland [30], who presents an implementation of Tabu Search to tackle the unconstrained formulation. This formulation is considered also in the implementation of evolutionary techniques in [2, 18, 19]. The use of local search techniques for the constrained portfolio selection problem has been proposed by several authors, including Chang et al. [4], Gilli and Këllezi [9] and Schaerf [31].

The cited works however use local search as a *monolithic* solver, exploring a search space composed of both continuous and discrete variables. Conversely, our hybrid solver focuses on the discrete variables, leaving the determination of the continuous ones to the QP solver. In addition, we consider here a more general problem w.r.t. the cited three papers, including also the possibility to specify a minimum number of assets (and not only the maximum).

Among the population-based methods developed for tackling the constrained formulation, we mention Streichert et al. [33], in which the cardinality constrained variant is considered, and memetic algorithm approaches introduced in [12, 15, 24]. These strategies, by being inherently effective in diversifying the search, exhibit good performance especially in multi-objective formulations, as shown by the family of Multi-Objective Evolutionary Algorithms [17, 8, 27, 33]. Finally, Ant Colony Optimisation has also been successfully applied to portfolio problems modelled with the cardinality constraint in [1, 23].

For the sake of completeness, we also mention interesting hybrid heuristic techniques based on linear programming that have been introduced in [32] and deal with a linear objective function formulation with integer variable domains. In this case, the value assigned to a variable represents the actual amount invested in the asset. The basic idea behind these approaches is to relax the discrete constraint on quantities, transforming the problem into a linear programming problem and find a solution to it. Fractional asset weights are then rounded to the closest admissible discrete quantity and a possible infeasible solution is repaired heuristically. More robust strategies use the solution to the continuous relaxation to feed a mixed integer-linear programming solver [16, 20].

[1] Measured by what we call average percentage loss, introduced in Section 5.1.

4 A Hybrid Local Search Solver for Portfolio Selection

Our master solver is based on local search, which works on the space induced by the vector Z only. For computing the actual quantities X, it invokes the QP (slave) solver, using as the input assets only those such that $z_i = 1$ in the current state.

In order to apply local search techniques we need to define the search space, the cost function, the neighbourhood structures, and the selection rule for the initial solution.

4.1 Search Space and Cost Function

The search space is composed of the all 2^n possible configurations of Z, with the exception of assignments that do not satisfy Constraints (4). These constraints are therefore implicitly enforced by the local search solver by excluding them from the search space. On the contrary, states that violate Constraints (1), (2), (3), or (5) are included, and these constraints are passed to the QP solver that handles them explicitly.

The QP solver receives as input only those assets included in the state under consideration, and it produces the assignment of values to the corresponding x_i variables. For all assets a_i that are not included in the state we obviously set $x_i = 0$. In addition, the QP solver also returns the computed risk f for the solution produced, which represents the cost of the state.

If the QP solver is unable to produce a feasible solution it returns the special value $f = +\infty$ (and the values x_i returned are not meaningful). In this case, we relax Constraint (1) and we build the configuration, using only the assets included that gives the highest return without violating the other constraints. This construction is done by a greedy algorithm that sorts the assets by the expected return and assigns the maximum quantity to each asset in turn, as long as the sum is smaller than 1.

In the latter case the cost is the degree of violation of Constraint (1) multiplied by a suitably large constant (that ensures that return related costs are always bigger than risk related ones).

4.2 Neighbourhood Structure

The neighbourhood relation we propose is based on addition, deletion and replacement of an asset. A move m is identified by a pair $\langle i, j \rangle$, where a_i is the asset to be added and a_j is the asset to be deleted ($i, j \in \{1, \ldots, n\}$). The value of i can also be 0, meaning that no asset is added. Analogously for j, if $j = 0$ it means that no asset is deleted.

Notice that not all pairs $m = \langle i, j \rangle$, with $i, j \in \{0, 1, \ldots, n\}$, correspond to a feasible move, since some values are meaningless, e.g. inserting an asset already present or setting both i and j to zero (*null move*). Moreover, moves that violates Constraints (4) are also considered infeasible, e.g. a delete move when the number of assets is equal to the minimum.

4.3 Initial Solution Construction

For the initial solution, we use three different strategies, that are employed at different stages of the search (as explained in Section 4.4). For all three, we ensure that Constraints (4) are always satisfied.

RandomCard: We draw at random a number k (between k_{min} and k_{max}), and we insert k randomly selected assets.

MaxReturn: We build the portfolio that produces the maximum possible return (independently of the risk)

PreviousPoint: We use the final solution of the previously computed point of the frontier

4.4 Local Search Techniques

We implemented three local search techniques, namely *Steepest Descent* (SD), *First Descent* (FD), and *Tabu Search* (TS).

The SD strategy relies on the exhaustive exploration of the neighbourhood and the selection of the neighbour that has the minimal value of f (breaking ties at random). The SD strategy stops as soon as no improving move is available, i.e., when a local minimum has been reached. FD behaves as SD with the difference that, as soon as an improving move is found, it is selected and the exploration of the current neighbourhood is interrupted.

For TS we use a dynamic-size tabu list to implement a short term prohibition mechanism and the standard aspiration criterion [10]. Like for SD, we search for the next state by exploring the full neighbourhood (excluding infeasible moves) at each iteration.

In order to make the solvers more robust, for all techniques, we make two runs for each value of R: one using the RandomCard initial solution construction, and the other one starting from the best of the previous point (PreviousPoint initial solution). For the very first point of the frontier (highest requested return and no previous point available) we use instead the MaxReturn construction.

5 Experimental Analysis

In this section, we first present the benchmark instances and the settings of our solver. In the following subsections, we show the comparison with all the previous works that use the same formulation. We conclude showing a search space analysis that tries to explain the behaviour of our solvers on the proposed instances.

5.1 Benchmark Instances

We experimented our techniques on two groups of instances obtained from real stock markets and used in previous works. The first is a group of five instances taken from the repository `ORlib` available at the URL `http://mscmga.ms.ai.`

Table 1. The benchmark instances

Inst.	Origin	assets	UEF
Group 1			
1	Hong Kong	31	$1.55936 \cdot 10^{-3}$
2	Germany	85	$0.412213 \cdot 10^{-3}$
3	UK	89	$0.454259 \cdot 10^{-3}$
4	USA	98	$0.502038 \cdot 10^{-3}$
5	Japan	225	$0.458285 \cdot 10^{-3}$
Group 2			
S1	USA (DataStream)	20	4.812528
S2	USA (DataStream)	30	8.892189
S3	USA (DataStream)	151	8.64933

ac.uk/ jeb/orlib/portfolio.html . These instances have been proposed by Chang et al. [4] and have been studied also in [1, 26, 31]. The second group of three instances have been provided to us by M. Schyns and are used in [5].

For the first group, a discretised UEF composed of 100 equally distributed values for the expected return R is provided along with the data. For the second group, we computed the discretised UEF ourselves using the QP solver with all assets available and no additional constraints.

As in previous works, we evaluate the quality of our solutions employing an aggregate indicator that measures the deviation of the CEF found by the algorithms w.r.t. the UEF on the whole set of frontier points. We call this measure *average percentage loss* (*apl*) and we define it as follows: let R_l be the expected return, $V(R_l)$ and $V_U(R_l)$ the values of the function f returned by the solver and the risk on the UEF, respectively, and $l = 1, \ldots, p$ where p is the number of points of the frontier; the average percentage loss is equal to $\frac{100}{p} \sum_{l=1}^{p} (V(R_l) - V_U(R_l))/V_U(R_l)$. Table 1 illustrates for all instances the original market, and the average variance of the UEF.

5.2 Experimental Setting of the Solvers

Experiments were performed on an Apple iMac computer equipped with an Intel Core 2 Duo (2.16 GHz) processor and running Mac OS X 10.4; the SD, FD and TS metaheuristics have been coded in C++ exploiting the framework EasyLocal++ [6], the QP solver has also been coded in C++ and is made publicly available from one of the authors' website[2]. The executables were obtained using the GNU C/C++ compiler (v. 4.0.1).

Concerning the algorithms setting, SD and FD have no parameter to be set; for TS we tuned its parameters by means of a statistical technique called F-race [3] and found that the algorithm is very robust with respect to parameter setting. We set the tabu list size in the range $[3 \ldots 10]$ and we stop the execution of TS when a maximum of 100 iterations without improvement was reached.

[2] http://www.diegm.uniud.it/digaspero/

5.3 Comparison with Previous Results

Due to the different formulations employed by the authors, the only papers we can compare with are those of Schaerf [31] and Moral-Escudero et al. [26], who employ the same set of constraints on the `ORlib` instances, and with Crama and Schyns [5] who deal with a slightly different setting and with a novel set of instances. Concerning Chang et al. [4], as already pointed out in [31], even though they work on the `ORlib` instances (and with the same constraints), a fair comparison with their solutions is not possible because the problem is solved by taking points along the frontier that are not homogeneously distributed.

Armañanzas and Lozano [1] work on a variant of the problem for which the values k_{min} and k_{max} coincide (i.e., $k_{min} = k_{max} = K$) on the `ORlib` instances. However, due to what we believe is an error in the implementation of their solution methods [3] they obtain a set of points that are infeasible w.r.t. Constraint (2). In details, they assign to the assets i for which $z_i = 1$ chosen by their ACO algorithm the quantity $x_i = (\delta_i - \epsilon_i)/K$, therefore since they set $\epsilon_i = 0.001, \delta_i = 1$ for all $i = 1, \ldots, n$, they obtain $\sum_{i=1}^{n} x_i = 0.999$ instead of 1. For this reason we could not compare our solvers with [1], nevertheless we are going to present some results on the behaviour of one of our solvers on the formulation proposed in that paper.

Comparison with Schaerf [31] and Moral-Escudero et al. [26]. For this comparison, we set the constraint values exactly as in [26, 31]: $\epsilon_i = 0.01$ and $\delta_i = 1$ for $i = 1, \ldots, n$, and $k_{max} = 10$ for all instances. The minimum cardinality is not considered in the cited work, and therefore we set it to $k_{min} = 1$ (i.e., no limitation).

Table 2 shows best results and running times obtained by our three solvers in comparison with previous work. Since Moral-Escudero et al. [26] report only the best outcomes of their solvers, in order to fairly compare with them we have to present the results as the *minimum* average percentage loss w.r.t. the UEF found by the algorithm.

The results of our solvers are the best CEFs found in 30 trials of the algorithm on each instance and the running times reported are those of the best trial (exactly as in [26]). Running times of [31] are obtained re-running Schaerf's software on our machine, those of Moral-Escudero et al. are taken from their paper, and are obtained using a PC having about the same performances.

Table 2 shows that we obtain results superior to [31] both in terms of risk and running times. This suggests that the hybrid solver outperforms monolithic local search ones. Regarding [26], we obtain with SD exactly the same results of their best solver, but in a much shorter time (on a comparable machine).

As already pointed out in [31], even though Chang et al. [4] solve the same instances (and with the same constraints), a fair comparison with their solutions is not possible. This is because they consider the CEF differently. Specifically, they do not solve a different instance for each value of R, but (following Perold [28]), they reformulate the problem without Constraint (1) and with the following

[3] We found the error in our analysis of the data provided to us by J. Lozano.

Table 2. Comparison of results with Schaerf [31] and Moral-Escudero et al. [26]

Inst.	FD + QP min	apl	time	SD + QP min	apl	time	TS + QP min	apl	time	GA + QP [26] min	apl	time	TS [31] min	apl	time
1	0.00366		1.3s	0.00321		4.3s	0.00321		17.2s	0.00321		415.1s	0.00409		251s
2	2.66104		5.3s	2.53139		20.3s	2.53139		61.3s	2.53180		552.7s	2.53617		531s
3	2.00146		5.4s	1.92146		23.6s	1.92133		69.5s	1.92150		886.3s	1.92597		583s
4	4.77157		7.6s	4.69371		27.6s	4.69371		80.0s	4.69507		1163.7s	4.69816		713s
5	0.24176		15.7s	0.20219		69.5s	0.20210		210.7s	0.20198		1465.8s	0.20258		1603s

objective function: $f(X) = \lambda f_1(X) + (1 - \lambda) f_2(X)$. The problem is then solved for different values of λ, and what they obtain is the solution for a set of values for R which are not homogeneously distributed.

Comparison with Crama and Schyns [5]. Since the results of Crama and Schyns [5] are presented in graphical form and make use of a slightly different cost function (i.e., they consider the standard deviation instead of the variance as the risk measure) we re-run their solver[4] on the three instances employed in their experimentation employing the same parameter setting reported in their paper. The constraints set in this experiment are as follows: $k_{min} = 1$, $k_{max} = 10$, $\epsilon_i = 0$, and $\delta_i = 0.25$.

In Table 3 we present the outcome of this comparison. For each algorithm we report in three columns the average and the standard deviation (in parentheses) of the average percentage loss w.r.t. the UEF, and the average time spent by the algorithm. The data was collected by running 30 times each algorithm on each instance and computing the whole CEF.

From the table it is clear that, in terms of solution quality, the family of our solvers outperforms the SA approach of Crama and Schyns. Looking at the times, we can see that SA, in general, exhibits shorter running times than our hybrid SD and TS approaches. This can be explained by the strategy employed by both our algorithms that thoroughly explore the full neighbourhood of each solution whereas the SA randomly picks out only some neighbours thus saving time in the evaluation of the cost function. Moreover, the slave QP procedure is more time-consuming than the solution evaluation carried out by Crama and Schyns, however it allows us a higher accuracy on the assignment of the assets.

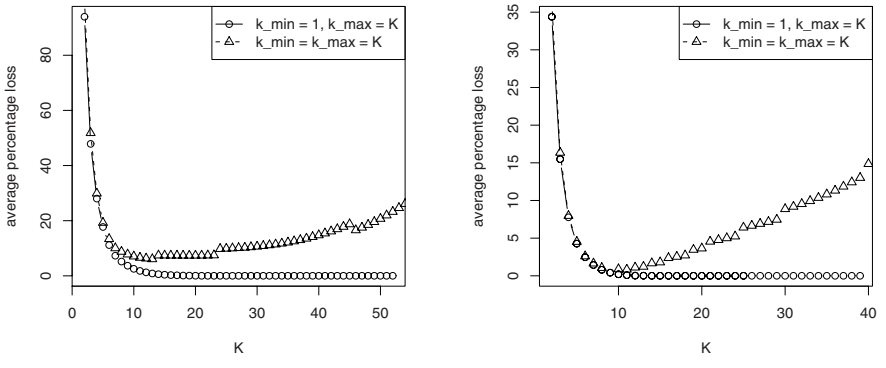
Results for fixed cardinality portfolios. As mentioned previously, even though we cannot compare our results with the work of Armañanzas and Lozano [1], we decided to show some results of the SD solver on the `ORlib` instances by setting the cardinality constraints so that they force the constructed portfolio to have exactly $k_{min} = k_{max} = K$ assets as in [1]. The quantity constraints are set as in the first set of experiments, i.e. $\epsilon_i = 0.01$, and $\delta_i = 1$.

In Figure 2 we plot the behaviour of the average percentage loss found by our SD + QP solver at different values of K on a selected pair of instances. The

[4] The executable was kindly provided to us by M. Schyns.

Table 3. Comparison of results with Crama and Schyns [5]

Inst.	FD + QP apl		time	SD + QP apl		time	TS + QP apl		time	SA [5] apl		time
S1	0.72	(0.094)	0.3s	0.35	(0.0)	1.4s	0.35	(0.0)	4.6s	1.13	(0.13)	3.2s
S2	1.79	(0.22)	0.5s	1.48	(0.0)	3.1s	1.48	(0.0)	8.5s	3.46	(0.17)	5.4s
S3	10.50	(0.51)	10.2s	8.87	(0.003)	53.3s	8.87	(0.0003)	124.3s	16.12	(0.43)	30.1s



(a) Results on instance 2. **(b)** Results on instance 5.

Fig. 2. Average percentage loss found by our SD + QP solver varying K

curve is compared with the average percentage loss computed by the same solver but relaxing the minimum cardinality constraint to $k_{min} = 1$ (i.e., just allowing to include an increasing number of assets in the portfolios, but not obliging the solver to compel to a fixed cardinality).

From the pictures we can notice an interesting phenomenon: the two curves are almost indistinguishable up to a value of K for which the fixed cardinality solutions tend to have an higher average percentage loss. In a sense, this sort of *minimum* represent the best compromise in the cardinality, i.e., the optimal fixed number of assets K that minimises the deviation from the best achievable returns (i.e, the UEF values).

5.4 Search Space Analysis

We study the search space main characteristics of the instances composing the benchmarks with the aim of providing an explanation for the observed algorithm behaviour and elaborating some guidelines for understanding the hardness of an instance when tackled with our hybrid local search. Once cardinality constraints are set, in general we are interested in studying the characteristics of the search space of the single instances of the problem along the frontier, i.e., at fixed values of return R. Among the 100 points composing the frontier, we took five samples homogeneously distributed along the frontier, in order

to estimate the characteristics of the search spaces encountered by our solver along the whole frontier. Moreover, constrained instances with different values of k_{min} and k_{max} have been considered. In our experiments, we chose $(k_{min}, k_{max}) \in \{(3,3), (6,6), (10,10), (1,3), (1,6), (1,10)\}$.

One of the most relevant search space characteristics is the number of global and local minima. The number of local minima is usually taken as an estimation of the ruggedness of the search space, that, in turn, is roughly negatively correlated with local search performance [13]. In order to estimate the number of minima in an instance, we run a deterministic version of SD (called SD_{det})[5] starting from initial states either produced by complete enumeration (for very small size instances) or by uniformly sampling the search space.

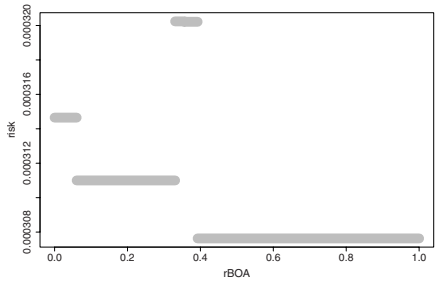
Our analysis shows that the instances of the benchmarks have a very small number of local minima, and only one global minimum (i.e., either a certified global minimum, when exhaustive enumeration is performed, or the best known solution, otherwise). Most of the analysed instances have only one minimum and the other instances have not more than six minima. We observed that the latter cases occur usually at low values of return R. Instance 4 is the one with the greatest number of local minima, while the remaining instances have very few cases with local minima.

This analysis may provide an explanation for the very similar performance exhibited by SD and TS in terms of solution quality. To strengthen this argument, we also studied global and local minima basins of attraction, in order to estimate the probability of reaching a global minimum [29]. Given a deterministic algorithm such as SD_{det}, the basin of attraction $\mathcal{B}(\overline{s})$ of a minimum $\overline{s}$, is defined as the set of states that, taken as initial states, give origin to trajectories that ends at point $\overline{s}$. The cardinality of $\mathcal{B}(\overline{s})$ represents its size (in this context, we always deal with finite spaces). The quantity $rBOA(\overline{s})$, defined as the ratio between the size of $\mathcal{B}(\overline{s})$ and the search space size, is an estimation of the reachability of state $\overline{s}$. If the initial solution is chosen at random, the probability of finding a global optimum s^* is exactly equal to $rBOA(s^*)$. Therefore, the higher is this ratio, the higher is the probability of success of the algorithm. Both SD and TS incorporate stochastic decision mechanisms and TS is also able to escape from local minima, therefore the estimation of basins of attraction size related to SD_{det} provides a lower bound on the probability of reaching the global optimum when using SD and TS.

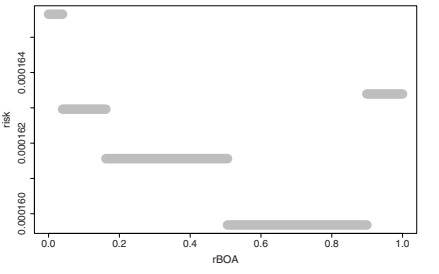
The outcome of our analysis is that global minima have usually a quite large basin of attraction. Representative examples of these results are depicted in Figures 3a, 3b, 3c and 3d; segments represent the basins of attraction: their length corresponds to $rBOA$ and their y-value is the objective value of the corresponding minimum. We can note that global minima have a quite large basin of attraction whose $rBOA$ ranges from 30% (in Figure 3c) to 60% (in Figure 3a).

It is worth remarking that these large basins are specific for our hybrid solver, and this is not the case for monolithic local search ones. The presence of large basins of attraction for the global optimum suggests that the best strategy for

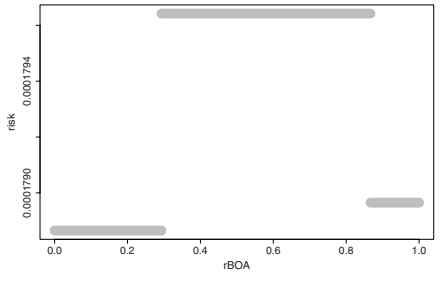
[5] Ties are broken by enforcing a lexicographic order of states.



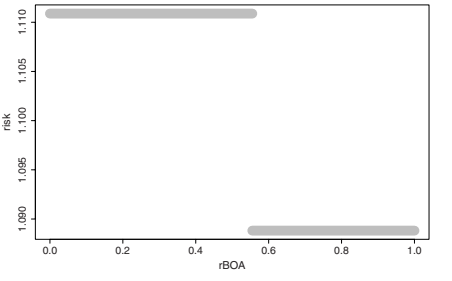
(a) Instance 4: $k_{min} = k_{max} = 3$, $R = 0.0037524115$.



(b) Instance 4: $k_{min} = 1$, $k_{max} = 6$, $R = 0.0019368822$.



(c) Instance 4: $k_{min} = 1$, $k_{max} = 10$, $R = 0.0037524115$.



(d) Instance S3: $k_{min} = 1$, $k_{max} = 6$, $R = 0.260588$.

Fig. 3. Basins of attraction of minima on two benchmark instances with different cardinality and return constraints

tackling these instances is simply to run SD with random restarts, and that there is no need for a more sophisticated solver such as TS.

However, since TS has better exploration capabilities than SD, it could still show superior performances on other, possibly more constrained, instances. Indeed, it is possible construct artificial instances with a large number of local minima and a small basin for the global one; it is straightforward to show that for such instances TS performs much better than SD for all values of R.

6 Conclusions and Future Work

Experiments show that our solver is comparable with (or superior to) the state of the art for the less constrained problem formulation (no minimum). Comparison for the general problem are subject of ongoing work. In the future, we plan to adapt this approach to tackle other formulations, such as the discrete formulation that is particularly interesting for some investors. This formulation enables us to take into account aspects of real-world finance, such as transaction costs. To this extent, instances including minimum lots will be investigated, since assets generally cannot be purchased in any quantity and the amount of money to

be invested in a single asset must be a multiple of a given minimum lot [20]. Moreover, we are going to include also asset preassignments, that will be useful for representing investor's subjective preferences.

We also aim at identifying difficult instances and verify whether more sophisticated local search metaheuristics, such as TS, could improve on the results of the simple SD strategy.

Acknowledgements

We thank Jose Lozano, Renata Mansini, Michael Schyns, and Ruben Ruiz-Torrubiano for helpful clarification about their work.

References

[1] R. Armañanzas and J.A. Lozano. A multiobjective approach to the portfolio optimization problem. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005)*, volume 2, pages 1388–1395. IEEE Press, 2005. doi: 10.1109/CEC.2005.1554852.

[2] S. Arnone, A. Loraschi, and A. Tettamanzi. A genetic approach to portfolio selection. *Neural Network World – International Journal on Neural and Mass-Parallel Computing and Information Systems*, 3(6):597–604, 1993.

[3] M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In W. B. Langdon et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 11–18, New York, 9–13 July 2002. Morgan Kaufmann Publishers.

[4] T.-J. Chang, N. Meade, J. E. Beasley, and Y. M. Sharaiha. Heuristics for cardinality constrained portfolio optimisation. *Computers & Operations Research*, 27 (13):1271–1302, 2000.

[5] Y. Crama and M. Schyns. Simulated annealing for complex portfolio selection problems. *European Journal of Operational Research*, 150:546–571, 2003.

[6] L. Di Gaspero and A. Schaerf. EasyLocal++: An object-oriented framework for flexible design of local search algorithms. *Software—Practice & Experience*, 33(8):733–765, 2003.

[7] G. di Tollo and A. Roli. Metaheuristics for the portfolio selection problem. Technical Report R-2006-005, Dipartimento di Scienze, Università "G. D'Annunzio" Chieti–Pescara, 2006.

[8] L. Dioşan. A multi-objective evolutionary approach to the portfolio optimization problem. In *Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation (CIMCA 2005)*, pages 183–188. IEEE Press, 2005.

[9] M. Gilli and E. Këllezi. A global optimization heuristic for portfolio choice with VaR and expected shortfall. In P. Pardalos and D.W. Hearn, editors, *Computational Methods in Decision-making, Economics and Finance*, Applied Optimization Series. Kluwer Academic Publishers, 2001.

[10] F. Glover and M. Laguna. *Tabu search*. Kluwer Academic Publishers, 1997.

[11] D. Goldfarb and A. Idnani. A numerically stable dual method for solving strictly convex quadratic programs. *Mathematical Programming*, 27:1–33, 1983.

[12] M.A. Gomez, C.X. Flores, and M.A. Osorio. Hybrid search for cardinality constrained portfolio optimization. In *Proceedings of the 8th annual conference on Genetic and Evolutionary Computation,*, pages 1865–1866. ACM Press, 2006. doi: 10.1145/1143997.1144302.

[13] H.H. Hoos and T. Stützle. *Stochastic Local Search Foundations and Applications*. Morgan Kaufmann Publishers, San Francisco, CA (USA), 2005. ISBN 1-55860-872-9.

[14] N.J. Jobst, M.D. Horniman, C.A. Lucas, and G. Mitra. Computational aspects of alternative portfolio selection models in the presence of discrete asset choice constraints. *Quantitative Finance*, 1:1–13, 2001.

[15] H. Kellerer and D.G. Maringer. Optimization of cardinality constrained portfolios with a hybrid local search algorithm. *OR Spectrum*, 25(4):481–495, 2003.

[16] H. Kellerer, R. Mansini, and M.G Speranza. On selecting a portfolio with fixed costs and minimum transaction lots. *Annals of Operations Research*, 99:287–304, 2000.

[17] D. Lin, S. Wang, and H. Yan. A multiobjective genetic algorithm for portfolio selection. Working paper, 2001.

[18] A. Loraschi and A. Tettamanzi. An evolutionary algorithm for portfolio selection within a downside risk framework. In C. Dunis, editor, *Forecasting Financial Markets*, Series in Financial Economics and Quantitative Analysis, pages 275–285. John Wiley & Sons, Chichester, UK, 1996.

[19] A. Loraschi, A. Tettamanzi, M. Tomassini, and P. Verda. Distributed genetic algorithms with an application to portfolio selection problems. In D. W. Pearson, N. C. Steele, and R. F. Albrecht, editors, *Proceedings of the International Conference on Artificial Neural Networks and Genetic Algorithms*, pages 384–387. Springer-Verlag, 1995.

[20] R. Mansini and M.G. Speranza. Heuristic algorithms for the portfolio selection problem with minimum transaction lots. *European Journal of Operational Research*, 114:219–233, 1999.

[21] R. Mansini and M.G. Speranza. An exact approach for portfolio selection with transaction costs and rounds. *IIE Transactions*, 37:919–929, 2005.

[22] R. Mansini, W. Ogryczak, and M.G. Speranza. Lp solvable models for portfolio optimization a classification and computational comparison. *IMA Journal of Management Mathematics*, 14:187–220, 2003.

[23] D.G. Maringer. Optimizing portfolios with ant systems. In *Proceedings of the International ICSC congress on computational intelligence: methods and applications (CIMA 2001)*, pages 288–294. ISCS Academic Press, 2001.

[24] D.G. Maringer and P. Winker. Portfolio optimization under different risk constraints with modified memetic algorithms. Technical Report 2003–005E, University of Erfurt, Faculty of Economics, Law and Social Sciences, 2003.

[25] H. Markowitz. Portfolio selection. *Journal of Finance*, 7(1):77–91, 1952.

[26] R. Moral-Escudero, R. Ruiz-Torrubiano, and A. Suárez. Selection of optimal investment with cardinality constraints. In *Proceedings of the IEEE World Congress on Evolutionary Computation (CEC 2006)*, pages 2382–2388, 2006.

[27] C.S. Ong, J.J. Huang, and G.H. Tzeng. A novel hybrid model for portfolio selection. *Applied Mathematics and Computation*, 169:1195–1210, October 2005.

[28] A.F. Perold. Large-scale portfolio optimization. *Management Science*, 30(10): 1143–1160, 1984.

[29] S. Prestwich and A. Roli. Symmetry breaking and local search spaces. In *Proceedings of the 2nd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2005)*, volume 3524 of *Lecture Notes in Computer Science*. Springer–Verlag, 2005.

[30] E. Rolland. A tabu search method for constrained real-number search: applications to portfolio selection. Working paper, 1996.

[31] A. Schaerf. Local search techniques for constrained portfolio selection problems. *Computational Economics*, 20(3):177–190, 2002.

[32] M.G. Speranza. A heuristic algorithm for a portfolio optimization model applied to the Milan stock market. *Computers & Operations Research*, 23(5):433–441, 1996. ISSN 0305-0548.

[33] F. Streichert, H. Ulmer, and A. Zell. Comparing discrete and continuous genotypes on the constrained portfolio selection problem. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004)*, volume 3103 of *Lecture Notes in Computer Science*, pages 1239–1250, Seattle, Washington, USA, 2004. Springer-Verlag.

The "Not-Too-Heavy Spanning Tree" Constraint

Grégoire Dooms and Irit Katriel

Department of Computer Science
Brown University
Providence, RI
{gdooms,irit}@cs.brown.edu

Abstract. We develop filtering algorithms for the Weight-Bounded Spanning Tree ($WBST(G, T, I, W)$) constraint, which is defined on undirected graph variables G and T, a scalar variable I and a vector of scalar variables W. It specifies that T is a spanning tree of G whose total weight is at most I, where W is a vector of the edge weights.

1 Introduction

Graphs are among the most common abstractions used for modeling real-life optimization problems. It is therefore likely that future constraint languages will include, in addition to set variables [2,14,15,16,17,23], also graph variables [6,9,14,22], i.e., variables whose assigned values are graphs. Such variables, of course, can only be useful if the language also supports constraints that are defined on them and capture frequently recurring graph properties. Such constraints, in turn, require efficient filtering algorithms.

Filtering algorithms were developed for several graph constraints, including the Sellmann's shorter-paths constraint [25] , Cambazard's simple-path constraint [5], and Beldiceanu et al.'s unweighted tree and forest constraints [3,4]. The $AllDifferent$ constraint [24] can also be viewed as a graph constraint, namely, the *bipartite matching* constraint. Recently, the authors developed filtering algorihtms for the $MST(G, T, W)$ constraint [10], which is defined on graph variables G and T and a vector W of scalar variables. The constraint specifies that T is a *minimum spanning tree* of G, while W is a vector of the edge weights of G (hence also of T). This constraint can be useful when modelling network design problems, in particular inverse optimization problems [12] where we wish to identify which edges of G must or must not be present in order for some tree to be a minimum spanning tree. It also generalizes earlier work by Aron and Van Hentenryck [1], who partially solved a special case of the MST filtering problem: For the case in which G is fixed but $E(T)$ and W are not, they identify the edges that have a support (we call these edges *possible*, and they call them *weak*).

In most network design applications, however, we do not insist on finding a spanning tree of *minimum* weight. Rather, we wish to find the lightest spanning tree that satisfies our feasibility criteria. For instance, if we are trying to construct a communication network, we might want to network to satisfy an arbitrary collection of constraints in addition to being a spanning tree, such as an upper bound on its diameter (e.g., [19]) or on the maximum degree of a node (e.g., [18]) or both (e.g., [21]). The additional constraints may exclude all *minimum* spanning trees of G. For applications of this kind,

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 59–70, 2007.

the MST constraint is not useful. Rather, we need an *optimization constraint*, more precisely a *minimization* constraint. Typically, a minimization constraint has a parameter Z that specifies an upper bound on the value of a solution. Whenever a solution satisfying all constraints is found, the upper bound of Z is decreased to the value of this solution. From then on, the filtering algorithm for the constraint removes values from the domains of variables if they do not belong to any *improving* solutions. On the other hand, when the filtering algorithm discovers a new lower bound on the value of any solution, it communicates this to the solver by setting the lower bound of Z to this value.

In this paper we introduce the Weight-Bounded Spanning Tree constraint, abbreviated $WBST(G, T, I, W)$, which is the optimization version of the MST constraint. It is defined on two graph variables G and T, a scalar variable I and a vector of scalar variables W. The constraint is satisfied if the graph T is a spanning tree of the graph G such that the total weight of T is at most I, where the weights of all edges in G (and hence also in T) are specified by the values in the vector W. We show that while filtering is NP-hard for this constraint in its most general form, it can be performed very efficiently in the special case where all graphs in the domains of G and T have the same node-sets (but may differ in their edge-sets).

Although they are semantically close, the structures of the solutions sets of MST and $WBST$ are quite different and therefore the filtering algorithms developed are also different. Perhaps the best indication of how different the two constraints are is the fact that in the most general case, when the node-sets of G and T are not fixed, MST can be filtered to bound consistency in polynomial time while for $WBST$, this task is NP-hard.

1.1 Set and Graph Variables

The value assigned to a set variable is a subset of a certain universe. For any finite universe, one can enumerate its subsets and replace the set variable by a scalar variable that is assigned the number of a subset instead of the subset itself. The drawback of this approach is that it might cause an exponential blowup in the size of the description of the domain.

A more compact representation of the domain $D(x)$ of a *set variable* x is to specify two sets of elements [15,16,23]: The set of elements that must belong to the set assigned to x (which we call the *lower bound* of the domain of x and denote by $\underline{D}(x)$) and the set of elements that may belong to this set (the *upper bound* of the domain of x, denoted $\overline{D}(x)$). The domain itself has a lattice structure corresponding to the partial order defined by set inclusion. In other words, for a set variable x with domain $D(x) = [\underline{D}(x), \overline{D}(x)]$, the value $v(x)$ that is assigned to x in any solution must be a set such that $\underline{D}(x) \subseteq v(x) \subseteq \overline{D}(x)$.

As the elements of the domain are not enumerated explicitly, the filtering task is to narrow the interval of possible sets by increasing the lower bound and decreasing the upper bound as much as possible without losing any solutions. In other words, we need to remove an element from the upper bound when it does not participate in any solution, and include an element from the upper bound in the lower bound if it belongs to the set in all solutions. *Thus, to filter the domain of a set variable to bound consistency, we need to compute the union and the intersection of all values that it may assume in a solution*.

A graph can be seen as two sets V and E with an inherent constraint specifying that $E \subseteq V \times V$. The domain $D(G)$ of a *graph variable* G is specified by two graphs: A lower bound graph $\underline{D}(G)$ and an upper bound graph $\overline{D}(G)$, such that the domain is the set of all subgraphs of the upper bound which are supergraphs of the lower bound.

1.2 Our Results

Let n and m be, respectively, the number of nodes and edges in the upper bounds of the domains of G and T and let $V(G)$ and $V(T)$ be the node-sets of G and T. We will assume that the domain of I and of each entry of W is an interval of rational numbers, specified by its endpoints. When the domain of a variable contains exactly one element, we say that the variable is *fixed* and denote it by a lowercase letter. We examine the filtering task for increasingly complex special cases of the $WBST$ constraint, beginning with the simple case in which the graph G as well as the weight of each edge (i.e., the entries of W) are fixed. We then gradually advance towards the most general case, in which none of the variables are fixed. The results we obtain on filtering the different cases are summarized below[1].

Case	Bound Consistency Computation
Fixed G and W	$O(m\alpha(m,n))$
Fixed W	$O(m\alpha(m,n))$
Fixed $V(G) = V(T)$	$O(m\alpha(m,n))$
No Variables are Fixed	NP-hard

Roadmap. In the rest of the paper we derive the results summarized in the table above. Note that the first two are special cases of the third, and are handled separately only to improve exposition of the complete algorithm. In the same spirit, we begin in the next section by describing filtering algorithms for two simple graph constraints, which will be later reused as components of subsequent algorithms. Finally, we explain why we have defined the $WBST$ constraint in the way we did, in terms of how it can be used in constraint programming.

2 Simple Graph Constraints

In this section we describe filtering algorithms for two simple graph constraints: the $Subgraph(T, G)$ constraint which specifies that T is a subgraph of G and the $Tree(T)$ constraint which holds if T is connected and acyclic.

Note that when the node-sets of G and T are fixed, $WBST(G, T, I, W)$ is equivalent to $Subgraph(T, G) \wedge Tree(T) \wedge Weight_{\leq}(T, W, I)$, where $Weight_{\leq}(T, W, I)$ specifies that the sum of the weights of the edges of T is at most I, and W is a vector of the edge-weights. However, the bound consistency algorithms we develop in this section do not combine with a bound consistency algorithm for $Weight_{\leq}(T, W, I)$ to form

[1] α is the inverse-Ackerman function that describes the complexity of the Union-Find data structure [27].

a bound consistency algorithm for the $WBST$ constraint: A value may be consistent with each constraint individually, while it does not participate in any solution to their conjunction. We discuss these constraints separately for didactic reasons. They allow us to first demonstrate bound consistency computations on simple examples, and they simplify the exposition of the complete algorithm, which uses them as subroutines.

2.1 The $Subgraph(T, G)$ Constraint

This constraint is bound consistent when $\overline{D}(T) \subseteq \overline{D}(G)$ and $\underline{D}(T) \subseteq \underline{D}(G)$. A filtering algorithm therefore needs to perform the following steps:

1. If $\underline{D}(T) \not\subseteq \overline{D}(G)$, the constraint has no solution.
2. Include every node or edge of $\overline{D}(G) \cap \underline{D}(T)$ in $\underline{D}(G)$.
3. Remove every node or edge of $\overline{D}(T) \setminus \overline{D}(G)$ from $\overline{D}(T)$.

The running time of this algorithm is linear in the sizes of $\overline{D}(T)$ and $\overline{D}(G)$. Once computed, bound consistency can be maintained dynamically as the domains of T and G shrink, at constant time per change[2]: Whenever a node or edge is removed from $\overline{D}(G)$, it should also be removed from $\overline{D}(T)$ and if it was present in $\underline{D}(T)$ then the constraint is inconsistent. Whenever a node or edge is placed in $\underline{D}(T)$ it should also be placed in $\underline{D}(G)$.

2.2 The $Tree(T)$ Constraint

If $\underline{D}(T)$ is empty, the constraint is bound consistent because any node or edge in $\overline{D}(T)$ belongs to a tree in $D(T)$; namely the tree consisting of a single node or a single edge. However, filtering may be necessary if $\underline{D}(T)$ is not empty.

If $\underline{D}(T)$ is not contained in a connected component of $\overline{D}(T)$, the constraint is inconsistent because T cannot be connected. Otherwise, all nodes and edges that are not in the same connected component with $\underline{D}(T)$ should be removed from $\overline{D}(T)$. Next, we need to find bridges and articulation nodes in $\overline{D}(T)$ whose removal disconnects two nodes from $\underline{D}(T)$. Since they must belong to the tree, we need to place them in $\underline{D}(T)$ as well. Finally, if $\underline{D}(T)$ contains a cycle the constraint is inconsistent, and an edge $e \in \overline{D}(T)$ between two nodes that belong to the same connected component of $\underline{D}(T)$ must be removed from $\overline{D}(T)$, because including it in the tree would introduce a cycle.

All steps above can be computed in linear time [26], but dynamic maintenance of bound consistency is not as simple as in the case of the $Subgraph$ constraint. For instance, removing an edge from $\overline{D}(T)$ may turn another edge into a bridge.

3 $WBST$ with Fixed Graph and Edge Weights

To simplify the exposition of the algorithm, we begin with the special case in which the variables G and W of the constraint are fixed (and are therefore denoted by lowercase

[2] We assume that simple operations on the domain of a graph variable such as removing an edge from the upper bound or inserting an upper bound edge into the lower bound, take constant time.

letters), i.e., the case $WBST(g, T, I, w)$. Since we assume that $D(G)$ contains exactly one graph and the edge-weights are fixed (i.e., $D(W)$ contains exactly one vector), we only need to filter the domains of T and I.

3.1 A Preprocessing Step

First, the algorithm applies the bound consistency algorithms for $Subgraph(T, G)$ and $Tree(T)$ described in Section 2. In the rest of this paper we will assume that bound consistency is maintained for $Subgraph(T, G)$ at no asymptotic cost, as described above.

Next, since $\underline{D}(T)$ is contained in the spanning tree in any solution, we reduce the problem to the case in which $\underline{D}(T)$ is initially empty, as follows. We contract all edges of $\underline{D}(T)$, in both $\overline{D}(T)$ and g. Let g' be the graph g after this contraction. We then subtract $w(\underline{D}(T))$, i.e., the total weight of edges in $\underline{D}(T)$, from each of $\underline{D}(I)$ and $\overline{D}(I)$. For any spanning tree t' of g' with weight $w(t')$, the edge-set $t' \cup \underline{D}(T)$ is a spanning tree of g of weight $w(t) = w(t') + w(\underline{D}(T))$.

3.2 Analysis of g

Let t be an MST of g of weight $w(t)$. We will use t to partition the edges of g into three sets $Mandatory(g, \overline{D}(I))$, $Possible(g, \overline{D}(I))$ and $Forbidden(g, \overline{D}(I))$, which are defined as follows.

Definition 1. *Let g be a connected graph. The sets $Mandatory(g, i)$, $Possible(g, i)$ and $Forbidden(g, i)$ contain, respectively, the edges that belong to all, some or none of the spanning trees of g with weight at most i.*

Clearly, a non-tree edge cannot belong to $Mandatory(g, \overline{D}(I))$ and a tree edge cannot belong to $Forbidden(g, \overline{D}(I))$. Thus, we determine which of the tree edges are mandatory and which of the non-tree edges are forbidden. For the first task, we can use techniques that resemble those used in Eppstein's algorithm for finding the k smallest spanning trees of a graph [11]. Let $e \in t$ be an edge whose removal from t disconnects t into two trees t_1 and t_2. Define the *replacement edge* $r_g(e)$ to be a minimum weight edge in g other than e which connects a node from t_1 and a node from t_2. Then we know that:

Lemma 1 ([11], Lemma 3). *For any edge e in an MST t of g such that $g \setminus \{e\}$ is connected, $(t \setminus \{e\}) \cup \{r_g(e)\}$ is an MST of $g \setminus \{e\}$.*

Lemma 2 ([11,28]). *Given a graph g and an MST t of g, the replacements $r_g(e)$ for all edges in t can be computed in time $O(m\alpha(m, n))$.*

For a non-tree edge $e \in g \setminus t$, define the replacement edge $r_g(e)$ to be a maximum weight edge in the unique path in t between u and v. Then we have that

Lemma 3. *For any edge $e \notin t$, $(t \cup \{e\}) \setminus \{r_g(e)\}$ is a minimum-weight spanning tree of g that contains the edge e.*

Proof. Contract e and find an MST of the remaining graph. By the cycle property, it will exclude one of the maximum weight edges on the cycle that was created by the contraction.

Finding the replacement edge for every non-tree edge can be done in linear-time on a RAM using the relevant component of the MST verification algorithm by Dixon et al. [8] or the simplified solution by King [20]: Both algorithms use a linear-time subroutine that determines the weight of the heaviest edge on the path between the endpoints of every non-tree edge. This implies the following $O(m\alpha(m,n))$-time algorithm: Find an MST t of g and compute the replacement of every edge of g with respect to t. An edge $e \in t$ is in $Mandatory(g, \overline{D}(I))$ iff $w(t) - w(e) + w(r_g(e)) > \overline{D}(I)$, i.e., $g \setminus \{e\}$ has only spanning trees which are too heavy to be in the solution. An edge $e \notin t$ is in $Forbidden(g, \overline{D}(I))$ iff $w(t) + w(e) - w(r_g(e)) > \overline{D}(I)$. All other edges are in $Possible(g, \overline{D}(I))$.

3.3 Filtering the Domains of T and I

We are now ready to use the results of the analysis of g to filter the domain of T. This entails the following steps:

1. For each edge $e \in Mandatory(g, \overline{D}(I))$, if $e \notin \overline{D}(T)$ then the constraint is inconsistent. Otherwise, place $e \in \underline{D}(T)$.
2. For each edge $e \in Forbidden(g, \overline{D}(I))$, remove e from $\overline{D}(T)$ (recall that we assume that $\underline{D}(T)$ is initially empty and note that an edge cannot be both mandatory and forbidden. So e is not in $\underline{D}(T)$).

The second step does not invalidate the completeness of the first: By definition, a forbidden edge is not the replacement of a non-mandatory edge. Hence, removing forbidden edges does not change the set $Mandatory(g, \overline{D}(I))$. As for $D(I)$, it is filtered by setting $D(I) \leftarrow D(I) \cap [\min(T), \infty]$, where $\min(T)$ is the minimum weight of a spanning tree of $\overline{D}(T)$ which contains $\underline{D}(T)$. Repeating the algorithm again will not result in more filtering: Since $\overline{D}(I)$ did not change, the sets $Mandatory(g, \overline{D}(I))$ and $Forbidden(g, \overline{D}(I))$ are also unchanged.

4 $WBST$ with Non-fixed Tree and Graph

We now consider the case in which both G and T are not fixed (but the edge weights still are). As before, we first apply the preprocessing step described above.

4.1 Analysis of $D(G)$

The main complication compared to the fixed-graph case is in the analysis of the set of graphs described by $D(G)$ in order to filter $D(T)$. We generalize the definition of the sets $Mandatory$, $Possible$ and $Forbidden$:

Definition 2. *For a set S of graphs, the set $Mandatory(S, i)$ contains the edges that belong to every spanning tree of weight at most i of any connected graph in S, the set $Forbidden(S, i)$ contains the edges that do not belong to any spanning tree of weight at most i of a connected graph in S and the set $Possible(S, i)$ contains all other edges appearing in at least one spanning tree of a connected graph in S.*

We will show that it suffices to analyze the upper bound of the graph domain, namely the graph $\overline{D}(G)$. The intuition behind the following lemmas is that when an edge is removed from the graph, this can only decrease the number of weight-bounded spanning trees in the graph.

Lemma 4 (Monotony of the $Mandatory$ set). *The removal of an edge from a graph cannot turn a mandatory edge into a possible or forbidden edge. Formally: Let g be a graph and $g' = g \setminus \{e\}$ the graph obtained by removing an edge $e = (u, v)$ from g. Then:*

$$\forall a \in g' : (a \in Mandatory(g, \overline{D}(I)) \Rightarrow$$

$$a \in Mandatory(g', \overline{D}(I)))$$

Proof. Let t be an MST of g and let t' be an MST of g' such that if $e \notin t$ then $t' = t$ and otherwise, $t' = (t \setminus \{e\}) \cup \{r_g(e)\}$. Note that $w(t') \geq w(t)$.

Let a be an edge in $g' \cap Mandatory(g, \overline{D}(I))$. By the results of Section 3.2, this implies that $a \in t$ and $w(t) - w(a) + w(r_g(a)) > \overline{D}(I)$. The weight $w(r_g(a))$ of the replacement edge of a in g is not higher than the weight $w(r_{g'}(a))$ of its replacement edge in g'. Hence, we can conclude that $w(t') - w(a) + w(r_{g'}(a)) \geq w(t) - w(a) + w(r_g(a)) > \overline{D}(I)$, which means that $a \in Mandatory(g', \overline{D}(I))$.

Lemma 5 (Monotony of the $Forbidden$ set). *The removal of an edge from a graph cannot turn a forbidden edge into a possible or mandatory edge. Formally: Let g be a graph and $g' = g \setminus \{e\}$ the graph obtained by removing an edge $e = (u, v)$ from g. Then:*

$$\forall a \in g' : (a \in Forbidden(g, \overline{D}(I)) \Rightarrow$$

$$a \in Forbidden(g', \overline{D}(I)))$$

Proof. Again, let t be an MST of g and let t' be an MST of g' such that if $e \notin t$ then $t' = t$ and otherwise, $t' = (t \setminus \{e\}) \cup \{r_g(e)\}$. Note that for every pair of nodes x, y, the weight of the heaviest edge on the path between x and y in t is not larger than the corresponding weight in t'.

Let a be an edge in $g' \cap Forbidden(g, \overline{D}(I))$. By the results of Section 3.2, this implies that $a \notin t$ and $w(t) + w(a) - w(r_g(a)) > \overline{D}(I)$. If $a \in t'$, then $w(t') > \overline{D}(I)$ so g' does not have any spanning tree with weight at most $\overline{D}(I)$, and all edges are forbidden. Otherwise, we distinguish between two cases. If $e \notin t$ then $t' = t$ and for every $a \notin t$, the replacement edge is the same in g and g', i.e., $r_{g'}(a) = r_g(a)$. So $w(t') + w(a) - w(r_{g'}(a)) = w(t) + w(a) - w(r_g(a)) > \overline{D}(I)$ and a is in $Forbidden(g', \overline{D}(I))$.

If $e \in t$, then $t' = (t \setminus \{e\}) \cup \{r_g(e)\}$ and $w(t') = w(t) - w(e) + w(r_g(e))$. For every pair of nodes x, y, the weight of the heaviest edge on the path between them in t' is at most $w(r_g(e)) - w(e)$ larger than the corresponding weight in t, i.e., $w(r_{g'}(a)) \leq w(r_g(a)) + w(r_g(e)) - w(e)$. We get that $w(t') + w(a) - w(r_{g'}(a)) = w(t) - w(e) + w(r_g(e)) + w(a) - w(r_{g'}(a)) \geq w(t) + w(a) - w(r_g(a)) > \overline{D}(I)$, so $a \in Forbidden(g', \overline{D}(I))$.

Corollary 1. *1. The set* $Forbidden(D(G), \overline{D}(I))$ *of edges that do not belong to any spanning tree of weight at most* $\overline{D}(I)$ *of any connected graph in* $D(G)$ *is* $Forbidden(D(G), \overline{D}(I)) = Forbidden(\overline{D}(G), \overline{D}(I))$.

2. The set $Mandatory(D(G), \overline{D}(I))$ *of edges that belong to any spanning tree of weight at most* $\overline{D}(I)$ *of any connected graph in* $D(G)$ *is* $Mandatory(D(G), \overline{D}(I)) = Mandatory(\overline{D}(G), \overline{D}(I))$.

3. The set $Possible(D(G), \overline{D}(I))$ *consists of the remaining edges in* $\overline{D}(G)$.

Proof. A direct consequence of Lemmas 4 and 5.

4.2 Filtering

The algorithm first computes the sets $Mandatory(D(G), \overline{D}(I))$, $Possible(D(G), \overline{D}(I))$ and $Forbidden(D(G), \overline{D}(I))$. It then uses them to filter the domains of T and I as in the case of fixed graph and tree, with a simple addition: Whenever an edge is placed in $\underline{D}(T)$, it is also placed in $\underline{D}(G)$.

5 *WBST* When Only $V(G)$ and $V(T)$ Are Fixed

We now turn to the most general case that we are able to solve in polynomial time: The one in which only the node-sets of G and T are fixed, but their edge-sets and the other two variables (I and W) are not. The weight of an edge e is now represented by the entry $W[e]$ of W, whose domain is an interval $[\underline{D}(W[e]), \overline{D}(W[e])]$. If we view such an edge e as an infinite set of parallel edges, one for each weight in $D(W[e])$, we get a problem which is similar to the one in the previous section, on infinite graphs T and G. There is, however, a subtle difference: If an edge is in $\underline{D}(T)$ or $\underline{D}(G)$, it means that *one* of the parallel edges it generates needs to be in the respective graph, and not all of them. Nevertheless, this does not affect the correctness of Corollary 1, which does not refer to $\underline{D}(G)$ at all. We will sketch how our algorithm can be applied to these graphs, which are infinite but have finite representations.

All steps that depend only on the topology of graphs (and do not involve edge weights) are unchanged. The other steps are modified as follows. In the preprocessing step, when contracting an edge $e \in \underline{D}(T)$, we subtract $\underline{D}(W[e])$ from the bounds of $D(I)$. When filtering G, we need to compute a minimum spanning tree and then search for the replacement edge for each MST edge. Here we need to minimize the weight of the spanning tree for all possible values of the weights so we assume that the weight of each edge e is $\underline{D}(W[e])$.

In the analysis of $\overline{D}(G)$, we again find an MST t using the minimum possible weight for each edge. When finding a replacement edge for a tree edge e, we search for the non-tree edge e' with minimal $\underline{D}(W[e'])$ and proceed as before (e is mandatory iff $w(t) - \underline{D}(W[e]) + \underline{D}(W[e']) > \overline{D}(I)$). When searching for a replacement edge for a non-tree edge e, we select the tree edge e' on the path between the endpoint of e with maximal $\underline{D}(W[e'])$. Then e is forbidden if $w(t) - \underline{D}(W[e]) + \underline{D}(W[e']) > \overline{D}(I)$).

Finally, we need to consider upper bounds on the weights of the tree edges; the weight selected for them must not be so high that the total weight of the spanning tree is above $\overline{D}(I)$. We reverse the contraction of edges that were in $\underline{D}(T)$ in the input, and find a minimum spanning tree t of $\overline{D}(G)$ that contains $\underline{D}(T)$. For every edge in $e \in \underline{D}(T)$, we set $D(W[e]) \leftarrow D(W[e]) \cap [-\infty, \overline{D}(I) - w(t) + \underline{D}(W[e])]$.

6 NP-Hardness of the Most General Case

As mentioned in the introduction, we wish to define constraints that allow efficient filtering. Theorem 1 states that if it is NP-hard to determine whether there exists a solution to a constraint defined on graph variables (and possibly other variables) then it is NP-hard to filter the domains of the variables to bound consistency.

Theorem 1. *Let $C(G_1, \ldots, G_k, X_1, \ldots X_\ell)$ be a constraint defined on graph variables $G_1, \ldots, G_k$ and scalar variables $X_1, \ldots, X_\ell$ whose domains are intervals of a totally ordered set. If there is a polynomial-time algorithm that narrows the domains of all variables to bound consistency then there is a polynomial-time algorithm that finds a single solution to C.*

Proof. Assume that we have an algorithm A that filters the variable domains to bound consistency in polynomial time. Then we can find a solution to the constraint as follows: First, use algorithm A to compute bound consistency (the constraint has a solution iff all domains are now non-empty). Next, repeat until all variable domains are fixed or there is a variable whose domain is empty:

1. If there is a graph variable G_i such that $\underline{D}(G_i) \neq \overline{D}(G_i)$, select a node v or an edge e from $\overline{D}(G_i) \setminus \underline{D}(G_i)$ and include it in $\underline{D}(G_i)$. Use algorithm A to compute bound consistency.
2. If there is a scalar variable X_i with $\underline{D}(X_i) \neq \overline{D}(X_i)$, set $\underline{D}(X_i) \leftarrow \overline{D}(X_i)$. Use algorithm A to compute bound consistency.

In each iteration, we select a variable whose domain was not fixed and narrow it. If it is a scalar variable X_i, we force it to be the upper endpoint of its domain (which belongs to a solution by the assumption that the domains are bound consistent). If it is a graph variable, we narrow its domain by excluding all graph that do not contain the selected node or edge (again, by the assumption that the domains are bound consistent, there is a solution in which this node or edge belongs to the graph assigned to G_i). Since the number of iterations is upper bounded by the number of scalar variables plus the sum of the sizes of the $\overline{D}(G_i)$'s, the number of calls to A, and hence the running time of the algorithm, is polynomial in the size of the input.

Now, it is NP-hard to check feasibility of $WBST$ in its most general form:

Lemma 6. *It is NP-hard to check whether a $WBST$ constraint has a solution.*

Proof. By reduction from STEINER TREE, which is the following NP-hard problem [13]: Given an undirected graph $H = (V, E)$ with edge-weights w, a subset $U \subseteq V$

of the nodes and a parameter k, determine whether the graph has a subtree of weight at most k that contains all nodes of U.

Given an instance of this problem, let $D(G) = D(T) = [U, H]$ and $D(I) = k$ (i.e., G and T must contain all nodes of U and may or may not contain the other nodes and the edges of H). Then there is a solution to the STEINER TREE problem iff there is a solution to the constraint $WBST(G, T, I, w)$.

Using Theorem 1, this implies NP-hardness of filtering $WBST$ to bound consistency when all variables are not fixed. In conclusion, we have shown the following:

Theorem 2. *Let G and T be graph variables let W be a collection of scalar variables whose domains are intervals of rational numbers, specified by their endpoints.*

If the node-sets of G and T are not fixed, it is NP-hard to filter $WBST(G, T, I, W)$ to bound-consistency. However, if the node-sets are fixed, there exists an algorithm that filters the constraint to bound-consistency in $O(m\alpha(m, n))$ time.

7 On Optimization Constraints

We now wish to explain our definition of the $WBST$ constraint. In particular, the fact that I is an *upper bound* on the weight of T rather than its exact weight. A Constraint Optimization Problem (COP) is a CSP that contains an optimization criteria such as "minimize X" or "maximize Y". When solving a COP, the solver is constantly looking for solutions which are *better than the best solution found so far*. Filtering, then, does not need to identify which variables belong to *some* solution, but rather those that belong to an *improving* solution. The $WBST$ constraint where I is an upper bound on the weight of T is therefore an *optimization constraint* which is suitable for this setting: Whenever the solver finds a solution of weight i, it sets $\overline{D}(i) \leftarrow \min\{\overline{D}(I), i\}$. By reducing $\overline{D}(I)$, it causes the filtering algorithm to remove more values from the domains of the other variables. On the other hand, if the filtering algorithm discovers that the weight of T cannot be less than i in any solution, it communicates this to the solver by setting $\underline{D}(I) \leftarrow \max\{\underline{D}(I), i\}$.

Now, what we wrote above explains how the $WBST$ filtering algorithm can be used in a branch and bound search. However, defining I to be the exact weight of T in the solution would not hurt the functionality of the constraint (by setting $\underline{D}(I) = 0$ we can allow any values up to $\overline{D}(I)$ as before). Would it not have been nicer to define the stronger variant of $WBST$, the *Exact-Weight Spanning-Tree* constraint $EWST(G, T, I, W)$, where I is *equal* to the weight of T? We show, by reduction from SUBSET-SUM, that $EWST$ is NP-hard to filter, even when the graph is fixed.

Lemma 7. *Given a graph $G = (V, E)$ with weights on the edges and a constant k, it is NP-hard to determine whether G has a spanning tree of weight k.*

Proof. By reduction from SUBSET-SUM, which is the following NP-hard problem [7]: Given a set S of integers and a target k, determine whether there is a subset of S that sums to k. We will assume that $k \neq 0$ (SUBSET-SUM remains NP-hard under this assumption).

Given an instance $(S = \{x_1, \ldots, x_n\}, k)$ of SUBSET-SUM, we construct a graph G_S that has a spanning tree of weight k iff S has a subset that sums to k. The graph has a node v_i for every element $x_i \in S$ plus two additional nodes, v_0 and $\tilde{v}$. For each $1 \leq i \leq n$, the graph contains the edge $(\tilde{v}, v_i)$ with weight x_i. In addition, v_0 is connected to every other node by an edge of weight 0. Clearly, for any spanning tree of the graph, the non-zero weight edges correspond to a subset of S that sums to the weight of the spanning tree.

Corollary 2. *It is NP-hard to determine whether an $EWST$ constraint has a solution.*

Proof. If there exists a polynomial-time algorithm that determines whether an $EWST$ constraint has a solution, it can be used to solve SUBSET-SUM in polynomial-time as follows: Let $D(G) = [G_S, G_S]$, $D(T) = [\emptyset, G_S]$, $D(I) = \{k\}$ and w be the edge-weights in G_S. The constraint $EWST(G, T, I, w)$ has a solution iff G_S has a spanning tree of weight k.

References

1. I. D. Aron and P. Van Hentenryck. A constraint satisfaction approach to the robust spanning tree problem with interval data. In *UAI*, pages 18–25, 2002.
2. F. Azevedo. Cardinal: A finite sets constraint solver. *Constraints*, 12(1):to appear, 2007.
3. N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In *CP-AI-OR 2005*, volume 3524 of *LNCS*, pages 64–78. Springer-Verlag, 2005.
4. N. Beldiceanu, I. Katriel, and X. Lorca. Undirected forest constraints. In *CP-AI-OR 2006*, volume 3990 of *LNCS*, pages 29–43. Springer-Verlag, 2006.
5. Hadrien Cambazard and Eric Bourreau. Conception d'une contrainte globale de chemin. In *10e Journ. nat. sur la rsolution pratique de problmes NP-complets (JNPC'04)*, pages 107–121, 2004.
6. A. Chabrier, E. Danna, C. Le Pape, and L. Perron. Solving a network design problem. *Annals of Operations Research*, 130:217–239, 2004.
7. T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, Cambridge, MA, USA, 1990.
8. B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–1192, 1992.
9. G. Dooms, Y. Deville, and P. Dupont. CP(Graph): Introducing a graph computation domain in constraint programming. In *CP 2005*, volume 3709 of *LNCS*, pages 211–225. Springer-Verlag, 2005.
10. G. Dooms and I. Katriel. The *minimum spanning tree* constraint. In Frederic Benhamou, editor, *12th International Conference on Principles and Practice of Constraint Programming (CP 2006)*, volume 4204 of *Lecture Notes in Computer Science*, pages 152–166, Nantes, France, 2006. Springer-Verlag.
11. D. Eppstein. Finding the k smallest spanning trees. In *SWAT '90*, pages 38–47, London, UK, 1990. Springer-Verlag.
12. D. Eppstein. Setting parameters by example. *SIAM J. Comput.*, 32(3):643–653, 2003.
13. M. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. Freeman, 1979.
14. C. Gervet. New structures of symbolic constraint objects: sets and graphs. In *Third Workshop on Constraint Logic Programming (WCLP'93)*, Marseille, 1993.

15. C. Gervet. Conjunto: Constraint propagation over set constraints with finite set domain variables. In *ICLP*, page 733, 1994.
16. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
17. C. Gervet and P. Van Hentenryck. Length-lex ordering for set csps. In *AAAI*. AAAI Press, 2006.
18. M. X. Goemans. Minimum bounded degree spanning trees. In *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 273–282, 2006.
19. L. Gouveia and T. L. Magnanti. Network flow models for designing diameter-constrained minimum-spanning and steiner trees. *Networks*, 41(3):159–173, 2003.
20. V. King. A simpler minimum spanning tree verification algorithm. In *WADS*, volume 955 of *LNCS*, pages 440–448. Springer, 1995.
21. J. Könemann, A. Levin, and A. Sinha. Approximating the degree-bounded minimum diameter spanning tree problem. *Algorithmica*, 41(2):117–129, 2004.
22. C. Lepape, L. Perron, J-C Regin, and P. Shaw. A robust and parallel solving of a network design problem. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, volume LNCS 2470, pages 633–648, 2002.
23. J.-F. Puget. Pecos: a high level constraint programming language. In *SPICIS'92*, 1992.
24. J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-94*, pages 362–367, 1994.
25. M. Sellmann. Cost-based filtering for shorter path constraints. In F. Rossi, editor, *CP 2003*, volume 2833 of *LNCS*, pages 694–708. Springer, 2003.
26. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
27. R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.
28. R. E. Tarjan. Applications of path compression on balanced trees. *J. ACM*, 26(4):690–715, 1979.

Eliminating Redundant Clauses in SAT Instances

Olivier Fourdrinoy, Éric Grégoire, Bertrand Mazure, and Lakhdar Saïs

CRIL CNRS & IRCICA
Université d'Artois
Rue Jean Souvraz SP18
F-62307 Lens Cedex France
{fourdrinoy,gregoire,mazure,sais}@cril.fr

Abstract. In this paper, we investigate to which extent the elimination of a class of redundant clauses in SAT instances could improve the efficiency of modern satisfiability provers. Since testing whether a SAT instance does not contain any redundant clause is NP-complete, a logically incomplete but polynomial-time procedure to remove redundant clauses is proposed as a pre-treatment of SAT solvers. It relies on the use of the linear-time unit propagation technique and often allows for significant performance improvements of the subsequent satisfiability checking procedure for really difficult real-world instances.

1 Introduction

The SAT problem, namely the issue of checking whether a set of Boolean clauses is satisfiable or not, is a central issue in many computer science and artificial intelligence domains, like e.g. theorem proving, planning, non-monotonic reasoning, VLSI correctness checking and knowledge-bases verification and validation. These last two decades, many approaches have been proposed to solve hard SAT instances, based on -logically complete or not- algorithms. Both local-search techniques (e.g. [1]) and elaborate variants of the Davis-Putnam-Loveland-Logemann's DPLL procedure [2] (e.g. [3,4]) can now solve many families of hard huge SAT instances from real-world applications.

Recently, several authors have focused on detecting possible hidden structural information inside real-world SAT instances (e.g backbones [5], backdoors [6], equivalences [7] and functional dependencies [8]), allowing to explain and improve the efficiency of SAT solvers on large real-world hard instances. Especially, the conjunctive normal form (CNF) encoding can conduct the structural information of the initial problem to be hidden [8]. More generally, it appears that many real-world SAT instances contain redundant information that can be safely removed in the sense that equivalent but easier to solve instances could be generated.

In this paper, we investigate to which extent the elimination of a class of redundant clauses in real-world SAT instances could improve the efficiency of modern satisfiability provers. A redundant clause is a clause that can be removed from the instance while keeping the ability to derive it from the remaining part of the instance. Since testing whether a SAT instance does not contain any redundant clause is NP-complete [7], an incomplete but polynomial-time procedure to remove redundant clauses is proposed as a pre-treatment of SAT solvers. It relies on the use of the linear time unit propagation technique. Interestingly, we show that it often allows for significant performance

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 71–83, 2007.

improvements of the subsequent satisfiability checking procedure for hard real-world instances.

The rest of the paper is organized as follows. After basic logical and SAT-related concepts are provided in Section 2, Section 3 focuses on redundancy in SAT instances, and the concept of redundancy modulo unit propagation is developed. In Section 4, our experimental studies are presented and analyzed. Related works are discussed in Section 5 before conclusive remarks and prospective future research works are given in the last Section.

2 Technical Background

Let $\mathcal{L}$ be a standard Boolean logical language built on a finite set of Boolean variables, denoted x, y, etc. Formulas will be denoted using letters such as c. Sets of formulas will be represented using Greek letters like Γ or Σ. An interpretation is a truth assignment function that assigns values from $\{true, false\}$ to every Boolean variable. A formula is consistent or satisfiable when there is at least one interpretation that satisfies it, i.e. that makes it become $true$. Such an interpretation is called a model of the instance. An interpretation will be denoted by upper-case letters like I and will be represented by the set of literals that it satisfies. Actually, any formula in $\mathcal{L}$ can be represented (while preserving satisfiability) using a set (interpreted as a conjunction) of clauses, where a clause is a finite disjunction of literals, where a literal is a Boolean variable that can be negated. A clause will also be represented by the set formed with its literals. Accordingly, the size of a clause c is given by the number of literals that it contains, and is noted $|c|$.

SAT is the NP-complete problem [9] that consists in checking whether a finite set of Boolean clauses of $\mathcal{L}$ is satisfiable or not, i.e. whether there exists an interpretation that satisfies all clauses in the set or not.

Logical entailment will be noted using the $\models$ symbol: let c be a clause of $\mathcal{L}$, $\Sigma \models c$ iff c is $true$ in all models of Σ. The empty clause will represent inconsistency and is noted $\bot$.

In the following, we often refer to the binary and Horn fragments of $\mathcal{L}$ for which the SAT issue can be solved in polynomial time [10,11,12]. A binary clause is a clause formed with at most two literals whereas a Horn clause is a clause containing at most one positive literal. A unit clause is a single literal clause.

In this paper, the *Unit Propagation* algorithm (in short UP) plays a central role. UP recursively simplifies a SAT instance by propagating -throughout the instance- the truth-value of unit clauses whose variables have been already assigned, as shown in algorithm 1.

We define entailment modulo Unit Propagation, noted $\models_{UP}$, the entailment relationship $\models$ restricted to the Unit Propagation technique.

Definition 1. *Let Σ be a SAT instance and c be a clause of $\mathcal{L}$, $\Sigma \models_{UP} c$ if and only if $\Sigma \wedge \neg c \models_{UP} \bot$ if and only if $UP(\Sigma \wedge \neg c)$ contains an empty clause.*

Clearly, $\models_{UP}$ is logically incomplete. It can be checked in polynomial time since UP is a linear-time process. Let c_1 and c_2 be two clauses of $\mathcal{L}$. When $c_1 \subseteq c_2$, we have that

Algorithm 1. Unit_Propagation

Input: a SAT instance Σ
Output: an UP-irredundant SAT instance Γ equivalent to Σ w.r.t. satisfiability s.t. Γ does not contain any unit clause

```
1  begin
2      if Σ contains an empty clause then return Σ;
3      else
4          if Σ contains a unit clause c = {l} then
5              remove all clauses containing l from Σ;
6              foreach c ∈ Σ s.t. ¬l ∈ c do
7                  c ⟵ c \ {¬l}
8              return Unit_Propagation(Σ);
9          else
10             return Σ;
11 end
```

$c_1 \vDash c_2$ and c_1 (resp. c_2) is said to subsume (resp. be subsumed by) c_2 (resp. c_1). A clause $c_1 \in \Sigma$ is subsumed in Σ iff there exists $c_2 \in \Sigma$ s.t. $c_1 \neq c_2$ and c_2 subsumes c_1. Σ is closed under subsumption iff for all $c \in \Sigma$, c is not subsumed in Σ.

3 Redundancy in SAT Instances

Intuitively, a SAT instance is redundant if it contains parts that can be logically inferred from it. Removing redundant parts of SAT instances in order to improve the satisfiability checking process entails at least two fundamental issues.

- First, it is not clear whether removing such parts will make satisfiability checking easier, or not. Some redundant information can actually improve the efficiency of SAT solvers (see e.g. [13,14]). For example, it is well-known that redundancy can help local search to escape from local minima.
- It is well-known that checking whether a SAT instance is irredundant or not is itself NP-complete [7]. It is thus as hard as solving the SAT instance itself.

In order to address these issues, we consider an incomplete algorithm that allows some redundant clauses to be detected and that remains polynomial. Intuitively, we should not aim at removing all kinds of redundant clauses. Some types of clauses are expected to facilitate the satisfiability testing since they belong to polynomial fragments of SAT, especially the binary and the Horn ones. Accordingly, we propose an approach that appears to be a two-levels trade-off: on the one hand, we run a redundancy detection and removal algorithm that is both fast and incomplete. On the other hand, we investigate whether it proves useful to eliminate redundant binary and Horn clauses or not.

Definition 2
Let Σ be a SAT instance and let $c \in \Sigma$, c is redundant in Σ if and only if $\Sigma \setminus \{c\} \vDash c$.

Algorithm 2. Compute an UP-irredundant formula

Input: a SAT instance Σ
Output: an UP-irredundant SAT instance Γ equivalent to Σ w.r.t. satisfiability

```
1 begin
2   Γ ⟵ Σ ;
3   forall clauses c = {l_1, ..., l_n} ∈ Σ sorted according to their decreasing sizes do
4     if UP(Γ \ {c} ∪ {¬l_1} ∪ ... ∪ {¬l_n}) contains an empty clause then
5       Γ ⟵ Γ \ {c} ;
6   return Γ;
7 end
```

Clearly, redundancy can be checked using a refutation procedure. Namely, c is redundant in Σ iff $\Sigma \setminus \{c\} \cup \{\neg c\} \vDash \bot$. We thus strengthen this refutation procedure by replacing $\vDash$ by $\vDash_{UP}$ in order to get an incomplete but polynomial-time redundancy checking approach.

Definition 3
Let Σ be a SAT instance and let $c \in \Sigma$, c is UP-redundant in Σ if and only if $\Sigma \setminus \{c\} \vDash_{UP} c$.

Accordingly, checking the UP-redundancy of c in Σ amounts to propagate the opposite of every literal of c throughout $\Sigma \setminus \{c\}$.

Let us consider Example 1, as depicted below. In this example, it is easy to show that $w \vee x$ is UP-redundant in Σ, while it is not subsumed in Σ. Let us consider $\Sigma \setminus \{w \vee x\} \wedge \neg w \wedge \neg x$. Clearly, $w \vee \neg y$ and $x \vee \neg z$ reduce to $\neg y \wedge \neg z$, respectively. Propagating these two literals generates a contradiction, showing that $w \vee x$ is UP-redundant in Σ. On the other hand, $w \vee x$ is clearly not subsumed in Σ since there is no other clause $c' \in \Sigma$ s.t. $c' \subseteq c$.

Example 1

$$\Sigma = \begin{cases} w \vee x \\ y \vee z \\ w \vee \neg y \\ x \vee \neg z \\ \dots \end{cases}$$

Accordingly, in Algorithm 2, a (basic) UP pre-treatment is described and can be motivated as follows. In the general case, there exists a possibly exponential number of different sets of irredundant formulas that can be extracted from the initial instance. Indeed, irredundancy and minimally inconsistency coincide on unsatisfiable formulas [7]. Clearly, the specific resulting simplified instance delivered by the Algorithm 1 depends on the order according to which clauses from Σ are considered. As small-size clauses allow one to reduce the search space in a more dramatic manner than longer ones, we have implemented a policy that checks longer clauses for redundancy, first. Accordingly, this amounts to considering the clauses Σ according to their decreasing sizes.

Example 2. Let Σ be the following SAT instance:

$$\Sigma = \begin{cases} w \vee x \\ w \vee x \vee y \vee z \\ w \vee \neg y \\ x \vee \neg z \end{cases}$$

- not considering clauses according to their decreasing sizes, but starting with the $w \vee x$ clause, the resulting UP-irredundant set of clauses would be the following Σ_1:

$$\Sigma_1 = \begin{cases} w \vee x \vee y \vee z \\ w \vee \neg y \\ x \vee \neg z \end{cases}$$

- whereas Algorithm 1, which considers $w \vee x \vee y \vee z$ first, delivers for this example a different -but same size- final set Σ_2 of clauses.

$$\Sigma_2 = \begin{cases} w \vee x \\ w \vee \neg y \\ x \vee \neg z \end{cases}$$

Starting with larger size clauses allows to obtain the smallest set of clauses in terms of number of clauses and also in terms of number of literals. We are sure that all subsumed clauses are removed in this way and only the subsuming clauses are preserved because the larger ones are first tested. As this example illustrates, subsumed clauses are removed, leading to shorter clauses in Σ_2, which is thus more constrained and, to some extent, easier to solve.

Property 1. Let Σ be a CNF formula. If Σ' is a formula obtained from Σ by applying Algorithm 2 then Σ' is closed under subsumption.

Proof. Suppose that there exist two clauses c and c' of Σ' such that c' subsumes c. We can deduce that $|c'| \leq |c|$. As the clauses of Σ checked for UP-redundancy are ordered according to their decreasing sizes, we deduce that c is UP-redundant. Consequently, $c \notin \Sigma'$, which contradicts the hypothesis. □

The converse of Property 1 is false. Indeed, the formula $\Sigma = \Sigma_2 \cup \{(y \vee e), (z \vee \neg e)\}$ is closed under subsumption but is not UP-irredundant.

Clearly, in the best cases, the pre-treatment could allow us to get rid of all non-polynomial clauses, and reduce the instance into a polynomial fragment. Since the size of the instances can be huge, we investigate whether polynomial fragments of $\mathcal{L}$ should be protected from redundancy checking or not. As a comparison study, several possible fragments have been considered for UP-redundancy checking: namely Σ, all non Horn clauses from Σ, all non-binary clauses from Σ, and all non-Horn and non-binary clauses from Σ.

Also, we have experimented an approach trying to maximize the number of UP triggerings. The intuition is as follows. A clause that contains literals that occur in many binary clauses will lead to a cascade of UP steps. For example, let $c = x \vee y \vee z$. When

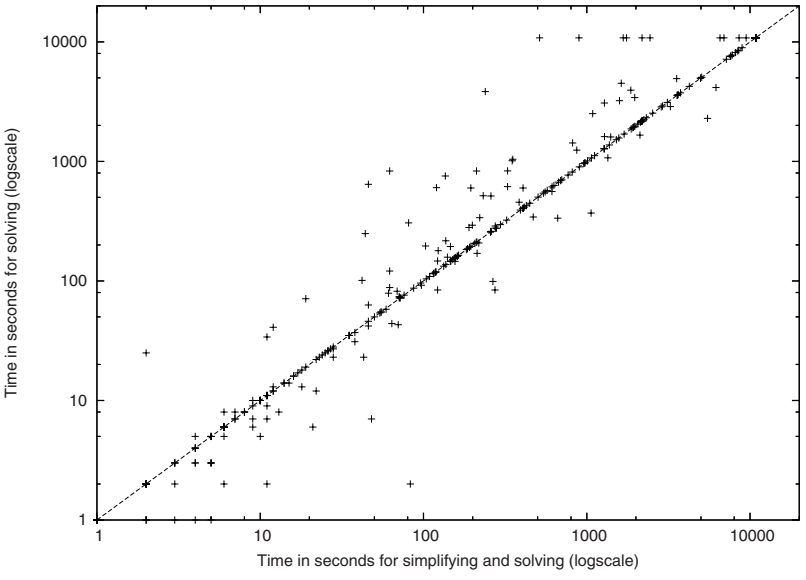


Fig. 1. Results for the 700 tested instances

x occurs in several binary clauses and when we check whether c is redundant using UP, each such binary clause will be simplified into a unit clause, recursively leading to other UP steps. Accordingly, we define a weight w associated to a clause c, noted $w(c)$ as the sum of the weights of each literal of c, where the weight of a literal is given by the number of binary clauses to which it belongs. Let us note that in order for a UP propagation step to occur when a clause $c_1 \in \Sigma$ is checked for redundancy using UP, there must be another clause $c_2 \in \Sigma$ s.t. $|c_2 - \{c_1 \cap c_2\}| = 1$. Clearly, when $|c_2 - \{c_1 \cap c_2\}| = 0$, c_1 is UP-redundant. Since computing and recording this necessary condition can be resource-consuming, we have implemented a more restrictive and easier-to-compute criterion based on the aforementioned weights. When c_2 is a binary clause, the previous condition is satisfied if and only if c_1 (to be checked for redundancy) contains a literal from c_2. Accordingly, when $w(c) = 0$, c is not checked for redundancy. This weight-heuristic has been mixed with a policy allowing clauses from polynomial classes (binary, Horn, binary and Horn) to be protected from redundancy checking.

4 Experimental Results

We have experimented the aforementioned UP-based pre-treatment extensively on the last SAT competitions benchmarks (`http://www.satcompetition.org`). We have tested more than 700 SAT instances that stem from various domains (industrial, random, hand-made, graph-coloring, etc.). Three of the winners of the last three competitions, namely ZChaff, Minisat and SatElite have been selected as SAT solvers. All

Table 1. Some typical instances

Instances	short name	#C	#V
gensys-icl004.shuffled-as.sat05-3825.cnf	gensys	15960	2401
rand_net60-30-5.miter.shuffled.cnf	rand_net	10681	3000
f3-b25-s0-10.cnf	f3-b25	12677	2125
ip50.shuffled-as.sat03-434.cnf	ip50	214786	66131
f2clk_40.shuffled-as.sat03-424.cnf	f2clk	80439	27568
f15-b3-s3-0.cnf	f15-b3	469519	132555
IBM_FV_2004_rule_batch_23_SAT_dat.k45.cnf	IBM_k45	381355	92106
IBM_FV_2004_rule_batch_22_SAT_dat.k70.cnf	IBM_k70	327635	63923
f15-b2-s2-0.cnf	f15-b2	425316	120367
IBM_FV_2004_rule_batch_22_SAT_dat.k75.cnf	IBM_k75	979230	246053
okgen-c2000-v400-s553070738-553070738.cnf	okgen	2000	400

experiments have been conducted on Pentium IV, 3Ghz under linux Fedora Core 4. The complete list of our experimental data and results are available at:
`http://www.cril.fr/~fourdrinoy/eliminating_redundant_clauses.html`

First, we have run the three SAT solvers on all benchmarks, collecting their computing times to solve each instance. A time-out was set to 3 hours. Then, we have run the UP pre-treatment on those benchmarks and collected both the simplification run-times and the subsequent run-times spent by each of the aforementioned solvers on the simplified instances. More precisely, we have experimented a series of different forms of UP pre-treament. In the first one, we have applied the UP-redundancy removing technique on all clauses. In the other ones, non-binary, non Horn clauses have been targeted, successively. We have also targeted clauses that are neither Horn nor binary. Finally, all those experimentations have been replayed using the additional triggering heuristic $w(c) > 0$.

In Fig 1, we show the gain on all 700 tested instances. On the x-axis, we represent the time for simplifying and solving an instance with its best policy. On the y-axis the best time for solving the initial -not yet simplified- instance is given. Accordingly, the line of centers represents the borderline of actual gain. Instances that are above the line of centers benefit from the simplification policy. Clearly, this figure shows that our technique is best employed for difficult instances requiring large amounts of CPU-time to solve them. Indeed, for those instances we obtain significant gains most often. In particular, all the dots horizontally aligned at 10000 seconds on the y-axis represent SAT instances that can not be solved -by no solver- without UP-redundant simplification.

Not surprisingly, our experiments show us that applying the simplification method on all clauses is often more time-consuming than focusing on non-polynomial clauses only and delivers the smallest simplified instances. However, this gain in size does not lead to a systematic run-time gain in solving the instances, including the simplification time. Indeed, these polynomial clauses might allow an efficient resolution of these instances.

Globally, our experiments show us that the best policy consists in applying the weight-based heuristic on all clauses.

In Tables 1 to 5, a more detailed typical sample of size-reduction of instances by the several aforementioned methods is provided. In Table 1, we provide for each

Table 2. Simplification time and size reduction

	No Restriction		Horn		Binary		Horn& Binary	
instances	T_s	#c$_r$(%)	T_s	#c$_r$(%)	T_s	#c$_r$(%)	T_s	#c$_r$(%)
gensys	1.26	2861(17.92)	1.18	2208(13.83)	1.21	2560(16.04)	1.13	2171(13.60)
rand_net	1.80	608(5.69)	0.62	178(1.66)	0.62	0(0)	0.30	0(0)
f3-b25	1.66	1502(11.84)	1.04	926(7.30)	1.64	1502(11.84)	1.03	926(7.30)
ip50	65.06	1823(0.84)	22.02	504(0.23)	14.11	194(0.09)	9.10	110(0.05)
f2clk	6.39	344(0.42)	2.08	119(0.14)	1.29	77(0.09)	0.75	54(0.06)
f15-b3	359.52	55816(11.88)	116.73	14167(3.01)	55.38	1010(0.21)	37.62	640(0.13)
IBM_k45	53.26	32796(8.59)	10.33	2122(0.55)	6.22	717(0.18)	5.03	715(0.18)
IBM_k70	36.68	22720(6.93)	5.36	4628(1.41)	3.81	0(0)	2.78	0(0)
f15-b2	306.06	50717(11.92)	100.14	12909(3.03)	47.74	979(0.23)	34.00	609(0.14)
IBM_k75	172.47	116841(11.93)	40.14	5597(0.57)	24.64	3912(0.39)	22.34	3911(0.39)
okgen	0.00	0(0)	0.00	0(0)	0.00	0(0)	0.00	0(0)

Weighting $w(c) > 0$

	No Restriction		Horn		Binary		Horn & Binary	
	T_s	#c$_r$(%)	T_s	#c$_r$(%)	T_s	#c$_r$(%)	T_s	#c$_r$(%)
gensys	0.65	2560(16.04)	0.63	2171(13.60)	0.64	2560(16.04)	0.64	2171(13.60)
rand_net	1.66	514(4.81)	0.58	148(1.38)	0.62	0(0)	0.30	0(0)
f3-b25	0.21	60(0.47)	0.15	44(0.34)	0.21	60(0.47)	0.14	44(0.34)
ip50	53.95	1823(0.84)	19.43	504(0.23)	14.24	194(0.09)	9.21	110(0.05)
f2clk	6.06	267(0.33)	1.96	100(0.12)	1.30	77(0.09)	0.80	54(0.06)
f15-b3	229.84	24384(5.19)	83.46	6393(1.36)	55.69	1010(0.21)	37.72	640(0.13)
IBM_k45	34.53	11049(2.89)	10.36	2122(0.55)	6.23	717(0.18)	4.98	715(0.18)
IBM_k70	15.25	11464(3.49)	5.36	4616(1.40)	3.77	0(0)	2.77	0(0)
f15-b2	209.55	22217(5.22)	77.22	5709(1.34)	51.04	979(0.23)	33.32	609(0.14)
IBM_k75	125.26	39640(4.04)	38.15	5597(0.57)	26.02	3912(0.39)	22.32	3911(0.39)
okgen	0	0(0)	0.00	0(0)	0.00	0(0)	0.00	0(0)

instance its numbers of clauses ($\#C$) and variables ($\#V$) and a short name to facilitate the presentation of results. In Table 2, the CPU time in seconds (T_s) needed to simplify the instance is given, together with the obtained size reduction, expressed in number of clauses ($\#c_r$) and expressed in percents. In the second column results are given for a policy that considers all clauses for simplification. In the next ones $Horn$, Bin, $Horn\&Bin$ represent the classes that are protected from simplification. The last columns provide the results for the same policies augmented with the weight heuristics.

In Tables 3 to 5, satisfiability checking run-times are provided for the same instances, using Zchaff, Minisat and SatElite, respectively. TO means "time-out". In the first column, T_b is the CPU-time to solve the initial instance. In the subsequent columns, the CPU-time to solve the simplified instance (T_r) is given together with the efficiency gains with respect to satisfiability checking: $\%_p$ and $\%_g$ being the gains without and with taking the simplification run-time into account. The symbol ∞ (resp. $-\infty$) represents the gain when the pre-processor-less (resp. pre-processor-based) approach fails to deliver a result while the pre-processor-based (resp. pre-processor-less) succeeds. When both approaches fail, the gain is represented by –.

5 Related Work

Introducing a fast -polynomial time- pre-processing step inside logically complete SAT solvers is not a new idea by itself. Mainly, C-SAT [15] was provided with a pre-processor that made use a form of bounded resolution procedure computing all resolvents whose

Table 3. Zchaff results

instances	T_b	No Restriction $T_r(\%_p,\%_g)$	Horn $T_r(\%_p,\%_g)$	Binary $T_r(\%_p,\%_g)$	Horn & Binary $T_r(\%_p,\%_g)$
gensys	(3418.17)	2847.1(16.70,16.66)	3353.69(1.88,1.85)	1988.37(41.82,41.79)	3683.58(-7.76,-7.79)
rand_net	(1334.19)	942.15(29.38,29.24)	1067.01(20.02,19.97)	1334.19(0,-0.04)	1334.19(0,-0.02)
f3-b25	(790.96)	137.26(82.64,82.43)	155.32(80.36,80.23)	134.91(82.94,82.73)	157.44(80.09,79.96)
ip50	(2571.18)	675.11(73.74,71.21)	474.64(81.53,80.68)	1023.82(60.18,59.63)	1945.87(24.31,23.96)
f2clk	(6447.19)	4542.32(29.54,29.44)	9457.25(-46.68,-46.72)	3978.14(38.29,38.27)	3848.02(40.31,40.30)
f15-b3	(7627.95)	5620.25(26.32,21.60)	2926.38(61.63,60.10)	10157.9(-33.16,-33.89)	2419.52(68.28,67.78)
IBM_k45	(5962.46)	2833.1(52.48,51.59)	3656.05(38.68,38.50)	3244.61(45.58,45.47)	4751.79(20.30,20.22)
IBM_k70	(TO)	514.83(∞,∞)	5377.91(∞,∞)	TO(–,–)	TO(–,–)
f15-b2	(TO)	2287.73(∞,∞)	8891.1(∞,∞)	TO(–,–)	3969.27(∞,∞)
IBM_k75	(TO)	TO(–,–)	TO(–,–)	TO(–,–)	TO(–,–)
okgen	(1309.66)	1309.66(0,-0.00)	1309.66(0,-0.00)	1309.66(0,-0.00)	1309.66(0,-0.00)

Weighting $w(c) > 0$

		No Restriction $T_r(\%_p,\%_g)$	Horn $T_r(\%_p,\%_g)$	Binary $T_r(\%_p,\%_g)$	Horn & Binary $T_r(\%_p,\%_g)$
gensys	(3418.17)	1986.98(41.87,41.85)	3896.93(-14.00,-14.02)	1967.22(42.44,42.42)	3873.89(-13.33,-13.35)
rand_net	(1334.19)	555.13(58.39,58.26)	614.75(53.92,53.87)	1334.19(0,-0.04)	1334.19(0,-0.02)
f3-b25	(790.96)	839.54(-6.14,-6.16)	811.37(-2.58,-2.59)	791.97(-0.12,-0.15)	813.05(-2.79,-2.81)
ip50	(2571.18)	708.81(72.43,70.33)	465.13(81.90,81.15)	1091.54(57.54,56.99)	1958(23.84,23.48)
f2clk	(6447.19)	5196.02(19.40,19.31)	5766.78(10.55,10.52)	4042.8(37.29,37.27)	3965.68(38.48,38.47)
15-b3	(7627.95)	TO($-\infty$,$-\infty$)	TO($-\infty$,$-\infty$)	10024(-31.41,-32.14)	2448.27(67.90,67.40)
IBM_k45	(5962.46)	4447.15(25.41,24.83)	3698.1(37.97,37.80)	3283.2(44.93,44.83)	4925.58(17.39,17.30)
IBM_k70	(TO)	4131.58(∞,∞)	5564.5(∞,∞)	TO(–,–)	TO(–,–)
f15-b2	(TO)	4456.24(∞,∞)	2880.15(∞,∞)	TO(–,–)	4028.04(∞,∞)
IBM_k75	(TO)	TO(–,–)	TO(–,–)	TO(–,–)	TO(–,–)
okgen	(1309.66)	1309.66(0,0)	1309.66(0,-0.00)	1309.66(0,-0.00)	1309.66(0,-0.00)

Table 4. Minisat results

instances	T_b	No Restriction $T_r(\%_p,\%_g)$	Horn $T_r(\%_p,\%_g)$	Binary $T_r(\%_p,\%_g)$	Horn & Binary $T_r(\%_p,\%_g)$
gensys	(7543.42)	4357.66(42.23,42.21)	7078.84(6.15,6.14)	4722.35(37.39,37.38)	7370.48(2.29,2.27)
rand_net	(41.15)	11.00(73.26,68.87)	35.12(14.66,13.14)	41.15(0,-1.52)	41.15(0,-0.73)
f3-b25	(755.90)	225.45(70.17,69.95)	246.97(67.32,67.18)	233.73(69.07,68.86)	243.39(67.80,67.66)
ip50	(88.43)	61.95(29.94,-43.62)	138.90(-57.06,-81.97)	64.47(27.09,11.13)	60.13(31.99,21.70)
f2clk	(280.88)	290.41(-3.39,-5.66)	188.53(32.87,32.13)	325.87(-16.01,-16.48)	274.77(2.17,1.90)
15-b3	(875.37)	531.31(39.30,-1.76)	561.40(35.86,22.53)	759.43(13.24,6.91)	555.47(36.54,32.24)
IBM_k45	(3940.82)	3729.01(5.37,4.02)	3568.43(9.44,9.18)	3625.13(8.01,7.85)	3541.57(10.13,10.00)
IBM_k70	(643.67)	76.16(88.16,82.46)	527.58(18.03,17.20)	643.67(0,-0.59)	643.67(0,-0.43)
f15-b2	(516.36)	334.07(35.30,-23.96)	257.93(50.04,30.65)	452.94(12.28,3.03)	369.69(28.40,21.81)
IBM_k75	(4035.72)	5096.73(-26.29,-30.56)	6324.08(-56.70,-57.69)	5782.49(-43.28,-43.89)	5534.54(-37.13,-37.69)
okgen	(46.43)	46.43(0,-0.02)	46.43(0,-0.01)	46.43(0,-0.01)	46.43(0,-0.01)

Weighting $w(c) > 0$

		No Restriction $T_r(\%_p,\%_g)$	Horn $T_r(\%_p,\%_g)$	Binary $T_r(\%_p,\%_g)$	Horn & Binary $T_r(\%_p,\%_g)$
gensys	(7543.42)	4742.55(37.12,37.12)	7434.11(1.44,1.44)	4615.46(38.81,38.80)	7610.16(-0.88,-0.89)
rand_net	(41.15)	27.00(34.38,30.33)	25.60(37.79,36.36)	41.15(0,-1.52)	41.15(0,-0.73)
f3-b25	(755.90)	745.80(1.33,1.30)	738.83(2.25,2.23)	773.19(-2.28,-2.31)	779.37(-3.10,-3.12)
ip50	(88.43)	54.28(38.61,-22.39)	138.28(-56.35,-78.33)	67.17(24.04,7.93)	52.71(40.39,29.97)
f2clk	(280.88)	224.82(19.95,17.79)	221.20(21.24,20.54)	299.23(-6.53,-6.99)	289.93(-3.22,-3.50)
15-b3	(875.37)	543.86(37.87,11.61)	687.06(21.51,11.97)	751.79(14.11,7.75)	498.80(43.01,38.70)
IBM_k45	(3940.82)	1826.6(53.64,52.77)	3324.82(15.63,15.36)	3637.35(7.70,7.54)	3714.29(5.74,5.62)
IBM_k70	(643.67)	31.51(95.10,92.73)	518.48(19.44,18.61)	643.67(0,-0.58)	643.67(0,-0.43)
f15-b2	(516.36)	378.88(26.62,-13.95)	155.70(69.84,54.89)	483.39(6.38,-3.49)	371.51(28.05,21.59)
IBM_k75	(4035.72)	4202.16(-4.12,-7.22)	5782.1(-43.27,-44.21)	5927.94(-46.88,-47.53)	5655.36(-40.13,-40.68)
okgen	(46.43)	46.43(0,0)	46.43(0,-0.00)	46.43(0,-0.00)	46.43(0,-0.00)

Table 5. SatElite results

instances	T_b	No Restriction $T_r(\%_p,\%_g)$	Horn $T_r(\%_p,\%_g)$	Binary $T_r(\%_p,\%_g)$	Horn & Binary $T_r(\%_p,\%_g)$
gensys	(5181.22)	4672.43(9.81,9.79)	7879.9(-52.08,-52.10)	5146.42(0.67,0.64)	7964.11(-53.71,-53.73)
rand_net	(131.01)	173.97(-32.79,-34.16)	110.02(16.01,15.54)	131.01(0,-0.48)	131.01(0,-0.23)
f3-b25	(1664.94)	2935.61(-76.31,-76.41)	1926.54(-15.71,-15.77)	2934.61(-76.25,-76.35)	1924.24(-15.57,-15.63)
ip50	(79.45)	110.37(-38.91,-120.79)	143.30(-80.36,-108.08)	150.33(-89.20,-106.97)	58.31(26.60,15.15)
f2clk	(323.15)	270.61(16.25,14.28)	291.93(9.66,9.01)	453.03(-40.19,-40.59)	518.12(-60.33,-60.56)
15-b3	(833.17)	244.23(70.68,27.53)	976.24(-17.17,-31.18)	281.31(66.23,59.58)	760.59(8.71,4.19)
IBM_k45	(7829.02)	4111.24(47.48,46.80)	4382.78(44.01,43.88)	3457.46(55.83,55.75)	3314.55(57.66,57.59)
IBM_k70	(712.10)	225.22(68.37,63.22)	757.96(-6.43,-7.19)	712.10(0,-0.53)	712.10(0,-0.39)
f15-b2	(794.85)	261.70(67.07,28.56)	575.30(27.62,15.02)	556.98(29.92,23.91)	441.52(44.45,40.17)
IBM_k75	(2877.51)	3714.45(-29.08,-35.07)	4099.39(-42.46,-43.85)	5505.93(-91.34,-92.20)	3796.42(-31.93,-32.71)
okgen	(323.06)	323.06(0,-0.00)	323.06(0,-0.00)	323.06(0,-0.00)	323.06(0,-0.00)

Weighting $w(c) > 0$

		No Restriction $T_r(\%_p,\%_g)$	Horn $T_r(\%_p,\%_g)$	Binary $T_r(\%_p,\%_g)$	Horn & Binary $T_r(\%_p,\%_g)$
gensys	(5181.22)	5052.73(2.47,2.46)	8046.2(-55.29,-55.30)	5163.37(0.34,0.33)	8231.05(-58.86,-58.87)
rand_net	(131.01)	63.00(51.91,50.63)	130.23(0.59,0.14)	131.01(0,-0.47)	131.01(0,-0.23)
f3-b25	(1664.94)	1701.57(-2.20,-2.21)	1698.4(-2.00,-2.01)	1745.22(-4.82,-4.83)	1723.25(-3.50,-3.51)
ip50	(79.45)	113.38(-42.70,-110.61)	136.52(-71.82,-96.28)	146.68(-84.61,-102.54)	60.46(23.90,12.30)
f2clk	(323.15)	313.56(2.96,1.09)	238.38(26.23,25.62)	466.84(-44.46,-44.86)	503.75(-55.88,-56.13)
15-b3	(833.17)	675.88(18.87,-8.70)	1276.87(-53.25,-63.27)	271.53(67.40,60.72)	733.52(11.96,7.43)
IBM_k45	(7829.02)	5836.14(25.45,25.01)	4364.69(44.24,44.11)	3341(57.32,57.24)	3311.19(57.70,57.64)
IBM_k70	(712.10)	406.32(42.94,40.79)	740.12(-3.93,-4.68)	712.10(0,-0.53)	712.10(0,-0.38)
f15-b2	(794.85)	316.32(60.20,33.83)	293.82(63.03,53.31)	549.23(30.90,24.47)	469.28(40.95,36.76)
IBM_k75	(2877.51)	3121.35(-8.47,-12.82)	4017.92(-39.63,-40.95)	5170.69(-79.69,-80.59)	3738.34(-29.91,-30.69)
okgen	(323.06)	323.06(0,0)	323.06(0,-0.00)	323.06(0,-0.00)	323.06(0,-0.00)

size are smaller than the size of their parents. At that time C-SAT was the most powerful solver to solve random k-SAT instances. Satz uses the same technique, but with a resolvent size limited to three [16]. Recently, Niklas Eén and Armin Biere have introduced a variable elimination technique with subsumption, self-subsuming resolution and variable elimination by substitution [17] as a pre-processing step for modern SAT solvers, extending a previous pre-processor NiVER by [18]. Another polynomial-time preprocessor has been introduced by James Crawford and is available in [19]. In [20], an algorithm is described that maintains a subsumption-free CNF clauses database by efficiently detecting and removing subsumption as the clauses are being added. An interesting path for future research would consist in comparing our approach with those other pre-processors from both theoretical and experimental points of view.

Due to its linear-time character, the unit propagation algorithm has been exploited in several ways in the context of SAT, in addition to being a key component of DPLL-like procedures. For example, C-SAT and Satz used a local treatment during important steps of the exploration of the search space, based on UP, to derive implied literals and detect local inconsistencies, and guide the selection of the next variable to be assigned [15,16]. In [21], a double UP schema is explored in the context of SAT solving. In [22,8], UP has been used as an efficient tool to detect functional dependencies in SAT instances. The UP technique has also been exploited in [23] in order to derive subclauses by using the UP implication graph of the SAT instance, and speed up the resolution process.

Bailleux, Roussel and Boufkhad have studied how clauses redundancy affects the resolution of random k-sat instances [24]. However, their study is specific to random

instances and cannot be exported to real-world ones. Moreover, they have explored redundancy and not UP-redundancy; their objective being to measure the redundancy degree of random 3-SAT instances at the crossover point. From a complexity point of view, a full study of redundancy in the Boolean framework is given in [25].

To some extent, our approach is also close to compilation techniques [26, 27, 28, 29] where the goal is to transform the Boolean instances into equivalent albeit *easier* ones to check or to infer from. The idea is to allow much time to be spent in the pre-processing step, transforming the instance into a polynomial-size new instance made of clauses belonging polynomial-time fragments of $\mathcal{L}$, only. Likewise, our approach aims to reduce the size of the non-polynomial fragments of the instances. However, it is a partial reduction since all clauses belonging to non-polynomial fragments are not necessary removed. Moreover, whereas compilation techniques allow a possibly exponential time to be spent in the pre-processing step, we make sure that our pre-processing technique remains a polynomial-time one.

6 Conclusions

Eliminating redundant clauses in SAT instances during a pre-treatment step in order to speed up the subsequent satisfiability checking process is a delicate matter. Indeed, redundancy checking is intractable in the worst case, and some redundant information can actually help to solve the SAT instances more efficiently. In this paper, we have thus proposed and experimented a two-levels trade-off. We rely on the efficiency albeit incomplete character of the unit propagation algorithm to get a fast pre-treatment that allows some -but not all- redundant clauses to be detected. We have shown from an experimental point of view the efficiency of a powerful weight-based heuristics for redundancy extraction under unit propagation. Such a pre-treatment can be envisioned as a compilation process that allows subsequent faster operations on the instances. Interestingly enough, the combined computing time spent by such a pre-treatment and the subsequent SAT checking often outperforms the SAT checking time for the initial instance on very difficult instances.

This piece of research opens other interesting perspectives. For example, such a pre-processing step can play a useful role in the computation of minimally inconsistent subformulas (MUSes) [30]. Also, we have focused on binary and Horn fragments as polynomial fragments. Considering other fragments like e.g. the reverse Horn and renamable Horn could be a fruitful path for future research.

Acknowledgments

This research has been supported in part by the EC under a Feder grant and by the Région Nord/Pas-de-Calais.

References

1. Selman, B., Levesque, H.J., Mitchell, D.G.: A new method for solving hard satisfiability problems. In: Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92). (1992) 440–446

2. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Communications of the ACM **5**(7) (1962) 394–397
3. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001) 530–535
4. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03). (2003) 502–518
5. Dubois, O., Dequen, G.: A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01). (2001) 248–253
6. Williams, R., Gomes, C.P., Selman, B.: Backdoors to typical case complexity. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03). (2003) 1173–1178
7. Liberatore, P.: The complexity of checking redundancy of CNF propositional formulae. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI'02). (2002) 262–266
8. Grégoire, É., Ostrowski, R., Mazure, B., Saïs, L.: Automatic extraction of functional dependencies. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04). (2004) 122–132
9. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, New York (USA), Association for Computing Machinery (1971) 151–158
10. Tarjan, R.E.: Depth first search and linear graph algorithms. SIAM J. Comput. **1** (1972) 146–160
11. Even, S., Itai, A., Shamir, A.: On the complexity of timetable and multicommodity flow problems. SIAM J. Comput. **5** (1976) 691–703
12. Dowling, W.H., Gallier, J.H.: Linear-time algorithms for testing satisfiability of propositional horn formulae. Journal of Logic Programming **1**(3) (1984) 267–284
13. Wei, W., Selman, B.: Accelerating random walks. In: Proceedings of 8th International Conference on the Principles and Practices of Constraint Programming (CP'2002). (2002) 216–232
14. Kautz, H.A., Ruan, Y., Achlioptas, D., Gomes, C.P., Selman, B., Stickel, M.E.: Balance and filtering in structured satisfiable problems. In: Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI'01). (2001) 351–358
15. Dubois, O., André, P., Boufkhad, Y., Carlier, Y.: SAT vs. UNSAT. In: Second DIMACS implementation challenge: cliques, coloring and satisfiability. Volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1996) 415–436
16. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI'97). (1997) 366–371
17. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05). (2005) 61–75
18. Subbarayan, S., Pradhan, D.K.: NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In: Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04). (2004) 276–291
19. Crawford, J.: A polynomial-time preprocessor ("compact") (1996) http://www.cirl.uoregon.edu/crawford/.
20. Zhang, W.: Configuration landscape analysis and backbone guided local search: Part i: Satisfiability and maximum satisfiability. Artificial Intelligence **158**(1) (2004) 1–26

21. Le Berre, D.: Exploiting the real power of unit propagation lookahead. In: Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT'01), Boston University, Massachusetts, USA (2001)
22. Ostrowski, R., Mazure, B., Saïs, L., Grégoire, É.: Eliminating redundancies in SAT search trees. In: Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'2003), Sacramento (2003) 100–104
23. Darras, S., Dequen, G., Devendeville, L., Mazure, B., Ostrowski, R., Saïs, L.: Using Boolean constraint propagation for sub-clauses deduction. In: Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05). (2005) 757–761
24. Boufkhad, Y., Roussel, O.: Redundancy in random SAT formulas. In: Proceedings of the 17th National Conference on Artificial Intelligence (AAAI'00). (2000) 273–278
25. Liberatore, P.: Redundancy in logic i: CNF propositional formulae. Artificial Intelligence **163**(2) (2005) 203–232
26. Selman, B., Kautz, H.A.: Knowledge compilation using horn approximations. In: Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91). (1991) 904–909
27. del Val, A.: Tractable databases: How to make propositional unit resolution complete through compilation. In: Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94). (1994) 551–561
28. Marquis, P.: Knowledge compilation using theory prime implicates. In: Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI'95), Montréal, Canada (1995) 837–843
29. Mazure, B., Marquis, P.: Theory reasoning within implicant cover compilations. In: Proceedings of the ECAI'96 Workshop on Advances in Propositional Deduction, Budapest, Hungary (1996) 65–69
30. Grégoire, É., Mazure, B., Piette, C.: Extracting MUSes. In: Proceedings of the 17th European Conference on Artificial Intelligence (ECAI'06), Trento, Italy (2006) 387–391

Cost-Bounded Binary Decision Diagrams for 0-1 Programming

Tarik Hadžić[1] and J.N. Hooker[2]

[1] IT University of Copenhagen
tarik@itu.dk
[2] Carnegie Mellon University
john@hooker.tepper.cmu.edu

Abstract. In recent work binary decision diagrams (BDDs) were introduced as a technique for postoptimality analysis for integer programming. In this paper we show that much smaller BDDs can be used for the same analysis by employing cost bounding techniques in their construction.

Binary decision diagrams (BDDs) have seen widespread application in logic circuit design and product configuration. They also have potential application to optimization, particularly to postoptimality analysis. A BDD can represent, often in compact form, the entire feasible set of an optimization problem. Optimal solutions correspond to shortest paths in the BDD. Due to the efficiency of this representation, one can rapidly extract a good deal of information about a problem and its solutions by querying the BDD.

This opens the door to fast, in-depth postoptimality analysis without having to re-solve the problem repeatedly—provided the BDD is of manageable size. For instance, one can identify all optimal or near-optimal solutions by finding all shortest or near-shortest paths in the BDD. One can quickly determine how much freedom there is to alter the solution without much increase in cost. This is particularly important in practice, since managers often require some flexibility in how they implement a solution. One can deduce, in real time, the consequences of fixing a variable to a particular value. One can conduct several types of sensitivity analysis to measure the effect of changing the problem data.

We present in [1] several techniques for postoptimality analysis using BDDs. We address here a key computational issue: how large does the BDD grow as the problem size increases, and how can one minimize this growth? In particular we examine the strategy of generating a BDD that represents only near-optimal solutions, since these are generally the solutions of greatest interest in practice. In principle, a BDD that exactly represents the set of near-optimal solutions need be no smaller than one that represents all solutions, and it can in fact be exponentially larger. At least in the problem domain we investigate, however, the BDD representing near-optimal solutions is significantly smaller. We also identify a family of *sound* BDDs that are even smaller but support valid postoptimality analysis—even though they do not exactly represent the set of

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 84–98, 2007.

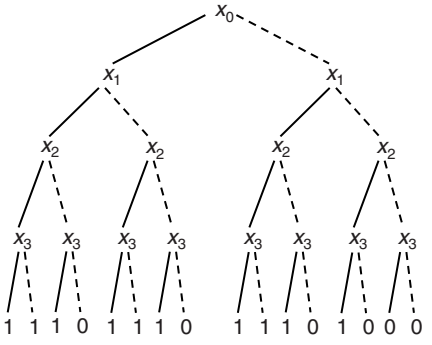


Fig. 1. Branching tree for $2x_0 + 3x_1 + 5x_2 + 5x_3 \geq 7$

near-optimal solutions. This allows for postoptimality analysis of much larger problem instances by using sound BDDs.

One advantage of BDD-based analysis is that it presupposes no structure in the problem, aside from separability of the objective function and finiteness of domains. The methods are the same regardless of whether the constraint and objective functions are linear, nonlinear, convex, or nonconvex. However, for purposes of experimentation we focus on 0-1 linear programming problems. The methods described here are readily extended to nonlinear constraints. They can also be extended to general integer variables by writing each variable as a vector of 0-1 variables.

1 Binary Decision Diagrams

A BDD is a directed graph that represents a boolean function. A given boolean function corresponds to a unique *reduced BDD* when the variable ordering is fixed. The same is true of a constraint set in 0-1 variables, since it can be viewed as a boolean function that is true when the constraints are satisfied and false otherwise.

The reduced BDD is essentially a compact representation of the branching tree for the constraint set. The leaf nodes of the tree are labelled by 1 or 0 to indicate that the constraint set is satisfied or violated. For example, the tree of Fig. 1 represents the 0-1 linear inequality

$$2x_0 + 3x_1 + 5x_2 + 5x_3 \geq 7\ . \tag{1}$$

The solid branches (*high edges*) correspond to setting $x_j = 1$ and the dashed branches (*low edges*) to setting $x_j = 0$.

The tree can be transformed to a reduced BDD by repeated application of two operations: (a) if both branches from a node lead to the same subtree, delete the node; (b) if two subtrees are identical, superimpose them. The reduced BDD for (1) appears in Fig. 2(a).

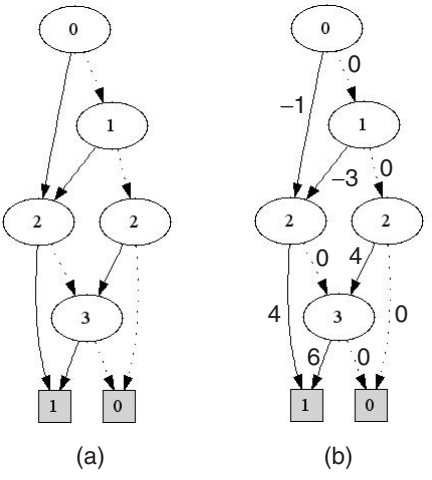


Fig. 2. (a) Reduced BDD for $2x_0 + 3x_1 + 5x_2 + 5x_3 \geq 7$ using the variable ordering x_0, x_1, x_2, x_3. (b) Same BDD with edge lengths corresponding to the objective function $2x_0 - 3x_1 + 4x_2 + 6x_3$.

Each path from the root to 1 in a BDD *represents* one or more solutions, namely all solutions in which x_j is set to 0 when the path contains a low edge from a node labelled x_j, and is set to 1 when the path contains a high edge from such a node. A BDD B represents the set $Sol(B)$ of all solutions that are represented by a path from the root to 1.

A reduced BDD can in principle be built by constructing the search tree and using intelligent caching to eliminate nodes and superimpose isomorphic subtrees. It is more efficient in practice, however, to combine the BDDs for elementary components of the boolean function. For example, if there are several constraints, one can build a BDD for each constraint and then conjoin the BDDs. Two BDDs can be conjoined in time that is roughly quadratic in the number of BDD nodes. Thus if the individual constraints have compact BDDs, this structure is exploited by constructing search trees for the individual constraints and conjoining the resulting BDDs, rather than constructing a search tree for the entire constraint set. Algorithms for building reduced BDDs in this fashion are presented in [2,1].

The BDD for a linear 0-1 inequality can be surprisingly compact. For instance, the 0-1 inequality

$$\begin{aligned} &300x_0 + 300x_1 + 285x_2 + 285x_3 + 265x_4 + 265x_5 + 230x_6 + \\ &\quad 23x_7 + 190x_8 + 200x_9 + 400x_{10} + 200x_{11} + 400x_{12} + \\ &\quad 200x_{13} + 400x_{14} + 200x_{15} + 400x_{16} + 200x_{17} + 400x_{18} \geq 2701 \end{aligned} \tag{2}$$

has a complex feasible set that contains 117,520 minimally feasible solutions (each of which becomes infeasible if any variable is flipped from 1 to 0), as reported in [3]. (Equivalently, if the right-hand side is ≤ 2700, the inequality has 117,520 minimal covers.) The BDD for (2) contains only 152 nodes.

A separable objective function $\sum_j c_j(x_j)$ can be minimized subject to a constraint set by finding a shortest path from the root to 1 in the corresponding BDD. If node u and u' have labels x_k and x_ℓ, respectively, then a high edge from u to u' has length $c^1[u,u'] = c_k(1) + c^*_{k+1,\ell-1}$, where $c^*_{k+1,\ell-1}$ is the cost of setting every skipped variable $x_{k+1},\ldots,x_{l-1}$ to a value that gives the lowest cost. More precisely:

$$c^v[u,u'] = c_k(v) + c^*_{k+1,\ell-1}$$

and

$$c^*_{pq} = \sum_{j=p}^{q} \min\{c_j(1), c_j(0)\}$$

A low edge from u to u' has length $c^0[u,u']$. For example, if we minimize

$$2x_0 - 3x_1 + 4x_2 + 6x_3 \tag{3}$$

subject to (1), the associated BDD has the edge lengths shown in Fig. 2(b). Note that the length of the high edge from the root node is $c_0(1) + c^*_{11} = 2 - 3 = -1$. The shortest path from the root node to 1 has length 1 and passes through the x_1 node and the x_2 node on the left. Its three edges indicate that $(x_0, x_1, x_2) = (0, 1, 1)$. This corresponds to optimal solution $(x_0, x_1, x_2, x_3) = (0, 1, 1, 0)$, where x_3 is set to zero to minimize the x_3 term in the objective function.

2 Previous Work

BDDs have been studied for decades [4,5]. Bryant [6] showed how to reduce a BDD to a unique canonical form, for a given variable ordering. Readable introductions to BDDs include [2,7].

There has been very little research into the application of BDDs to optimization. Becker at al. [8] used BDDs to identify separating cuts for 0-1 linear programming problems in a branch-and-cut context. They generated BDDs for a subset of constraints and obtained a cut $ux \geq u_0$ that is violated by the solution $\bar{x}$ of the linear relaxation of the problem. The cut is obtained by using subgradient optimization to find an assignment of costs u_i to edges of the BDD for which $ux < u_0$, where u_0 is the length of a shortest path to 1 in the BDD.

In [1] we show how BDDs can be used for various types of postoptimality analysis. One type is cost-based domain analysis, which computes the projection of the set of near-optimal solutions onto any given variable. Near-optimal solutions are those whose objective function value is within Δ of the optimal value, where Δ is specified by the user. This type of analysis shows the extent to which the values of variables can be changed without increasing cost more than Δ. We also show how to perform conditional domain analysis, which computes the projections after restricting the domain of one or more variables. This shows the consequences of making certain decisions on possible values for the remaining variables. We illustrate these techniques on capital budgeting, network reliability, and investment problems, the last two of which are nonlinear and nonconvex, and all of which involve general integer variables.

3 Projection and Postoptimality Analysis

Consider a 0-1 programming problem with a separable objective function:

$$\begin{aligned} &\min \sum_{j=1}^{n} c_j(x_j) \\ &g_i(x) \geq b_i, \quad i = 1, \ldots, m \\ &x_j \in \{0,1\}, \quad j = 1, \ldots, n \end{aligned} \tag{4}$$

We refer to any 0-1 n-tuple as a *solution* of (4), any solution that satisfies the constraints as a *feasible solution*. If Sol is the set of feasible solutions, let Sol_j be the projection of Sol onto variable x_j. Then Sol_j can be easily deduced from the reduced BDD B of (4). Sol_j contains the value 0 if at least one path from the root to 1 in B contains a low edge from a node labelled x_j, and it contains 1 if at least one path contains a high edge from a node labelled x_j.

If c^* is the optimal value of (4), then for a given tolerance Δ the set of near-optimal feasible solutions of (4) is

$$Sol_\Delta = \left\{ x \in Sol \;\middle|\; \sum c_j(x_j) \leq c^* + \Delta \right\}$$

In general, *cost-based domain analysis* derives the projection $Sol_{\Delta j}$ of Sol_Δ onto any x_j for any Δ between 0 and some maximum tolerance $\Delta_{\max}$. It may also incorporate conditional domain analysis subject to a partial assignment $x_J = v_J$ specified by the user, where $J \subset \{1, \ldots, n\}$. This projection is

$$Sol_{\Delta j}(x_J = v_J) = \{x_j \mid x \in Sol_\Delta, x_J = v_J\}$$

Algorithms for implementing cost-based domain analysis are presented in [1], and they are based on compiling the solution set of (4) into a BDD B. $Sol_{\Delta j}$ can be efficiently computed by examining paths of length at most $c^* + \Delta$ from the root to 1 in B. $Sol_{\Delta j}$ contains 0 if at least one such path contains a low edge from a node labelled x_j, and it contains 1 if at least one such path contains a high edge from a node labelled x_j [9]. Similar analysis for $Sol_{\Delta j}(x_J = v_J)$ is performed on a restricted BDD that can be efficiently constructed from B.

It is obvious that domain-analysis is correct if B represents exactly the solution set Sol. However, the main computational issue is obtaining a BDD B of manageable size. It is therefore useful to find smaller BDDs that do not necessarily represent Sol but still yield the same outputs $Sol_{\Delta j}$. We will say that any BDD B' over which the algorithms from [1] compute the required $Sol_{\Delta j}$ *yields correct cost-based domain analysis*.

4 Cost-Bounded BDDs

To obtain correct cost-based domain analysis from a smaller BDD, a straightforward strategy is to use a BDD that represents only near-optimal solutions.

A BDD that represents all solutions is not necessary, since $Sol_{\Delta j}$ contains projections of near-optimal solutions only. Thus if B represents the solution set of (4), we denote by $B_{cx \leq b}$ the BDD representing the set of near-optimal solutions, where $b = c^* + \Delta_{max}$. $B_{cx \leq b}$ can be built by adding $cx \leq b$ to the constraint set of (4) and constructing a BDD in the usual fashion. We will refer to this exact representation of near-optimal solution set as an *exact BDD*.

Although $Sol(B_{cx \leq b})$ is smaller than $Sol(B)$, $B_{cx \leq b}$ is not necessarily smaller than B. In fact, it can be exponentially larger. For example, consider a 0-1 programming model (4) over $n = p^2$ variables $\{x_{kl} \mid k, l = 1, \ldots, p\}$. Let there be no constraints, i.e., the solution set contains all 0-1 tuples, so that resulting BDD B is a single 1 node (a tautology). Let the objective function of (4) be

$$\sum_{k=1}^{p} \sum_{\ell=1}^{p} c_{k\ell} x_{k\ell}$$

where $c_{k\ell} = 2^{k-1} + 2^{\ell+p-1}$, so that $c^* = 0$. If we let

$$\bar{b} = \tfrac{1}{2} \sum_{k=1}^{p} \sum_{\ell=1}^{p} c_{k\ell} = \tfrac{1}{2} p(2^{2p} - 1)$$

then Theorem 6 from [10] states that a BDD B_1 representing function $cx \geq \bar{b}$ has the width of at least $\Omega(2^{\sqrt{n}/2})$ and is therefore exponentially large. A BDD B_2 representing $cx \leq \bar{b} - 1$ is a negation of B_1, obtained by just swapping terminal nodes 0 and 1, and has the same width $\Omega(2^{\sqrt{n}/2})$. Therefore, if we take $b = \bar{b} - 1$, $B_{cx \leq b}$ is exponentially larger than B.

On the other hand, exact BDD can also be exponentially smaller than B. For example, if the objective function is $\sum_{k\ell} x_{k\ell}$ and the constraint set consists of $\sum_{k\ell} c_{k\ell} x_{k\ell} \leq \bar{b} - 1$, then $c^* = 0$ and B has the width $\Omega(2^{\sqrt{n}/2})$. However, $Sol(B_{cx \leq b})$ contains one solution when $b = c^* = 0$, namely $x = 0$, and $B_{cx \leq b}$ therefore has a linear number of nodes.

5 Sound BDDs

As part of a strategy for overcoming the possible size explosion of an exact BDD, we suggest a family of *sound* BDDs to be used for cost-based domain analysis. A sound BDD for a given Δ_{max} is any BDD B' for which

$$Sol(B') \cap \{x \in \{0,1\}^n \mid cx \leq b\} = Sol(B_{cx \leq b}) \tag{5}$$

where again $b = c^* + \Delta_{max}$. Clearly,

Lemma 1. *Any sound BDD yields correct cost-based domain analysis.*

This is because if B' is sound, the elements of B' with cost at most b are precisely the elements of $Sol(B)$ with cost at most b. Thus when B' is used to compute $Sol_{\Delta j}$, the result is the same as when using $B_{cx \leq b}$ or B. Note that $Sol(B')$ need

not be a subset of $Sol(B)$. We can add or remove from $Sol(B')$ any element with cost greater than b without violating soundness.

A smallest sound BDD is no larger than either B or $B_{cx\leq b}$ (as both are sound themselves), and it may be significantly smaller than both.

We are unaware of a polynomial-time exact algorithm for computing a smallest sound BDD, but we offer a heuristic method that uses two polynomial-time operations, *pruning* and *contraction*, each of which reduces the size of a BDD while preserving soundness.

6 Pruning

Pruning a BDD removes edges that belong only to paths that are longer than the cost bound b. Pruning therefore reduces the size of the BDD without removing any solutions with cost less than or equal to b.

Define a path from the root to 1 to be *admissible* with respect to b if it represents at least one solution x with $cx \leq b$. An edge in the BDD is admissible if it lies in at least one admissible path. *Pruning* is the operation of removing an inadmissible edge. A BDD is *pruned* if it contains no inadmissible edges.

Pruning clearly preserves soundness, but some pruned and sound BDDs for the same constraint set may be smaller than others. Consider BDDs in Fig. 3 defined over two variables x_1 and x_2 and the objective function $x_1 + x_2$. Then if $b = 1$, both BDDs in Fig. 3 are pruned and sound.

Fig. 4 displays an algorithm that generates a pruned BDD, given a starting BDD B and a cost bound $cx \leq b$ as input. In the algorithm, $L[j]$ is the set of nodes of B with label x_j. Pruning starts with the last layer of nodes $L[n-1]$ and proceeds to first layer $L[0]$. Each round of pruning creates a new B, which is stored as B_{old} before the next round starts. For a given node u, $l(u)$ and $h(u)$ are its low and high children, respectively, in B_{old}. If node u has label x_j, the algorithm checks whether the low edge $(u, l(u))$ is too expensive by checking whether the shortest path from the root to 1 through that edge is longer than b. If the edge $(u, l(u))$ is too expensive, it is deleted by redirecting it to a terminal node 0, i.e., by replacing it with $(u, 0)$. A similar test is performed for the high edge.

The algorithm checks whether edge $(u, l(u))$ is too expensive by checking whether $U[u] + c^0[u, l(u)] + D[l(u)] > b$, where $c^0[u, l(u)]$ is the length of the edge, $U(u)$ is the length of the shortest path from u up to the root in B_{old}, and $D(l(u))$ is the length of the shortest path from $l(u)$ down to 1 in B_{old}. By convention, $D(0) = \infty$. Replacement of $(u, l(u))$ with $(u, 0)$ is implemented by calls to a standard BDD node creation function that ensures that the resulting BDD is reduced [1].

When $\Delta_{\max} = 0$, pruning retains only edges that belong to a shortest path. If all edges in a BDD belong to a shortest path from the root to 1, then all paths from the root to 1 are shortest paths, due to the Lemma 2. As a consequence, the number of all paths in a pruned BDD is bounded by the number of optimal solutions, as every path represents at least one optimal solution.

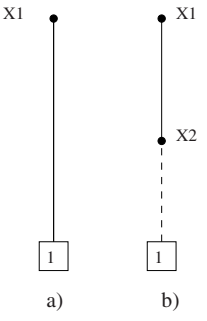


Fig. 3. Two pruned and sound BDDs over variables x_1, x_2. If $b = 1$, each edge in (a) and (b) is part of an admissible path representing the solution $(x_1, x_2) = (1, 0)$.

Function **prune**(B, c, b)
  $B_{old} \leftarrow 0$
  While $B \neq B_{old}$
    $B_{old} \leftarrow B$, update $D[\cdot], U[\cdot]$
    For $j = n - 1$ to 0:
      For $u \in L[j]$:
        If $U[u] + c^0[u, l(u)] + D[l(u)] > b$ then
          replace $(u, l(u), h(u))$ with $(u, 0, h(u))$ in B
        Else if $U[u] + c^1[u, h(u)] + D[h(u)] > b$ then
          replace $(u, l(u), h(u))$ with $(u, l(u), 0)$ in B
        $D[u] \leftarrow min\{c^0[u, l(u)] + D[l(u)], c^1[u, h(u)] + D[h(u)]\}$
  Return B.

Fig. 4. Algorithm for pruning a BDD B with respect to $cx \leq b$

Lemma 2. *If every edge in a directed acyclic graph G belongs to a shortest source-terminus (s-t) path, then every s-t path in G is shortest.*

Proof. Suppose to the contrary there is an s-t path P in G that is not shortest. Let P' be the shortest subpath of P that is part of no shortest s-t path. Then P' contains at least two edges, which means that P' can be broken into two subpaths A and B, and we write $P' = A + B$ (Fig. 5). Since P' is minimal, A is part of a shortest s-t path $A_s + A + A_t$, and B is part of a shortest s-t path $B_s + B + B_t$. But

$$B_s < A_s + A \tag{6}$$

(where $<$ means "is shorter than"), since otherwise $A + B$ is part of a shortest s-t path $A_s + A + B + B_t$. But (6) implies

$$B_s + A_t < A_s + A + A_t$$

which contradicts the fact that $A_s + A + A_t$ is a shortest path.

Even more, when $\Delta_{\max} = 0$ we can efficiently construct exact BDD $B_{cx \leq b}$ from the pruned BDD B_Δ. Whenever an edge (u_1, u_2) skips a variable x_i, and if a

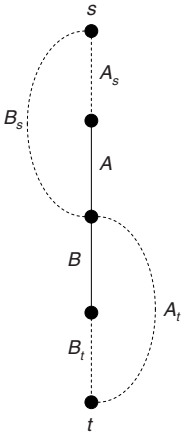


Fig. 5. Illustration of Lemma 2

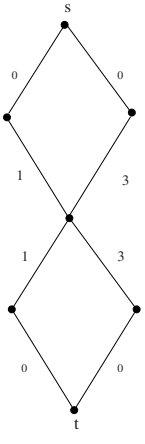


Fig. 6. Example for pruning when $\Delta_{max} > 0$. Even though every edge belongs to a path of length at most 4, there are paths of length 6.

cost of $c_i(0)$ is cheaper than $c_i(1)$, we forbid assignment $x_i = 1$ by inserting a node u_i that breaks the edge (u_1, u_2) into (u_1, u_i) and (u_i, u_2) in such a way that the low child of u is u_2, $l(u_i) = u_2$, while the high child is 0, $h(u_i) = 0$. This operation increases the number of nodes by 1. Hence, it suffices for every skipping edge and every skipped variable covered by that edge to insert a node to get a BDD B'_Δ that represents exactly the set of optimal solutions. The exact BDD $B_{cx \leq b}$ is obtained by reducing B'_Δ.

It follows that exact BDD cannot be exponentially bigger than pruned BDD when $\Delta_{\max} = 0$. A simple overestimation gives us a bound on the number of nodes: $|B_{cx \leq b}| \leq |B_\Delta| + |E| \cdot n$, where E is the set of edges in B_Δ. Namely, for each edge in E we can insert at most n nodes.

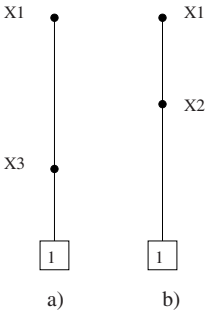


Fig. 7. Two contracted and sound BDDs over variables x_1, x_2, x_3 and objective function $x_1 + x_2 + x_3$. If $b = 1$, the contraction of a node introduces a solution $(x_1, x_2, x_3) = (1, 0, 0)$ with cost 1.

When $\Delta_{\max} > 0$, even when all inadmissible edges are removed, some paths of length less than or equal to b may remain. This is because even if every edge of a graph belongs to an s-t path of length at most b, the graph may yet contain an s-t path longer than b. Consider for example the graph of Fig. 6. Every edge belongs to an s-t path with length at most 4, but there is nonetheless an s-t path of length 6.

7 Contraction

A node u in a BDD B is *contractible* with respect to parent node u_p, child node u_c, and cost restriction $cx \leq b$ if replacing edges (u_p, u) and (u, u_c) with an edge (u_p, u_c) introduces no solutions with cost less than or equal to b. That is, if B^c is the BDD resulting from the replacement, $cx > b$ for all $x \in Sol(B^c) \setminus Sol(B)$. To *contract* node u is to replace (u_p, u) and (u, u_c) with (u_p, u_c). A BDD is *contracted* if it contains no contractible nodes.

Contraction obviously preserves soundness, as it adds only solutions with cost greater than b. Yet as with pruning, there are more than one sound and contracted BDD for the same constraint set. Consider BDDs in Fig. 7 defined over variables x_1, x_2, and x_3, and objective function $x_1 + x_2 + x_3$. If $b = 1$, both BDDs are sound and contracted.

An algorithm that implements contraction is presented in Fig. 8. The algorithm repeatedly contracts nodes u with only one non-zero child, until no contracting is possible.

Lemma 3. *The algorithm of Fig. 8 removes only contractible nodes.*

Proof. Suppose that the algorithm contracts node $h(u)$ by replacing $(u, h(u))$ with $(u, h(h(u))$. It suffices to show that the replacement adds no solutions with cost less than or equal to b. (The argument is similar for the other three cases.) Suppose that u has label x_k, $h(u)$ has label x_ℓ, and $h(h(u))$ has label x_m. Any

Function **contract**(B, c, b)
  $B_{old} \leftarrow 0$
  While $B \neq B_{old}$
    $B_{old} \leftarrow B$, update $D[\cdot], U[\cdot]$
    For $j = n-1$ to 0
      For all $u \in L[j]$
        If $l(h(u)) = 0$ then
          If $U[u] + c^1[u, h(u)] + c^0[h(u), h(h(u))] + D[h(h(u)] > b$ then
            Replace $(u, h(u)), (h(u), h(h(u)))$ with $(u, h(h(u)))$ in B
        If $h(h(u)) = 0$ then
          If $U[u] + c^1[u, h(u)] + c^1[h(u), l(h(u))] + D[l(h(u)] > b$ then
            Replace $(u, h(u)), (h(u), l(h(u)))$ with $(u, l(h(u)))$ in B
        If $l(l(u)) = 0$ then
          If $U[u] + c^0[u, l(u)] + c^0[l(u), h(h(u))] + D[h(h(u)] > b$ then
            Replace $(u, l(u)), (l(u), h(l(u)))$ with $(u, h(l(u)))$ in B
        If $h(l(u)) = 0$ then
          If $U[u] + c^0[u, l(u)] + c^1[h(u), l(l(u))] + D[l(l(u)] > b$ then
            Replace $(u, l(u)), (l(u), l(l(u)))$ with $(u, l(l(u)))$ in B
  Return B

Fig. 8. Algorithm for contracting a BDD B with respect to cost bound $cx \leq b$

Function **compile**(b)
  $B_\Delta \leftarrow 1$
  for $i = 1$ to m
    $B_i \leftarrow \text{BDD}(g_i)$
    $B_\Delta \leftarrow B_\Delta \wedge B_i$
    if $(|B_\Delta| > T$ or $i = m)$
      $B_\Delta \leftarrow$ **prune**(B_Δ, c, b)
      $B_\Delta \leftarrow$ **contract**(B_Δ, c, b)
  return B_Δ

Fig. 9. A simple compilation scheme to obtain a sound BDD, for problem (4) and cost bound b, that is pruned and contracted

path through the new edge $(u, h(h(u)))$ represents a solution that is also represented by a path through the original edges $(u, h(u))$ and $(h(u), h(h(u)))$, unless $x_\ell = 0$. So we need only check that any solution with $x_\ell = 0$ represented by a path through $(u, h(h(u)))$ has cost greater than b. But the cost of any such solution is at least

$$\begin{aligned} &U[u] + c_k(1) + c^*_{k+1,\ell-1} + c_\ell(0) + c^*_{\ell+1,m-1} + D[h(h(u))] \\ &= U[u] + c^1[u, h(u)] + c^0[h(u), h(h(u))] + D[h(h(u))] \end{aligned}$$

The algorithm ensures that node $h(u)$ is not contracted unless this quantity is greater than b.

Both pruning and contraction algorithm have worst-case running time that is quadratic in the size of the underlying BDD. Both algorithms explore all the BDD nodes in each internal iteration. There is at most linear number of these iterations, since each time at least one edge is removed.

Fig. 9 presents simple algorithm for compiling a pruned and contracted BDD for (4). The variable ordering is fixed to $x_1 < \ldots < x_n$, and BDD(g_i) denotes an atomic compilation step that generates a BDD from the syntactical definition of $g_i(x) \geq b_i$. Pruning and contraction are applied not only to the final BDD, but to intermediate BDDs when their size exceeds a threshold T.

8 Computational Results

We carried out a number of experiments to analyze the effect of cost bounding on the size of BDDs. In particular, we wish to test the benefits of replacing an exact cost-bounded BDD with a sound BDD obtained by pruning and contraction.

Table 1. Experimental results for 0-1 linear programs. Each instance has n variables and m constraints. The coefficients a_j of 0-1 inequalities $\sum_{j=1}^{n} a_j x_j \geq b$ are drawn uniformly from $[0, r]$, and b is chosen such that $b = \alpha \cdot \sum_{j=1}^{n} a_j$ where α indicates the tightness of the constraints. The optimal value is c^*, and the largest feasible objective function value is $c_{\max}$. The size of the original BDD B is not shown when it is too large to compute.

$n = 20$, $m = 5$
$r = 50$, $\alpha = 0.3$
$c^* = 101$, $c_{\max} = 588$

Δ	$\lvert B_\Delta \rvert$	$\lvert B_{cx \leq b} \rvert$	$\lvert B \rvert$
0	5	20	8566
40	524	742	8566
80	3456	4328	8566
120	7037	11217	8566
200	8563	16285	8566
240	8566	13557	8566

$n = 30$, $m = 6$
$r = 60$, $\alpha = 0.3$
$c^* = 36$, $c_{\max} = 812$

Δ	$\lvert B_\Delta \rvert$	$\lvert B_{cx \leq b} \rvert$	$\lvert B \rvert$
0	10	30	925610
50	2006	3428	925610
150	262364	226683	925610
200	568863	674285	925610
250	808425	1295465	925610
200	905602	1755378	925610

$n = 40$, $m = 8$
$r = 80$, $\alpha = 0.3$
$c^* = 110$, $c_{\max} = 1241$

Δ	$\lvert B_\Delta \rvert$	$\lvert B_{cx \leq b} \rvert$
0	12	40
15	402	1143
35	1160	3003
70	7327	11040
100	223008	404713
140	52123	

$n = 50$, $m = 5$
$r = 100$, $\alpha = 0.1$
$c^* = 83$, $c_{\max} = 2531$

Δ	$\lvert B_\Delta \rvert$	$\lvert B_{cx \leq b} \rvert$	$\lvert B \rvert$
0	12	83	4891332
100	103623	163835	4891332
200	1595641	2383624	4891332

$n = 60$, $m = 10$
$r = 100$, $\alpha = 0.1$
$c^* = 67$, $c_{\max} = 3179$

Δ	$\lvert B_\Delta \rvert$	$\lvert B_{cx \leq b} \rvert$
0	7	60
50	1814	5519
100	78023	111401

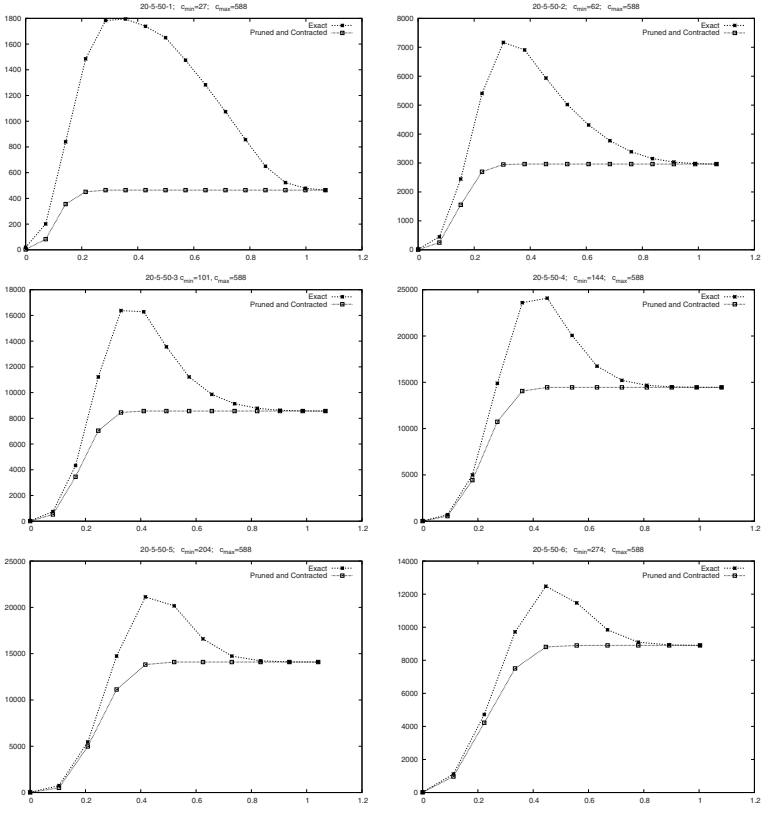


Fig. 10. Illustration of how the size of exact BDDs $B_{cx\leq b}$ (upper curve) and pruned and contracted BDDs B_Δ (lower curve) in 0-1 linear instances depends on Δ. Here there are 20 variables, 5 constraints, and $r = 50$. The horizontal axis indicates Δ as a fraction of $c_{\max} - c^*$. The tightness parameter α takes the values $0.1, 0.2, \ldots, 0.6$ in the six plots. The difference between $|B_{cx\leq b}|$ and $|B_\Delta|$ is larger when the constraints are more relaxed (α is small).

We performed experiments over randomly generated 0-1 linear programs with multiple inequalities and a few 0-1 instances from the MIPLIB library.[1] All the experiments were carried out on a Pentium-III 1 GHz machine with 2GB of memory, running Linux. To load instances we used customized version of a BDD-based configuration library CLab [11], running over BDD package BuDDy [12].

For each instance we compared three BDDs: the BDD B representing the original constraint set, the exact bounded BDD $B_{cx\leq b}$, and the sound BDD B_Δ that results from pruning and contracting B. We show results for several values of Δ. The optimal value c^* and the largest feasible value $c_{\max}$ of the objective function are shown for comparison.

[1] Available at http://miplib.zib.de/miplib2003.php

Table 2. Experimental results for 0-1 MIPLIB instances with $\Delta = 0$

instance	$\lvert B_\Delta \rvert$	$\lvert B_{cx \le b} \rvert$	$\lvert B \rvert$
lseu	19	99	-
p0033	21	41	375
p0201	84	737	310420
stein27	4882	6260	25202
stein45	1176	1765	5102257

The results show that both $B_{cx \le b}$ and B_Δ are substantially smaller than B for rather large cost tolerances Δ. Also B_Δ is almost always significantly smaller than $B_{cx \le b}$. For example, in an instance with 30 variables and six constraints, one can explore all solutions within a range of 50 of the optimal value 36 by constructing a sound BDD that is only a tiny fraction of the size the full BDD (2006 nodes versus 925,610 nodes). In problems with 40 and 60 variables, the full BDD is too large to compute, while sound BDDs are of easily manageable size for a wide range of objective functions values.

Fig. 10 illustrates how $|B_{cx \le b}|$ and $|B_\Delta|$ compare over the full range of objective function values. Note that $|B_{cx \le b}|$ is actually larger than $|B|$ for larger values of Δ, even though $B_{cx \le b}$ represents fewer solutions than B. The important fact for postoptimality analysis, however, is that $|B_{cx \le b}|$, and especially $|B_\Delta|$, are much smaller than $|B|$ for small Δ.

Table 2 shows the results for a few 0-1 problems in MIPLIB for fixed $\Delta = 0$. A BDD for $\Delta = 0$ is useful for identifying all optimal solutions. We observe significant savings in space for both B_Δ and $B_{cx \le b}$ in comparison to B. For instance *lseu*, we were not able to generate B, while both B_Δ and $B_{cx \le b}$ are quite small.

The response times of algorithms for calculating valid domains and postoptimality analysis depend linearly on the size of an underlying BDD. In our experience, response time over a BDD having 10 000 nodes is within tens of milliseconds. A response time of up to one second corresponds to a BDD with about 250 000 nodes. In terms of memory consumption, one BDD node takes 20 bytes of memory, hence 50 000 nodes take 1 MB.

9 Conclusion

We conclude that cost-bounded BDDs can yield significant computational advantages for cost-based domain analysis of 0-1 linear programming problems of moderate size. Sound BDDs obtained by pruning and contraction produce more significant savings. There is evidence that problems for which the original BDD is intractable may often be easily analyzed using sound BDDs.

If a valid bound on the optimal value is available, cost-bounded BDDs could be a competitive method for solution as well as postoptimality analysis, particularly when the problem is nonlinear. This is a topic for future research.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable suggestions.

References

1. Hadzic, T., Hooker, J.: Postoptimality analysis for integer programming using binary decision diagrams. Technical report, Carnegie Mellon University (2006) Presented at GICOLAG workshop (Global Optimization: Integrating Convexity, Optimization, Logic Programming, and Computational Algebraic Geometry), Vienna.
2. Andersen, H.R.: An introduction to binary decision diagrams. Lecture notes, available online, IT University of Copenhagen (1997)
3. Barth, P.: Logic-based 0-1 Constraint Solving in Constraint Logic Programming. Kluwer, Dordrecht (1995)
4. Akers, S.B.: Binary decision diagrams. IEEE Transactions on Computers **C-27** (1978) 509–516
5. Lee, C.Y.: Representation of switching circuits by binary-decision programs. Bell Systems Technical Journal **38** (1959) 985?–999
6. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **C-35** (1986) 677–?691
7. Bryant, R.E.: Symbolic boolean manipulation with ordered binary decision diagrams. ACM Computing Surveys **24** (1992) 293– 318
8. Becker, Behle, Eisenbrand, Wimmer: BDDs in a branch and cut framework. In: International Workshop on Experimental and Efficient Algorithms (WEA), LNCS. Volume 4. (2005)
9. Hadzic, T., Andersen, H.R.: A BDD-based Polytime Algorithm for Cost-Bounded Interactive Configuration. In: AAAI-Press. (2006)
10. Hosaka, K., Takenaga, Y., Kaneda, T., Yajima, S.: Size of ordered binary decision diagrams representing threshold functions. Theoretical Computer Science **180** (1997) 47–60
11. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration. `http://www.itu.dk/people/rmj/clab/` (2007)
12. Lind-Nielsen, J.: BuDDy - A Binary Decision Diagram Package. `http://sourceforge.net/projects/buddy` (online)

YIELDS: A Yet Improved Limited Discrepancy Search for CSPs*

Wafa Karoui[1,2], Marie-José Huguet[1], Pierre Lopez[1], and Wady Naanaa[3]

[1] Univ. de Toulouse, LAAS-CNRS, 7 avenue du colonel Roche, 31077 Toulouse, France
[2] Unité ROI, Ecole Polytechnique de Tunisie, La Marsa, Tunisie
[3] Faculté des Sciences de Monastir, Boulevard de l'environnement, Tunisie
{wakaroui,huguet,lopez}@laas.fr, naanaa.wady@planet.tn

Abstract. In this paper, we introduce a Yet ImprovEd Limited Discrepancy Search (YIELDS), a complete algorithm for solving Constraint Satisfaction Problems. As indicated in its name, YIELDS is an improved version of Limited Discrepancy Search (LDS). It integrates constraint propagation and variable order learning. The learning scheme, which is the main contribution of this paper, takes benefit from failures encountered during search in order to enhance the efficiency of variable ordering heuristic. As a result, we obtain a search which needs less discrepancies than LDS to find a solution or to state a problem is intractable. This method is then less redundant than LDS.

The efficiency of YIELDS is experimentally validated, comparing it with several solving algorithms: Depth-bounded Discrepancy Search, Forward Checking, and Maintaining Arc-Consistency. Experiments carried out on randomly generated binary CSPs and real problems clearly indicate that YIELDS often outperforms the algorithms with which it is compared, especially for tractable problems.

1 Introduction and Motivations

Constraint Satisfaction Problems (CSPs) provide a general framework for modeling and solving numerous combinatorial problems. Basically, a CSP consists of a set of variables, each of which can take a value chosen among a set of potential values called its domain. The constraints express restrictions on which combinations of values are allowed. The problem is to find an assignment of values to variables, from their respective domains, such that all the constraints are satisfied [4, 19].

CSPs are known to be NP-complete problems. Nevertheless, since CSPs crop up in various domains, many search algorithms for solving them have been developed. In this paper, we are interested in complete methods which have the advantage of finding at least a solution to a problem if such a solution exists. A widely studied class of complete algorithms relies to depth first search (DFS).

* To be published in the proceedings of The Fourth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'07), Brussels, Belgium.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 99–111, 2007.

Forward Checking (FC) [9] and *Maintaining Arc-Consistency* (MAC) [17] are two sophisticated algorithms belonging to the DFS class. Each of them enforces during search a kind of local consistency to prune the search tree and therefore to fasten problem solving. Another algorithm belonging to the DFS class is *Limited Discrepancy Search* (LDS) [10]. Since value ordering heuristics cannot avoid bad instantiations (*i.e.,* choosing, for a given variable, a value that does not participate in any solution), LDS tackles this problem by gradually increasing the number of less-preferred options from the heuristic view-point (*discrepancies*) [13] [15] [20].

Lots of recent research works try to improve this type of methods. One important area is concerned with learning from failures [2] [8]. In this context and in order to design a more efficient search method we proposed a subtantial improvement of LDS. We have then defined *YIELDS* (Yet Improved Limited Discrepancy Search) [12]. YIELDS integrates constraint propagation as well as a variable order learning scheme in order to reduce the size of the search tree. The goal is to minimize the number of discrepancies needed to obtain a solution or to state a problem is intractable. More precisely, this paper is dedicated to the refinement of the learning mechanism of YIELDS and its evaluation on various types of instances including randomly generated data and real problems.

The paper is organized as follows. The next section specifies the main concepts and reviews the existing methods to solve constraint satisfaction problems, especially those following Depth-First Search. Section 3 is the core of the paper: it describes how to better explore the search space using Yet Improved Discrepancy Search. Experimental experience is reported in Section 4. Section 5 describes related works and Section 6 concludes the paper.

2 Background

Constraint Satisfaction Problem. A CSP is defined by a tuple (X, D, C) where: $X = \{X_1, \ldots, X_n\}$ is a finite set of variables; $D = \{D_1, \ldots, D_n\}$ is the set of domains for each variable, each D_i being the set of discrete values for variable X_i; $C = \{C_1, \ldots, C_m\}$ is a set of constraints.

An instantiation of a subset of variables corresponds to an assignment of these variables by a value from their domain. An instantiation is said to be complete when it concerns all the variables from X. Otherwise, it is called a partial instantiation. A solution is a complete instantiation satisfying the constraints. An inconsistency in the problem is raised as soon as a partial instantiation cannot be extended to a complete one.

Tree Search Methods. Methods based on Depth-First Search are generally used to quickly obtain a solution of a CSP. One has to successively instantiate variables which leads to the development of a search tree in which the root corresponds to uninstantiated variables and leaves to solutions. DFS methods are based on several key-concepts:

- At each node of the tree, one has to determine how to choose a variable to be instantiated and how to choose its value; this can be processed by variable and value ordering heuristics, respectively.
- After each instantiation, one has to define what level of constraint propagation can be used to prune the tree.
- Moreover, when a dead-end occurs (*i.e.,* an inconsistency) one has to specify how to backtrack in the tree to develop another branch or to restart in order to continue the search.

This type of methods is usually stopped either as soon as a solution is obtained or when the complete tree has been explored without finding any solution. In the worst case, it needs an exponential time in the number of variables.

Chronological Backtracking (CB) is a well-known method based on DFS for solving CSPs. CB extends a partial instantiation by assigning to a new variable a value which is consistent with the previous instantiated variables (look-back scheme). When an inconsistency appears it goes back to the latest instantiated variable trying another value.

Limited Discrepancy Search (LDS) is based on DFS for variable instantiation and on the concept of discrepancy to expand the search tree. A discrepancy is realized when the search assigns to a variable a value which is not ranked as the best by the value ordering heuristic. The LDS method is based on the idea of gradually increasing the number of allowed discrepancies while restarting:

- It begins by exploring the path obtained by following the values advocated by the value ordering heuristic: this path corresponds to zero discrepancy.
- If this path does not lead to a solution, the search explores the paths that involve a single discrepancy.
- The method iterates increasing the number of allowed discrepancies.

For binary trees, counting discrepancies is a quite simple task: exploring the branch associated with the best boolean value, according to a value ordering heuristic, involves no discrepancy, while exploring the remaining branch implies a single discrepancy. For non binary trees, the values are ranked according to a value ordering heuristic such that the best value has rank 1; exploring the branch associated to value of rank $k \geq 1$ leads to make $k - 1$ discrepancies.

LDS can be stopped either as soon as a first solution is found or when the complete tree is expanded using the maximum number of allowed discrepancies. Note that an improvement of LDS is Depth-bounded Discrepancy Search (DDS) which first favours discrepancies at the top of the search tree (*i.e.,* on the most important variables).

Ordering heuristics. They aim to reduce the search space to find a solution. Depending on their type they provide an order for the selection of the next variable to consider or the next value for a variable instantiation. These heuristics can be static (*i.e.,* the orders are defined at the beginning of the search) or dynamic (*i.e.,* the orders may change during search). The efficiency of DFS methods such as CB or LDS clearly depends on the chosen ordering heuristic.

For CSPs, several common variable and value ordering heuristics are defined such as dom (*i.e., min-domain*) or dom/deg for variable ordering, and min-conflict for value ordering (see [2] and [8]).

Propagations. To limit the search space and then to speed up the search process, constraint propagation mechanisms can be joined to each variable instantiation. The goal of these propagations is to filter the domain of some not yet instantiated variables. Various levels of constraint propagation can be considered in a tree search method. The most common are:

- *Forward Checking* (FC) which suppresses inconsistent values in the domain of not yet instantiated variables linked to the instantiated one by a constraint.
- *Arc-Consistency* (AC) which corresponds to suppress inconsistent values in the domain of all uninstantiated variables.

These constraint propagation mechanisms can be added both in Chronological Backtracking and in Discrepancy Search. In the rest of the paper, CB-FC refers to the CB method including FC propagation while CB-AC includes AC propagations (CB-AC corresponds to MAC algorithm [17]).

3 The Proposed Approach

3.1 Overcoming the Limits of LDS

The objective of our approach is to minimize the number of discrepancies needed to reach a solution or to declare that the problem is intractable. To do that, we propose to use the dead ends encountered during a step of the LDS method to order the problem variables for the following steps. In fact in LDS, only the heuristic on the order of variables selects the path in the search space (see Algorithms 1 and 2). This means that when we increment the number of discrepancies and reiterate LDS, we have frequently the same initial variable to instantiate. If we assume that this variable is the failure reason and that it eliminates values required for the solution, it is useless to develop again its branch.

Algorithm 1. LDS(X, D, C, k-max, Sol)

```
1: k ← 0
2: Sol ← NIL
3: while (Sol = NIL) and (k ≤ k_max) do
4:     Sol ← LDS_iteration(X, D, C, k, Sol)
5:     k ← k+1
6: end while
7: return Sol
```

To avoid this kind of situations, we associate a weight, initially equal to zero, to each variable. This weight is incremented every time this variable fails because of the limit on the number of allowed discrepancies: we cannot diverge on this

Algorithm 2. LDS_iteration(X, D, C, k, Sol)

```
1: if X = ∅ then
2:    return Sol
3: else
4:    x_i ← BestVariable(X) // variable instantiation heuristic
5:    v_i ← BestValue(D_i, k) // value instantiation heuristic
6:    if v_i ≠ NIL then
7:       D' ← Update(X\{x_i}, D, C, (x_i, v_i)) // constraint propagation
8:       I ← LDS_iteration(X\{x_i}, D', C, k, Sol ⋃ {(x_i, v_i)})
9:       if I ≠ NIL then
10:         return I
11:      else
12:         if k > 0 then
13:            D_i ← D_i\{v_i}
14:            return LDS_iteration(X, D, C, k-1, Sol) // can diverge
15:         end if
16:      end if
17:   end if
18:   return NIL
19: end if
```

variable despite its domain of values is not empty. In the following iterations, this variable will be privileged and will be placed higher in the branch developed by the LDS method. Like this, we will avoid the situation of inconsistency caused by this variable. Therefore, the choice of variable to be instantiated is based, first, on the usual heuristic (dom for example), second, on the variable weight, finally, if tied variables still remain, the order of indexation can be considered.

Thus, by introducing the notion of weight, our purpose is to correct the heuristic for variable instantiation guiding it to variables concretely constrained. These variables greatly influence the solution search. Therefore, we correct mistakes of the heuristic by adding the weight notion which can be considered as a type of dynamic learning. Like this, we can exploit previous failures and take useful information for the following steps. To speed up the process, difficult and intractable subproblems are pushed up at the top of the search tree.

This improvement of LDS method besides its effects on the variable ordering, stops the LDS iterations when an inconsistency is found. In fact, if an inconsistency arises with k allowed discrepancies, other iterations, from k+1 to the maximum number of discrepancies, are unnecessary since they will discover again the same inconsistency.

For LDS, when we authorize a fixed number of discrepancies we can consume, completely or not, these authorized discrepancies. If the totality of discrepancies is consumed and no solution is found, a new iteration of LDS is launched incrementing the number of allowed discrepancies. In contrast, if the allowed discrepancies are not consumed, it is not necessary to continue to reiterate LDS with a greater number of discrepancies even if no solution has been found. In such situation, one can be sure that the problem is intractable (all the feasible values of each variable have been tried without using the number of allowed discrepancies).

3.2 The YIELDS Algorithm

The YIELDS method is based on a learning from failures technique. This learning produces a new way to go all over the search space contributing to speed up the resolution. Moreover, the propagation mechanisms lead us to stop the search before we reach the maximum number of discrepancies in the case of intractable problems, without missing solution if it does exist.

The completeness of YIELDS can be proved: if the problem has a solution, YIELDS does find it. In fact, when the problem is tractable, the learning technique has produced a permutation on the order of variables. The iteration of YIELDS which has discovered the solution, called YIELDS(k), is based on a variable ordering O, learnt during the previous iterations. We can say that YIELDS(k) is equivalent to CB-FC directly associated with the variable ordering O.

When the problem is intractable, YIELDS stops the search with anticipation. The last iteration does not consume all allowed discrepancies: it can be compared to a call to CB-FC because the bound on discrepancies was not at the origin of the break. Like this, the method is complete (see Algorithms 3 and 4).

Algorithm 3. YIELDS(X, D, C, k_max, Sol)

```
 1: k ← 0
 2: Sol ← NIL
 3: Exceed ← False
 4: while (Sol = NIL) and (k ≤ k_max) do
 5:    Sol ← YIELDS_iteration(X, D, C, k, Sol)
 6:    k ← k+1
 7:    if !Exceed then
 8:       exit
 9:    end if
10: end while
11: return Sol
```

The principle of YIELDS is exactly the same as LDS: it considers, initially, branches of the tree which cumulate the smallest number of discrepancies. The first difference is that a weight (initially the same for all variables) is associated to each variable and every time a variable fails because of the limit on discrepancies, its weight is incremented to guide next choices of the heuristic. The second difference is that the number of discrepancies is not blindly incremented until the maximum of discrepancies allowed by the search tree is reached (as it is done in LDS). Thus, the new method consumes less discrepancies than LDS or even DDS.

Definition 1. *Let $P = (X, D, C)$ be a binary CSP of n variables and w_i a weight associated to each variable x_i. The weight vector W of P is the vector composed of weights of all variables of the problem:*

$$W(P) = [w_0, w_1, ..., w_{n-1}]$$

Algorithm 4. YIELDS_iteration(X, D, C, k, Sol)

```
1: if X = ∅ then
2:    return Sol
3: else
4:    x_i ← First_VariableOrdering(X,Weight)
5:    v_i ← First_ValueOrdering(D_i, k)
6:    if v_i ≠ NIL then
7:       D' ← Update(X\{x_i}, D, C, (x_i, v_i))
8:       I ← YIELDS_iteration(X\{x_i}, D', C, k, Sol ⋃ {(x_i, v_i)})
9:       if I ≠ NIL then
10:         return I
11:      else
12:         if k > 0 then
13:            D_i ← D_i \{v_i}
14:            return YIELDS_iteration(X, D, C, k-1, Sol)
15:         else
16:            Weight[x_i] ← Weight[x_i]+1
17:            Exceed ← True // impossible to diverge
18:         end if
19:      end if
20:   end if
21:   return NIL
22: end if
```

Definition 2. *Let $W1$ and $W2$ be two weight vectors of P, a binary CSP. The variation of weights of P is given by ΔW the vector difference between $W1$ and $W2$:*

$$\Delta W(P) = W2(P) - W1(P)$$

Proposition 1. *Let assume that variable weights are initially equal and that they are incremented every time that we do not found a variable value which respects the limit on the number of authorized discrepancies. Let consider two successive iterations of YIELDS for the resolution of P a binary CSP. If the variation of weights $\Delta W(P)$ between these iterations is equal to the null vector, then we can be sure that:*

1. *The process of learning comes to end.*
2. *P is an intractable problem.*

Proof: Since $\Delta W(P) = 0$ the last iteration was not interrupted because of the limit on the number of authorized discrepancies (see Algorithm 2). In addition, it is obvious that an iteration of YIELDS without the limit on the number of discrepancies corresponds to CB-FC which is a complete method. Therefore, if the last iteration corresponds to a complete method and that no solution has been found yet, the problem is intractable. □

3.3 Illustrative Examples

As an example for an intractable problem, let consider a CSP composed of three variables x_0, x_1, x_2 and four values $0, 1, 2, 3$ presented by its incompatibility diagram (see Figure 1).

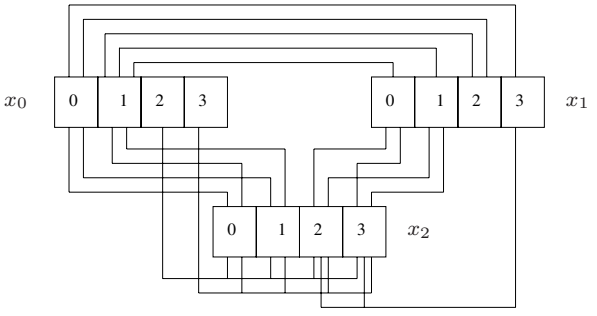


Fig. 1. Incompatibility diagram

The variable ordering initially follows the ascending order, then it is based first on min-domain order (dom), then on min-domain plus weights order (see Table 1), while min-conflict heuristic is applied for value ordering. In this example, the weights do not influence the variables order which is always the same (ascending order). Reminding the way retained for counting discrepancies (see Section 2), the maximum number of discrepancies is here equal to 9.

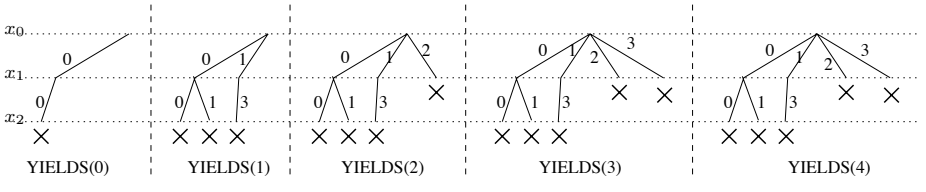


Fig. 2. Illustration of YIELDS on an intractable problem

Table 1. Variable weights for the intractable problem

Weight	Initial	YIELDS(0)	YIELDS(1)	YIELDS(2)	YIELDS(3)	YIELDS(4)
$W[x_0]$	0	1	2	3	4	4
$W[x_1]$	0	1	3	3	3	3
$W[x_2]$	0	0	0	0	0	0

From iterations YIELDS(0) till YIELDS(4), YIELDS develops the same search trees as LDS (see Figure 2). In YIELDS(4), we can see that even if we authorize 4 discrepancies, only 3 are used. The iterations of the YIELDS method are interrupted but not because of the limit on discrepancies. In such a context, YIELDS stops the search. LDS would continue iterations until LDS(9) and would repeat exactly the same search tree.

As an example for a tractable problem, let consider the CSP (X, D, C) defined by $X = \{x_0, x_1, x_2\}$, $D = \{D_0, D_1, D_2\}$ where $D_0 = D_1 = D_2 = \{0, 1, 2, 3, 4\}$. The set of contraints C is represented by the following set of incompatible tuples: $\{(x_0, 0), (x_1, 4)\} \cup \{(x_0, 0), (x_2, 4)\} \cup \{(x_0, 1), (x_1, 4)\} \cup \{(x_0, 1), (x_2, 4)\}$

$\cup \{(x_0,2),(x_1,4)\} \cup \{(x_0,2),(x_2,4)\} \cup \{(x_0,3),(x_1,4)\} \cup \{(x_0,3),(x_2,4)\} \cup \{(x_0,4),(x_2,2)\} \cup \{(x_0,4),(x_2,3)\} \cup \{(x_1,0),(x_2,0)\} \cup \{(x_1,0),(x_2,1)\} \cup \{(x_1,0),(x_2,2)\} \cup \{(x_1,0),(x_2,3)\} \cup \{(x_1,1),(x_2,0)\} \cup \{(x_1,1),(x_2,1)\} \cup \{(x_1,1),(x_2,2)\} \cup \{(x_1,1),(x_2,3)\} \cup \{(x_1,2),(x_2,0)\} \cup \{(x_1,2),(x_2,1)\} \cup \{(x_1,2),(x_2,2)\} \cup \{(x_1,2),(x_2,3)\} \cup \{(x_1,3),(x_2,0)\} \cup \{(x_1,3),(x_2,1)\} \cup \{(x_1,3),(x_2,2)\} \cup \{(x_1,3),(x_2,3)\}.$

In this example, we use the same ordering heuristics as previously. Applying CB-FC to solve this CSP, the resulting search tree consists of 24 expanded nodes (EN) (see Figure 3). Applying LDS, we obtain a bigger search tree of 95 EN. If we apply YIELDS, we obtain a search tree of only 13 EN (see Figure 4) due to the increasing of x_1 priority which contributes to speed up the search.

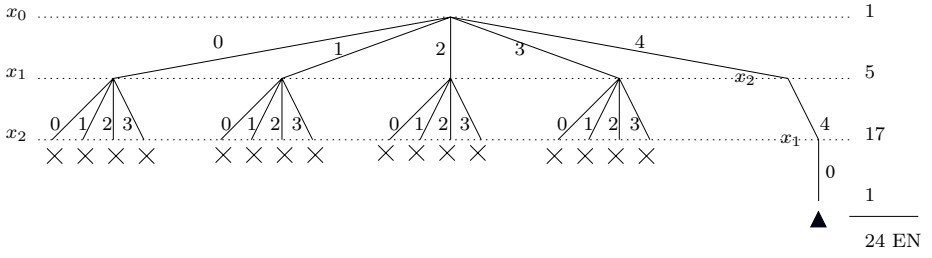


Fig. 3. CB-FC search tree

Table 2. Variable weights for the tractable problem

Weight	Initial	YIELDS(0)	YIELDS(1)	YIELDS(2)
$W[x_0]$	0	1	2	2
$W[x_1]$	0	1	3	3
$W[x_2]$	0	0	0	0

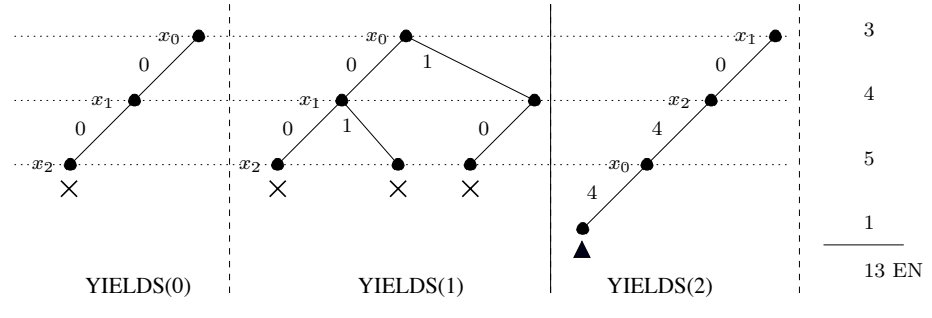


Fig. 4. YIELDS search tree

4 Experimental Results

The problems investigated in our experiments are random binary CSPs, latin squares, and job-shop problems. We compared YIELDS with standard versions of DDS, CB-FC, and CB-AC. The arc-consistency algorithm underlying CB-AC is AC-3.1 [1] [21]. The variable ordering heuristic used by all algorithms is dom. For value ordering, we used the min-conflict heuristic. The evaluation criteria are the number of expanded nodes (NED) and CPU time in seconds. Reported results are expressed as average values. All algorithms were implemented in C++. They were run under Windows XP Professional on a 1.6 GHz PC having 1 Go of RAM.

Random Binary CSPs

For random binary CSPs, we used the generator developed by Frost *et al.* which is available in [3]. Problems are generated according to model B. We experimented on problems involving $n = 30$ variables, a uniform domain size of $d = 25$. The problem density p_1 (*i.e.,* the ratio of the number of constraints in the constraint graph over that of all possible constraints) varied from 0.1 (sparse problems: from line 1 to line 3 in Table 3) to 0.3 (dense problems: from line 4 to the end in Table 3). The constraint tightness p_2 (*i.e.,* the ratio of the number of disallowed tuples over that of all possible tuples) varied so that we obtain instances around the peak of complexity. The size of samples is 100 problem instances for each data point.

Table 3. Random binary CSPs instances

instances	DDS		YIELDS		CB-FC		CB-AC	
<n,d,C,T>	NED	CPU	NED	CPU	NED	CPU	NED	CPU
<30,25,44,531>(35% sat)	10210510	90.61	6273	0.04	1427970	10.2	71	0.08
<30,25,44,526>(48% sat)	21876033	207.8	8526	0.06	1732513	11.92	250	0.19
<30,25,44,518>(73% sat)	1447242	11.72	3543	0.02	178168	1.26	270	0.21
<30,25,131,322>(39% sat)	$\gg$	$\gg$	1342742	12	1898943	16.45	203862	152.66
<30,25,131,320>(56% sat)	$\gg$	$\gg$	1360525	11.76	1374413	11.92	94277	79.92
<30,25,131,318>(74% sat)	$\gg$	$\gg$	1503581	12.39	1577180	13.24	54870	39.9
<30,25,131,322>(sat)	$\gg$	$\gg$	326739	3.07	1101429	9.37	46583	35
<30,25,131,320>(sat)	$\gg$	$\gg$	337996	3.05	827566	6.98	55165	58.46
<30,25,131,318>(sat)	$\gg$	$\gg$	341994	3.12	843548	7.06	16876	11.87

For all considered problems, the results clearly indicated that YIELDS outperforms DDS on sparse and dense problems (in Table 3, "$\gg$" means that execution times are of several hours).

For sparse problems, YIELDS is faster than CB-AC and CB-FC, albeit CB-AC develops less nodes.

For dense problems, YIELDS is also faster than CB-AC and CB-FC. However the advantage is less significant as we move toward dense problems. If we isolate tractable problems (last three lines in Table 3), results become particularly

interesting and YIELDS clearly outperforms other considered methods. For intractable problems, CB-FC remains the better method.

Latin Squares and Job-Shop Scheduling problems
For the job-shop problems, we investigated the Sadeh instances [18]. For tested instances, YIELDS is clearly better than CB-FC and CB-AC (see Table 4).

Table 4. Job-Shop instances

instances	CB-FC		YIELDS		CB-AC	
	NND	CPU	NND	CPU	NND	CPU
enddr1-10-5-1	802233	362	68997	<1	53186	897
enddr1-10-5-10	176015	94	57	<1	113486	457
ewddr2-10-5-1	156388	58	910	<1	92729	480
ewddr2-10-5-10	104032	41	55	<1	64535	372
e0ddr1-10-5-1	1262247624	13030	17133261	113	6262916	1752

We also studied experiments on *Latin Squares* obtained by the generator of [7]. Selected problems have an order of 10. Results showed that YIELDS is always faster than all considered methods (see Table 5).

Table 5. Latin Squares instances

instances	CB-FC		YIELDS		CB-AC	
	NND	CPU	NND	CPU	NND	CPU
qg.1030	7940160	74	158808	16	68276	72.5
qg.1032	26070985	239	128424	71	80215	105
qg.1034	400490	3.91	1775	0.02	181934	212
qg.1036	18976	0.19	364	0.01	13609	17.6
qg.1038	22795	0.21	114	0.01	14393	18

5 Related Works

Many research works try to improve known methods integrating learning from failures. In this context, the following methods were proposed:

1. Squeaky Wheel Optimization (SWO) [11] which is a general optimization approach for local search. In SWO, a greedy constructor produces an initial solution in which difficult elements are identified and guides the construction of a new solution (the process is iterated until some stopping criterion is met). This strategy has not completeness guarantee.
2. Impact-Based Search (IBS) [16] which is also a general search method based on a probing-like integer programming technique. In IBS, the reduction of the search space following a variable instantiation is used to prioritize the variables to consider. This method differs from ours by the used information for learning and by the nature of restarts.

3. F-O-Opt (failure-driven algorithm for Open Hidden-variable Weighted Constraint Optimization Problems) which is one of the algorithm proposed in [5] for open constraint optimization. The context for this search method is dynamic and constraints are updated while searching so used learning technics are local and different.
4. Last Conflict reasoning (LC) [2] which is a learning search method. This method was improved by Grimes and Wallace in [8] including restarts to the original method. Unlike our method, this method learns from constraints and, in Grimes improvement, added restarts are not relied to problem properties. In our method, learning is based on variables and restarts are based on dicrepancies. Gathered information on discrepancies variation may represent an additional information on the considered problem and contribute to accelerate the search.

6 Conclusion and Further Work

In this paper we present a novel method, Yet Improved Discrepancy Search (YIELDS), which takes advantages from failures to guide the search. The goal of this method is to correct the variable ordering heuristic exploiting some fails and detects whether a problem is intractable without doing all the iterations of LDS. We propose an effective YIELDS algorithm and describe how to integrate it into a classical LDS algorithm.

An experimental study carried out on numerous random and real CSPs have shown how it is possible to obtain good results.

In the near future, we plan to set up an association of two learning ways, weights and no-goods which, in our opinion, will constitute a helpful tool for the proposed method. In addition, we think that a careful computational study on other known benchmarks will present an interesting issue to better illustrate the usefulness of YIELDS. Comparisons with some related works are also planned.

References

1. C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings IJCAI-01*, pages 309–315, Seattle, USA, 2001
2. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings ECAI'04*, pages 146–150, Valencia, Spain, 2004
3. D. Frost, C. Bessière, R. Dechter, and J.-C. Régin, Random uniform CSP generators, `http://www.lirmm.fr/~bessiere/generator.html`
4. R. Dechter. *Constraint processing*, Morgan Kaufmann, San Francisco, 2003
5. B. Faltings and S. Macho-Gonzalez, Open constraint satisfaction. In Van Hentenryck, P., ed., *Proceedings of CP'2002, LNCS No. 2470*, Springer, pages 356–370, 2002
6. I.P. Gent and P. Prosser. Inside MAC and FC. APES Research Group Report APES-20-2000, 2000

7. C.P. Gomes. Generator of Quasigroup Completion Problem and related problems, `http://www.cs.cornell.edu/gomes/new-demos.htm`
8. D. Grimes and R. J. Wallace. Learning from failures in constraint satisfaction search. *AAAI Workshop on Learning for Search*, Boston, Massachusetts, USA, 2006
9. R. Haralick and G. Elliot, Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence*, 14:263–313, 1980
10. W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *Proceedings IJCAI-95*, pages 607–613, Montréal, Canada, 1995
11. D. Joslin and D. Clements. Squeaky Wheel Optimisation. In *Proceedings Sixteenth National Conference on Artificial Intelligence-AAAI'98*, pages 340–346, 1998.
12. W. Karoui, M.J. Huguet, P. Lopez et W. Naanaa. Amélioration par apprentissage de la recherche à divergences limitées. In *Proceedings JFPC'05*, pages 109–118, Lens, France, 2005
13. R.E. Korf. Improved limited discrepancy search. In *Proceedings AAAI-96/IAAI-96, Vol. 1*, pages 286–291, Portland, Oregon, USA, 1996
14. C. Likitvivatanavong, Y. Zhang, J. Bowen, and E.C. Freuder. Arc-consistency in MAC: A new perspective. In *Proceedings First International Workshop on Constraint Propagation and Implementation*, Toronto, Canada, 2004
15. N. Prcovic. Quelques variantes de LDS. In *Proceedings JNPC'02*, pages 195–208, Nice, France, 2002
16. P. Refalo. Impact-based search strategies for constraint programming. In Wallace, M., ed., *Principles and Practice of Constraints Programming-CP'04, LNCS No. 3258*, Springer, pages 557–571, 2004
17. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings PPCP-94*, Seattle, USA, 1994
18. N. Sadeh and M.S. Fox. Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. *Artificial Intelligence*, 86:1–41, 1996
19. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press Ltd, London, 1993
20. T. Walsh. Depth-bounded discrepancy search. In *Proceedings IJCAI-97*, pages 1388–1395, Nagoya, Japan, 1997
21. Y. Zhang and R.H.C. Yap. Making AC-3 an optimal algorithm. In *Proceedings IJCAI-01*, pages 316–321, Seattle, USA, 2001

A Global Constraint for Total Weighted Completion Time

András Kovács[1,3] and J. Christopher Beck[2]

[1] Projet Contraintes, INRIA Rocquencourt, France
[2] Dept. of Mechanical and Industrial Engineering, University of Toronto, Canada
[3] Computer and Automation Research Institute, Hungarian Academy of Sciences
akovacs@sztaki.hu, jcb@mie.utoronto.ca

Abstract. We introduce a novel global constraint for the total weighted completion time of activities on a single unary capacity resource. For propagating the constraint, an $O(n^4)$ algorithm is proposed, which makes use of the preemptive mean busy time relaxation of the scheduling problem. The solution to this problem is used to test if an activity can start at each start time in its domain in solutions that respect the upper bound on the cost of the schedule. Empirical results show that the proposed global constraint significantly improves the performance of constraint-based approaches to single-machine scheduling for minimizing the total weighted completion time. Since our eventual goal is to use the global constraint as part of a larger optimization problem, we view this performance as very promising. We also sketch the application of the global constraint to cumulative resources and to problems with multiple machines.

1 Introduction

Many successful applications of constraint programming (CP) to optimization problems exhibit a "maximum type" optimization criteria, characterized by minimizing the maximum value of a set of variables (e.g., makespan, maximum tardiness, or peak resource usage in scheduling). Such criteria exhibit strong back propagation: placing an upper bound on the cost variable results in the pruning of the domain (i.e., the reduction of the maximum value) of the constituent variables. CP has not been as successful for other practically important optimization criteria such as "sum type" objective functions characterized by the minimization of the sum of a set of variables. Examples in the scheduling domain include total weighted completion time, weighted tardiness, weighted earliness and tardiness, and the number of late jobs. Nearly all CP-based approaches to scheduling with these criteria use only the basic sum constraint to propagate the objective function. However, back propagation of the sum constraint is weak because it is often the case that the maximum value of each decision variable is supported by the minimum values of all the other decision variables. The significance of more efficient global constraints for back propagation has been emphasized by Focacci et al. in [10,11].

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 112–126, 2007.

Our purpose is to develop algorithms for propagating "sum type" objective functions in constraint-based scheduling. In this paper, we address the total weighted completion time criterion on a single unary resource. The total weighted completion time criterion has equivalents in a wide range of applications. In container loading problems, it is a frequent requirement that the center of gravity of the loaded container has to be situated as low as possible and, depending on the means of transport, either above the axes of the vehicle or in the center of the container. Along each axis of the coordinate system, the location of the center of gravity of box-shaped goods corresponds to the average weighted completion time of the activities in a schedule. The weight of the activities equals the physical weight of the goods, while their duration corresponds to the length, and their resource requirement to the cross section of the loaded goods. In lot-sizing problems, different items are produced on a single machine, with specific deadlines. The cost of a solution is composed of a holding cost and a setup or ordering cost. The holding cost is computed as the total weighted difference of deadlines and actual production times. Apart from a constant factor, this is equivalent to the weighted distance of the activities from a remote point in time, which corresponds to the weighted completion time in a reversed schedule. In all these applications, the total weighted completion time constraint appears as only one component of a complex satisfaction or optimization problem, in conjunction with various other constraints. This justifies our ambition to develop a generic constraint propagation algorithm, instead of customized search algorithms for specific problems.

The remainder of this paper is organized as follows. In the next section, we introduce the notation used in the paper. In Section 3 we review the related literature. This is followed by the presentation of the proposed constraint propagation algorithm (Section 4). In Section 5, we evaluate the performance of our algorithm on a set of benchmark problems from the literature. In Section 6, we sketch extensions of this work to cumulative resources and multiple resource problems. Finally, conclusions are drawn and directions of future research are outlined.

2 Definitions and Notations

While the proposed constraint has potential applications in various fields, we present this work in the context of a single, unary capacity resource scheduling problem where the optimization criterion is the minimization of total weighted completion time.

This scheduling problem involves n activities, A_i, to be executed without preemption on a single, unary resource. Each activity is characterized by its processing time, p_i, and a non-negative weight, w_i. The start time variable of A_i will be denoted by S_i. When appropriate, we call the current lower bound on a start time variable S_i the *release time* of the activity, and denote it by r_i. The total weighted completion time of the activities will be denoted by C. We assume

that all data are integral. Thus, the constraint that enforces $C = \sum_i w_i(S_i + p_i)$ on activities takes the following form.

$$\text{COMPLETION}([S_1, ..., S_n], [p_1, ..., p_n], [w_1, ..., w_n], C)$$

Throughout this paper we assume that p_i and w_i are constants. In applications where this assumption is restrictive, the lower bounds can be used during the propagation. Our algorithm filters the domain of the S_i variables, while it tightens only the lower bound of C. The minimum and maximum values in the current domain of a variable X will be denoted by $\underline{X}$ and $\bar{X}$, respectively.

3 Related Literature

The complexity, approximability, and algorithmic aspects of total weighted completion time scheduling problems have been studied extensively. The most widely discussed problem variants are the single and parallel machine versions with release dates. The classical scheduling notations for these problems are $1|r_i|\sum w_iC_i$ and $P|r_i|\sum w_iC_i$, respectively. Both variants are known to be NP-hard in the strong sense, even with uniform weights. Various polynomially solvable cases have been identified: without release dates, ordering the activities according to the *Weighted Shortest Processing Time* (WSPT) rule, i.e., by non-decreasing p_i/w_i yields an optimal solution. The preemptive version of the single machine problem with release dates and unit weights $(1|r_i, pmtn|\sum C_i)$ is polynomially solvable using *Shortest Remaining Processing Time* rule, but adding non-uniform weights renders it NP-hard. A comprehensive overview of the complexity of related scheduling problems is presented in [7].

Linear programming (LP) and combinatorial lower bounds for the single machine problem have been studied and compared by Goemans et al. [12] and Dyer & Wolsey [9]. The *preemptive time-indexed relaxation* corresponds to an assignment problem in which variables indicate whether activity A_i is processed at time t. In an alternative LP relaxation, the *non-preemptive time-indexed formulation*, variables express if activity A_i is completed at time t. Dyer & Wolsey [9] have shown that the latter is strictly stronger than the former. Since these LP formulations only include continuous variables, but their size depends both on the number of activities and the number of time units, they can be solved in pseudo-polynomial time.

A different LP relaxation has been proposed by Schulz [18], using completion time variables. Subsequently, Goemans et al. [12] proved that this relaxation is equivalent to the preemptive time-indexed formulation, by showing that a preemptive schedule that minimizes the mean busy time (see Section 4) yields the optimal solution for both relaxations. Moreover, this preemptive schedule can be found in $O(n \log n)$ time, where n is the number of activities. The authors also propose two randomized algorithms (and their de-randomized counterparts) to convert the preemptive schedule into a feasible solution of the original problem, and prove that these algorithms lead to 1.69 and 1.75-approximations, respectively. These results also imply a guarantee on the quality of the lower bound.

Polynomial time approximation schemes for the single and parallel machines case, as well as for some other variants are presented in Afrati et al. [1]. The time complexity of the algorithm to achieve a $(1+\varepsilon)$-approximation for a fixed ε is $O(n \log n)$, but the complexity increases super-exponentially with ε.

Papers presenting complete solution methods for different versions of the total weighted completion time problem include a classical work of Belouadah et al. [6] and more recent papers by Jouglet et al. [13] and Pan & Shi [17] for a single machine, Nessah et al. [15] for identical machines, and Della Croce et al. [8], as well as Akkan and Karabatı [2] for the two-machine flowshop problem. Most of these algorithms make use of lower bounds similar to the ones discussed above, as well as various dominance rules and customized branching strategies.

The literature of global constraint propagation algorithms for "sum type" objective functions in scheduling is scarce. Notable exceptions are the works of Focacci et al. [10,11] on embedding relaxations of the *Traveling Salesman Problem* into global constraints. Baptiste et al. [4] proposed a branch-and-bound method for minimizing the total tardiness on a single machine. While building the schedule chronologically, the algorithm makes use of constraint propagation to filter the set of possible next activities by examining how a given choice affects the value of the lower bound. Baptiste et al. [5] address the minimization of the number of late activities on a single resource, and generalize some well-known resource constraint propagation techniques for the case where there are some activities that complete after their due dates. The authors also propose propagation rules to infer if activities are on time or late, but the applicability of these inference techniques is restricted by the fact that they incorporate dominance rules that might be invalid in more general contexts. For propagating the weighted earliness/tardiness cost function in general resource constrained project scheduling problems, Kéri & Kis [14] defined a simple method for tightening time windows of activities by eliminating values that would lead to solutions with a cost higher than the current upper bound.

4 Propagating Total Weighted Completion Time on a Unary Resource

Our propagation algorithm relies on solving the *preemptive mean busy time* relaxation [12] of the scheduling problem. This relaxed problem minimizes $\sum_i w_i M_i$, where M_i denotes the mean busy time of activity A_i, i.e., the average point in time at which the machine is busy processing A_i. This is easily calculated by finding the mean of each time point at which activity A_i is executed.

The underlying idea of our constraint propagator is to exploit the above relaxation to obtain a lower bound on the solution value of the original problem. However, instead of computing only one lower bound, we recompute the lower bound for restricted versions of the problem in which the value of a start time variable S_i is bound to a given value t. We denote such restricted problems by $\Pi\langle S_i = t\rangle$, and the resulting lower bound on C by $C\langle S_i = t\rangle$. Our domain filtering mechanism will rely on the following proposition.

Proposition 1. *If* $\bar{C} < \underline{C}\langle S_i = t\rangle$*, then* t *can be removed from the domain of* S_i*.*

In what follows, we first present an algorithm to compute the optimal relaxed schedule, and then show how this relaxed schedule can be quickly recomputed for the restricted problems. These algorithms will be illustrated using the sample problem introduced in Fig. 1.

	r_i	p_i	w_i	w_i/p_i
A_0	0	5	10	2
A_1	4	2	10	5
A_2	5	4	12	3
A_3	13	2	8	4

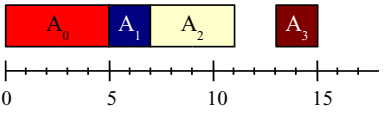


Fig. 1. Left: The input data for the sample problem. Right: The optimal solution of the sample problem. The total weighted completion time is 372.

4.1 Computing a Lower Bound

The optimal solution of the preemptive mean busy time relaxation can be computed in $O(n \log n)$ time [12]. The algorithm maintains a priority queue of the activities sorted by non-increasing w_i/p_i. At each point of time, t, the queue contains the activities A_i with $r_i \leq t$ that have not yet been completely processed. Scheduling decisions must be made each time a new activity is released or an activity is completely processed. In either case, the queue is updated and a *fragment* of the first activity in the queue, lasting until the next decision point, is inserted into the schedule. If the queue is empty, but there are activities not yet released, a gap is created. Technically, gaps are represented as fragments of a zero-weight, zero-release-time activity, and will be called *empty fragments*. We assume that the schedule ends with a sufficiently long empty fragment. Since there are at most $2n$ release time and activity completion events, and updating the queue requires $O(\log n)$ time, the algorithm runs in $O(n \log n)$ time.

The optimal relaxed schedule for the sample problem is presented in Fig. 2. The objective value of this relaxed solution of 362. The fragments of activity A_i are denoted by $\alpha_i, \alpha_i', \alpha_i''$, etc. Empty fragments are named $\varepsilon, \varepsilon', \varepsilon''$, etc.

4.2 Incrementally Recomputing the Lower Bound

The above algorithm can easily be modified to compute optimal relaxed solutions for restricted problems $\Pi\langle S_i = t\rangle$, by assigning $r_i = t$ and $w_i = \infty$. This gives activity A_i the largest w_i/p_i ratio among all the activities, ensuring that it starts at t and is not preempted. Relaxed solutions for various restrictions on the sample problems are presented in Fig. 3.

We apply the above method only once for each activity A_i, restricting it to start exactly at its release time, i.e., for $\Pi\langle S_i = r_i\rangle$. For other possible start

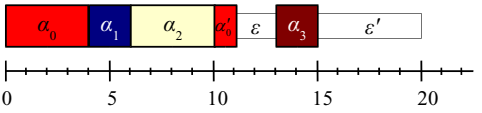


Fig. 2. The solution of the relaxed problem, with objective value 362. This is a lower bound on the original problem.

times, we incrementally convert the relaxed solution of $\Pi\langle S_i = t\rangle$ into a solution of $\Pi\langle S_i = t+1\rangle$ (or even directly for $\Pi\langle S_i = t+\Delta\rangle$ with $\Delta \geq 1$). This recomputation is based on the observation that one can represent the transformation of the relaxed solutions of $\Pi\langle S_i = t\rangle$ to that for $\Pi\langle S_i = t+1\rangle$ by a permutation of unit-duration sections of activities as follows.

Definition 1. *The permutation $\pi = (\alpha_0, \alpha_1, ..., \alpha_K)$ transforms a preemptive schedule by moving the first unit of each activity fragment α_k to the place of the first unit of $\alpha_{(k+1) \bmod (K+1)}$. If the moved unit is placed next to a fragment of the same activity then they are merged, otherwise a new fragment is created.*

In Fig. 3, the corresponding permutations are displayed next to each relaxed solution. For example, moving from $\Pi\langle S_i = 0\rangle$ to $\Pi\langle S_i = 1\rangle$ requires the movements of the first unit of the fragments as follows: $\alpha_0 \rightarrow \alpha_1, \alpha_1 \rightarrow \alpha_2, \alpha_2 \rightarrow \varepsilon, \varepsilon \rightarrow \alpha_0$. The final move creates a new empty fragment, ε''.

Lemma 1. *The permutation from $\Pi\langle S_i = t\rangle$ to $\Pi\langle S_i = t+1\rangle$ starts with $\alpha_0 = A_i$, while its further elements can be computed as*

$$\alpha_{k+1} = \textit{the leftmost fragment with } S(\alpha_{k+1}) > S(\alpha_k) \textit{ and } \frac{w(\alpha_{k+1})}{p(\alpha_{k+1})} < \frac{w(\alpha_k)}{p(\alpha_k)}.$$

The permutation ends when it reaches a fragment α_K with $r(\alpha_K) \leq t$. Recall that the empty fragment at the end of the schedule always satisfies this condition, and $w(A_i) = \infty$ is assumed.

Applying permutation $\pi = (\alpha_0, ..., \alpha_K)$ increases the total mean busy time of the preemptive schedule by

$$C(\pi) = \sum_{k=0}^{K-1} (S(\alpha_{k+1}) - S(\alpha_k)) \frac{w(\alpha_k)}{p(\alpha_k)} \; - \; (S(\alpha_K) - S(\alpha_0)) \frac{w(\alpha_K)}{p(\alpha_K)},$$

where $S(\alpha_k)$ denotes the start time of fragment α_k. Thus, the cost of the new relaxed solution is $\underline{C}\langle S_i = t+1\rangle = \underline{C}\langle S_i = t\rangle + C(\pi)$.

Transformations of consecutive relaxed solutions $\Pi\langle S_i = t\rangle$ to $\Pi\langle S_i = t+1\rangle$ and $\Pi\langle S_i = t+1\rangle$ to $\Pi\langle S_i = t+2\rangle$ may be identical. Such is the case for $i = 0$ and $t = 0$ in the sample problem shown in Fig. 3. Note that the Δ-fold application of permutation π shifts fragments α_k that satisfy $S(\alpha_{k+1}) = S(\alpha_k) + p(\alpha_k)$ with Δ units to the right, while it relocates a Δ-long portion of every other fragment

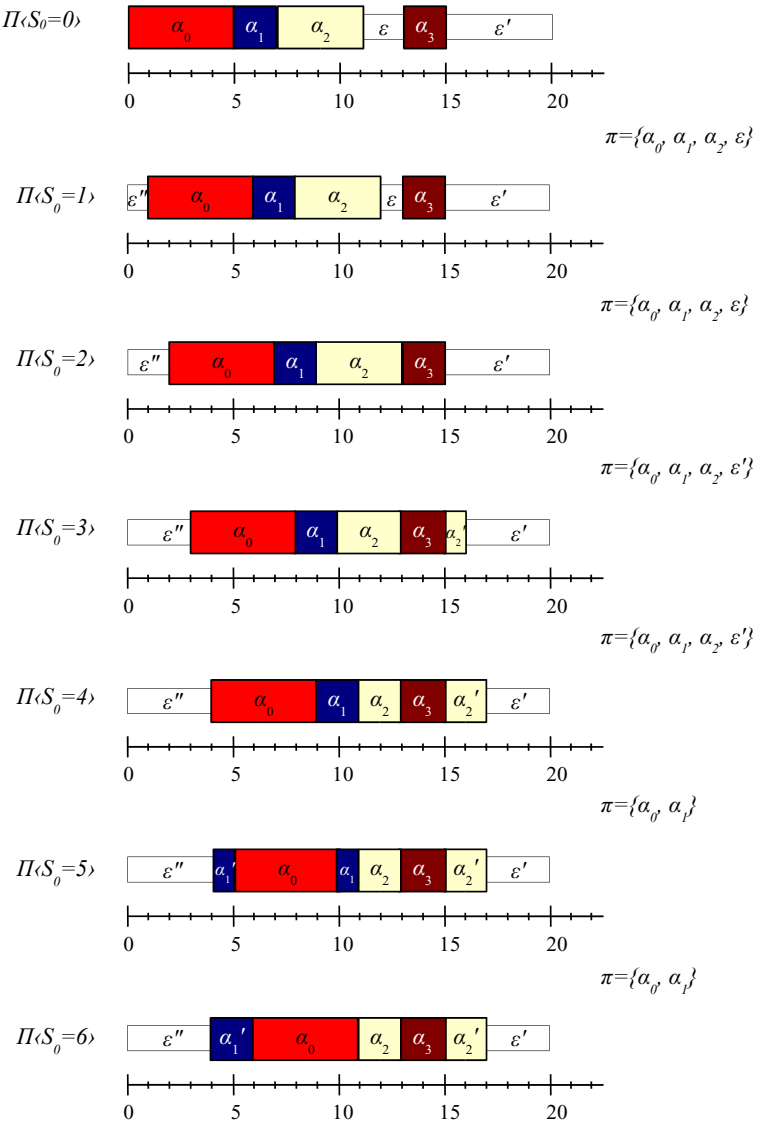


Fig. 3. Relaxed solutions of various restricted problems. The corresponding restrictions are displayed on the left, the permutations applied are shown on the left of the schedule.

into newly created fragments. Thus, a permutation π on an optimal preemptive schedule results in optimal relaxed solutions for subsequent problems exactly Δ times, where Δ is the minimum of the following values:

(1) $p(\alpha_k)$ of fragments $\alpha_k \in \pi$ with $S(\alpha_{(k+1) \bmod (K+1)}) \neq S(\alpha_k) + p(\alpha_k)$;
(2) $r(\alpha_k) - S(\alpha_0)$ for fragments $\alpha_k \in \pi$ with $1 \leq k < K$.

Condition (1) must be respected because the Δ-long, relocated portion of a fragment cannot be longer than the fragment itself. On the other hand, violating condition (2) would lead to sub-optimality: after the $(r(\alpha_{k^*}) - S(\alpha_0))$-fold application of permutation π, it is a different permutation $\pi' = (\alpha_0, ..., \alpha_{k^*})$ that results in optimal solutions for the subsequent relaxed problems.

The lower bound cost for the new restricted problem can be calculated as $\underline{C}\langle S_i = t + \Delta\rangle = \underline{C}\langle S_i = t\rangle + \Delta \cdot C(\pi)$, and linear interpolation can be applied for determining $\underline{C}\langle S_i = t'\rangle$ for $t' \in (t, t + \Delta)$.

4.3 From a Lower Bound to Domain Filtering

If both $\underline{C}\langle S_i = t\rangle > \bar{C}$ and $\underline{C}\langle S_i = t + \Delta\rangle > \bar{C}$ hold then the complete interval $[t, t + \Delta]$ can be removed from the domain of S_i. If only the first (second) condition holds, then the first (second), proportional part of the interval is removed. No removal is made otherwise. The new lower bound on the total weighted completion time is computed as $\underline{C} = \max_i \min_t C\langle S_i = t\rangle$.

Thus, each recomputation step consist of determining the permutation to apply and the corresponding Δ, followed by filtering the appropriate variable domains. These steps are iterated until activity A_i reaches the end of its time window, and the same procedure is repeated on each activity. The pseudo-code of the proposed constraint propagation algorithm is presented in Fig. 4.

4.4 Computational Complexity

In order to determine the computational complexity of the propagation algorithm, we first give a bound on the number of recomputation steps during the filtering of the domain of one start time variable S_i. We distinguish between two kinds of recomputation steps, depending on whether condition (1) or condition (2) bounds Δ, and call these (1)-type and (2)-type steps, respectively. Recomputation steps where (1) and (2) are equally bounding are considered to be of the (1)-type.

Now, let us define the number of inversions $I(\sigma)$ as the number of fragment pairs (α_1, α_2) in the preemptive schedule σ such that $S(A_i) \leq S(\alpha_1) < S(\alpha_2)$ and $w(\alpha_1)/p(\alpha_1) < w(\alpha_2)/p(\alpha_2)$. Since there are at most $2n$ fragments in σ, $I(\sigma)$ is at most $O(n^2)$. Observe that $I(\sigma)$ is strictly decreased by (1)-type recomputation steps, while it is not affected by (2)-type steps. Therefore, the number of (1)-type steps is at most $O(n^2)$, while the number of (2)-type steps is bounded by the number of different activity release times, which is not greater than n.

Thus, the complete run of the constraint propagation algorithm takes at most $O(n^4)$ time: for each of the n activities, there are at worst $O(n^2)$ recomputation steps and each step is carried out in at worst $O(n)$ time.

4.5 Implementation Details

While the pseudo-code depicted in Fig. 4 captures the underlying ideas of the proposed propagation algorithm, its performance can be increased in various

```
PROCEDURE RecomputeSchedule(σ - schedule, A_i - activity, t - time)
  LET permutation π = (A_i)
  WHILE r(last(π)) > t OR size(π) = 1
      α := leftmost fragment in σ fragments with S(α) > S(last(π))
           and w(α)/p(α) < w(last(π))/p(last(π))
      Append α to π
  LET Δ := min(min(p(α_k) | α_k ∈ π : S(α_{k+1 mod K+1}) ≠ S(α_k) + p(α_k)),
               min(r(α_k) − S(α_0) | α_k ∈ π, 1 ≤ k < K))
  σ' := Schedule obtained by performing π^Δ on σ
  RETURN ⟨σ', Δ⟩


PROCEDURE Propagate()
  FORALL activity A_i
      LET t := r_i
      σ_t := schedule computed by the lower bounding procedure for Π⟨S_i = t⟩
      WHILE t < S̄_i
           ⟨σ_{t+Δ}, Δ⟩ := RecomputeSchedule(σ, A_i, t)
           IF C⟨S_i = t⟩ > C̄ and C⟨S_i = t + Δ⟩ > C̄ THEN
               Remove [t, t + Δ] from domain(S_i)
           ELSE IF C⟨S_i = t⟩ > C̄ THEN
               Remove [t, t + ⌈(Δ (C̄ − C⟨S_i=t⟩)/(C⟨S_i=t+Δ⟩ − C⟨S_i=t⟩))⌉ − 1] from domain(S_i)
           ELSE IF C⟨S_i = t + Δ⟩ > C̄ THEN
               Remove [t + ⌊Δ (C̄ − C⟨S_i=t⟩)/(C⟨S_i=t+Δ⟩ − C⟨S_i=t⟩)⌋ + 1, t + Δ] from domain(S_i)
           t := t + Δ
  C ≥ max_i min_t C⟨S_i = t⟩
```

Fig. 4. Pseudo-code of the constraint propagation algorithm

ways. We have improved the average complexity of our implementation of the algorithm with the following changes:

- Common branching strategies bind activity start times in a chronological order, which results in a bound head of the schedule. This bound head is ignored when building the relaxed solutions, only its cost is taken into account;
- Relaxed solutions are saved during each run of the propagator; filtering the domain of S_i is attempted again only after $\underline{S}_j$, $j \neq i$ have increased, or $\bar{C}$ decreased sufficiently to modify the relaxed solutions.

5 Computational Experiments

We ran computational experiments to measure the efficiency of the proposed propagation algorithm on the single-machine total weighted completion time problem with release times. The propagator has been implemented in C++ and embedded it into ILOG Solver and Scheduler versions 6.1. For propagating the

resource constraint, we used the edge-finding algorithm. We applied an adapted version of the *SetTimes* branching heuristic: in each search node from the set of not yet scheduled (and not postponed) activities, the heuristic selects the activity that has the smallest earliest start time (EST) and then breaks ties by choosing the activity with the highest w/p ratio. Two branches are created according to whether the start time of this activity is bound to its EST or it is postponed.

We compared the performance of three different models. The first used the standard weighted sum constraint for propagating the optimization criterion (WS). The second calculated the lower bound presented in Section 4.1 (WS+LB) at each node and used it for bounding. The third model made use of the proposed COMPLETION constraint.

These algorithms were tested on benchmark instances from the online repository [16]. The same instances or the some problem generation method have been used in various previous works [3,6,13,17]. The repository contains 10 single-machine problem instances for each combination of parameters n and R, where n denotes the number of activities and takes values between 20 and 200 in increments of 10, while R is the relative range of the release time, chosen from $\{0.2, 0.4, 0.6, 0.8, 1.0, 1.25, 1.5, 1.75, 2.0, 3.0\}$. Activity durations are randomly chosen from $U[1, 100]$, weights from $U[1, 10]$, and release times from $U[0, 50.5nR]$, where $U[a, b]$ denotes the integer uniform distribution over interval $[a, b]$. Out of these 1900 instances in total, we ran experiments on the 300 instances with $n \leq 70$ and every second value of parameter R. The experiments were run on a 1.86 GHz Pentium M computer with 1 GB of RAM, with a time limit of 120 seconds imposed.

The experimental results are presented in Table 1, where each row contains combined results for the 10 instances with the given number of activities n and release time range R. For each of the three models, the table displays the number of the instances that could be solved to optimality (column *Solved*), the average number of search nodes (*Nodes*), and average search time in seconds (*Time*). The average is computed only on the instances that the algorithm solved. From these results we can conclude that the COMPLETION constraint adds significant pruning strength to the constraint-based approach. This pruning not only paid off in the means of the number of search nodes, but also decreased solution time on *every instance*, compared to both other models. While the classical WS model fails on some of the 20-activity instances, the COMPLETION constraint enabled us to solve – with one exception – all problems with at most 40 activities, and also performed well on the 50-activity instances.

The results also illustrate that instances with release time range $R \in \{0.6, 1.0\}$ are significantly more complicated for models WS+LB and COMPLETION than other instances. This is explained by the fact that with $R \ll 1$, activities in the second half of the schedule can simply be ordered by non-increasing w_i/p_i. On this section of the schedule, the lower bound is exact and our propagator achieves completeness. On the other hand, $R \gg 1$ leads to problems where only a few activities can be chosen for scheduling at any point in time, which makes the instance easily solvable as well.

Table 1. Experimental results: number of instances solved, average number of search nodes and average search time for the different versions of the branch and bound. Dash '-' means that none of the instances could be solved within 120 CPU seconds.

n	R	WS			WS+LB			COMPLETION		
		Solved	Nodes	Time	Solved	Nodes	Time	Solved	Nodes	Time
20	20	-	-	-	10	591	0.00	10	46	0.00
	60	2	1842024	62.50	10	1872	0.00	10	95	0.00
	100	10	114359	3.60	10	1756	0.10	10	109	0.00
	150	10	2518	0.00	10	223	0.00	10	67	0.00
	200	10	140	0.00	10	102	0.00	10	51	0.00
30	20	-	-	-	10	1718	0.10	10	114	0.00
	60	-	-	-	10	20674	2.90	10	417	0.00
	100	5	845422	58.60	9	113523	17.00	10	7046	2.90
	150	10	21555	1.30	10	1912	0.10	10	189	0.00
	200	10	2633	0.10	10	857	0.00	10	159	0.00
40	20	-	-	-	10	8434	1.30	10	241	0.00
	60	-	-	-	8	290209	57.12	9	4815	4.77
	100	2	164455	29.00	4	55760	12.00	10	27366	15.60
	150	10	40160	2.80	10	8943	1.30	10	592	0.20
	200	10	60602	3.70	10	3685	0.20	10	374	0.00
50	20	-	-	-	8	89148	22.75	10	1557	2.30
	60	-	-	-	-	-	-	8	31486	55.25
	100	-	-	-	-	-	-	2	26807	41.00
	150	3	92954	8.33	7	113198	26.71	10	12180	15.10
	200	8	36056	3.25	9	6898	1.22	10	1498	0.80
60	20	-	-	-	3	161594	49.66	10	16159	33.20
	60	-	-	-	-	-	-	-	-	
	100	-	-	-	-	-	-	-	-	
	150	-	-	-	4	194053	60.25	8	43353	32.75
	200	4	120345	12.75	9	48066	11.55	10	5290	4.30
70	20	-	-	-	2	72540	30.00	6	2591	10.83
	60	-	-	-	-	-	-	-	-	
	100	-	-	-	-	-	-	1	12125	46.00
	150	-	-	-	2	184557	64.50	3	8940	14.33
	200	4	228762	43.75	7	108701	34.14	9	14448	12.22

Our goal in this work is to develop a widely applicable constraint rather than to solve the single machine weighted completion time problem. However, it is instructive to compare these results directly against state-of-the-art, dedicated techniques for solving the single machine problem. Our algorithms comparable favorably to existing LP-based methods [3] that are able to solve instances with at most 30–35 activities, and earlier branch-and-bound methods [6], which solve problems with up to 40–50 activities. On the other hand, our approach is outperformed by two different, recent solution methods. One is a branch-and-bound algorithm combined with powerful dominance rules, constraint propagation, and no-good recording by Jouglet et al. [13], which has originally been developed for solving the more general total weighted tardiness problem. The other is a

dynamic programming approach enhanced with dominance rules and constraint propagation by Pan and Shi [17]. These two approaches are able to solve instances with up to 100 and 200 activities, respectively. It should be noted that a part of the contributions of these works, especially the strong dominance rules are orthogonal and complementary to the COMPLETION constraint. We expect that combining such approaches with the COMPLETION constraint would lead to further performance improvements.

6 Extensions to Other Scheduling Models

Our future work will focus on the application and extension of these results to more complex scheduling models and other application domains, such as constraining the location of the center of gravity in container loading. Below we sketch two possible extensions.

6.1 Extension to Cumulative Resources

To extend the COMPLETION constraint to cumulative resources (i.e., resources with a non-unary capacity), we define a variation:

$$\text{COMPLETION}_m([S_1, ..., S_n], [p_1, ..., p_n], [\varrho_1, ..., \varrho_n], [w_1, ..., w_n], R, C)$$

As with the unary case, the scheduling problem involves n activities A_i to be executed without preemption on a single, cumulative resource. Each activity is characterized by its processing time p_i, a non-negative weight factor w_i, and its resource requirement ϱ_i. R represents the capacity of the resource. The total weighted completion time of the activities will be denoted by C. We assume that ϱ_i and R are constants, however our approach is easily adapted by reasoning with the lower bound of ϱ_i and the upper bound of R.

As above, we use the mean busy time relaxation to obtain a lower bound on C. A preemptive schedule is prepared chronologically, by choosing at each decision point the k available activities that have the highest w_i/p_i ratio, so that $\sum_{i=1}^{k-1} \varrho_i < R \leq \sum_{i=1}^{k} \varrho_i$ holds. A schedule fragment is created in which activities $A_1, ..., A_{k-1}$ are processed at rates $\varrho_1, ..., \varrho_{k-1}$, and A_k is processed at rate $R - \sum_{i=1}^{k-1} \varrho_i$. This fragment lasts until the next decision point, which corresponds to a release time, an activity completion, or a point in time where the remaining volume of an activity decreases below its previous processing rate. The complexity of the lower bounding algorithm is $O(n^2)$.

However, the cumulative extension of the COMPLETION constraint is more challenging than the unary version, because recomputing the mean busy time relaxation for each relevant value of the start time variables imposes an extensive computational burden in the cumulative case. We are currently investigating ways of partially relaxing the release times or resource requirements in order to facilitate quicker computations, at the price of reduced pruning strength.

6.2 Extension to Multiple Resource Problems

In a simple multiple resource problem, each activity requires one or more resources and has a weight. The obvious approach therefore is simply to have one COMPLETION constraint on each resource and represent the sum of completion times criterion as the sum of the sum of completion times on each resource, correcting for activities that require more than one resource.

More interesting is the extension to multiple resource project scheduling problems with more complex temporal relations amongst activities. For example, in a job shop scheduling problem, a job is made up of a sequence of activities linked by precedence constraints. The standard weighted completion model associates the weight and the completion time of a job to the final activity in the job, assigning zero weight to all other activities. We propose a more generic approach that allows us to use the COMPLETION constraint as defined above, and leads to more efficient pruning.

In our approach, activities can be assigned arbitrary weights, under the condition that the sum of activity weights within a job must equal the weight of the job. One COMPLETION constraint is placed on each resource that processes weighted activities, and the overall cost of the solution is the sum of total weighted completion times on each resource. The difference between activity and job completion times is compensated by a bias computed as the scalar product of activity weights and the temporal distance of activity and job completions. This way, weighted activities contribute to the overall cost of the solution both directly by their weighted completion time, and indirectly by delaying other weighted activities that have a lower w_i/p_i ratio. On the other hand, zero-weight activities do not affect the cost in either way, unless they satisfy $\bar{S}_i < \underline{S}_i + p_i$, in which case they must use the resource in the interval $[\bar{S}_i,\ \underline{S}_i + p_i]$. This can be represented by a fragment α of appropriate duration, infinite weight, and $r(\alpha) = \bar{S}_i$.

Intuitively, the above considerations imply that the relaxed cost on each resource increases super-linearly with the total activity weight on the resource. Therefore, the strongest pruning is achieved when weighted activities are concentrated on a small number of resources. Such a distribution of the weights can be attained by a preprocessing procedure to assign weights. The procedure selects the most utilized resource R_1 and the set of activities $\mathscr{A}_1$ that require R_1, but such that none of their job-successors require R_1. To each activity in $\mathscr{A}_1$, we assign the weight of the corresponding job. The procedure continues with repeating these steps on the second, third, etc., most utilized resource while there are jobs not yet covered. Finally, all the remaining activities receive zero weights.

This approach does the opposite of what the classical weight-on-finals approach does in the case where job-final activities are uniformly spread over the resources. The superiority of the proposed approach can be best demonstrated on such problems: in the extreme, where there is at most one job-final activity on each resource, the classical approach results in no propagation at all. In contrast, the proposed method can still tighten variable domains by considering various weighted activities on the same resource. Our future work will focus on

elaborating the details of the assignment of weights to activities in the above sketched framework.

7 Conclusions

In this paper, we proposed an algorithm for propagating the COMPLETION constraint, which represents the sum of weighted completion times on a single unary capacity resource. The propagation of the constraint exploits a lower bound arising from the optimal solution to the preemptive mean busy time scheduling problem which can be found in polynomial time. Using this lower bound, we propose an algorithm that updates the lower bound on the cost by incrementally recomputing the optimal preemptive mean busy time schedule for a carefully structured subset of the possible start times of each activity. The time complexity of this algorithm is $O(n^4)$.

Empirical results on a set of single resource, minimum weighted completion time benchmarks in the literature show that the COMPLETION constraint significantly improves the performance of constraint-based approaches, which is a considerable result in the field of scheduling with "sum type" objective functions, an area where constraint programming has not yet been especially strong.

Our future work will examine the extension of this approach to cumulative resources, scheduling problems with multiple machines such as the job shop scheduling problem, and other problems with "sum type" cost constraints.

Acknowledgment

A part of this work was carried out while A. Kovács was with the Cork Constraint Computation Centre, supported by an ERCIM fellowship. The authors acknowledge the support of the EU FP6 Net-WMS project and the Canadian Natural Sciences and Engineering Research Council, and ILOG, S.A.

References

1. F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, M. Sviridenko. Approximation Schemes for Minimizing Average Weighted Completion Time with Release Dates. In Proc. of the 40th IEEE Symposium on Foundations of Computer Science, pp. 32–44, 1999.
2. C. Akkan, S. Karabatı. The Two-machine Flowshop Total Completion Time Problem: Improved Lower Bounds and a Branch-and-bound Algorithm. European Journal of Operational Research 159:420–429, 2004.
3. J.M. van den Akker, C.A.J. Hurkens, M.W.P. Savelsbergh. Time-indexed Formulations for Machine Scheduling Problems: Column Generation. INFORMS Journal on Computing 12:111–124, 2000.
4. Ph. Baptiste, J. Carlier, A. Jouglet. A Branch-and-Bound Procedure to Minimize Total Tardiness on One Machine with Arbitrary Release Dates. European Journal of Operational Research 158:595–608, 2004.

5. Ph. Baptiste, L. Peridy, E. Pinson. A Branch and Bound to Mininimze the Number of Late Jobs on a Single Machine with Release Time Constraints. European Journal of Operational Research 144(1):1–11, 2003.
6. H. Belouadah, M.E. Posner, C.N. Potts. Scheduling with Release Dates on a Single Machine to Minimize Total Weighted Completion Time. Discrete Applied Mathematics 36:213–231, 1992.
7. B. Chen, C.N. Potts, G.J. Woeginger. A Review of Machine Scheduling: Complexity, Algorithms and Approximation. In: Handbook of Combinatorial Optimization, Vol. 3, pp. 21–169, Kluwer, 1998.
8. F. Della Croce, M. Ghirardi, R. Tadei. An Improved Branch-and-bound Algorithm for the Two Machine Total Completion Time Flow Shop Problem. European Journal of Operational Research 139:293–301, 2002.
9. M. Dyer, L.A. Wolsey. Formulating the Single Machine Sequencing Problem with Release Dates as Mixed Integer Program. Discrete Applied Mathematics 26:255–270, 1990.
10. F. Focacci, A. Lodi, M. Milano. Embedding Relaxations in Global Constraints for Solving TSP and TSPTW. Annals of Mathematics and Artificial Intelligence 34(4):291–311, 2002.
11. F. Focacci, A. Lodi, M. Milano. Optimization-Oriented Global Constraints. Constraints 7(3-4):351–365, 2002.
12. M.X. Goemans, M. Queyranne, A.S. Schulz, M. Skutella, Y. Wang. Single Machine Scheduling with Release Dates. SIAM Journal on Discrete Mathematics 15(2):165–192, 2002.
13. A. Jouglet, Ph. Baptiste, J. Carlier. Branch-and-Bound Algorithms for Total Weighted Tardiness. In: Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Chapter 13, Chapman & Hall / CRC, 2004.
14. A. Kéri, T. Kis. Primal-dual Combined with Constraint Propagation for Solving RCPSPWET. In Proc. of the 2nd Multidisciplinary International Conference on Scheduling: Theory and Applications, pp. 748–751, 2005.
15. R. Nessah, F. Yalaoui, C. Chu. A Branch-and-bound Algorithm to Minimize Total Weighted Completion Time on Identical Parallel Machines with Job Release Dates. Computers and Operations Research, in print.
16. Y. Pan. Test Instances for the Dynamic Single-machine Sequencing Problem to Minimize Total Weighted Completion Time. Available at `http://www.cs.wisc.edu/~yunpeng/test/sm/dwct/instances.htm`
17. Y. Pan, L. Shi. New Hybrid Optimization Algorithms for Machine Scheduling Problems. IEEE Transactions on Automation Science and Engineering, in print.
18. A.S. Schulz. Scheduling to Minimize Total Weighted Completion Time: Performance Guarantees of LP-Based Heuristics and Lower Bounds. Proc. of the 5th Int. Conf. on Integer Programming and Combinatorial Optimization, pp. 301–315, 1996.

Computing Tight Time Windows for RCPSPWET with the Primal-Dual Method*

András Kéri[1] and Tamás Kis[2,**]

[1] Chair of Business Administration, Transport and Logistics, Technical University of Dresden, Andreas Schubert str. 23, D-01069 Dresden, Germany
andras.keri@mailbox.tu-dresden.de

[2] Computer and Automation Institute, Kende str. 13-17, H-1111 Budapest, Hungary
tamas.kis@sztaki.hu

Abstract. In this paper we combine OR and CP techniques to solve the Resource-Constrained Project Scheduling Problem with Earliness-Tardiness costs and general temporal constraints. Namely, we modify the Primal-Dual algorithm for solving the maximum-cost flow problem in a network to deduce tight time windows for activities with respect to a finite upper bound on the optimal objective function value. We compare our method to the only exact method in the literature. Our results show that time window computations and additional domain filtering techniques may improve the performance of tree-search based methods.

1 Introduction

Suppose we have n activities $V = \{1, \ldots, n\}$ with given *processing time* $p_j \geq 0$, *due-date* $d_j \geq 0$, *unit earliness cost* $e_j \geq 0$ and *unit tardiness cost* $t_j \geq 0$ for each $j \in V$. Furthermore, there is a set $A \subset V \times V$ of pairwise temporal relations between the activities with time lags $\delta_{i,j}$, $(i,j) \in A$, where the $\delta_{i,j}$ can be 0, positive or negative, which enables the modeling of minimum and maximum time lags. In addition, there is a finite set R of resources to carry out the activities. Each resource $k \in R$ has a finite capacity $b_k > 0$, and for each activity $i \in V$, its resource requirements are given by non-negative numbers $r_{i,k}$, $k \in R$. All the problem data are integral.

We have to determine the non-negative *start time* S_j of each activity j subject to (i) temporal constraints: $S_j - S_i \geq \delta_{i,j}$, $\forall (i,j) \in A$, and (ii) resource constraints: $\sum_{i \in \mathcal{A}(S,t)} r_{i,k} \leq b_k$, for each $k \in R$ and any time point t, where $\mathcal{A}(S,t)$ denotes the set of those activities that are being executed at time point t, i.e., $\mathcal{A}(S,t) = \{i \in V \mid S_i \leq t < S_i + p_i\}$. We say that activity j is *early* if $S_j + p_j < d_j$ and *tardy* if $S_j + p_j > d_j$. The cost incurred when activity j is early is $e_j \cdot \max\{0, -S_j + d_j - p_j\}$, while the cost of being late is $t_j \cdot \max\{0, S_j - d_j + p_j\}$.

* The research of Tamás Kis has been supported by the János Bolyai Research Grant No. BO/00380/05 and by the NKFP Grant No. 2/010/2004 (VITAL).

** Corresponding author.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 127–140, 2007.

The objective is to minimize the total earliness-tardiness cost over all activities. To summarize, the *Resource-Constrained Project Scheduling Problem with Earliness-Tardiness costs (RCPSPWET)* studied in this paper can be stated as follows:

$$\min \sum_{j \in V} (e_j \cdot \max\{0, -S_j + d_j - p_j\} + t_j \cdot \max\{0, S_j - d_j + p_j\})$$

subject to the constraints (i) and (ii). This problem is NP-hard in general and becomes polynomial if there are no resource constraints [21].

The objective function is *non-regular*, as it is not monotone in the starting (or completion) times of the activities. A classification of non-regular objective functions can be found in [15].

We are aware of only one exact method for RCPSPWET [18] with general temporal constraints, which is nevertheless similar to that of [14] for the maximum net present value problem. In fact, almost the same branch and bound procedure can be applied to the two different problems [15]. The lower/upper bounds can be computed by the *steepest ascent method* [18], [19]. In addition, the procedure makes various deductions to cut off a large part of the search tree. This includes the subset dominance rule and inference of new arcs between pairs of resource-disjunct activities. The special case with precedence constraints only (no maximal time lags) has been studied by Vanhoucke et al. [20] and then by Kéri and Kis [9], [10]. In particular, Kéri and Kis has proposed a fast approximation method for determining approximate time windows for activities, but that method seems to work only when there are no directed cycles in the precedence graph. Notice that when maximum time lags are allowed, then cycles of non-positive lengths are allowed. For the job-shop special case Beck and Refalo [3] have proposed various hybrid solution techniques. Their procedures exploit that there are only end-to-start precedence constraints and that all resources are of unit capacity. Some of their procedures (CRS family) also make use of the fact that only the last activities of the jobs affect the cost function directly. From a different perspective, the resource-relaxed problem is a special case of the *Simple Temporal Problem with Preferences*, proposed by Khatib et al. [11]. In that problem, between two events X_i and X_j there can be hard minimum and maximum time lags, or a *preference function* $F_{(X_i,X_j)}(t)$ indicating the preference of scheduling X_j t time units after X_i. For piecewise linear concave preference functions, a solution which maximizes the sum of preferences can be found by solving a linear program as observed by Morris et al. [13]. Kumar [12] has recently noticed that the linear program is equivalent to a minimum cost flow problem in an appropriate network.

Our main contribution is an algorithm for computing tight time windows for the activities using only the best known objective function value. Our method is based on the Primal-Dual method for computing a maximum-cost flow in a network. Time windows are then used for applying additional domain filtering techniques, published in the literature, in order to narrow further the time windows of activities. Narrower time windows permit, on the one hand, to infer

more arcs between resource-disjunct activities, and on the other hand to compute stronger lower bounds. As a result, in many cases it suffices to explore a smaller portion of the search tree to find an optimal solution and prove its optimality.

The structure of the paper is as follows: In Sections 2 we summarize known facts about the resource-relaxed problem. In Section 3 we describe the network computations and in particular we provide the method for computing tight time windows. Section 4 sketches the branch and bound procedure that we apply. We do not give all details as these are well documented in the literature. The performance of our method is evaluated in Section 5.

2 Preliminaries

Below we formalize the resource-relaxed problem as a linear program and recognize that it is dual to a maximum cost flow problem. The same conclusion has been reached in [21]. We pay special attention to the modeling of time windows.

We introduce two dummy activities, 0 and $n+1$, and let $V^+ = V \cup \{0, n+1\}$. Let A^+ contain all the temporal constraints from A and the new constraints $(0, j)$ with $\delta_{0,j} = 0$, $(j, 0)$ with $\delta_{j,0} = -\infty$, $(j, n+1)$ with $\delta_{j,n+1} = -d_j + p_j$ and $(n+1, j)$ with $\delta_{n+1,j} = d_j - p_j$ for $j \in V$. Let $A^- = A^+ \setminus \{(j, n+1), (n+1, j) \mid j \in V \cup \{0\}\}$. Moreover, let $e_0 = t_0 = \infty$ and $d_0 = 0$. After these preparations, the linear program modeling the resource-relaxed problem is:

$$v(S) = \min \sum_{j \in V \cup \{0\}} e_j w_{n+1,j} + t_j w_{j,n+1} \tag{1}$$

subject to

$$S_j - S_i \geq \delta_{i,j}, \quad \forall (i,j) \in A^-, \tag{2}$$

$$S_j - S_{n+1} + w_{n+1,j} \geq \delta_{n+1,j}, \quad \forall j \in V \cup \{0\}, \tag{3}$$

$$S_{n+1} - S_j + w_{j,n+1} \geq \delta_{j,n+1}, \quad \forall j \in V \cup \{0\}, \tag{4}$$

$$w_{n+1,j}, w_{j,n+1} \geq 0, \quad \forall j \in V, \tag{5}$$

$$S_{n+1} = 0. \tag{6}$$

Inequalities (2) represent the temporal constraints, while (3) and (4) express the earliness and tardiness of activity j, respectively. Since $e_0 = t_0 = \infty$ and $d_0 = 0$, in any optimal solution $S_0 = 0$. We deliberately do not set S_0 to 0 explicitly to facilitate fast re-optimization as explained in Section 3.1. Then the dual problem is:

$$\max \sum_{(i,j) \in A^+} \delta_{i,j} X_{i,j} \tag{7}$$

subject to

$$\sum_{(j,i) \in A^+} X_{j,i} - \sum_{(i,j) \in A^+} X_{i,j} = 0, \quad \forall i \in V^+, \tag{8}$$

$$X_{n+1,j} \leq e_j \quad \forall j \in V \cup \{0\}, \tag{9}$$
$$X_{j,n+1} \leq t_j \quad \forall j \in V \cup \{0\}, \tag{10}$$
$$X_{i,j} \geq 0, \quad \forall (i,j) \in A^+. \tag{11}$$

Clearly, this is a maximum cost flow problem in the network $N(\delta) = (V^+, A^+, \delta)$. We will refer to (7)-(11) as the *primal problem* and to (1)-(6) as the *dual problem*. It is known that for non-negative e_j and t_j values, the optimal $w_{j,n+1}$ and $w_{n+1,j}$ values are uniquely determined by the optimal S values by the formulas $w_{n+1,j} = \max\{0, -S_j + d_j - p_j\}$ and $w_{j,n+1} = \max\{0, S_j - d_j + p_j\}$. Therefore, the optimal dual solution can be characterized by the values of the S_j variables.

We recapitulate the complementary slackness optimality conditions for later reference. The *reduced cost* of arc $(i,j) \in A^+$ is defined as $rc_{i,j}(S) = \delta_{i,j} + S_i - S_j$. Let $u_{i,j}$ denote the upper bound on arc (i,j). Notice that $u_{i,j} = \infty$ except on the arcs $(n+1, j)$ and $(j, n+1)$, $j \in V$. It is known that a pair of primal and dual optimal solutions must satisfy the conditions:

If $rc_{i,j}(S) < 0$, then $X_{i,j} = 0$.
If $rc_{i,j}(S) > 0$, then $X_{i,j} = u_{i,j}$.
If $0 < X_{i,j} < u_{i,j}$, then $rc_{i,j}(S) = 0$.

With respect to any $\bar{X} \in \mathbb{R}^{|A^+|}$, define the *excess* of node i as $ex(i) = \sum_{(j,i)\in A^+} \bar{X}_{j,i} - \sum_{(i,j)\in A^+} \bar{X}_{i,j}$. Node i is a *source node* if $ex(i) > 0$, and it is a *sink node* if $ex(i) < 0$.

3 Network Computations

If we relax the resource constraints, we are left with a much easier problem, which can be solved in polynomial time [21]. Various methods have been suggested in the literature for finding the optimal solution of the resource-relaxed problem quickly [18], [20]. However, in branch and bound based procedures only a few new arcs are introduced when generating child nodes, and we can use the optimal solution of the parent node to initialize the search in a child node. This idea occurs e.g., in [14] for speeding up the steepest ascent method applied to the net present value problem. We will show how to recompute the optimal solutions using the primal-dual maximum cost flow algorithm. On the other hand, when an upper bound on the optimum has been found, we also wish to compute tight time windows for activities, that is, the earliest and latest time points beyond which it is no use to execute the activity because the objective function is guaranteed to be greater than the known upper bound.

3.1 Updating the Primal and Dual Optimal Solutions

Branch and bound algorithms explore a tree defined by a branching scheme (cf. Section 4). In each node visited by the algorithm a lower bound is computed with respect to the constraints of the original problem and additional constraints

determined during the search. Let $N(E,\delta) = (V^+, A^+ \cup E, \delta)$ be the network associated with a tree-node q, where E contains those arcs added by the branch and bound algorithm along the path leading to the node (Section 4), and $\delta : A^+ \to \mathbb{Z}$ defines the arc lengths. We generate child nodes by adding new temporal constraints to the network, that is, we obtain each child node by adding a set of arcs E' along with their lengths to $N(E,\delta)$. Since we know the lower bound in node q, the optimal primal, X^*, and dual, S^*, solutions are available and can be used to speed up the computation of the lower bound in each child node. Namely, if E' is the set of arcs added to $N(E,\delta)$ with lengths δ', first we update S^* by these arcs to obtain a dual feasible solution S for $N(E'',\delta'')$, where $E'' = E \cup E'$ and $\delta''_{i,j} := \delta_{i,j}$ for each $(i,j) \in (E \cup A^+) \setminus E'$, $\delta''_{i,j} := \max\{\delta_{i,j}, \delta'_{i,j}\}$ for each $(i,j) \in E' \cap (E \cup A^+)$ and $\delta''_{i,j} := \delta'_{i,j}$ for each $(i,j) \in E' \setminus (E \cup A^+)$. To this end, let $S := S^*$ and we check each arc $(i,j) \in A^+ \cup E \cup E'$ wether $S_j - S_i \geq \delta''_{i,j}$. If not, then update $S_j := S_i + \delta''_{i,j}$. We repeat this procedure until no more changes occur or a positive cycle is found in which case the dual problem has no feasible solution. For details, see [1].

With a feasible dual solution, we can update the primal solution to ensure that the complementary slackness conditions are satisfied. Namely, if $rc_{i,j}(S) < 0$ then let $X_{i,j} = 0$; if $rc_{i,j}(S) > 0$ then let $X_{i,j} = u_{i,j}$; while if $rc_{i,j}(S) = 0$, $X_{i,j} = X^*_{i,j}$. Clearly, X can be computed in linear time in the size of $N(E'',\delta'')$.

Finally, we can apply any variant of the primal-dual maximum cost flow algorithm to X and S to obtain a pair of optimal solutions for $N(E'',\delta'')$.

In practice this method is very efficient as E' usually contains only a few arcs and therefore only a small fraction of the distances $S_i - S_j$, $(i,j) \in A^+ \cup E \cup E'$ change. Therefore, X and X^* usually differ only on a few arcs (cf. definition of $rc_{i,j}(S)$).

3.2 Tightening the Domains of Variables

Suppose the algorithm for solving RCPSPWET has found one ore more feasible solutions (respecting constraints (i) and (ii)). Therefore, a finite upper bound UB is available on the value of the optimal solution. The *domain* of activity $j \in V$ in node q of the search tree is the time interval $[\delta_{0,j}(q), -\delta_{j,0}(q)]$ in which S_j may take a value (see the next section). However, if, say, $t_j > 0$, then possibly $\delta_{0,j}(q) < S_j$ or $S_j < -\delta_{j,0}(q)$ in any feasible solution with objective function value smaller than UB and reachable from node q. We propose a simple procedure for tightening the domains of activities using only the temporal constraints and the upper bound UB.

Consider the network $N(E,\delta) = (V^+, A^+ \cup E, \delta)$ associated with a node of the branch and bound tree. For each activity j, define the function

$$f_j(s) = \min\{v(S) \mid S_j = s \text{ and } S \text{ is dual feasible for } N(E,\delta)\}.$$

Proposition 1. *$f_j(s)$ is convex on the interval $[\delta_{0,j}, -\delta_{j,0}]$.*

With these functions, define the quantities

$$S_j^{\min} = \min\{s \in \mathbb{Z} \mid s \geq \delta_{0,j} \text{ and } f_j(s) \leq UB\}$$

and

$$S_j^{\max} = \max\{s \in \mathbb{Z} \mid s \leq -\delta_{j,0}, \text{ and } f_j(s) \leq UB\},$$

where UB is the best known upper bound. We discuss the computation of $S_j^{\max}$ in detail, the method for $S_j^{\min}$ being similar. After the computations we can set $\delta_{0,j} = S_j^{\min}$ and $\delta_{j,0} = -S_j^{\max}$.

Let X^* and S^* constitute a pair of primal and dual optimal solutions for $N(E, \delta)$. We may assume that $-\delta_{j,0} \in [\delta_{0,j}, \infty]$, otherwise the dual problem is infeasible. If the common primal and dual objective function value is greater than or equal to UB, then the dual feasible solutions of the maximum cost flow problem with respect to $N(E, \delta)$ cannot contain a better solution than the one with cost UB. In this case there is no use to compute time windows. So assume that the optimal objective function value is smaller than UB.

Algorithm 1. Computation of $S_j^{\max}$ (input: $N(E, \delta)$, UB; output: $S_j^{\max}$)

1: Let $\delta_{0,j}^0 = -\delta_{j,0}$, $X_{0,j}^0 = \infty$, $\delta_{i,j}^0 = \delta_{i,j}$ and $X_{i,j}^0 = X_{i,j}^*$ for $(i,j) \in A^+ \setminus \{(0,j)\}$.
2: Apply the *Primal-Dual Maximum Cost Flow algorithm* (sketched below) to $N(E, \delta)$ starting from X^0, and S^*. Stop the algorithm when either the solution becomes primal feasible or the dual objective function value becomes greater than UB after the change of the dual variables. Upon termination, let $\bar{X}$ and $\bar{S}$ denote the values of the primal and the dual variables, respectively.
3: Let S' denote the value of the dual variables before the last increase. If $v(\bar{S}) - UB > 0$, $\lambda = (v(\bar{S}) - UB)/(v(\bar{S}) - v(S'))$, otherwise let $\lambda = 0$. Then, $S_j^{\max} := \lfloor \lambda S'_j + (1-\lambda)\overline{S}_j \rfloor$.

Algorithm 1 for computing $S_j^{\max}$ consists of three main steps. In the first step $\delta_{0,j}$ is increased to $-\delta_{j,0}$. With this modified $\delta_{0,j}$ value, $rc_{0,j}(S^*) > 0$ unless $S_j^* = -\delta_{0,j}$ in which case there is nothing to compute. To maintain the complementary slackness optimality conditions, $X_{0,j}^0$ is set to its upper bound. This way 0 becomes a sink node and j becomes a source node. All other nodes have zero excess.

The *Primal-Dual Maximum Cost Flow algorithm* applied in the second step of the algorithm alternates between sending flow from a source node to a sink node, and increasing the value of a subset $W \subset V^+$ of dual variables taking into account the reduced costs. Namely, it tries to send flow from a source node to a sink node through directed paths in the residual network using only arcs with zero reduced costs. If no more flow can be sent from a source to a sink node, then there is a set of nodes W reachable from a source node through these paths, but W does not contain a sink node. In our case the source node is always j and the sink node is always 0. Then the dual variables corresponding to the nodes in W are increased by the same amount ε such that complementary slackness is maintained. Namely, $\varepsilon = \min\{\varepsilon_1, \varepsilon_2\}$, where $\varepsilon_1 = \min\{rc_{i,j}(S) \mid (i,j) \in A^+, i \in V^+ \setminus W, j \in W, rc_{i,j}(S) > 0\}$ and $\varepsilon_2 = \min\{-rc_{i,j}(S) \mid (i,j) \in A^+, i \in W, j \in V^+ \setminus W, rc_{i,j}(S) < 0\}$.

In the third step, if $v(\bar{S}) \leq UB$, then $S_j^{\max} = \bar{S}_j$. Otherwise, when $v(S') < UB < v(\bar{S})$, the algorithm computes the point between S'_j and $\bar{S}_j$ such that

the dual objective reaches UB. To make this more precise, first notice that the choice of λ ensures $\lambda v(S') + (1-\lambda)v(\bar{S}) = UB$. Moreover, we have:

Lemma 1. *The dual objective function value satisfies* $v(\lambda S' + (1-\lambda)\overline{S}) = \lambda v(S') + (1-\lambda)v(\bar{S})$.

Proof. If $v(\bar{S}) \leq UB$ then $\lambda = 0$ and we are done. So assume $v(\bar{S}) > UB$. Let W be the set of nodes whose dual variable has been changed just before the termination of the algorithm. Then node $n+1 \notin W$. This can be seen by induction as we start out from an optimal solution S^* in which $rc_{n+1,0}(S^*) = 0$. Therefore, if $n+1$ was reachable from node j on a path consisting of arcs with zero reduced costs, node 0 would be reachable as well. Now, since $n+1 \notin W$, the choice of ε and the definition of the $rc_{n+1,i}(S)$ implies that $\varepsilon \leq \min\{d_i - p_i - S_i \mid i \in W \text{ and } S_i < d_i - p_i\}$. Consequently, for $i \in W$ with $S_i \geq d_i - p_i$, increasing S_i by ε increases the dual objective function value by εt_i, and for $i \in W$ with $S_i < d_i - p_i$, increasing S_i by ε decreases the dual objective function value by εe_i. Therefore, the dual objective changes linearly when increasing the value of a subset W of dual variables by the same amount of ε. Consequently, the objective function value changes linearly during the last increase of the dual variables and the statement follows. □

We still have to show the following:

Lemma 2. $v(\lambda S' + (1-\lambda)\overline{S})$ *equals the smallest dual objective function value over all dual feasible solutions* $\tilde{S}$ *with* $\tilde{S}_j \geq \lambda S'_j + (1-\lambda)\overline{S}_j$.

Proof. Since the complementary slackness conditions are maintained by the primal-dual maximum cost flow algorithm, if we let $S = \lambda S' + (1-\lambda)\overline{S}$, $\delta_{0,j} = S_j$, $X_{0,j} = \sum_{(j,i)\in A^+} \bar{X}_{j,i} - \sum_{(i,j)\in A^+\setminus\{(0,j)\}} \bar{X}_{i,j}$, and $X_a = \bar{X}_a$ for $a \in A^+ \setminus \{(0,j)\}$, X and S will be primal and dual feasible with respect to $N(E,\delta)$, and satisfy the complementary slackness conditions. Therefore, X and S constitute a pair of primal and dual optimal solutions with value $v(\lambda S' + (1-\lambda)\overline{S})$ and $S_j = \lambda S'_j + (1-\lambda)\overline{S}_j$. □

If $\lambda S'_j + (1-\lambda)\overline{S}_j$ is fractional, then clearly, we can round it down to obtain $S_j^{\max}$.

The algorithm performs at most $n + |A^+|$ dual changes, and between two changes it solves a maximum flow problem with successive augmenting paths (which is theoretically not efficient). Clearly, the maximum flow problem may be solved by any polynomial time algorithm and thus the entire algorithm may be implemented to run in polynomial time. However, in practice, our method is quite efficient as there are only a few flow augmentations between any two changes of dual variables. We illustrate all this by means of a small example:

Example 1. A simple AoN-network is depicted in Figure 1. For the sake of simplicity we suppose that i) each arc is an end-to-start precedence relation ($\delta_{i,j} = p_i$) and ii) the earliness and tardiness costs are equal for each activity ($e_j = t_j$). The optimal solution (of the resource-relaxed problem) is

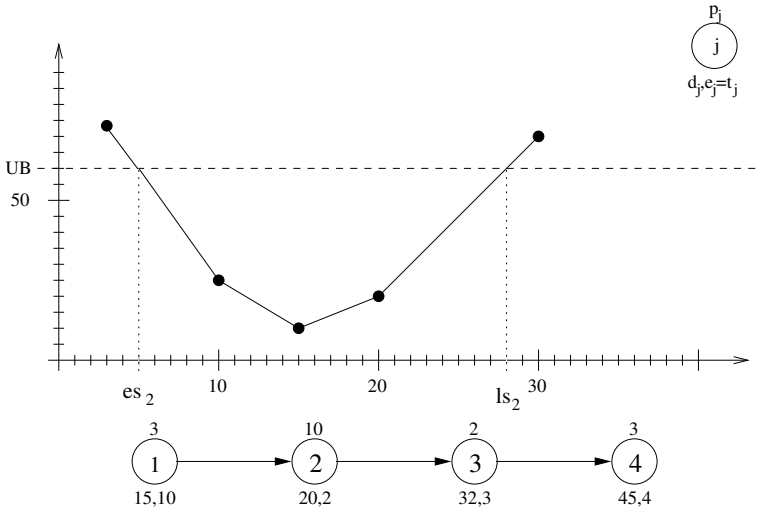


Fig. 1. Illustration of computing $S_j^{\max}$

$S^* =< 12, 15, 30, 42 >$ with cost $c = 10$. Assume that we have an upper bound $UB = 60$.

To compute $S_2^{\max}$, we start to shift activity 2 to the right. We can shift it until $S_2 = 20$, where arc $(2, 3)$ becomes tight, thus $\varepsilon = 5$. The new schedule is $S^1 =< 12, 20, 30, 42 >$, the cost is $c = 30$. Since this is smaller than the upper bound, we continue with shifting activities $\{2, 3\}$, until arc $(3, 4)$ becomes tight ($S_3 = 40$). $\varepsilon = 8$, the schedule is $S^2 =< 12, 30, 40, 42 >$ with cost $c = 70$. $c > UB$, so we can calculate $S_2^{\max}$ from S_2^1 and S_2^2. Regarding that the per-unit tardiness cost of activities $\{2, 3\}$ together is $t_2 + t_3 = 5$, we get that $S_2^{\max} = 28$, and the corresponding schedule is $< 12, 28, 38, 42 >$ with cost $c = 60$.

4 Exact Algorithm Based on Branch and Bound

A subset C of activities are in conflict, or C is a *conflicting set*, if there exists $k \in R$ with $\sum_{i \in C} r_{i,k} > b_k$. Two activities, i and j, are *resource-disjunct* if $\{i, j\}$ is a conflicting set. If $\mathcal{A}(S, t)$ is a conflicting set, then the inclusion-wise minimal subsets B of $\mathcal{A}(S, t)$ such that $\mathcal{A}(S, t) \setminus B$ is non-conflicting are called *minimal delaying alternatives* [5]. Algorithm 2 is an extension of Algorithm 3.5.1 of [15] for project scheduling with general precedence relations and non-regular objective functions (see also [6] and [14]). The extension consists of the application of domain filtering techniques in lines (15)-(18).

In Algorithm 2, L is a list of nodes and S_{best}^* is the best resource feasible solution found so far. At the beginning, L contains only the root node and S_{best}^* is empty. The search proceeds in a depth-first manner by always choosing the last node q appended to the end of L. Each node is processed only once. First

Algorithm 2. Branch-and-Bound (input: UB, output: S^*_{best})

```
1:  L := {r}, S*_best := ∅
2:  Compute all-pairs-longest-paths δ(r)
3:  while L ≠ ∅ do
4:     Remove the last node q from L
5:     if S*(q) is resource feasible then
6:        if v(S*(q)) < UB then
7:           UB := v(S*(q)), S*_best := S*(q)
8:        end if
9:     else
10:       Determine the earliest time point t when A(S*(q), t) is conflicting
11:       K := ∅
12:       for each minimal delaying alternative B ⊂ A(S*(q), t) do
13:          for all i ∈ A(S*(q), t) \ B do
14:             Generate node q' from node q with additional temporal constraints (i, j)
                with δ_{i,j}(q') = p_i for each j ∈ B
15:             if UB < ∞ then
16:                Tighten the variable domains with Network Computations
17:                Tighten the variable domains with Unit-Interval Tests and Edge-
                   Finding
18:             end if
19:             Compute all-pairs-longest-paths δ(q')
20:             Perform Immediate-Selection for resource-disjunct activities
21:             Re-compute S*(q')
22:             if node q' is infeasible or dominated, or v(S*(q')) ≥ UB then
23:                Delete node q'
24:             else
25:                K := K ∪ {q'}
26:             end if
27:          end for
28:       end for
29:       Append the nodes q' ∈ K to L in non-increasing order of v(S*(q'))
30:    end if
31: end while
```

it is checked whether the solution associated with the node, $S^*(q)$, is resource feasible. If it is, the node will be fathomed. But before, it is checked whether $S^*(q)$ represents a better solution than S^*_{best} (line 6). If $S^*(q)$ is not resource feasible, then the algorithm finds the first time point t when $\mathcal{A}(S^*(q), t)$ is conflicting. Then for each minimal delaying alternative with respect to $\mathcal{A}(S^*(q), t)$, a new node q' is generated (line 14) [8], [6]. If there is a feasible solution, i.e., $UB < \infty$, we tighten the time windows of activities by the network computations described in Section 3.2 and also by two well-known domain filtering techniques: the Unit-Interval Tests and Edge-Finding (see [16], [2] and [7]), lines (15)-(18). All these domain filtering techniques are invoked only once for each activity, and not until a fixpoint is reached. We have found by experimentation that it does not pay off repeating these computations until no more changes occur. Then we

compute the longest paths between all pairs of activities in $V \cup \{0\}$ and store it in the $(n+1) \times (n+1)$ matrix $\delta(q')$. Using $\delta(q')$, we perform immediate selections between pairs of resource-disjunct activities. To this end, we apply the techniques of [15], pages 53-57 (see also [14]) which extend those of [4] and [6]. Notice that in the end of the computations, $\delta_{0,i}(q')$ equals the earliest start time of activity i, while $-\delta_{i,0}(q')$ is its latest start time. The node is *infeasible* if $\delta_{0,i}(q') + \delta_{i,0}(q') > 0$, or $\delta_{i,i}(q') > 0$ hold for any $i \in V$. The child node is dropped if it is infeasible, or dominated by some node already explored in the tree (we apply the subset dominance rule of [14] which is a strengthening of that proposed in [6]), or its lower bound is not smaller than the current upper bound (line 23). Those child nodes that are not deleted get appended to L in non-increasing order of their lower bounds (line 29).

We apply Algorithm 2 in two phases. In the first phase, when no feasible solution is known, we invoke the algorithm with $UB = \infty$ and stop it immediately when it encounters the first resource and time feasible solution, or proves that the problem is infeasible. If a feasible solution is found, we invoke Algorithm 2 the second time with UB equal to the value of the known feasible solution. Notice that the domain filtering techniques in lines (15)-(18) become active only during the second invocation.

5 Computational Results

To evaluate the performance of our algorithm, we used two groups of RCPSP/max test instances, UBO20 and UBO50 generated by ProGen/max [17]. Since these test instances were generated for the makespen objective, we complemented each of them by due-date and per unit earliness and tardiness cost information. Namely, for an instance with n activities we generated n random numbers chosen uniformly from the interval $[1, 1.5C_{\max}]$, where $C_{\max}$ is the makespan of the resource-relaxed problem. Then we sorted these numbers in ascending order and assigned the i-th member of the sorted list to the i-th activity of the instance. The per unit earliness and tardiness costs were chosen uniformly at random from the interval $[1, 100]$. We implemented the complete algorithm in C++ . We run the tests on a PC with 2.4 GHz Xeon CPU, using Windows 2000 operating system. To make a fair comparison with the method of Schwindt [18], we also solved all instances with his program which he kindly provided to us. In all of the following tables, column "TW" represents our algorithm.

5.1 Results on UBO20 Instances

All of these instances consists of 20 activities. We run both algorithms with a 100-second time limit. Out of the 90 instances, 20 were infeasible. As we can see in Table 1, our solver (TW) was capable to solve all the 70 feasible instance within 22 seconds at most, while the code of Schwindt exceeded the 100-second time limit in three cases.

Having no access to the number of search-tree nodes generated by the solver of Schwindt, we do not provide a comparison of the sizes of search-trees.

Table 1. Results on UBO20 instances

	TW	Schwindt
# inst. solved to opt. in $100s$	70	67
max solution time	$21.97s$	$> 100s$[a]
avg. solution time[b]	$0.22s$	$1.32s$

[a] There were 3 instances which took more than $100s$.
[b] On instances both solver solved to optimality.

5.2 Results on UBO50 Instances

Each of the 90 instances in this group consists of 50 activities, and we performed a more thorough evaluation. We run the tests with 100, 200, 400-second time limits. Out of the 90 instances, there were 17 instances which did not admit a feasible solution. Both algorithms proved infeasibility of these instances within a few seconds. For all other instances our algorithm found a feasible solution already within 100 seconds. However, the method of Schwindt did not find a feasible solution for 3 additional instances within $100s$, and for 1 additional instance within $200s$, while it found a feasible solution for all feasible instances within $400s$, see Table 2.

Table 2. Number of feasible UBO50 instances for which no feasible solution was found

Time limit	TW	Schwindt
$100s$	0	3
$200s$	0	1
$400s$	0	0

Table 3 shows the number of instances solved to optimality by the algorithms. Our solver was able to solve more instances to optimality with respect to all three time limits. Moreover, as the time limit increased, our solver solved more new instances to optimality than that of Schwindt.

Table 3. Number of UBO50 instances solved to optimality.

Time limit	TW	Schwindt	Both
$100s$	32	24	24
$200s$	35	26	26
$400s$	39	28	28

Table 4 depicts the average running times on those instances solved to optimality by both solvers within the respective time limits. As can be seen, our solver is faster on average.

Table 4. Average solution times on UBO50 instances

Time limit	TW	Schwindt
100s	8.64s	22.38s
200s	11.89s	33.01s
400s	13.32s	53.19s

Finally, Table 5 compares the algorithms on those instances not solved to optimality by either of two solvers. The columns "TW", "Equal", and "Schwindt" provide the number of instances (within this set of instances) on which our algorithm found a better solution, the two solvers found solutions of equal total tardiness, and the program of Schwindt found a better solution, respectively, within the given time limits. Our method clearly performs better than that of Schwindt in this respect, too.

Table 5. UBO50: Number of better non-optimal solutions found

	TW	Equal	Schwindt
100s	22	13	6
200s	21	12	5
400s	21	8	5

The above results show that our solver outperforms that of Schwindt in all aspects examined.

6 Conclusions

In this paper we have described a simple method for computing tight time windows for activities when solving the RCPSPWET problem which enables us to apply many domain filtering techniques known in the Constraint-based Scheduling community. Our preliminary computational results indicate that it is worthwhile to make these computations as our method clearly outperforms the only method published in the literature.

A major question is how to speed up our method. We have observed that in 60%-70% of the cases our method is able to reduce the domains of activities and this is independent from the depth of the node in the search-tree. Nevertheless, it would be interesting to find out in advance for which activities to perform the computations in a search-tree node.

Acknowledgments. The authors are grateful to the three anonymous referees for helpful comments that helped to improve the presentation.

References

1. R.K. Ahuja, T.L. Magnanti, J.B. Orlin, *Network Flows: Theory, Algorithms and Applications*, Prentice Hall, Englewood Cliffs, New Jersey, 1993.
2. Ph. Baptiste, C. Le Pape, W.P.M. Nuijten, *Constraint-Based Scheduling. Applying Constraint Programming to Scheduling Problems*, Kluwer Academic Publishers, Boston, 2001.
3. J.C. Beck, P. Refalo, A hybrid approach to scheduling with earliness and tardiness costs, *Annals of Operations Research*, 118 (2003) 49-71.
4. P. Brucker, S. Knust, A. Schoo, O. Thiele, A branch and bound algorithm for the resource-constrained project scheduling problem, *Eur. J. Oper. Res.* 107 (1998) 272-288.
5. E.L. Demeulemeester, W.S. Herroelen, A branch and bound procedure for the multiple resource constrained project scheduling problem, *Management Sci.* 38 (1992) 1803-1818.
6. B. De Reyck, W.S. Herroelen, A branch and bound procedure for the resource-constrained project scheduling problem with generalized precedence relations, *Eur. J. Oper. Res.* 111 (1998) 152-174.
7. U. Dorndorf, *Project scheduling with time windows: From theory to applications*, Physica-Verlag, 2002.
8. O. Icmeli, S. Erengüç, A branch and bound procedure for the resource-constrained project scheduling problem with discounted cash flows, *Management Sci.* 42 (1996) 1395-1408.
9. A. Kéri, T. Kis, Primal-dual combined with constraint propagation for solving RCPSPWET. In: G. Kendall, L. Lei and M. Pinedo (eds.), *Proc. of the 2nd Multidisciplinary International Conference on Scheduling: Theory and Applications*, New York University, New York, July, 2005, pp. 748-751 (electronic edition).
10. A. Kéri, T. Kis, Primal-Dual combined with constraint propagation for solving RCPSPWET, In: H-D. Haasis, H. Kopfer, J. Schönberger (eds.), *Operation Research Proceedings*, September 7-9, 2005, Bremen, Germany, Springer, pp. 685-690.
11. L. Khatib, P. Morris, R. Morris, F. Rossi, Temporal Constraint Reasoning with Preferences, *Proc. Seventieth Int. Joint Conf. Artif. Intell. (IJCAI'01)*, August 4-10, 2001, Seattle, USA, pp. 322-327.
12. T.K.S. Kumar, Fast (incremental) algorithms for useful classes of simple temporal problems with preferences, *Proc. Twentieth Int. Joint Conf. Artif. Intell. (IJCAI'07)*, January 6-12, 2007, Hyderabad, India, pp. 1954-1959.
13. P. Morris, R. Morris, L Khatib, S. Ramakrishnan, A. Bachmann, Strategies for global optimization of temporal preferences, *LNCS 3258: Proc. Tenth Int. Conf. Princip. Practice Artif. Intell. (CP 2004)*, Toronto, Canada, September 27-October 1, 2004, pp. 408-422.
14. K. Neumann, J. Zimmermann, Exact and truncated branch and bound procedures for resource-constrained project scheduling with discounted cash flows and general temporal constraints, *Central Eur. J. of Oper. Res.*, 10:4 (2002) 357-380.
15. K. Neumann, C. Schwindt, J. Zimmermann, *Project Scheduling with Time Windows and Scarce Resources*, 2nd ed. Springer, Berlin, 2003.
16. W.P.M. Nuijten, *Time and resource constrained scheduling: A constraint satisfaction approach*, PhD Thesis, Eindhoven University of Technology, 1994.

17. C. Schwindt, ProGen/max: A New Problem Generator for Different Resource-Constrained Project Scheduling Problems with Minimal and Maximal Time Lags, *Technical report* WIOR-449, University of Karlsruhe, Karlsruhe, 1996. (URL: http://www.wior.uni-karlsruhe.de/LS_Neumann/Forschung/ProGenMax/rcpspmax.html)
18. C. Schwindt, Minimizing earliness-tardiness costs of resource constrained projects, In: K. Inderfurth, G. Schwödiauer, W. Domschke, F. Juhnke, P. Kleinschmidt, G. Wäscher (eds.), *Operations Research Proceedings* 1999. Springer, Berlin, 2000, pp. 402-407.
19. C. Schwindt, J. Zimmermann, A steepest ascent approach to maximizing the net present value of projects, *Math. Methods of Oper. Res.* 53 (2001) 435-450.
20. M. Vanhoucke, E.L. Demeulemeester, W.S. Herroelen, An exact procedure for the resource-constrained weighted earliness-tardiness project scheduling problem, *Annals of Oper. Res.* 102 (2001) 179-196.
21. M. Wennink, *Algorithmic support for automated planning boards*, PhD Thesis, Eindhoven University of Technology, 1995.

Necessary Condition for Path Partitioning Constraints

Nicolas Beldiceanu and Xavier Lorca

École des Mines de Nantes, LINA FRE CNRS 2729, FR – 44307 Nantes Cedex 3
{Nicolas.Beldiceanu,Xavier.Lorca}@emn.fr

Abstract. Given a directed graph $\mathcal{G}$, the K node-disjoint paths problem consists in finding a partition of $\mathcal{G}$ into K node-disjoint paths, such that each path ends up in a given subset of nodes in $\mathcal{G}$. This article provides a necessary condition for the K node-disjoint paths problem which combines (1) the structure of the reduced graph associated with $\mathcal{G}$, (2) the structure of each strongly connected component of $\mathcal{G}$ with respect to dominance relation between nodes, and (3) the way the nodes of two strongly connected components are inter-connected. This necessary condition is next used to deal with a path partitioning constraint.

1 Introduction

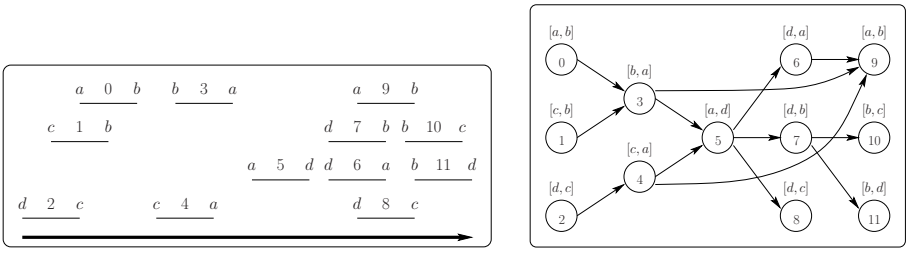
Graph node-partitioning constraints are ubiquitous in many practical applications such as *vehicle-routing* [2] or *network robustness* [17]. Partitioning patterns are usually cycles [3,10], trees [4,12] or paths [13,15]. Within the context of a constraint that partitions a digraph into node-disjoint paths (named *path partitioning* constraint in the following), necessary conditions preventing circuits and enforcing the fact that each node has no more than one predecessor were already introduced in [5,13,15]. However, none of these necessary conditions really consider the number of paths to build. The aim of this article is to come up with necessary conditions related to the number of paths required to partition a directed graph.

Consider a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, a *path partition* of $\mathcal{G}$ is a collection of node-disjoint paths $P_1 = (V_1, A_1), \ldots, P_K = (V_K, A_K)$ in $\mathcal{G}$ whose union is $\mathcal{V}$, i.e., $V_i \cap V_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=1}^{K} V_i = \mathcal{V}$. The K *node-disjoint paths problem* [16,18] (denoted K-NDP in the following) determines a path partition of $\mathcal{G}$ such that $\mathtt{K} \in [\underline{\mathtt{K}}, \overline{\mathtt{K}}]$, where $\underline{\mathtt{K}}$ is the minimum number of paths in a path partition of $\mathcal{G}$ and $\overline{\mathtt{K}}$ is the size of a given subset $\mathcal{T} \subseteq \mathcal{V}$ of *potential final nodes* for paths.

Finding a compatible path partition of $\mathcal{G}$ is NP-complete [9,16], even when fixing the number of paths to $\mathtt{K} = 2$. However, there are many known particular graphs classes on which the path partition problem is solvable in polynomial time [1,14,19]. Within the context of a general path partitioning constraint, we cannot make any assumption on the graph class of $\mathcal{G}$ but, we still want to come up with necessary conditions based on the structure of $\mathcal{G}$ (e.g., the reduced graph[1] of $\mathcal{G}$, the dominance relation [11,13] between nodes of $\mathcal{G}$).

[1] The *reduced graph* G_r of a given directed graph G is derived by the following transformation of G: To each strongly connected component (*scc*) of G we associate a node of G_r, to each arc of G that connects different *scc*'s corresponds an arc in G_r.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 141–154, 2007.



(a) **(b)**

Fig. 1. Part (a) A set of fixed tasks in time, where beside its position on the time axis, each task t is described by a triple s, i, d which respectively denotes the *start place*, the *identifier*, and the *destination place* of task t. Part (b) gives the corresponding digraph.

The K node-disjoint paths problem has many practical uses. A typical application of the problem consists to cover a set of tasks (e.g., deliveries, flights) by a minimum number of resources (e.g., lorries, planes). Each task is defined as an interval with an *earliest start*, a *latest end* and a *fixed duration*. In addition, a task has also a *start place* and a *destination place*. Within a solution, a task t_1 can immediately precede a task t_2 if (1) the earliest end of t_1 precedes the latest start of t_2, and if (2) the destination place of t_1 is equal to the start place of t_2.

Example 1. As an instance of the previous application, consider the set of fixed tasks depicted by Figure 1a. Figure 1b provides the corresponding graph $\mathcal{G}$ which can be covered by 6 node-disjoint paths. Observe that $\mathcal{G}$ is acyclic since all tasks are completely fixed in time.

In the context of paths partitioning constraints [5,7,13], the contribution of this article is to show how to combine *network flow* and *dominance* theory to get a general necessary condition for the path partitioning constraint. In the following, we first introduce, in Section 2, some recalls on network flow theory provided by [6, p.72]. Next, in Section 3, a flow-based approach is studied in the case of directed acyclic graphs (DAGs). Then, Section 4 generalizes our flow-based approach to the non-acyclic case, dealing with bottlenecks of the reduced graph of the graph to partition: Sections 4.1 and 4.2 show how to get tighter bounds on the arcs of the network according to (1) the dominance relation between nodes of the graph to partition and (2) the way the nodes of two strongly connected components are inter-connected. Next, Section 5 shows how to partially exploit the flow-based relaxation of the K node-disjoint paths problem to improve the filtering related to a path partitioning constraint.

2 Preliminaries

We first recall three definitions taken from [6, p.72] in order to define basic notions of network flow theory. Moreover, we recall the well-known Hoffman theorem that generalised the flow-conservation to any cocycle of a network. These notions will be used throughout the article.

Definition 1. *Let A be a set of nodes of a graph $G = (U, E)$, let $\omega^+(A)$ be the set of emanating arcs from A, and let $\omega^-(A)$ be the set of entering arc in A. A* cocycle *is the set of arcs $\omega(A) = \omega^+(A) \cup \omega^-(A)$.*

Definition 2 (Network)
Let $N = (U \cup \{s, t\}, E)$ be a directed graph for which every arc $(i, j) \in E$ has a non-negative, integer-valued lower bound *l_{ij} and a non-negative, integer-valued* capacity *c_{ij} (greater than or equal to l_{ij}). For each node i, distinct from s and t, there is a path in N from s to i and a path from i to t. There is also an arc from t to s, named* backward *arc of N.*

Definition 3 (Flow)
A flow *of a network $N = (U \cup \{s, t\}, E)$ is defined by an integer function $f : E \mapsto \mathbb{N}$ such that:*

1. Capacity constraints*: For any $(i, j) \in E$, $l_{ij} \leq f(i, j) \leq c_{ij}$.*
2. Flow conservation*:*

$$\forall x \in U, \sum_{(i,x)\in\omega^-(\{x\})} f(i, x) = \sum_{(x,j)\in\omega^+(\{x\})} f(x, j) \tag{1}$$

A global flow *$\mathcal{F}(N)$ of a network N is provided by:*

$$\mathcal{F}(N) = \sum_{(s,j)\in\omega^+(\{s\})} f(s, j) \tag{2}$$

Theorem 1 (Hoffman). *Given a network $N = (U, E)$ defined by an integer function $f : E \mapsto \mathbb{N}$ such that for all $(i, j) \in \mathcal{E}$, $f_{ij} \in [\ell_{ij}, c_{ij}]$, there exists a feasible flow in N iff for any cocycle $\omega(A)$ in N, we have:*

$$\sum_{(i,j)\in\omega^+(A)} c_{ij} - \sum_{(i,j)\in\omega^-(A)} \ell_{ij} \geq 0 \tag{3}$$

3 K-NDP Problem in Directed Acyclic Graphs

This Section focuses the case of directed acyclic graphs for which the K-NDP problem can be solved in polynomial time [1,14,19]. When the number of paths K is not fixed, the key point of any approach solving the K-NDP problem is the evaluation of the lower bound on the minimum number of paths for partitioning the digraph $\mathcal{G}$.

A first way to evaluate a lower bound considers the size of a maximum antichain[2] (also named *width*) of the digraph $\mathcal{G}$ (see [8] for a characterisation). Indeed, once the antichain is traversed, there is no way back since $\mathcal{G}$ is acyclic. However, even if the width of $\mathcal{G}$ constitutes a sharp lower bound for the path covering problem (see [14, p.219]), this is not true anymore in the context of the node-disjoint paths problem. Next example illustrates this point:

[2] An *antichain* in a partially ordered set P is a subset A of P such that every pair of members of A is incomparable, i.e., for any x, y in A, neither $x \leq y$ nor $y \leq x$ (in our context, there is neither a path from x to y, nor a path from y to x).

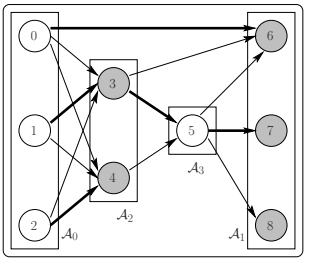


Fig. 2. A directed acyclic graph $\mathcal{G}$, where potential final nodes are depicted in grey. Thick arcs depict a path partition of $\mathcal{G}$ minimising the number of node-disjoint paths.

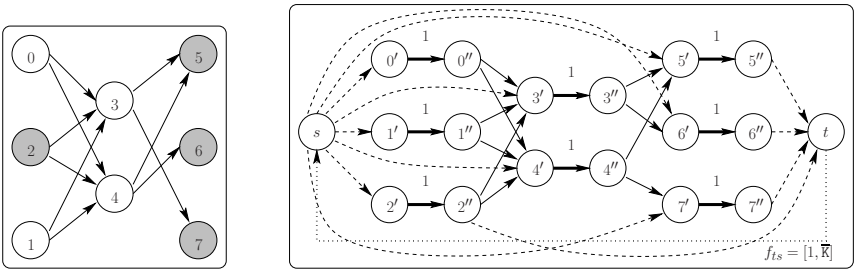
Example 2. In Figure 2, from the maximum antichains $\mathcal{A}_0 = \{0, 1, 2\}$ and $\mathcal{A}_1 = \{6, 7, 8\}$ of $\mathcal{G}_r$, the width of $\mathcal{G}$ is equal to 3 and then, there exists a path covering of $\mathcal{G}$ with 3 paths (e.g., $\langle 0, 6\rangle$, $\langle 1, 3, 5, 7\rangle$, $\langle 2, 4, 5, 8\rangle$). But observe that, in the context of a path partition, the antichains $\mathcal{A}_2 = \{3, 4\}$ and $\mathcal{A}_3 = \{5\}$ constitute a bottleneck between any node of $\mathcal{A}_0$ and any node of $\mathcal{A}_1$. Thus, partitioning $\mathcal{G}$ with 3 node-disjoint paths is clearly impossible. In fact, 4 node-disjoint paths are required to partition $\mathcal{G}$ (e.g., $\langle 0, 6\rangle$, $\langle 1, 3, 5, 7\rangle$, $\langle 2, 4\rangle$, and $\langle 8\rangle$).

Within the context of directed acyclic graphs, this section introduces a classical flow--based approach (Example 3) in order to provide a necessary and sufficient condition for the K-NDP problem, as well as a sharp lower bound on the minimum number of node-disjoint paths partitioning $\mathcal{G}$. In this approach, one has to build from every K-path partition of the graph, a K-flow in a derived graph. A simple idea is to add for each node i two arcs (s, i) and (i, t) (where s and t are two extra-nodes). Then, for every path $\langle i_1, \ldots, i_n\rangle$ of the K-partition, one can add a flow arc (s, i_1) and a flow arc (i_n, t), which leads to a K-flow. The problem of this construction is that the resulting graph always admit a zero flow, which leads to a trivial lower bound. In order to impose a non-null flow through each node of the graph we split each node i into i' and i'' and add the arc (i', i''). Thus, because of the flow conservation, there will be a non-null flow from i' to i''.

Example 3. Figure 3 illustrates Definitions 4 and 5. Each node i of Figure 3a is splitted in two nodes i' and i'' in Figure 3b. The dotted arc of Figure 3b corresponds to the *backward* arc of Definition 5, with a $[1, \overline{\mathtt{K}}]$ flow, thick arcs correspond to *node-splitting* arcs, with an unit flow, dashed arcs to *path-extremities* arcs, with a $[0, \overline{\mathtt{K}}]$ flow, and plain arcs to *dag's* arcs, with a $[0, \overline{\mathtt{K}}]$ flow.

Definition 4. *Given a directed acyclic graph $\mathcal{G}$, the network $\mathcal{N} = (\mathcal{X}, \mathcal{E}, c)$ is defined in the following way:*

- *To each node i of $\mathcal{G}$ corresponds two nodes i' (named* input *of node i) and i'' (named* output *of node i) of $\mathcal{X}$, as well as a* node-splitting *arc (i', i'') of $\mathcal{E}$. Let $\mathcal{I}'$ and $\mathcal{I}''$ respectively denote the two corresponding set of nodes. Finally, two additional nodes s (*source*) and t (*sink*) belong to $\mathcal{X}$.*



(a) $\mathcal{G}$ **(b)** $\mathcal{N}$

Fig. 3. For the digraph $\mathcal{G}$ depicted by Part (a), Part (b) provides the corresponding network $\mathcal{N}$. Dashed arcs represent *path-extremities* arcs, thick arcs represent *node-splitting* arcs, the dotted arc (t, s) represents the *backward* arc of the network, and plain arcs represent *dag's* arcs.

- *To each arc* (i, j) *of* $\mathcal{G}$ *corresponds an arc* (i'', j') *of* $\mathcal{E}$*, where* i'' *is the output node of* i *and* j' *the input node of* j*.*
- *There is an arc* (s, i') *in* $\mathcal{E}$*, for all input nodes* $i' \in \mathcal{I}'$*, as well as an arc* (i'', t) *for all output nodes* $i'' \in \mathcal{I}''$ *of a node* i *that belongs to* $\mathcal{T}$*.*
- *A* backward *arc from* t *to* s *is added.*

We are now in position to define the function f associated with the network $\mathcal{N}$. For this purpose, we distinguish several classes of arcs respectively corresponding to *backward*, *intra-scc*, *inter-scc*, and *path-extremities* arcs.

Definition 5. *Given a directed acyclic graph* $\mathcal{G}$*, the flow on each arc of the corresponding network* $\mathcal{N}$ *is defined in the following way:*

1. Backward *arc:* $f(t, s) = f_{ts} \in [1, \overline{\mathtt{K}}]$*, where* $\overline{\mathtt{K}} = |\mathcal{T}|$*.*
2. Node-splitting *arcs:* $f(i', i'') = f_i = 1$*.*
3. Dag's *arcs:* $f(i'', j') = f_{ij} \in [\ell_{ij}, c_{ij}]$ *where* $\ell_{ij} = 0$ *and* $c_{ij} = \overline{\mathtt{K}}$*.*
4. Path-extremities *arcs:* $f(s, i) = f(i, t) = f_{si} = f_{it} \in [0, \overline{\mathtt{K}}]$*, for all* (s, i) *and* (i, t) *in* $\mathcal{E}$*.*

Observe that any cycle in $\mathcal{N}$ contains the arc (t, s), because $\mathcal{G}$ is a directed acyclic graph. Then, according to flow conservation, the global flow of $\mathcal{N}$ is provided by:

$$\mathcal{F}(\mathcal{N}) = \sum_{(s,j) \in \omega^+(\{s\})} f_{sj} = f_{ts} \tag{4}$$

In the following, the global flow of $\mathcal{N}$ corresponds to the flow f_{ts} on backward arc (t, s).

Theorem 2. *A directed acyclic graph* $\mathcal{G}$ *can be partitioned in* $\mathtt{K}$ *node-disjoint paths iff there exists a feasible flow* $f_{ts} = \mathtt{K}$ *in the network* $\mathcal{N}$*.*

Proof. We ensure the condition is necessary. Assume (a) there exists a path partition of $\mathcal{G}$ of size $\mathtt{K} \in [1, \overline{\mathtt{K}}]$ and (b) there does not exist a feasible flow in $\mathcal{N} = (\mathcal{X}, \mathcal{E})$ with

$\mathcal{F}(\mathcal{N}) \in [1, \overline{\mathtt{K}}]$. We show there is a contradiction. By assumption of (b), there does not exist a feasible flow in $\mathcal{N}$ with $\mathcal{F}(\mathcal{N}) = \mathtt{K}$ then, there exists a set of nodes $A \subseteq \mathcal{X}$ such that $s \in A$, for which Hoffman theorem ensures:

$$U_{\omega^+(A)} = \sum_{(i,j) \in \omega^+(A)} c_{ij} < \sum_{(i,j) \in \omega^-(A)} \ell_{ij} = U_{\omega^-(A)} \tag{5}$$

For the cocycle $\omega(A)$, (t, s) is the unique arc entering a node of A (because $s \in A$ and $\mathcal{N} \setminus \{(t, s)\}$ is acyclic). Moreover by definition of $\mathcal{N}$, $f_{ts} \in [1, \overline{\mathtt{K}}]$, thus:

$$1 \leq U_{\omega^+(A)} < U_{\omega^-(A)} \leq \overline{\mathtt{K}} \tag{6}$$

This means the minimum flow entering $\omega(A)$ exceeds the maximum capacity emanating from $\omega(A)$. The contradiction is obvious by considering that $\omega(A)$ is cycle free.

Then, we prove that the condition is sufficient. We show that, from any feasible flow of $\mathcal{N}$, $\mathcal{F}(\mathcal{N}) = f_{ts} = \mathtt{K}$, we can build a partition of $\mathcal{G}$ in $\mathtt{K}$ node-disjoint paths. For this purpose, consider a feasible flow f_{ts} in $\mathcal{N}$ and a DAG $\mathcal{N}' = (\mathcal{X}', \mathcal{E}')$ defined from the flow f_{ts} in the following way:

- $\mathcal{X}' = \mathcal{X} \setminus \{s, t\}$.
- $\mathcal{E}' = \{(i, j) \in \mathcal{E} \setminus (t, s) \mid f_{ij} = 1\}$.

The flow f_{ts} ensures $\mathcal{N}'$ is a DAG of $\mathtt{K}$ connected components such that each one is an elementary path. This property is directly derived from the fact that for each dag's arc (i, j) of $\mathcal{N}$, either $f_{ij} = 1$ or $f_{ij} = 0$ (because in the case of DAGs each node-splitting arc (i', i'') as a flow $f_i = 1$). Then, by contracting each node-splitting arc of $\mathcal{N}'$ in one single node, $\mathcal{N}'$ becomes a partial graph of $\mathcal{G}$ induced by $\mathcal{E}' \subseteq \mathcal{A}$, and composed by $\mathtt{K}$ node-disjoint paths. □

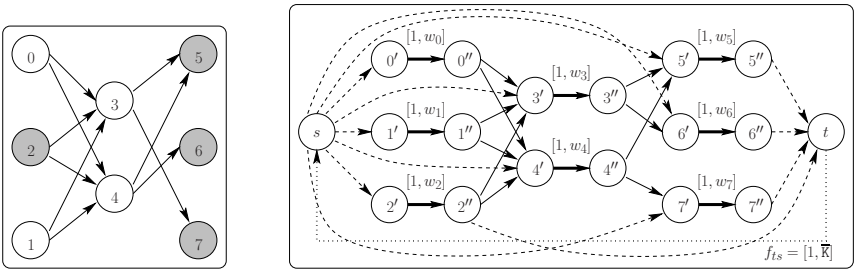
A lower bound on the number of the node-disjoint paths partitioning $\mathcal{G}$ is introduced by the following Lemma:

Lemma 1. *Given a directed acyclic graph $\mathcal{G}$ and its corresponding network $\mathcal{N}$, the lower bound on the number of node-disjoint paths partitioning $\mathcal{G}$ is given by the minimum feasible flow $f^\star_{ts}$ of $\mathcal{N}$: $f^\star_{ts} = \underline{\mathtt{K}}$.*

Proof. Directly provided by Theorem 2. □

4 K-NDP Problem in Non-acyclic Graphs

As we saw in the previous section, in the case of a non-fixed k, the key point of any approach solving the K-NDP problem is the evaluation of a lower bound on the minimum number of paths for partitioning the digraph $\mathcal{G}$. Indeed, consider a digraph $\mathcal{G}$ and a set of potential final nodes $\mathcal{T}$. From any path partition $\mathcal{P}$ of size $p \in [1, |\mathcal{T}|]$, one can build a path partition $\mathcal{P}'$ of size $p' \in [p + 1, |\mathcal{T}|]$ by eventually decomposing one of its path with respect to its potential final nodes. However, finding from $\mathcal{P}$, a path partition $\mathcal{P}''$ of size $p'' \in [1, p - 1]$ is NP-complete (assume $p = 2$ then, $p'' = 1$ and as a consequence, we have to build a Hamiltonian path).



(a) $\mathcal{G}_r$ **(b)** $\mathcal{N}$

Fig. 4. For the digraph $\mathcal{G}_r$ depicted by Part (a), Part (b) provides the corresponding network $\mathcal{N}$. Dashed arcs represent *path-extremities* arcs, thick arcs represent *node-splitting* arcs, dotted arc (t, s) represents the *backward* arc of the network.

In Section 3, a flow model was proposed in a case of directed acyclic graph. This section shows how to improve this model by considering the reduced graph $\mathcal{G}_r$ associated to the digraph $\mathcal{G}$. A node of the reduced graph $\mathcal{G}_r$ corresponds to a strongly connected component of the initial graph $\mathcal{G}$. Therefore we will need distinguish dag's arcs (i.e., the arcs of $\mathcal{G}_r$) and node-splitting arcs (i.e., the arcs that represents *scc*'s of $\mathcal{G}$) and apply the lower bound to the former. Thus, Definitions 4 and 5 are extended to the case of non-acyclic graphs in the following way (Figure 4):

- The network $\mathcal{N} = (\mathcal{X}, \mathcal{E}, c)$ is derived from the reduced graph $\mathcal{G}_r$ associated to $\mathcal{G}$.
- The flow associated to each node-splitting arc (i', i'') of $\mathcal{N}$ is now evaluated as follows: $f(i', i'') = f_i \in [\ell_i, c_i]$ for each node-splitting arc (i', i'') associated with a *scc* C_i of $\mathcal{G}$, where ℓ_i (resp. c_i) corresponds to a lower bound (resp. upper bound) on the minimum (resp. maximum) number of node-disjoint paths partitioning C_i. We have $f_i \in [1, w_i]$ where w_i denotes the number of nodes in C_i which are either potential final nodes in $\mathcal{G}$, or nodes directly connected, by one single arc of $\mathcal{G}$, to another *scc* C_j of $\mathcal{G}$ ($j \neq i$).

Then, the necessary and sufficient condition introduced by Theorem 2 can be directly generalized to a necessary condition in the case of non-acyclic graphs:

Theorem 3. *If there exists a path partition of $\mathcal{G}$ of size $\mathtt{K} \in [\underline{\mathtt{K}}, \overline{\mathtt{K}}]$ then, there exists a feasible flow $f_{ts} = \mathtt{K}$ in the network $\mathcal{N}$ corresponding to $\mathcal{G}$.*

Similarly, the sharp lower bound introduced by Lemma 1 is generalized to a non-sharp lower bound on the number of node-disjoint paths partitioning $\mathcal{G}$:

Lemma 2. *Given a digraph $\mathcal{G}$ and its corresponding network $\mathcal{N}$, a lower bound on the number of node-disjoint paths partitioning $\mathcal{G}$ is given by the minimum feasible flow $f^\star_{ts}$ of $\mathcal{N}$: $f^\star_{ts} \leq \underline{\mathtt{K}}$.*

In the following, we propose two ways for improving the evaluation of the minimum feasible flow in the network $\mathcal{N}$ associated to the digraph $\mathcal{G}$: Firstly, in Section 4.1, we show how to estimate the number of paths partitioning each *scc* of $\mathcal{G}$ in order to improve

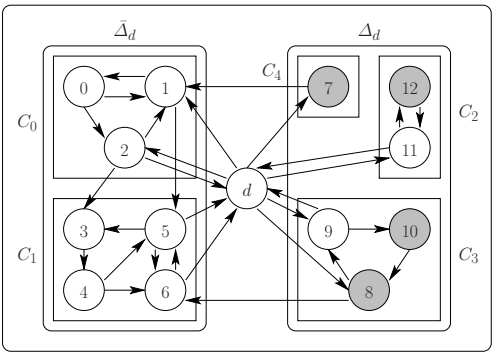


Fig. 5. This figure depicts a *scc* C_i of $\mathcal{G}$, a dominator node d of $D_i = \{1, 4, 8, 9, 11, d\}$. Grey nodes depict the set $\mathcal{T}_i$. The rectangles contained in Δ_d and $\bar{\Delta}_d$ represent *scc* created by the removal of d in C_i.

the evaluation of the flow going throughout each node-splitting arc of $\mathcal{N}$. Secondly, in Section 4.2, we show to estimate the number of paths between two *scc*'s of $\mathcal{G}$ in order to improve the evaluation of the flow going throughout each dag's arcs of $\mathcal{N}$.

4.1 Estimating the Number of Paths Partitioning a *scc*

A first way to improve the tightness of the relaxation of the K-NDP problem, when some *scc*'s of $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ involve more than one node (i.e., $\mathcal{G}$ is not a DAG), is to refine the bounds on the flow $f_i \in [\ell_i, c_i]$ for each node-splitting arc representing the *scc* C_i of $\mathcal{G}$. This section shows how to improve the tightness of ℓ_i (i.e., the minimum number of paths for partitioning C_i) which was originally set to 1 in Definition 5. The idea is to identify a node d of C_i, whose removal increases the number of *scc*'s of $C_i \setminus \{d\}$ (i.e., the graph corresponding to the *scc* C_i from which we remove the node d), in order to re-apply Lemma 2 on the new DAG built from the reduced graph of $C_i \setminus \{d\}$. This is achieved by using the dominance relation between nodes of C_i that we now introduce:

Definition 6 ([11]). *Given a digraph G and two nodes i, j of G such that there is at least one path from i to j, a node d is a* dominator *of node j with respect to node i iff there is no path from i to j in $G \setminus \{d\}$. The set of dominators of j with respect to i is denoted by $DOM_{\langle \mathcal{G}, i \rangle}(j)$.*

Notations 1. *Given a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, let C_i be a scc of $\mathcal{G}$:*

- *$\mathcal{C}_i = (\mathcal{V}_i, \mathcal{A}_i)$ denotes the sub-graph of $\mathcal{G}$ corresponding to C_i.*
- *$\mathcal{T}_i$ denotes the subset of nodes in $\mathcal{V}_i$ that are either potential final nodes of $\mathcal{G}$, or nodes reaching, by one single arc, another scc C_j of $\mathcal{G}$.*
- *Let D_i be the set of dominator nodes of $\mathcal{C}_i$ defined according to the node set $\mathcal{T}_i$:*

$$D_i = \{d \mid \exists v \in \mathcal{V}_i, \forall w \in \mathcal{T}_i, d \in DOM_{\langle \mathcal{C}_i, v \rangle}(w)\} \tag{7}$$

So, the removal of a dominator $d \in D_i$ creates two kinds of *scc*'s (Figure 5 illustrates this point on an example):

- Δ_d is the, possibly empty, set of new *scc*'s from which at least one node of $\mathcal{T}_i$ can be reached by at least one path that does not contain d. Let $\Delta_d^\star$ denotes the set of nodes involved in the *scc*'s of Δ_d.
- $\bar{\Delta}_d$ is the, possibly empty, set of new *scc*'s from which no node of $\mathcal{T}_i$ can be reached by a path that does not contain d. Let $\bar{\Delta}_d^\star$ denotes the set of nodes involved in the *scc*'s of $\bar{\Delta}_d$.

Example 4. In the *scc* C_i of $\mathcal{G}$ depicted by Figure 5, we have $\mathcal{T}_i = \{7, 8, 10, 12\}$. The set of dominator nodes of C_i with respect to $\mathcal{T}_i$ is $D_i = \{1, 4, 8, 9, 11, d\}$. For instance, any path from node 3 to a node of $\mathcal{T}_i$ encounters nodes 4 and node d. Thus, nodes 4 and node d are dominators of node 3 with respect to all the nodes of $\mathcal{T}_i$. Figure 5 also illustrates the partitioning of a *scc* C_i in strongly connected components, produced by the removal of node d:

- $\Delta_d = \{\mathcal{C}_2^d, \mathcal{C}_3^d, \mathcal{C}_4^d\}$ and $\Delta_d^\star = \{7, 8, 9, 10, 11, 12\}$,
- $\bar{\Delta}_d = \{\mathcal{C}_0^d, \mathcal{C}_1^d\}$ and $\bar{\Delta}_d^\star = \{0, 1, 2, 3, 4, 5, 6\}$.

We now introduce a proposition that will allow us to reduce the problem of estimating the minimum number of paths partitioning $\mathcal{C}_i$ to the problem of finding the minimum number of paths partitioning Δ_d.

Proposition 1. *If there exists a path partition of $\mathcal{G}$ then, for each dominator node $d \in D_i$ of a scc C_i, there exists a Hamiltonian path, finishing on a predecessor of d, in the induced sub-graph of $\mathcal{G}$ by the set of nodes $\bar{\Delta}_d^\star$.*

Proof. Remember that, by construction of $\bar{\Delta}_d^\star$, a path cannot finish on any node of $\bar{\Delta}_d^\star$. Since d dominates any node of $\mathcal{T}_i \subset \Delta_d^\star$ with respect to any node of $\bar{\Delta}_d^\star$, d is the only possible output for the nodes of $\bar{\Delta}_d^\star$. Consequently, if there does not exist a Hamiltonian path in $\bar{\Delta}_d^\star$ then node d is reached by at least two node-disjoint paths covering $\bar{\Delta}_d^\star$: A contradiction. □

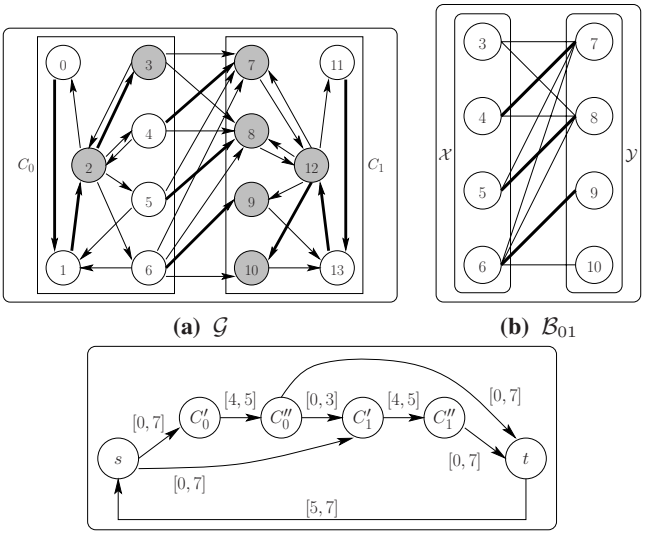
Proposition 2. *Consider the reduced graph $\mathcal{G}_{\Delta_d}$ (resp. $\mathcal{G}_{\bar{\Delta}_d}$) associated with the induced sub-graph of a scc C_i of $\mathcal{G}$ by the nodes of $\Delta_d^\star$ (resp. $\bar{\Delta}_d^\star$). Let x_{Δ_d} be a lower bound of the minimum number of node-disjoint paths partitioning $\mathcal{G}_{\Delta_d}$. A lower bound of the minimum number of node-disjoint paths partitioning $\mathcal{C}_i$ (denoted l_i^d), with respect to a dominator node d of C_i, is defined by:*

- $x_{\Delta_d} - 1 \leq l_i^d$, *if there exists an arc (u, v) in $\mathcal{C}_i$ such that $u \in \Delta_d^\star$, $v \in \bar{\Delta}_d^\star$, and the scc containing node v is a source node in $\mathcal{G}_{\bar{\Delta}_d}$.*
- $x_{\Delta_d} \leq l_i^d$, *otherwise.*

The lower bound x_{Δ_d} is defined by Lemma 2 according to the flow-based relaxation of the digraph $\mathcal{G}_{\Delta_d}$ with respect to the set of potential final nodes provided by $\mathcal{T}_i$. Finally, the maximum value in the set of l_i^d's ($d \in D_i$) provides a lower bound of the minimum path partition covering $\mathcal{C}_i$:

Proposition 3. *Let ℓ_i be the minimum number of node-disjoint paths partitioning the scc $\mathcal{C}_i$ of $\mathcal{G}$ and let D_i be the set of dominator nodes of C_i then, a lower bound of ℓ_i is:*

$$1 \leq max(\{l_i^d \mid d \in D_i\}) \leq \ell_i \tag{8}$$



(a) $\mathcal{G}$ **(b)** $\mathcal{B}_{01}$

(c) Network $\mathcal{N}$ associated to $\mathcal{G}$

Fig. 6. Part (a) depicts a digraph $\mathcal{G}$ composed by two *scc*'s C_0 and C_1. Thick arcs represent a possible solution maximizing the number of node-disjoint path partitioning $\mathcal{G}$ (5 paths). Bold nodes represent potential final nodes of $\mathcal{G}$. Part (b) depicts the bipartite graph $\mathcal{B}_{01}$ extracted from the *scc*'s C_0 and C_1. Part (c) depicts the network $\mathcal{N}$ associated to $\mathcal{G}$.

4.2 Estimating the Number of Paths Between Two *scc*'s

A second way to refine the relaxation of the K-NDP problem is to adjust the bounds on the flow $f_{ij} \in [l_{ij}, c_{ij}]$, for each inter-scc arc. This section shows how to improve the tightness of the upper bound c_{ij}, which was originally set to $\overline{\mathtt{K}} = |\mathcal{T}|$ in Definition 5 (i.e., the number of potential final nodes in $\mathcal{G}$). This is achieved by computing the cardinality of a maximum bipartite matching between nodes i incidents to an arc emanating from a *scc* C_i, and entering a node j of a *scc* C_j.

Notations 2. *Given two distinct scc's C_i and C_j of $\mathcal{G}$ and a node i of $\mathcal{C}_i$,*

- φ^+_{ij} *denotes the number of arcs emanating from i and entering a node j of C_j.*
- φ^-_{ij} *denotes the number of arcs entering node j of C_j and emanating from node i.*

Given two distinct *scc*'s C_i and C_j of $\mathcal{G}$ such that there is at least one arc from a node of C_i to a node of C_j, the maximum number of paths emanating from $\mathcal{C}_i$ and directly entering C_j can be computed from a bipartite graph, extracted from C_i and C_j, defined in the following way:

Definition 7. *The bipartite graph $\mathcal{B}_{ij} = (\mathcal{X}, \mathcal{Y}, \mathcal{E})$ associated with a pair of distinct scc's $\mathcal{C}_i$ and $\mathcal{C}_j$ of $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, such that there is an arc from a node of C_i to a node of C_j, is defined by:*

- $\mathcal{X} = \{i \in \mathcal{C}_i \mid \varphi^+_{ij} > 0\}$,

- $\mathcal{Y} = \{j \in \mathcal{C}_j \mid \varphi^-_{ij} > 0\}$,
- $\mathcal{E} = \{(i,j) \in \mathcal{A} \mid i \in C_i,\ j \in \mathcal{C}_j\}$.

Proposition 4. *Let $\mathcal{M}_{ij}$ denotes the cardinality of maximum matching in the bipartite graph $\mathcal{B}_{ij}$ associated with a pair of distinct scc's, C_i and C_j, of $\mathcal{G}$. The capacity c_{ij}, going through an inter-scc arc (i'', j') of the network $\mathcal{N}$ associated with $\mathcal{G}$ is bounded by $\mathcal{M}_{ij}$: $c_{ij} \leq \mathcal{M}_{ij}$.*

Example 5. From the digraph $\mathcal{G}$ depicted by Figure 6a, we build the bipartite graph $\mathcal{B}_{01}$ depicted by Figure 6b. A maximum cardinality matching in $\mathcal{B}_{01}$, of size $\mathcal{M}_{01} = 3$, is depicted by thick arcs of Figure 6b. The minimum number of node-disjoint paths partitioning $\mathcal{G}$ is equal to 5 (thick arcs of Figure 6a depict the following paths: $\{0,1,2,3\}$, $\{4,7\}$, $\{5,8\}$, $\{6,9\}$ and $\{11,13,12,10\}$). Figure 6c depicts the resulting network $\mathcal{N}$ associated to $\mathcal{G}$. Notice that for the intra-scc arc (C'_0, C''_0), the lower bound on the minimum number of paths covering *scc* C_0 is refined to the value 4 according to the dominator node 2 (see Section 4.1). Similarly, the lower bound on the minimum number of paths covering *scc* C_1 is refined to the value 4 according to the dominator node 12. In the same way, the upper bound on the maximum number of paths emanating *scc* C_0 and entering *scc* C_1 (depicted by the capacity of the inter-scc arc (C''_0, C'_1)) is refined to the value $\mathcal{M}_{01} = 3$.

Thus the value of a minimum flow in $\mathcal{N}$ is 5. Indeed, assume the value is 4 (i.e. $f_{ts} = 4$) then, 4 flow units are pushed through the arc (s, C'_0) because $\ell_0 = 4$, and we know that at most only 3 units have to reach C'_1: There is a contradiction with $\ell_1 = 4$. Thus, there does not exists a feasible flow with $f_{ts} = 4$.

5 A *path* Partitioning Constraint

A *path* partitioning constraint can be defined by a directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$, a number of paths NPATH, and a set $\mathcal{T} \subseteq \mathcal{V}$ of potential final nodes. W.l.o.g., our formal description of the *path* constraint follows the description of the *tree* constraint introduced in [4]. This allows to directly reuse all the filtering developed for the *tree* constraint which prevents the creation of circuits. In addition, we can also reuse the *in-degree* constraint of [5] that enforces one single predecessor for each node of the digraph.

Thus, the *path* constraint has the form path(NPATH, NODES), where NPATH is a domain variable[3] specifying the number of paths in the path partition, and NODES is the collection of n nodes NODES$[1], \ldots,$ NODES$[n]$ of the given digraph $\mathcal{G}$. Each node $v_i =$ NODES$[i]$ has the following attributes, which complete the description of $\mathcal{G}$:

- I is a unique integer in $[1, n]$. It can be interpreted as the *label* of node v_i.
- S is a domain variable (a *successor* variable) whose domain consists of elements (nodes labels) of $[1, n]$. It can be interpreted as the *unique successor* of node v_i. If $i \in dom($NODES$[i]$.S$)$, then we say that v_i is a *potential final node*.

[3] A *domain variable* V is a variable that ranges over a finite set of integers denoted by $dom(V)$. $min(V)$ and $max(V)$ respectively denote the minimum and the maximum value of V.

For each $i \in [1, n]$, the terms NODES$[i]$.I and NODES$[i]$.S respectively denote the I and S attributes of NODES$[i]$. Moreover, the digraph $\mathcal{G}$ associated with the constraint can be formally defined in the following way:

Definition 8. *The n-nodes, m-arcs directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ and the set $\mathcal{T}$ of a* path(NPATH, NODES) *constraint are defined by:*

- $\mathcal{V} = \{i \mid i \in [1, n]\}$,
- $\mathcal{A} = \{(i, j) \mid j \in dom(\text{NODES}[i].\text{S}),\ i \neq j)\}$,
- $\mathcal{T} = \{i \mid i \in dom(\text{NODES}[i].\text{S})\}$.

Finally, to complete the description of the constraint, we introduce the formal definition of a *ground* path partitioning constraint. For this purpose, we directly reason on the digraph $\mathcal{G}$ that models the *path* constraint.

Definition 9. *A* ground instance *of a* path(NPATH, NODES) *constraint is* satisfied *iff:*

- $\forall i \in [1, n] : \text{NODES}[i].\text{I} = i$.
- $\mathcal{G}$ *consists of* NPATH *connected components such that each one is an elementary path that ends up in a potential final node.*

The rest of this section is organised as follows: Section 5.1 provides two necessary conditions for partitioning the directed graph $\mathcal{G}$ associated with a *path* constraint, derived from the necessary condition introduced in Theorem 3. Section 5.2 shows how to exploit this necessary condition in order to filter NPATH as well as the successor variables modelling the associated digraph $\mathcal{G}$.

5.1 Feasibility

Based on Theorem 3 of Section 4, we introduce an algorithm for checking the feasibility of the *path* constraint according to the maximum number of allowed paths max(NPATH):

- Check if there exists a feasible flow in the network $\mathcal{N}$ associated with $\mathcal{G}$.
- For each *scc* C_i of $\mathcal{G}$, for each dominator node d of C_i, there exists at most one path for covering the digraph $\mathcal{G}_{\bar{\Delta}_d}$ (derived from Proposition 1).[4]

5.2 Filtering Algorithm

This section shows how to filter the domains of the successor variables NODES$[1]$.S, ..., NODES$[n]$.S and of the variable NPATH from the digraph $\mathcal{G}$ associated with a *path* partitioning constraint:

- Adjust the minimum value of NPATH to the value of a minimum feasible flow in the network $\mathcal{N}$ associated with $\mathcal{G}$.

[4] Remember that $\mathcal{G}_{\bar{\Delta}_d}$ is the reduced graph associated with the induced sub-graph of $\mathcal{G}$ by the nodes of $\bar{\Delta}^{\star}_d$.

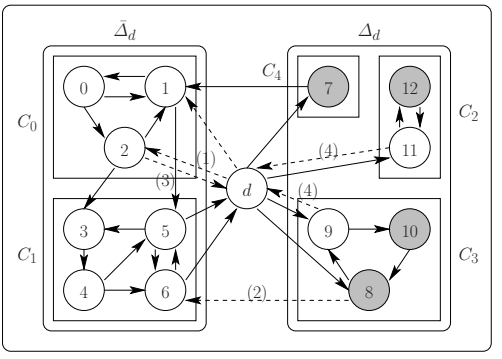


Fig. 7. From Figure 5 of Section 4.1, dotted arcs depict infeasible arcs of $\mathcal{G}$ detected by the algorithm introduced in Section 5.2.

– For each *scc* C_i of $\mathcal{G}$, for each dominator node $d \in C_i$, remove each arc (i, j) from $\mathcal{G}$ such that:
  - Case (1): $i = d$ and $j \in \bar{\Delta}^\star_d$.
  - Case (2): $i \in \Delta^\star_d$ and $j \in \bar{\Delta}^\star_d$ such that the *scc*, created by the removal of d and containing j, is not a source in the digraph $\mathcal{G}_{\bar{\Delta}_d}$.
  - Case (3): $j = d$ and $i \in \bar{\Delta}^\star_d$ such that the *scc*, created by the removal of d and containing i, is not a sink in the digraph $\mathcal{G}_{\bar{\Delta}_d}$.
  - Case (4): $i \in \Delta^\star_d$, $j = d$ and, $\bar{\Delta}^\star_d \neq \emptyset$.

Example 6. Figure 7 illustrates infeasible arcs detected by the previous algorithm. Arcs $(d, 1)$ and $(d, 2)$ are detected by Case (1), arc $(8, 6)$ is detected by Case (2), arc $(2, d)$ is detected by Case (3) and, arcs $(9, d)$ and $(11, d)$ are detected by Case (4).

Observe that one could also remove any arc (i, j) of $\mathcal{G}$, such that i and j belong to two distinct *scc*'s C_i and C_j, if the corresponding inter-scc arc in $\mathcal{N}$ between C_i and C_j cannot carry any flow (in any feasible flow of $\mathcal{N}$). We did not include this filtering within the previous algorithm since we do not know how to do this efficiently by computing one single feasible flow.

6 Conclusion

The filtering algorithms proposed for previously existing *path* constraints were only based on the prevention of circuits, as well as the in-degree constraints (each node has at most one predecessor). This article considers the *path* partitioning problem where we should not exceed a given number of paths. Within this context it came up with a flow model combining the structure of the reduced graph associated with $\mathcal{G}$, the structure of each strongly connected components of $\mathcal{G}$ with respect to dominance relation between nodes, and the way the nodes of two strongly connected components are inter-connected.

However, several questions remain open. Firstly, in our approach, each dominator node is considered independently from the others, and one can assume there exists a

strong interaction between the dominator nodes of $\mathcal{G}$, for which one can improve the tightness of the bounds on each intra-scc arcs of $\mathcal{N}$, as well as the related filtering. Secondly, in Section 5.2, we suggest a filtering related to the detection of the arcs that do not belong to any feasible flow of $\mathcal{N}$. This problem is polynomial, but the existence of an efficient algorithm (i.e., an algorithm which is not based on a repetitive test of each arc of $\mathcal{N}$) is not known to our knowledge.

Acknowledgements

We wish to thank the referees for helpful suggestions related to the form of the paper.

References

1. S. R. Arikati and C. P. Rangan. Linear algorithm for optimal path cover problem on interval graphs. *Inf. Process. Lett.*, 35(3):149–153, 1990.
2. M. Balinski and R. Quandt. On an Integer Program for Delivery Problem. *Operations Research*, 12(2):300–304, 1964.
3. N. Beldiceanu and E. Contejean. Introducing global constraint in CHIP. *Mathl. Comput. Modelling*, 20(12):97–123, 1994.
4. N. Beldiceanu, P. Flener, and X. Lorca. The *tree* constraint. In *CP-AI-OR'05*, volume 3524 of *LNCS*, pages 64–78. Springer-Verlag, 2005.
5. N. Beldiceanu, P. Flener, and X. Lorca. Combining tree partitioning, precedence, incomparability, and degree constraints, with an application to phylogenetic and ordered-path problems. Technical Report 2006-020, Department of Information Technology, Uppsala University, Sweden, April 2006.
6. C. Berge. *Graphes et Hypergraphes*. Dunod, Paris, 1970. In French.
7. H. Cambazard and E. Bourreau. Conception d'une contrainte globale de chemin. In *JNPC'04*, pages 107–120, 2004. In French.
8. R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.
9. S. Fortune, J. E. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. *Theor. Comput. Sci.*, 10:111–121, 1980.
10. L. G. Kaya and J. N. Hooker. A filter for the circuit constraint. In *CP'06*, volume 4204, pages 706–710. Springer-Verlag, 2006.
11. T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans Program. Lang. Syst.*, 1(1):121–141, 1979.
12. P. Prosser and C. Unsworth. Rooted tree and spanning tree constraints. Technical Report cppod-13-2006, CP Pod research group, May 2006.
13. L. Quesada, P. Van Roy, Y. Deville, and R. Collet. Using dominators for solving constrained path problems. In *PADL'06*, volume 3819 of *LNCS*, pages 73–87, 2006.
14. A. Schrijver. *Combinatorial Optimization*. Springer, Berlin, 2003.
15. M. Sellmann. Cost-based filtering for shortest path constraints. In *CP'03*, volume 2833 of *LNCS*, pages 694–708. Springer-Verlag, 2003.
16. G. Steiner. On the k-path partition of graphs. *Theor. Comput. Sci.*, 290(3):2147–2155, 2003.
17. J. Suurballe. Disjoint Paths in a Network. *Networks*, 4:125–145, 1974.
18. J. Vygen. NP-completeness of some edge-disjoint paths problems. *Discrete Appl. Math.*, 61(1):83–90, 1995.
19. J.-H. Yan and G. J. Chang. The path-partition problem in block graphs. *Inf. Process. Lett.*, 52(6):317–322, 1994.

A Constraint Programming Approach to the Hospitals / Residents Problem

David F. Manlove[⋆,⋆⋆], Gregg O'Malley[⋆⋆], Patrick Prosser, and Chris Unsworth[⋆⋆⋆]

Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, UK
davidm/gregg/pat/chrisu@dcs.gla.ac.uk.

Abstract. An instance I of the Hospitals / Residents problem (HR) involves a set of residents (graduating medical students) and a set of hospitals, where each hospital has a given capacity. The residents have preferences for the hospitals, as do hospitals for residents. A solution of I is a *stable matching*, which is an assignment of residents to hospitals that respects the capacity conditions and preference lists in a precise way. In this paper we present constraint encodings for HR that give rise to important structural properties. We also present a computational study using both randomly-generated and real-world instances. We provide additional motivation for our models by indicating how side constraints can be added easily in order to solve hard variants of HR.

1 Introduction

Gale and Shapley described in their seminal paper [7] the classical Hospitals / Residents problem (HR), referred to by the authors as the College Admissions problem. An instance of HR involves a set of *residents* (i.e. graduating medical students) and a set of *hospitals*. Each resident ranks in order of preference a subset of the hospitals. Each hospital has an integral *capacity*, and ranks in order of preference those residents who ranked it. We seek to match each resident to an acceptable hospital, in such a way that a hospital's capacity is never exceeded. Moreover the matching must be *stable* – a formal definition of stability follows, but informally stability ensures that no resident and hospital, not already matched together, would rather be assigned to one another than remain with their assignees. Such a resident and hospital could form a private arrangement outside the matching, undermining its integrity. Gale and Shapley [7] described a linear-time algorithm for finding a stable matching, given an instance of HR.

Many centralised matching schemes that automate the process of assigning residents to hospitals employ algorithms that solve HR and its variants [25]. For example, the National Resident Matching Program (NRMP) in the US [23] is perhaps the largest such scheme. The NRMP has been in operation since

⋆ Supported by RSE / Scottish Executive Personal Research Fellowship and EPSRC grant EP/E011993/1.
⋆⋆ Supported by EPSRC grant GR/R84597/01.
⋆⋆⋆ Supported by an EPSRC studentship.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 155–170, 2007.

1952 and handles the annual allocation of some 31,000 residents to hospitals. Counterparts of the NRMP elsewhere are the Canadian Resident Matching Service (CaRMS) [5] and the Scottish Foundation Allocation Scheme (SFAS) [13]. Similar matching schemes are also used in educational and vocational contexts.

A special case of HR occurs when each hospital has capacity 1 – this is the Stable Marriage problem with Incomplete lists (SMI). In this context, residents are referred to as *men*, whilst hospitals are referred to as *women*. A special case of SMI occurs when the numbers of men and women are equal, and each man finds all women acceptable and vice versa – this is the classical Stable Marriage problem (SM), also introduced by Gale and Shapley [7]. A specialised linear-time algorithm for SM, known as the Gale / Shapley (GS) algorithm [7], can be generalised to the SMI case [12, Section 1.4.2]. Using a process known as "cloning hospitals" (described in more detail in Section 3), a given instance I of HR may be transformed to an instance J of SMI, and the GS algorithm can be applied to J in order to give a stable matching in I. However in general this method expands the instance size, so that in practice specialised algorithms (such as those described in [12, Section 1.6]; see also Figure 2) are used to solve HR directly and achieve a better worst-case time complexity.

Over the last few decades, stable matching problems, and SM in particular, have been the focus of much attention in the literature [7,15,12,26]. Several encodings of SM and its variants as a Constraint Satisfaction Problem (CSP) have been formulated [1,8,16,9,10,11,19,29,30]. Moreover, recent papers have focussed on distributed variants of SM (including the Stable Roommmates problem, a non-bipartite extension of SM) where preference lists are to be kept private [27,28,3,4]. However, no encoding for HR has been considered before now.

This paper is concerned with a Constraint Programming (CP) approach to solving HR. We firstly present in Section 3 a cloned model for HR, indicating how existing formulations of SMI as a CSP [8] can be used in order to model HR. We then present in Section 4 a constraint-based model of HR that deals directly with an HR instance without cloning, achieving improved time and space complexities. We show that the effect of Arc Consistency (AC) propagation [2] applied to this model yields the same structure as the action of established algorithms for HR [7,12]. As a consequence, a stable matching for the given HR instance can be obtained without search (in fact we can in general obtain two complementary stable matchings following AC propagation, with optimality properties for the residents and hospitals respectively). We also demonstrate how a failure-free enumeration can be used to find all solutions for a given HR instance without search. These results therefore extend analogous results presented in [8] for SMI. In Section 5, we present a specialised n-ary constraint for HR, comparing and constrasting the time and space requirements for establishing AC with the models presented in Sections 3 and 4. Then, in Section 6, we describe the results of an empirical study which compares the various models presented in this paper in practice, on both randomly-generated and real-world data.

The models in Sections 4 and 5 are non-trivial extensions of earlier constraint models presented for SMI [8,19,29,30]. In the SMI case, clearly each woman can be assigned at most one man, but to model an HR instance without cloning,

Residents' preferences	M_0	M_z	Hospitals' preferences
$r_1 : h_1\ h_3$	–	–	$h_1 : (2) : \underline{r_3}\ r_7\ \underline{r_5}\ \underline{r_2}\ r_4\ r_6\ r_1$
$r_2 : \underline{h_1}\ h_5\ \underline{h_4}\ \underline{h_3}$	h_1	h_3	$h_2 : (3) : r_5\ \underline{r_6}\ r_3\ \underline{r_4}$
$r_3 : \underline{h_1}\ h_2\ h_5$	h_1	h_1	$h_3 : (1) : \underline{r_2}\ \underline{r_5}\ r_6\ r_1\ r_7$
$r_4 : h_1\ \underline{h_2}\ h_4$	h_2	h_2	$h_4 : (1) : \underline{r_8}\ \underline{r_2}\ r_4\ \underline{r_7}$
$r_5 : \underline{h_3}\ \underline{h_1}\ h_2$	h_3	h_1	$h_5 : (1) : r_3\ \underline{r_7}\ r_6\ \underline{r_8}\ r_2$
$r_6 : h_3\ \underline{h_2}\ h_1\ h_5$	h_2	h_2	
$r_7 : h_3\ \underline{h_4}\ \underline{h_5}\ h_1$	h_4	h_5	
$r_8 : \underline{h_5}\ \underline{h_4}$	h_5	h_4	

Fig. 1. An HR instance. The GS-list entries are underlined, and the middle two columns indicate the residents' assigned hospitals in M_0 and M_z (r_1 is unassigned in both).

the main challenges are to maintain a representation of the *set* of assignees of a given hospital h_j, and of the identity of the worst resident assigned to h_j.

The benefits of our approach are two-fold: firstly, the CSP models presented here for HR indicate that AC propagation using a CP toolkit yields the same structure as given by established linear-time algorithms for HR, from which all solutions for a given instance can be generated in a failure-free manner without search. Secondly, and more importantly, our models can be used as a basis on which additional constraints can be imposed, covering variants of HR that arise naturally in practical applications, but which cannot be accommodated easily by existing algorithms. These include variants of HR that are NP-hard, and for which no polynomial-time algorithm is currently known. Examples of such variants, where appropriate side-constraints are suggested in three cases, are given in Section 7 to provide additional motivation for our approach.

In the next section we present notation and terminology relating to HR, which will be assumed in the remainder of this paper, and we also present some important structural and algorithmic results.

2 Definitions and Fundamental Results

We now give a formal definition of HR. An instance I of HR comprises a set $R = \{r_1, \ldots, r_n\}$ of *residents* and a set $H = \{h_1, \ldots, h_m\}$ of *hospitals*. Each resident $r_i \in R$ has an *acceptable* set of hospitals $A_i \subseteq H$; moreover r_i ranks A_i in strict order of preference. For each $h_j \in H$, denote by $B_j \subseteq R$ those residents who find h_j acceptable; h_j ranks B_j in strict order of preference. Finally, each hospital $h_j \in H$ has an associated *capacity*, denoted by $c_j \in \mathbb{Z}^+$, indicating the number of *posts* that h_j has. For each $r_i \in R$, let l_i^r denote the length of r_i's preference list, and for each $h_j \in H$, let l_j^h denote the length of h_j's preference list; we assume that $c_j \leq l_j^h$. Let L denote the total length of the residents' preference lists in I. Given $r_i \in R$ and $h_j \in A_i$, define $rank(r_i, h_j)$ to be the position of h_j in r_i's preference list; $rank(h_j, r_i)$ is defined similarly. An example HR instance is shown in Figure 1 (the hospital capacities are indicated in brackets).

An *assignment* M is a subset of $R \times H$ such that $(r_i, h_j) \in M$ implies that $h_j \in A_i$ (i.e. r_i finds h_j acceptable). If $(r_i, h_j) \in M$, we say that r_i is *assigned to* h_j, and h_j is *assigned* r_i. For any $q \in R \cup H$, we denote by $M(q)$ the set of assignees of q in M. If $r_i \in R$ and $M(r_i) = \emptyset$, we say that r_i is *unassigned*, otherwise r_i is *assigned*. Similarly, any hospital $h_j \in H$ is *under-subscribed*, *full* or *over-subscribed* according as $|M(h_j)|$ is less than, equal to, or greater than c_j, respectively.

A *matching* M is an assignment such that $|M(r_i)| \leq 1$ for each $r_i \in R$ and $|M(h_j)| \leq c_j$ for each $h_j \in H$ (i.e. each resident is assigned to at most one hospital, and no hospital is over-subscribed). For convenience, given a resident $r_i \in R$ such that $M(r_i) \neq \emptyset$, where there is no ambiguity the notation $M(r_i)$ is also used to refer to the single member of $M(r_i)$.

A *blocking pair* relative to a matching M is a (resident,hospital) pair $(r_i, h_j) \in (R \times H) \backslash M$ such that (i) $h_j \in A_i$, (ii) either r_i is unassigned in M or prefers h_j to $M(r_i)$, and (iii) either h_j is under-subscribed or prefers r_i to at least one member of $M(h_j)$. A matching is *stable* if it admits no blocking pair.

Gale and Shapley [7] described an algorithm for finding a stable matching in a given HR instance I, which is known as the *resident-oriented* Gale/Shapley (RGS) algorithm [12, Section 1.6.3]. This algorithm finds the *resident-optimal* stable matching M_0 in I, in which each assigned resident is assigned to the best hospital that he could obtain in any stable matching. On the other hand, the *hospital-oriented* (HGS) algorithm [12, Section 1.6.2] is a second algorithm for HR that finds the *hospital-optimal* stable matching M_z in I, in which each hospital is assigned the best set of residents that it could obtain in any stable matching. Figure 1 includes columns that give M_0 and M_z for the example HR instance shown. In general, the optimality property of each of M_0 and M_z is achieved at the expense of the hospitals and residents respectively (the "pessimality" of each of these matchings for the relevant parties is discussed in Sections 1.6.2 and 1.6.5 of [12]). The RGS and HGS algorithms for HR are shown in Figure 2 (the term "delete the pair (r_i, h_j)" refers to the operations of deleting r_i from h_j's preference list and vice versa). Using a suitable choice of data structures (extending those described in [12, Section 1.2.3]), both the RGS and the HGS algorithms can be implemented to run in $O(L)$ time and $O(nm)$ space.

The deletions made by each of the RGS and HGS algorithms have the effect of reducing the original set of preference lists in I. The reduced lists returned by the RGS (respectively HGS) algorithm are known as the *RGS-lists* (respectively *HGS-lists*). The intersection of the RGS-lists and the HGS-lists yields the *GS-lists*. (E.g. the GS-lists for the HR instance shown in Figure 1 are represented as underlined preference list entries.) The GS-lists in I have several useful properties, which are summarised below (these properties follow as a consequence of Lemmas 1.6.2 and 1.6.4, and Theorems 1.6.1 and 1.6.2 of [12]):

Theorem 1. *For a given instance of HR,*
(i) all stable matchings are contained in the GS-lists;
(ii) in M_0, each resident with a non-empty GS-list is assigned to the first hospital on his GS-list, whilst each resident with an empty GS-list is unassigned;

$M = \emptyset$;
while (some $r_i \in R$ is unassigned
and r_i has a non-empty list)
h_j = first hospital on r_i's list;
/* r_i *applies to* h_j */
$M = M \cup \{(r_i, h_j)\}$;
if (h_j is over-subscribed)
r_k = worst resident assigned to h_j;
$M = M \backslash \{(r_k, h_j)\}$;
if (h_j is full)
r_k = worst resident assigned to h_j;
for (each successor r_z of r_k on h_j's list)
delete the pair (r_z, h_j);

$M = \emptyset$;
while (some $h_j \in H$ is under-subscribed
and some $r_i \in B_j$ is not assigned to h_j)
r_i = first such resident on h_j's list;
/* h_j *offers a post to* r_i */
if (r_i is assigned)
$h_k = M(r_i)$;
$M = M \backslash \{(r_i, h_k)\}$;
$M = M \cup \{(r_i, h_j)\}$;
for (each successor h_z of h_j on r_i's list)
delete the pair (r_i, h_z);

Fig. 2. RGS algorithm for HR; HGS algorithm for HR

(iii) in M_z, each hospital h_j is assigned the first m_j members of its GS-list, where $m_j = \min\{c_j, g_j^h\}$ and g_j^h is the length of h_j's GS-list.

Given any $q \in R \cup H$, we denote by $GS(q)$ the set of hospitals or residents (as appropriate) that belong to q's GS-list in I.

Additional important results concern residents who are unassigned, and hospitals that are under-subscribed, in stable matchings in I. These results are collectively known as the *Rural Hospitals Theorem* [12, Section 1.6.4], and may be stated as follows:

Theorem 2. *For a given instance of HR,*
(i) each hospital is assigned the same number of residents in all stable matchings;
(ii) exactly the same residents are unassigned in all stable matchings;
(iii) any hospital that is under-subscribed in one stable matching is assigned precisely the same set of residents in all stable matchings.

3 A Cloned Model

In this section we indicate how an instance of HR may be reduced to an instance of SMI by "cloning" hospitals. This technique is described in [12, p.38]; see also [26, pp.131-132]. For completeness, we briefly restate the construction here. Let I be an instance of HR. We form an instance J of SMI by replacing each hospital $h_j \in H$ by c_j women in J, denoted by h_j^k $(1 \leq k \leq c_j)$. The preference list of h_j^k in J is identical to that of h_j in I. Each resident r_i in I corresponds to a man r_i in J, and each hospital h_j in r_i's list in I is replaced by $h_j^1\ h_j^2\ \ldots\ h_j^{c_j}$, in that order, in J. It may then be shown that the stable matchings in I are in one-one correspondence with the stable matchings in J.

In order to obtain the GS-lists of I, we can model J using the "conflict matrices" encoding of SMI as presented in [8]. In general AC may be established in $O(ed^r)$ time, where e is the number of constraints, d is the domain size, and r is the arity of each constraint [2]. Due to the cloning technique, the number

1.	$y_{j,k} < y_{j,k+1}$	$(1 \leq j \leq m, 1 \leq k \leq c_j - 1)$
2.	$y_{j,k} \geq q \Rightarrow x_i \leq p$	$(1 \leq j \leq m, 1 \leq k \leq c_j, 1 \leq q \leq l_j^h)$
3.	$x_i \neq p \Rightarrow y_{j,k} \neq q$	$(1 \leq i \leq n, 1 \leq p \leq l_i^r, 1 \leq k \leq c_j)$
4.	$(x_i \geq p \wedge y_{j,k-1} < q) \Rightarrow y_{j,k} \leq q$	$(1 \leq i \leq n, 1 \leq p \leq l_i^r, 1 \leq k \leq c_j)$
5.	$y_{j,c_j} < q \Rightarrow x_i \neq p$	$(1 \leq j \leq m, c_j \leq q \leq l_j^h)$

Fig. 3. Constraints for the CSP model of an HR instance

of women in J is $\sum_{j=1}^{m} c_j = O(cm)$, where $c = \max\{c_j : h_j \in H\}$. Given the construction of the encoding in J [8], it follows that $e = O(nmc)$, $d = O(n+m)$ and $r = 2$, so that the time and space complexities for finding the GS-lists in I using the cloned model are $O((n+m)^4 c)$ and $O((nmc)^2)$ respectively.

4 A Direct CSP-Based Model

We now present a direct CSP encoding of an HR instance that avoids cloning. Let I be an instance of HR. For $r_i \in R$ and $h_j \in H$, we use the terminology r_i *applies (or is assigned) to* h_j*'s* k^{th} *post* $(1 \leq k \leq c_j)$ in the case that h_j prefers exactly $k-1$ members of $M(h_j)$ to r_i. Also given a matching M, we denote the resident who is assigned to h_j's k^{th} post in M by $M_k(h_j)$ $(1 \leq k \leq |M(h_j)|)$.

We construct a CSP instance J with variables $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_{j,k} : 1 \leq j \leq m \wedge 0 \leq k \leq c_j\}$, whose domains are initially defined as follows:

$$
\begin{array}{ll}
dom(x_i) = \{1, 2, \ldots, l_i^r\} \cup \{m+1\} & (1 \leq i \leq n) \\
dom(y_{j,0}) = \{0\} & (1 \leq j \leq m) \\
dom(y_{j,k}) = \{k, k+1, \ldots, l_j^h\} \cup \{n+k\} & (1 \leq j \leq m \wedge 1 \leq k \leq c_j).
\end{array}
$$

For the x_i variables $(1 \leq i \leq n)$, the value $m+1$ corresponds to the case that r_i's GS-list is empty, whilst the remaining values correspond to the ranks of preference list entries that belong to the GS-lists. A similar meaning applies to the $y_{j,k}$ variables $(1 \leq j \leq m,\ 1 \leq k \leq c_j)$, except that the value $n+k$ corresponds to the case that h_j's GS-list contains fewer than k entries.

More specificially, if $\min(dom(x_i)) \geq p$ $(1 \leq p \leq l_i^r)$, then during the RGS algorithm, r_i applies to his p^{th}-choice hospital or worse, so that in M_0, either r_i is assigned to such a hospital or is unassigned. Similarly if $\max(dom(x_i)) \leq p$, then during the HGS algorithm, r_i was offered a post by his p^{th}-choice hospital or better, so that r_i is assigned to such a hospital in M_z.

From the hospitals' point of view, if $\min(dom(y_{j,k})) \geq q$ $(1 \leq q \leq l_j^h)$, then during the HGS algorithm, h_j offers its k^{th} post to its q^{th}-choice resident or worse, so that in M_z, either h_j's k^{th} post is filled by such a resident, or is unfilled. Similarly if $\max(dom(y_{j,k})) \leq q$, then during the RGS algorithm, some resident r_i applied to h_j's k^{th} post, where $rank(h_j, r_i) \leq q$, so that h_j's k^{th} post is filled by r_i or better in M_0.

The constraints in J are given in Figure 3 (in the context of Constraints 2-5, p denotes the rank of h_j in r_i's list and q denotes the rank of r_i in h_j's list). An interpretation of the constraints is now given. Constraint 1 ensures that h_j's filled posts are occupied by residents in preference order, and that if post $k-1$ is unfilled then so is post k. Constraint 2 states that if h_j's k^{th} post is filled by a resident no better than r_i or is unfilled, then r_i must be assigned to a hospital no worse than h_j. Constraints 3 and 5 reflect the consistency of deletions carried out by the HGS and RGS algorithms respectively (i.e. if h_j is deleted from r_i's list, then r_i is deleted from h_j's list, and vice versa). Finally Constraint 4 states that if r_i is assigned to a hospital no better than h_j or is unassigned, and h_j's first $k-1$ posts are filled by residents better than r_i, then h_j's k^{th} post must be filled by a resident at least as good as r_i.

It turns out that establishing AC in J yields a set of domains that correspond to the GS-lists in I. To demonstrate this, we define some additional notation. For each j $(1 \leq j \leq m)$, define $S_j = \{rank(h_j, r_i) : r_i \in GS(h_j)\}$. Let d_j denote the number of residents assigned to hospital h_j in any stable matching in I. For each k $(1 \leq k \leq d_j)$, let $q_{j,k} = rank(h_j, M_{z_k}(h_j))$ and $t_{j,k} = rank(h_j, M_{0_k}(h_j))$. The *GS-domains* for the variables in J are defined as follows:

$$dom(x_i) = \begin{cases} \{rank(r_i, h_j) : h_j \in GS(r_i)\}, & \text{if } GS(r_i) \neq \emptyset \\ \{m+1\}, & \text{otherwise} \end{cases}$$

$$dom(y_{j,k}) = \begin{cases} \{s \in S_j : q_{j,k} \leq s \leq t_{j,k}\}, & \text{if } 1 \leq k \leq d_j \\ \{n+k\}, & \text{if } d_j + 1 \leq k \leq c_j. \end{cases}$$

We prove in [20] (we omit the proof here for space reasons) that, following AC propagation in J, the domain of each variable is a subset of its GS-domain, and conversely, the GS-domains are arc consistent in J. Given that AC algorithms find the unique maximal set of arc consistent domains [2], we therefore have:

Theorem 3. *Let I be an instance of HR, and let J be a CSP instance obtained by the encoding of this section. Then the domains remaining after AC propagation in J correspond exactly to the GS-lists in I.*

For example, in the context of the HR instance given in Figure 1, the GS-domains for x_2, $y_{1,1}$ and $y_{1,2}$ are $\{1,3,4\}$, $\{1\}$ and $\{3,4\}$ respectively. In general, following AC propagation in J, matchings M_0 and M_z may be obtained as follows. Let $x_i \in X$. If $x_i = m+1$, resident r_i is unassigned in both M_0 and M_z. Otherwise, in M_0 (respectively M_z), r_i is assigned to the hospital h_j such that $rank(r_i, h_j) = p$, where $p = \min(dom(x_i))$ (respectively $p = \max(dom(x_i))$).

In the context of the time complexity function for establishing AC as mentioned in Section 3, for this encoding we have $e = O(Lc)$ and $d = O(n+m)$ (recall that L is the total length of the residents' preference lists in I). The constraints shown in Figure 3 may be revised in $O(1)$ time, assuming that upper and lower bounds for the variables' domains are maintained throughout propagation. It follows by [31] that the time complexity for establishing AC in this model is $O(Lc(n+m))$. Since the space complexity is $O(Lc)$, the model presented in this section is more efficient than the cloned model in terms of both time and space.

The next result, proved in [20] (we also omit the proof here), states that the encoding presented above can be used to enumerate all the solutions of I in a failure-free manner using AC propagation with a value-ordering heuristic.

Theorem 4. *Let I be an instance of HR and let J be a CSP instance obtained by the encoding of this section. Then the following search process enumerates all solutions in I without repetition and without ever failing due to an inconsistency:*

- *AC is established as a preprocessing step, and after each branching decision including the decision to remove a value from a domain;*
- *if all domains are arc consistent and some variable x_i has two or more values in its domain then search proceeds by setting x_i to the minimum value p in its domain. On backtracking, the value p is removed from the domain of x_i;*
- *when a solution is found, it is reported and backtracking is forced.*

5 A Specialised n-Ary Constraint

We now present a specialised n-ary constraint HRN for the Hospitals / Residents problem. A model based on HRN requires only one constraint for the whole problem. We assume that this constraint will be processed by an AC5 [31] type arc consistency algorithm. That is, the algorithm has a stack of calls to revise constraints, and if a variable v loses a value then a call to all constraints involving v will be added to the stack along with the removed value.

5.1 Preliminaries

Our model involves a constrained integer variable x_i corresponding to each resident $r_i \in R$, where the domain values represent ranks, as in Section 4. In addition, we associate a single constrained integer variable y_j corresponding to each hospital $h_j \in H$ with similar meanings for the domain values. In this model only the x variables are search variables, meaning that a solution consists of a single value being assigned to each x variable, but the y variables may have multiple values remaining in their associated domains.

We assume that we have the following functions, each being of $O(1)$ complexity, that operate over constrained integer variables:

- $getMin(v)$ delivers the smallest value in $dom(v)$.
- $getMax(v)$ delivers the largest value in $dom(v)$.
- $getValue(v, a)$ returns the a^{th} smallest value in $dom(v)$, if $|dom(v)| < a$ then $getMax(v)$ is returned.
- $setMax(v, a)$ removes all values greater than a from $dom(v)$.
- $remVal(v, a)$ removes the value a from $dom(v)$.
- $PL(r_i, k)$ returns the k^{th} entry in r_i's preference list.
- $swap(a, b)$ swaps the values of the variables a and b.

```
1. init()
2.    for i := 1 to n loop
3.       apply(i);
4.    end loop;
5.    for j := 1 to m loop
6.       offer(j);
7.    end loop;
```

Fig. 4. Method *init*

The HRN constraint also requires the following data structures:

- $\check{x}$ is an array of n reversible integer variables containing the previous lower bounds of all x variables. All are initially set to $min(x)-1$. On backtracking the values in $\check{x}$ are restored by the solver.
- $\check{y}$ is an array of m reversible integer variables containing the value that represents y's least favourite resident to be offered a post at y. For hospital h_j, $\check{y}_j$ will equal the c_j^{th} lowest value in $dom(y_j)$. All are initially set to $min(y)-1$. On backtracking the values in $\check{y}$ are restored by the solver.
- *post* : an $m \times c$ matrix of reversible integer variables which stores applications for hospital posts. Each array element is initialised to ∞ (i.e. the largest integer). Row $post_j$ stores the applications for hospital h_j and entry $post_{j,k}$ stores the k^{th} best application received by hospital h_j.

To implement a constraint we require two methods: one that is called at the head of search to initialise the constraint and one that is called when a value is removed from a constrained variable. We now give the first of these methods:

The *init* method (Figure 4) is called at the head of search. Each resident applies to their favourite hospital (lines 2-4) via the *apply*(i) function (details given later), then each hospital makes an offer to their c favourite residents (lines 5-7) via the *offer*(j) function (details given later).

As HRN constrains two sets of variables we require two different method to call when a value is removed from one of the variable's domains. These methods are given below:

The *deltaX* method, shown in Figure 5(a), is called when some value a, where $a < m+1$, is removed from $dom(x_i)$. The index j of the hospital a represents is found (line 2), and r_i is then removed from the domain of h_j (line 3). If a represents the last hospital r_i applied to (line 4), then r_i will make a new application to its new favourite via the *apply*(i) function (line 5). Note that

```
1.  deltaX(i,a)
2.     j := PL(r_i, a);
3.     remValue(y_j,rank(h_j, r_i));
4.     if a = x̌_i then
5.        apply(i);
```

Fig. 5. (a) Method *deltaX*.

```
1.  deltaY(j,a)
2.     i := PL(h_j, a);
3.     remValue(x_i,rank(r_i, h_j));
4.     if a ≤ y̌_j then
5.        offer(j);
```

(b) Method *deltaY*.

```
1. apply(i)
2. for k := x̆_i + 1 to min(x_i) loop
3.    j := PL(r_i, k);
4.    apply(j,rank(h_j, r_i));
5.    if post_{j,c_j} < ∞ then
6.       setMax(y_j,post_{j,c_j});
7. end loop;
8. x̆_i := min(x_i);
```

```
1. apply(j,a)
2. for k := 1 to c_j loop
3.    if post_{j,k} = a then
4.       a := n + 1;
5.    if post_{j,k} > a then
6.       swap(post_{j,k}, a);
7. end loop;
```

Fig. 6. (a) Function *apply*(i). (b) Function *apply*(j, a).

either the deletion on line 3 or an indirect deletion via a call to the *apply*(i) function (details given later) could cause a reduction in the domain of some y variable and thus a call to *deltaY* will be placed on the call stack.

The *deltaY* method, shown in Figure 5(b), is called when some value a, where $a < n + 1$, is removed from $dom(y_j)$. The index i of the resident a represents is found (line 2) and h_j is then removed from the domain of r_i (line 3). If a represents a resident h_j that made an offer to (line 4), then h_j will make a new set of offers via the *offer*(j) function (line 5). Note that either the deletion on line 3 or an indirect deletion via a call to the *offer*(j) function (details given later), could cause a reduction in the domain of some x variable and thus a call to *deltaX*. Therefore the propagation of this constraint results from the mutual recursion between methods *deltaX* and *deltaY*.

The *apply*(i) function of Figure 6(a) is called either at the head of search (via the *init* method) or when the lower bound of x_i changes (via the *deltaX* method). Resident r_i will apply to each hospital that it prefers to any other in its domain, and to which it has not previously applied to (line 2). First the hospital h_j to be applied to is found (line 3), then resident r_i makes an application to hospital h_j via a call to the *apply*(j, a) function(line 4). If c_y applications have been made to hospital h_j (line 5) then h_j must not consider any resident worse then its c_j^{th} favourite applicant (line 6). $\breve{x}_i$ is then updated with the current lower bound of x_i (line 8). As the runtime of this function is dependent on the number of domain reductions made since the previous call to this function, it therefore has $O(1)$ complexity per deletion.

The *apply*(j, a) function of Figure 6(b) is called only by the *apply*(i) function when hospital h_j receives an application from its a^{th} choice resident. The hospital's preference for this applicant is placed in the list of applicants in ascending order. If more than c_j applications have been received then the worst applicant will drop off the end of the array and will effectively be removed from the list. This function runs in $O(c)$ time.

Figure 7 gives the *offer*(j) function which can be called either at the head of search (via the *init* method) or when a resident that was previously offered a place has been removed from $dom(y_j)$ (via the *deltaX* method). Hospital h_j will offer a post to r_i, the c_j^{th} favourite resident still in its domain, and to all other residents that it prefers to r_i to which it has not yet offered a place to. $\breve{y}_j$ is then

```
1.  offer(j)
2.  for k := y̌_i + 1 to getValue(h_j,c_j) loop
3.    i := PL(h_j, k);
4.    setMax(x_i,rank(r_i, h_j))
5.  end loop;
6.  y̌_j := getValue(h_j,c_j);
```

Fig. 7. Function *offer*(j)

Table 1. Summary of time and space complexities for the HR models of this paper

Model:	Cloned	CBM	HRN
Time:	$O((n+m)^4\, c)$	$O(Lc(n+m))$	$O(Lc)$
Space:	$O((nmc)^2)$	$O(Lc)$	$O(nm)$

updated to its preference for r_i. This function contains one loop which cycles at most c_j times, therefore it runs in $O(c)$ time.

5.2 Complexity

The *deltaX* and *deltaY* methods contains no loops, but each calls a function which runs in $O(c)$ time. Thus *deltaX* and *deltaY* both have a complexity of $O(c)$. The *deltaX* method can be called at most once for each value in the domain of an x_i variable, and similarly *deltaY* can be called at most once for each value in the domain of the y_j variable. Therefore we have a time complexity of $O(Lc)$. Hence the time complexity for the HRN constraint improves those of the models presented in earlier sections. The space complexity of this encoding is dominated by the ranking arrays, and is $O(nm)$. However, if preference lists are short we may economically trade time for space, or use some sparse data structure, or a hash table to map preferences to indices.

Table 1 summarises the time and space complexities for the HR models in this paper (the columns refer respectively to the models in Sections 3, 4 and 5).

5.3 Searching for All Solutions

Arc consistency processing on the HRN constraint yields the *GS-domains* as defined in Section 4. A search process need only consider the resident variables (and need not instantiate the hospital variables), following a similar process to that outlined in Theorem 4.

6 Computational Experience

The three encodings presented in this paper were implemented using the JSolver toolkit, i.e. the Java version of ILOG Solver, in order to carry out an empirical analysis. The objective was to compare the runtimes for these models as applied

Table 2. Average computation times in seconds to find all solutions to 100 randomly-generated HR instances with attributes $n/m/c$

	50/13/4	100/20/5	500/63/8	1k/100/10	5k/250/20	20k/550/37	50k/1.2k/42
Cloned	5.84	–	–	–	–	–	–
CBM	0.24	0.36	1.69	4.75	–	–	–
HRN	0.12	0.15	0.19	0.22	0.53	1.42	4.2

to randomly-generated and real-world data. Our studies were carried out using a 2.8Ghz Pentium 4 processor with 512 Mb of RAM, running Microsoft Windows XP Professional and Java2 SDK 1.4.2.6 with an increased heap size of 512 Mb.

Random problem instances were generated with varying number of residents n, number of hospitals m, capacity c (uniform for each hospital), and a fixed residents' preference list size of 10. Hence we classify problems via the triple $n/m/c$. Instances were generated as follows. First, a uniformly random preference list of length 10 was produced for each resident, then a preference list was produced for each hospital by randomly permuting their acceptable residents. A sample size of 100 was used for each value of $n/m/c$.

Table 2 shows the mean time in seconds to construct the model and find all solutions, for the each of the four models applied to random instances with varying $n/m/c$ attributes. A table entry of $-$ signifies that there was insufficient space to create the model of that size using the specified encoding. Table 3 shows the time to establish AC (shown as "AC") and find all solutions (shown as "ALL") to three anonymised HR instances arising from SFAS [13]. The first column indicates $n/m/c$, where c is the average hospital capacity; also $l_i^r \leq 5$ in each case. (For each instance, the Cloned model ran out of memory.)

The results indicate that the HRN model was typically able to handle larger problem instances than the other models, and the average runtime was faster than for the other models in all cases. The HRN model was also applied to instances as large as $500k/11.8k/85$, finding all solutions on average in 35 seconds. As mentioned in the Introduction, instances of the NRMP typically involve around 31,000 residents and 2,300 hospitals, with residents' preference lists of size between 4 and 7 [23]. The HRN model finds all solutions to problems of size $200k/3k/67$ in 22 seconds on average. This leads us to believe that Constraint Programming is indeed a suitable technology for the HR problem.

7 Motivation: Side-Constraints

It is natural to build additional constraints on top of the constraint models of HR presented in this paper, in order to cope with generalisations of HR for which the RGS and HGS algorithms are inapplicable. In this section we present several variants of HR that are either NP-hard or for which no polynomial-time algorithm is currently known. In the first three cases we suggest additional

side-constraints that can be added to any of our base models in order to cope with the more general problem, providing additional motivation for our approach.

Resident-exchange-stable HR. During a previous run of the SFAS matching scheme, two residents complained that, had they swapped their given hospitals, they could each have been better off. Such a swap would not have been permitted by the hospitals, of course, as it would have violated the stability criterion. However it would be desirable to avoid such a situation arising if possible, and this leads to the problem of finding a *resident-exchange stable matching* given an instance I of HR. This is a stable matching M in I such that there are no two assigned residents r_i, r_j such that r_i prefers $M(r_j)$ to $M(r_i)$, and r_j prefers $M(r_i)$ to $M(r_j)$. It is known that a such a matching need not exist in I, and indeed the problem of deciding whether such a matching exists in I is NP-complete [14,21], even if each hospital has capacity 1. For any two residents r_i, r_j and for any two hospitals h_k, h_l such that r_i prefers h_l to h_k and r_j prefers h_l to h_k, the additional constraint $x_i = p_1 \Rightarrow x_j \neq p_2$ should be added, where $rank(r_i, h_k) = p_1$ and $rank(r_j, h_l) = p_2$.

HR with forbidden pairs. Let F be a set of (resident,hospital) pairs in an instance I of HR. An adminstrator of a matching scheme may wish to exclude the pairs in F from any matching. Hence a matching M in I must not include any member of F, however a pair in F could still form a blocking pair (hence we cannot simply delete pairs in F from the preference lists). The task is to find a matching in I that is stable in the usual sense. Clearly a stable matching need not exist, given an instance of HR with forbidden pairs. However given an instance of SMI with forbidden pairs, there exists a linear-time algorithm to find a stable matching or report that none exists [6], and it is straightforward to extend this algorithm to HR. However no polynomial-time algorithm is currently known for the problem of finding a matching M in I (in the usual sense) with the fewest number of forbidden pairs. One possibility for modelling this problem is to add new variables $T = \{t_{i,p} : 1 \leq i \leq n \wedge 1 \leq p \leq l_i^r\}$, each with domain $\{0, 1\}$, and a constraint $x_i = p \Rightarrow t_{i,p} = 1$, for each $(r_i, h_j) \in F$, where $rank(r_i, h_j) = p$, and then minimise the sum of the values of the variables in T.

HR with groups. An extension of HR that has practical relevance arises when residents may form groups, and may decide that they are only prepared to be matched to a given hospital if the whole group is matched to it. More formally,

Table 3. Time taken to establish AC and find all solutions to three SFAS instances

	# Solutions	CBM		HRN	
		AC	ALL	AC	ALL
502/41/13.2	1	1.61	1.64	0.17	0.17
510/43/11.5	1	1.64	1.7	0.17	0.17
245/34/3.9	1	0.26	0.26	0.12	0.12

each hospital $h_j \in H$ may have one or more associated groups $G_j \subseteq R$. A matching M must satisfy the additional property that if $(r_i, h_j) \in M$ for some $r_i \in G_j$, then $(r_k, h_j) \in M$ for all $r_k \in G_j$. No polynomial-time algorithm for this problem is currently known. However this variant can be modelled as follows. For any group $G_j = \{r_{i_1}, \ldots, r_{i_k}\}$, add the constraint $x_{i_a} = p_{i_a} \Rightarrow x_{i_b} = p_{i_b}$ $(1 \leq a \neq b \leq k)$ where $rank(r_{i_a}, h_j) = p_{i_a}$ and $rank(r_{i_b}, h_j) = p_{i_b}$. A particular case of this problem is the Hospitals / Residents problem with Couples (HRC), described below.

Other generalisations of HR. The Hospitals / Residents problem with Ties (HRT) arises when ties are permitted in the preference lists of hospitals and/or residents. For example, a popular hospital may be indifferent among several applicants. The SFAS scheme [13] already permits ties in the hospitals' lists. However it is known [18] that, in the presence of ties, stable matchings can be of different sizes, and the problem of finding a maximum stable matching is NP-hard, even for very restricted instances of SMI with ties. It has already been demonstrated [9,10] that the earlier encodings of [8] can be extended to the case where preference lists in a given SMI instance may involve ties. We have begun to consider the corresponding extension of the models presented in Sections 4 and 5 to the HRT case, and further details will appear elsewhere.

HRC (in which couples submit joint preference lists over pairs of hospitals) is another generalisation of HR. Again it is possible that an instance need not admit a stable matching (where the stability definition is extended to the couples case), and the problem of deciding whether such a matching exists is NP-complete [24]. A constraint-based solution to this problem is motivated by the NRMP, which permits couples to submit joint preference lists.

8 Conclusions and Future Work

In this paper we have presented three CP models of an HR instance. The empirical results for the models as presented in Section 6 are broadly in line with what may be expected, given the summary of time and space complexities presented in Table 1. Our results indicate that, as is the case for SMI [8], CSP encodings of HR are "tractable", a notion that has been explored in detail by Green and Cohen [11]. However it remains open as to whether there exists a CSP encoding of HR that gives rise to the GS-lists, for which AC may be established in $O(L)$ time and using $O(nm)$ space. The time complexity of $O(L)$ is optimal, since SM is a special case of HR, and a lower bound of $\Omega(L)$ holds for the problem of finding a stable matching, given an instance of SM [22].

Acknowledgement

The authors are grateful to ILOG SA for providing access to the JSolver toolkit via an Academic Grant Licence.

References

1. B. Aldershof, O.M. Carducci, and D.C. Lorenc. Refined inequalities for stable marriage. *Constraints*, 4:281–292, 1999.
2. C. Bessière and J-C. Régin. Arc consistency for general constraint networks: Preliminary results. In *Proceedings of IJCAI '97*, vol. 1, pp. 398–404, 1997.
3. I. Brito and P. Meseguer. Distributed stable matching problems. In *Proceedings of CP '05*, *LNCS* vol. 3705, pp. 152–166. Springer, 2005.
4. I. Brito and P. Meseguer. Distributed stable matching problems with ties and incomplete lists. In *Proceedings of CP '06*, *LNCS* vol. 4204, pp. 675–679. Springer, 2006.
5. Canadian Resident Matching Service. How the matching algorithm works. Web document available at `http://www.carms.ca/matching/algorith.htm`.
6. V.M.F. Dias, G.D. da Fonseca, C.M.H. de Figueiredo and J.L. Szwarcfiter. The stable marriage problem with restricted pairs. *Theoretical Computer Science*, 306(1-3):391–405, 2003.
7. D. Gale and L.S. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.
8. I.P. Gent, R.W. Irving, D.F. Manlove, P. Prosser, and B.M. Smith. A constraint programming approach to the stable marriage problem. In *Proceedings of CP '01*, *LNCS* vol. 2239, pp. 225–239. Springer, 2001.
9. I.P. Gent and P. Prosser. An empirical study of the stable marriage problem with ties and incomplete lists. In *Proceedings of ECAI '02*, pp. 141–145. IOS Press, 2002.
10. I.P. Gent and P. Prosser. SAT encodings of the stable marriage problem with ties and incomplete lists. In *Proceedings of SAT '02*, pp. 133–140, 2002.
11. M.J. Green and D.A. Cohen. Tractability by approximating constraint languages. In *Proceedings of CP '03*, *LNCS* vol. 2833, pp. 392–406. Springer, 2003.
12. D. Gusfield and R.W. Irving. *The Stable Marriage Problem: Structure and Algorithms*. MIT Press, 1989.
13. R.W. Irving. Matching medical students to pairs of hospitals: a new variation on a well-known theme. In *Proceedings of ESA '98*, *LNCS* vol. 1461, pp. 381–392. Springer, 1998.
14. R.W. Irving The Man-Exchange Stable Marriage Problem. Technical Report TR-2004-177, University of Glasgow, Department of Computing Science, 2004.
15. D.E. Knuth. Mariages Stables *Les Presses de L'Université de Montréal*, 1976.
16. I.J. Lustig and J. Puget. Program does not equal program: constraint programming and its relationship to mathematical programming. *Interfaces*, 31:29–53, 2001.
17. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
18. D.F. Manlove, R.W. Irving, K. Iwama, S. Miyazaki, and Y. Morita. Hard variants of stable marriage. *Theoretical Computer Science*, 276 (1-2) : 261–279, 2002.
19. D.F. Manlove and G. O'Malley. Modelling and solving the stable marriage problem using constraint programming. In *Proceedings of the Fifth Workshop on Modelling and Solving Problems with Constraints*, held at IJCAI '05, pp. 10–17, 2005.
20. D.F. Manlove, G. O'Malley, P. Prosser and C. Unsworth. A Constraint Programming Approach to the Hospitals / Residents Problem. Technical Report TR-2007-236, University of Glasgow, Department of Computing Science, 2007.
21. E. McDermid, C. Cheng and I. Suzuki. Hardness results on the man-exchange stable marriage problem with short preference lists. *Information Processing Letters*, 101:13–19, 2007.

22. C. Ng and D.S. Hirschberg. Lower bounds for the stable marriage problem and its variants. *SIAM Journal on Computing*, 19:71–77, 1990.
23. National Resident Matching Program. About the NRMP. Web document available at `http://www.nrmp.org/about_nrmp/how.html`.
24. E. Ronn. NP-complete stable matching problems. *Journal of Algorithms*, 11:285–304, 1990.
25. A.E. Roth. The evolution of the labor market for medical interns and residents: a case study in game theory. *Journal of Political Economy*, 92(6):991–1016, 1984.
26. A.E. Roth and M.A.O. Sotomayor. *Two-sided matching: a study in game-theoretic modeling and analysis*. Cambridge University Press, 1990.
27. M.-C. Silaghi, M. Zanker and R. Barták. Desk-mates (stable matching) with privacy of preferences, and a new distributed CSP framework. In *Proceedings of the CP 2004 workshop on CSP Techniques with Immediate Application (CSPIA)*, pp. 83–96, 2004.
28. M.-C. Silaghi, A. Abhyankar, M. Zanker and R. Barták. Desk-mates (stable matching) with privacy of preferences, and a new distributed CSP framework. In *Proceedings of FLAIRS 2005*, pp. 671-677. AIII Press, 2005.
29. C. Unsworth and P. Prosser. An n-ary constraint for the stable marriage problem. In *Proceedings of the Fifth Workshop on Modelling and Solving Problems with Constraints*, held at IJCAI '05, pp. 32–38, 2005.
30. C. Unsworth and P. Prosser. A specialised binary constraint for the stable marriage problem. In *Proceedings of SARA '05*, *LNAI* vol. 3607, pp. 218-233. Springer, 2005.
31. P. van Hentenryck, Y. Deville, and C-M. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.

Best-First AND/OR Search for 0/1 Integer Programming

Radu Marinescu and Rina Dechter

School of Information and Computer Science
University of California, Irvine, CA 92697-3425
{radum,dechter}@ics.uci.edu

Abstract. AND/OR search spaces are a unifying paradigm for advanced algorithmic schemes for graphical models. The main virtue of this representation is its sensitivity to the structure of the model, which can translate into exponential time savings for search algorithms. In this paper we introduce an AND/OR search algorithm that explores a context-minimal AND/OR search graph in a *best-first* manner for solving 0/1 Integer Linear Programs (0/1 ILP). We also extend to the 0/1 ILP domain the *depth-first* AND/OR Branch-and-Bound search with caching algorithm which was recently proposed by [1] for solving optimization tasks in graphical models. The effectiveness of the best-first AND/OR search approach compared to depth-first AND/OR Branch-and-Bound search is demonstrated on a variety of benchmarks for 0/1 ILPs, including instances from the MIPLIB library, real-world combinatorial auctions, random uncapacitated warehouse location problems and MAX-SAT instances.

1 Introduction

In constraint optimization the goal is to minimize (or maximize) an objective function, subject to a set of constraints on the possible values of a set of independent decision variables. An important class of constraint optimization problems are the 0/1 Integer Linear Programming problems (0/1 ILP) [2] where the objective is to optimize a linear function of binary integer variables, subject to a set of linear equality or inequality constraints defined on subsets of variables. The classical approach to solving 0/1 ILPs is the *Branch-and-Bound* method [3] which maintains the best solution found so far, while discarding partial solutions which cannot improve on the best.

The AND/OR search space for graphical models [4] is a framework for search that is sensitive to the independencies in the model, often resulting in exponentially reduced complexities. It is based on a pseudo-tree that captures independencies in the graphical model, resulting in a search space exponential in the depth of the pseudo-tree, rather than in the number of variables.

The AND/OR Branch-and-Bound search ($\texttt{AOBB}_t$) was first introduced by [5] as a Branch-and-Bound algorithm that explores an AND/OR search tree in a depth-first manner for solving optimization tasks in graphical models. The AND/OR Branch-and-Bound search with caching algorithm ($\texttt{AOBB}_g$) due to [1] improves $\texttt{AOBB}_t$ by allowing the algorithm to save previously computed results and retrieve them when the same subproblems are encountered again. These algorithms are restricted to a static variable ordering determined by the underlying pseudo-tree. More recently, [6,7] proposed several extensions of $\texttt{AOBB}_t$ that incorporate dynamic variable ordering heuristics and

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 171–185, 2007.

explore dynamic AND/OR search trees. Two such extensions, *AND/OR Branch-and-Bound with Partial Variable Ordering* (AOBB$_t$+PVO) and *AND/OR Branch-and-Bound with Full Dynamic Variable Ordering* (AOBB$_t$+DVO) were shown to outperform significantly the static AOBB$_t$ algorithm as well as state-of-the-art classic OR Branch-and-Bound algorithms on various domains, including 0/1 ILPs.

In this paper we present and evaluate a new AND/OR search algorithm, that explores an AND/OR search graph in a *best-first* manner for solving 0/1 ILPs. Under conditions of admissibility and monotonicity of the guiding heuristic function, best-first search is known to expand the minimal number of nodes, at the expense of using additional memory [8]. In practice, these savings in number of nodes may often translate into impressive time savings as well. Since variable selection can have a dramatic impact on search performance, we also introduce a best-first AND/OR search algorithm that explores an AND/OR search tree, rather than a graph, and combines the AND/OR decomposition principle with dynamic variable selection heuristics, in a similar fashion as the dynamic AND/OR Branch-and-Bound algorithms described in [6,7]. We also adapt the static AOBB$_g$ algorithm for solving 0/1 ILPs.

We demonstrate empirically the efficiency of our best-first AND/OR search approach compared to depth-first AND/OR Branch-and-Bound search on several benchmarks for 0/1 ILP, including test instances from the MIPLIB library, combinatorial auctions simulating radio spectrum allocation, random uncapacitated warehouse location problems and MAX-SAT instances from the SATLIB library.

The paper is organized as follows. In Section 2 we present background on 0/1 ILP and AND/OR search spaces. In Section 3 we introduce the best-first AND/OR search algorithm as well as the extension to 0/1 ILP of the depth-first AND/OR Branch-and-Bound search with caching. In Section 4 we present a best-first AND/OR search algorithm that incorporates dynamic variable orderings. Section 5 shows our empirical evaluation and Section 6 concludes.

2 Background

2.1 Integer Linear Programming

A *Linear Program* (LP) consists of a set of continuous variables and a set of linear constraints (equalities or inequalities). The goal is to optimize a global linear cost function subject to the constraints. One of the standard forms of a linear program is:

$$min\{c^\top x \mid Ax \leq b, x \geq 0\} \tag{1}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$ and $x \in \mathbb{R}^n$. Here c represents the cost vector and x is the vector of decision variables. The vector b and the matrix A define the m linear constraints. Linear programs are usually solved by Dantzig's SIMPLEX method [9].

An *Integer Linear Programming* (ILP) problem is a linear program where all the decision variables are constrained to have integer values at the optimal solution. An important special case is a decision variable x_i that is integer with $0 \leq x_i \leq 1$. This forces x_i to be either 0 or 1 at the solution. Variables like x_i are called *0/1* or *binary integer variables*. A *0/1 Integer Linear Programming* problem is an ILP where all the

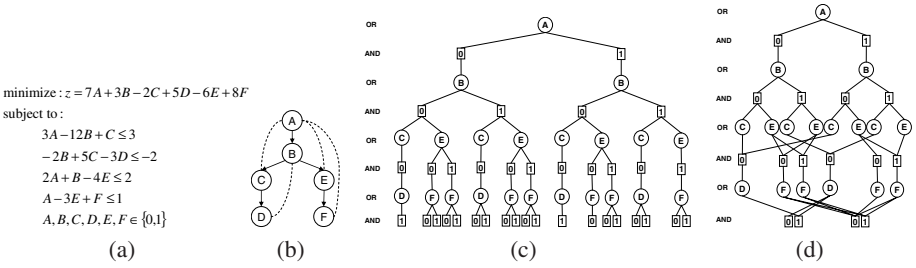


Fig. 1. The AND/OR search space

decision variables are binary. 0/1 ILPs can formulate many practical problems such as capital budgeting [10], cargo loading [11], processor allocation in distributed systems [12], combinatorial auctions [13,14] or maximum satisfiability problems [15,16].

With every 0/1 ILP instance we can associate an *interaction graph* G which has a node for each variable and connects any two nodes whose variables appear in the scope of the same constraint. The *induced graph* of G relative to an ordering d of its variables, denoted $G^*(d)$, is obtained by processing the nodes in reverse order of d. For each node all its earlier neighbors are connected, including neighbors connected by previously added edges. Given a graph and an ordering of its nodes, the *width* of a node is the number of edges connecting it to nodes lower in the ordering. The *induced width* of a graph, denoted $w^*(d)$, is the maximum width of nodes in the induced graph.

In the remainder, we will consider the *minimization* of a 0/1 ILP instance defined by a linear objective function $z = \sum_{i=1}^{n} c_i X_i$ subject to m linear constraints $\mathcal{F} = \{F_1, ..., F_m\}$, over n decision variables $\mathcal{X} = \{X_1, ..., X_n\}$ with binary domains $\mathcal{D} = \{D_1, ..., D_n\}$. We use the notation $\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, z \rangle$ to refer to any 0/1 ILP instance.

2.2 AND/OR Search Spaces for 0/1 Integer Linear Programs

The common way of solving 0/1 Integer Linear Programs is by search, namely to instantiate variables one at a time following a static or dynamic variable ordering. In the simplest case, this process defines an OR search tree, whose nodes represent states in the space of partial assignments. This search space does not capture independencies that appear in the structure of the problem. To remedy this problem an AND/OR search space was recently introduced in the context of general graphical models [4]. The AND/OR search space is defined using a backbone *pseudo-tree* [17].

Definition 1 (pseudo-tree). *Given an undirected graph $G = (V, E)$, a directed rooted tree $T = (V, E')$ defined on all its nodes is called* pseudo-tree *if any arc of G which is not included in E' is a back-arc, namely it connects a node to an ancestor in T.*

We will next specialize the AND/OR search space for a 0/1 ILP which is a special type of a graphical model.

AND/OR Search Trees. Given a 0/1 ILP instance $\langle \mathcal{X}, \mathcal{D}, \mathcal{F}, z \rangle$, its interaction graph G and a pseudo-tree T of G, the associated AND/OR search tree S_T has alternating levels of OR nodes and AND nodes. The OR nodes are labeled by X_i and correspond to the

variables. The AND nodes are labeled by $\langle X_i, x_i \rangle$ and correspond to value assignments in the domains of the variables. The structure of the AND/OR tree is based on the underlying pseudo-tree T of G. The root of the AND/OR search tree is an OR node, labeled with the root of T.

The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$, consistent along the path from the root, $path(x_i) = (\langle X_1, x_1 \rangle, ..., \langle X_{i-1}, x_{i-1} \rangle)$. The children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of variable X_i in T. Semantically, the OR states represent alternative ways of solving the problem, whereas the AND states represent problem decomposition into independent subproblems, all of which need be solved. When the pseudo-tree is a chain, the AND/OR search tree coincides with the regular OR search tree.

A *solution tree* SOL_{S_T} of S_T is an AND/OR subtree such that: (i) it contains the root of S_T; (ii) if a nonterminal AND node $n \in S_T$ is in SOL_{S_T} then all of its children are in SOL_{S_T}; (iii) if a nonterminal OR node $n \in S_T$ is in SOL_{S_T} then exactly one of its children is in SOL_{S_T}.

Example 1. For illustration consider the 0/1 ILP with 6 decision variables A, B, C, D, E, F and 4 linear constraints $F_1(A, B, C)$, $F_2(B, C, D)$, $F_3(A, B, E)$, $F_4(A, E, F)$ from Figure 1(a). The objective function to be minimized is $z = 7A + B - 2C + 5D - 6E + 8F$. The pseudo-tree arrangement of the interaction graph, together with the back-arcs (dotted lines) are given in Figure 1(b). Figure 1(c) shows the corresponding AND/OR search tree.

Arc Labels and Node Values. The arcs from OR nodes X_i to AND nodes $\langle X_i, x_i \rangle$ in the AND/OR search tree S_T are annotated by *labels* derived from the objective function.

Definition 2 (label). *Given a 0/1 ILP instance with objective function* $z = \sum_{i=1}^{n} c_i X_i$ *and a corresponding AND/OR search tree* S_T*, the* label $l(n, m)$ *of the arc from the OR node* $n = X_i$ *to the AND node* $m = \langle X_i, x_i \rangle$ *is defined as* $l(n, m) = c_i \cdot x_i$.

Given a labeled AND/OR search tree, each node can be associated with a *value* [4].

Definition 3 (value). *The* value $v(n)$ *of a node* $n \in S_T$ *is defined recursively as follows: (i) if* $n = \langle X_i, x_i \rangle$ *is a terminal AND node then* $v(n) = 0$*; (ii) if* $n = \langle X_i, x_i \rangle$ *is an internal AND node then* $v(n) = \sum_{m \in succ(n)} v(m)$*; (iii) if* $n = X_i$ *is an internal OR node then* $v(n) = min_{m \in succ(n)}(l(n, m) + v(m))$*, where* $succ(n)$ *are the children of* n *in* S_T.

It is easy to see that the value $v(n)$ of a node in the AND/OR search tree S_T is the minimal cost solution to the subproblem rooted at n, subject to the current variable instantiation along the path from the root to n. If n is the root of S_T, then $v(n)$ is the minimal cost solution to the initial problem [6].

Clearly, the AND/OR search tree can be traversed to compute each node's value either by a depth-first or best-first search algorithm.

AND/OR Search Graphs. The AND/OR search tree may contain nodes that root identical subtrees (in particular, subproblems with identical optimal solutions). These are called *unifiable*. When unifiable nodes are merged, the search tree becomes a graph and its size becomes smaller. Some unifiable nodes can be identified based on their *contexts*.

Algorithm 1. AOBB$_g$

Data: A 0/1 ILP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{F}, z)$, pseudo-tree T, root s.
Result: Minimal cost solution to $\mathcal{P}$.
1. Create a list OPEN, consisting solely of the start node s. Set $v(s) = \infty$.
2. **until** s is labeled SOLVED, **do**:
   (a) Remove the first node n from OPEN and add it to CLOSED.
   (b) If n is an AND node, then set $v(n) = cache(n)$.
   (c) Try to prune the subtree below n, as follows: if for some ancestor m of n in CLOSED, $f_h(m) \geq v(m)$, then set $v(n) = \infty$ and continue from step (e).
   (d) Expand node n generating all its successor nodes n_i. For each new node n_i compute $h(n_i)$; if n_i is an AND node then set $v(n_i) = 0$, else if n_i is an OR node then set $v(n_i) = \infty$; add n_i on top of OPEN.
   (e) Create a set S. If n has no successors then label n SOLVED and add it to S.
   (f) **until** S is empty, **do**:
      i. Remove the first node m from S.
      ii. Update the value $v(p)$ of the parent p of m as follows:
         A. **if** p is an AND node **then** $v(p) = v(p) + v(m)$.
         B. **if** p is an OR node **then** $v(p) = min(v(p), l(p, m) + v(m))$. Save the AND value $v(m)$ in cache by setting $cache(m) = v(m)$, if $v(m) \neq \infty$.
      iii. Remove m from the successors of p. If p has no successors left, label p SOLVED and add it to S. Remove m from CLOSED.
3. **return** $v(s)$.

Definition 4 (context). *Given a 0/1 ILP instance and the corresponding AND/OR search tree S_T relative to a pseudo-tree T, the* context *of any AND node $\langle X_i, x_i \rangle \in S_T$, denoted by $context(X_i)$, is defined as the set of ancestors of X_i in T, including X_i, that are connected to descendants of X_i.*

It is easy to verify that any two nodes having the same context represent the same subproblem. Therefore, we can solve P_{X_i}, the subproblem rooted at X_i, once and use its optimal solution whenever the same subproblem is encountered again.

The *context-minimal* AND/OR search graph, denoted by G_T, is obtained by merging all the AND nodes that have the same context. It can be shown [4] that the size of the largest context is bounded by the induced width w^* of the interaction graph, extended with the pseudo-tree extra arcs, over the ordering given by the depth-first traversal of T (i.e. induced width of the pseudo-tree). Therefore,

Theorem 1 (complexity). *The complexity of any search algorithm traversing a context-minimal AND/OR search graph is time and space $O(exp(w^*))$, where w^* is the induced width of the underlying pseudo-tree [4].*

Example 2. Consider the context-minimal AND/OR search graph in Figure 1(d) of the pseudo-tree from Figure 1(b). Its size is far smaller that that of the AND/OR tree from Figure 1(c) (16 nodes vs. 36 nodes). The contexts of the nodes can be read from the pseudo-tree, as follows: $context(A)$ = {A}, $context(B)$ = {B,A}, $context(C)$ = {C,B}, $context(D)$ = {D}, $context(E)$ = {E,A} and $context(F)$ = {F}.

3 Algorithms Exploring the Context-Minimal AND/OR Graph

In this section we introduce two algorithms that explore a context-minimal AND/OR search graph in either a *depth-first* or *best-first* manner for solving optimization problems from the class of 0/1 ILP. First, we present the depth-first AND/OR Branch-and-Bound search algorithm (AOBB$_g$) which extends the 0/1 ILP algorithm presented in [6]

for searching AND/OR trees to searching AND/OR graphs. The algorithm specializes recent AND/OR graph search algorithms for general constraint optimization problems described in [1] to the 0/1 ILP case.

3.1 Depth-First AND/OR Branch-and-Bound Search

The AND/OR Branch-and-Bound search algorithm, denoted by $\texttt{AOBB}_g$, that explores the context-minimal AND/OR search graph in a depth-first manner is described in Algorithm 1. Its pruning strategy is similar to that of the Branch-and-Bound algorithm searching AND/OR trees developed in [6]. Specifically, each node n along the path from the root has associated a *static* heuristic function $h(n)$ underestimating $v(n)$ that can be computed efficiently by solving the linear relaxation (i.e. relaxing the integrality restrictions) of the subproblem rooted at n. The algorithm also improves the heuristic function dynamically during search. The *dynamic heuristic function* $f_h(n)$ is computed based on the search space below n that has already been explored, as described in [6], and is used to prune unpromising portions of the search space that cannot improve the best solution found so far.

$\texttt{AOBB}_g$ is restricted to a static variable ordering determined by the underlying pseudo-tree and explores the context-minimal AND/OR search graph via *full caching*. The algorithm saves previously computed results and retrieves them when the same nodes are encountered again, during search. A simple way of implementing the caching mechanism is to have a *cache table* for each variable X_k recording its context. Specifically, let us assume that the context of X_k is $context(X_k) = \{X_i, ..., X_k\}$. A cache table entry corresponds to a particular instantiation $\{x_i, ..., x_k\}$ of the variables in $context(X_k)$ and records the optimal cost solution to the subproblem P_{X_k}.

However, some tables might never get cache hits. These are called *dead-caches* [18,1]. In the context-minimal AND/OR search graph, dead-caches appear at nodes that have only one incoming arc. $\texttt{AOBB}_g$ needs to record only nodes that are likely to have additional incoming arcs, and some of these nodes can be determined by inspecting the pseudo-tree. Namely, if the context of a node includes that of its parent, then there is no need to store anything for that node, because it would be a dead-cache. For example, node B in the AND/OR search graph from Figure 1(d) is a dead-cache because its context includes the context of its parent A in the pseudo-tree from Figure 1(b).

If the memory requirements are prohibitive, rather than using full caching, $\texttt{AOBB}_g$ can be modified to use a memory bounded caching scheme that saves only those nodes whose context size can fit in the available memory, as suggested by [1].

3.2 Best-First AND/OR Search

The context-minimal AND/OR search graph can be traversed in a best-first rather than depth-first manner to compute the optimal cost solution to a 0/1 ILP. It is known that under conditions of admissibility and monotonicity of the guiding heuristic function, best-first search algorithms are guaranteed to expand the minimal number of nodes, at the expense of using additional memory [8].

Algorithm 2. AOBF$_g$

Data: A 0/1 ILP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{F}, z)$, pseudo-tree T, root s.
Result: Minimal cost solution to $\mathcal{P}$.
1. Create explicit graph G'_T, consisting solely of the start node s. Set $v(s) = h(s)$.
2. **until** s is labeled SOLVED, **do**:
   (a) Compute a *partial solution tree* by tracing down the *marked* arcs in G'_T from s and select any nonterminal tip node n.
   (b) Expand node n and add any new successor node n_i to G'_T. For each new node n_i set $v(n_i) = h(n_i)$. Label SOLVED any of these successors that are terminal nodes.
   (c) Create a set S containing node n.
   (d) **until** S is empty, **do**:
      i. Remove from S a node m such that m has no descendants in G'_T still in S.
      ii. Revise the value $v(m)$ as follows:
         A. **if** m is an AND node **then** $v(m) = \sum_{m_j \in succ(m)} v(m_j)$. If all the successor nodes are labeled SOLVED, then label node m SOLVED.
         B. **if** m is an OR node **then** $v(m) = min_{m_j \in succ(m)}(l(m, m_j) + v(m_j))$ and mark the arc through which this minimum is achieved. If the marked successor is labeled SOLVED, then label m SOLVED.
      iii. If m has been marked SOLVED or if the revised value $v(m)$ is different than the previous one, then add to S all those parents of m such that m is one of their successors through a marked arc.
3. **return** $v(s)$.

Our best-first AND/OR graph search algorithm, denoted by AOBF$_g$, that traverses the context-minimal AND/OR search graph is described in Algorithm 2. It specializes Nillson's AO* algorithm [19] to solving 0/1 ILPs and interleaves forward expansion of the best partial solution tree with a cost revision step that updates estimated node values. First, a top-down, graph-growing operation (`step 2.a`) finds the best partial solution tree by tracing down through the marked arcs of the explicit AND/OR search graph G'_T. These previously computed marks indicate the current best partial solution tree from each node in G'_T. One of the nonterminal leaf nodes n of this best partial solution tree is then expanded, and a static heuristic estimate $h(n_i)$ is assigned to its successors (`step 2.b`). The successors of an AND node $n = \langle X_j, x_j \rangle$ are X_j's children in the pseudo-tree, while the successors of an OR node $n = X_j$ correspond to X_j's domain values. Notice that when expanding an OR node, the algorithm does not generate AND children that are already present in the explicit search graph G'_T. All these identical AND nodes in G'_T are easily recognized based on their contexts.

The second operation in AOBF$_g$ is a bottom-up, cost revision, arc marking, SOLVE-labeling procedure (`step 2.c`). Starting with the node just expanded n, the procedure revises its value $v(n)$ (using the newly computed values of its successors) and marks the outgoing arcs on the estimated best path to terminal nodes. This revised value is then propagated upwards in the graph. The revised cost $v(n)$ is an updated estimate of the cost of an optimal solution to the subproblem rooted at n. If we assume the monotone restriction on h, the algorithm considers only those ancestors that root best partial solution subtrees containing descendants with revised values (`step 2.d.iii`). The optimal cost solution to the initial problem is obtained when the root node s is solved.

The static heuristic function $h(n)$ is obtained by solving the linear relaxation of the subproblem P_n rooted at node n in the search graph, subject to the current variable instantiation of the best partial solution tree. If P_n is infeasible then we assume $h(n) = \infty$. The bottom-up operation of AOBF$_g$ will then propagate this high cost upward, which eliminates any chances that a subtree containing this node might be selected as an estimated best solution subtree.

4 Dynamic Variable Orderings

It is well known that variable selection may influence dramatically search performance. Recent work by [6,7] showed how several dynamic variable orderings affect depth-first Branch-and-Bound search on AND/OR trees. One extension, called AND/OR Branch-and-Bound with Partial Variable Ordering (AOBB$_t$+PVO) that orders dynamically the variables forming chains in the pseudo-tree, was shown to outperform significantly static AND/OR as well as state-of-the-art OR Branch-and-Bound solvers for general COPs and in particular for 0/1 ILPs [6,7]. Next, we extend the idea of partial variable ordering to best-first search on AND/OR trees.

Partial Variable Orderings. AOBF$_g$ described in the previous section is restricted to a static variable ordering determined by the pseudo-tree arrangement. The mechanism of identifying unifiable AND nodes based solely on their contexts is hard to extend when variables are instantiated in a different order than that dictated by the pseudo-tree, and therefore it cannot be used to accommodate dynamic variable orderings. If we explore the AND/OR search tree we can use dynamic variable orderings while exploring the AND/OR search tree in a best-first manner.

Best-first AND/OR search with Partial Variable Ordering (AOBF$_t$+PVO) traverses an AND/OR search tree by combining the static graph-based problem decomposition given by a pseudo-tree with a dynamic semantic variable selection heuristic. We illustrate the idea with an example. Consider the pseudo-tree from Figure 1(a) inducing the following variable group ordering: {A,B}, {C,D}, {E,F}; which dictates that variables {A,B} should be considered before {C,D} and {E,F}. Variables in each group can be dynamically ordered based on a second, independent semantic heuristic (e.g., min reduced cost, min pseudo cost, etc.). Notice that after variables {A,B} are instantiated, the problem decomposes into two independent components that can be solved separately.

5 Experiments

In this section we evaluate empirically the performance of the best-first AND/OR search algorithms on several benchmarks for 0/1 ILP including problem instances from the MIPLIB library[1], combinatorial auctions, uncapacitated warehouse location problems and MAX-SAT problems. All our experiments were done on a 2.4GHz Pentium IV with 2GB of RAM, running Windows XP.

We consider two classes of best-first search algorithms exploring an AND/OR search tree and using either a static variable ordering (SVO) or a partial variable ordering (PVO). The algorithms are denoted by AOBF$_t$+SVO and AOBF$_t$+PVO, respectively. We also consider two classes of depth-first and best-first search algorithms traversing context-minimal AND/OR search graphs, both restricted to a static variable ordering and denoted by AOBB$_g$+SVO and AOBF$_g$+SVO, respectively. For comparison we include results obtained with two depth-first AND/OR Branch-and-Bound algorithms without caching developed recently in [6] and denoted by AOBB$_t$+SVO and AOBB$_t$+PVO, respectively. The guiding heuristic of the AND/OR search algorithms is

[1] http://miplib.zib.de/miplib2003.php

Table 1. Results for MIPLIB problem instances

miplib	n c	w^* h		BB (lp_solve)	AOBB$_t$ SVO	AOBF$_t$ SVO	AOBB$_g$ SVO	AOBF$_g$ SVO	AOBB$_t$ PVO	AOBF$_t$ PVO
p0033	33	19	time	5.34	0.31	0.27	**0.19**	0.39	0.28	0.33
	15	21	nodes	15,832	438	403	339	281	428	374
p0040	40	19	time	**0.08**	0.11	0.11	0.11	0.09	0.27	0.18
	23	23	nodes	134	113	100	113	100	142	121
p0201	201	120	time	98.21	91.36	**71.62**	90.52	76.05	84.36	91.45
	133	142	nodes	23,742	15,187	10,387	15,130	10,387	9,653	8,261
lseu	89	57	time	282.27	89.04	**35.44**	86.88	36.50	44.85	36.45
	28	68	nodes	386,122	70,322	21,396	63,906	19,692	30,202	18,383

computed by solving the linear relaxation of the current subproblem. We used the SIMPLEX implementation from the open-source `lp_solve`[2] library. The guiding pseudo-trees used by the AND/OR algorithms were constructed using the hypergraph partitioning heuristic described in [6].

To ensure a fair comparison of the 0/1 ILP algorithms we used as reference the ILP Branch-and-Bound solver (`BB`) available in the `lp_solve` library and did not rely on a commercial ILP solver like CPLEX[3], whose excellent performance is determined, in many cases, by its powerful nogood recording mechanism (e.g., Gomoroy fractional cuts). For the MAX-SAT instances we compare, in addition, with three specialized solvers: `MaxSolver` [16], a DPLL-based algorithm that uses a 0/1 non-linear integer formulation of the MAX-SAT problem, `toolbar3` [20], a classic OR Branch-and-Bound algorithm that solves MAX-SAT as a Weighted CSP problem, and `PBS` [21], a DPLL-based solver capable of propagating and learning pseudo-boolean constraints as well as clauses. `MaxSolver` and `toolbar3` were shown to perform very well on random MAX-SAT instances with high graph connectivity [20], whereas `PBS` exhibits better performance on relatively sparse MAX-SAT instances [16].

The algorithms `BB`, `AOBB`$_t$`+PVO` and `AOBF`$_t$`+PVO` used a dynamic semantic variable selection heuristic based on *reduced costs* (i.e. dual values) [2]. Specifically, the next fractional variable to instantiate has the smallest reduced cost. Ties are broken lexicographically.

We report the average effort, as CPU time (in seconds) and number of nodes visited (which is equivalent to the number of times the SIMPLEX routine was called to solve the linear relaxation of the current subproblem), required for proving optimality of the solution. We also record the number of variables (n), the number of constraints (c), the depth of the pseudo-trees (h) and the induced width of the graphs (w^*) obtained for the test instances. The best performance points are highlighted.

5.1 MIPLIB

MIPLIB is a library of Mixed Integer Linear Programming instances that is commonly used for benchmarking integer programming algorithms. For our purpose we selected

[2] lp_solve 5.5.0.9 is available at http://lpsolve.sourceforge.net/5.5/

[3] http://www.ilog.com/products/cplex/

Table 2. Results for combinatorial auction problem instances

auction	n c	w^* h		BB (lp_solve)	AOBB$_t$ SVO	AOBF$_t$ SVO	AOBB$_g$ SVO	AOBF$_g$ SVO	AOBB$_t$ PVO	AOBF$_t$ PVO
reg-upv	203	145	time	5.95	7.83	6.82	8.08	6.79	7.02	**3.66**
b200g50	87	162	nodes	658	500	310	500	310	533	189
reg-upv	251	166	time	45.42	19.24	14.92	19.37	15.07	16.59	**8.31**
b250g75	124	190	nodes	3,321	663	333	663	333	620	170
reg-upv	304	173	time	198.07	155.76	90.61	148.34	91.67	125.95	**48.67**
b300g100	157	204	nodes	7,756	2,561	1,084	2,561	1,084	2,617	569
reg-npv	202	140	time	4.41	4.58	3.52	4.56	3.64	4.75	**1.66**
b200g50	88	161	nodes	491	280	158	280	158	367	64
reg-npv	251	160	time	18.04	15.52	10.06	15.39	10.21	15.35	**4.55**
b250g75	120	187	nodes	1,177	593	250	593	250	659	95
reg-npv	302	172	time	185.65	69.81	50.55	69.27	51.24	62.17	**24.14**
b300g100	156	206	nodes	7,131	1,195	537	1,195	537	1,335	237

four 0/1 ILP instances of increasing difficulty. Table 1 reports a summary of the experiment. We see that, overall, the best-first AND/OR search algorithms explore the smallest search space, which sometimes translates into significant time savings. For example, on `lseu`, one of the hardest instances, AOBF$_t$+SVO causes a speedup of 2.5 over AOBB$_t$+SVO, while exploring a search space 3 times smaller. Similarly, AOBF$_g$+SVO is 2.4 times faster than AOBB$_g$+SVO, while AOBF$_t$+PVO is only slightly better than AOBB$_t$+PVO. We observe that caching did not help much on these instances, namely the difference in number of nodes expanded by traversing the AND/OR tree versus context-minimal AND/OR graph was not significant enough to outweigh the overhead. This indeed can be explained by the relatively high induced widths.

5.2 Combinatorial Auctions

In **combinatorial auctions** (CA), an auctioneer has a set of goods, $M = \{1, 2, ..., m\}$ to sell and the buyers submit a set of bids, $\mathcal{B} = \{B_1, B_2, ..., B_n\}$. A bid is a tuple $B_j = \langle S_j, p_j \rangle$, where $S_j \subseteq M$ is a set of goods and $p_j \geq 0$ is a price. The winner determination problem is to label the bids as winning or loosing so as to maximize the sum of the accepted bid prices under the constraint that each good is allocated to at most one bid. We used the 0/1 ILP formulation described in [6].

Table 2 shows the results for experiments with 6 classes of moderate size combinatorial auctions from [6]. These auctions were drawn from the `regions` distribution of the CATS 2.0 test suite [14] and simulate the auction of radio spectrum in which a government sells the right to use specific segments of spectrum in different geographical areas. We observe that AOBF$_t$+PVO is the best performing algorithm, exploring the smallest search space. If we look for example at the 300 bid problem instances from the `reg-npv` distribution, AOBF$_t$+PVO is on average about 2.5 times faster than the other AND/OR algorithms and the search space explored is about 4 times smaller. When compared with the classic OR Branch-and-Bound algorithm, AOBF$_t$+PVO causes an even higher speedup, exploring a search space 30 times smaller. Notice that the AND/OR

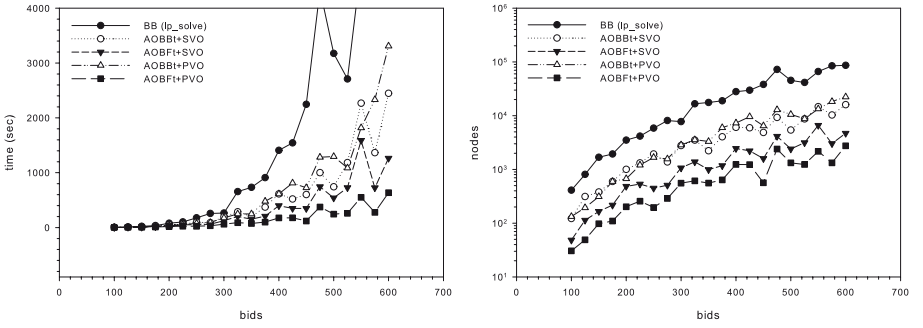


Fig. 2. Results for `regions-upv` auctions with 100 goods and increasing number of bids

graph search algorithms $\texttt{AOBB}_g\texttt{+SVO}$ and $\texttt{AOBF}_g\texttt{+SVO}$ expanded the same number of nodes as the AND/OR tree search algorithms $\texttt{AOBB}_t\texttt{+SVO}$ and $\texttt{AOBF}_t\texttt{+SVO}$, respectively. This indicates that, for these problem classes, the context-minimal AND/OR search graph is a tree and all cache entries are actually dead. Therefore the gain observed moving from depth-first Branch-and-Bound to best-first search is primarily due to the optimal cost, which bounds the horizon of best-first, but not depth-first search.

Figure 2 displays the results for experiments with `regions-upv` auctions having 100 goods and increasing number of bids. Each data point represents an average over 10 random samples. We observe that $\texttt{AOBF}_t\texttt{+PVO}$ is the best performing algorithm and, on some of the hardest instances, it outperforms its competitors with up to one order of magnitude in terms of both CPU time and size of the search space explored. This demonstrates the power of the dynamic variable selection heuristic which is able in this case to cut the search tree dramatically. The best-first and depth-first AND/OR algorithms using static variable orderings explored the same search space, namely the context-minimal AND/OR graph was a tree (in Figure 2 we only plotted $\texttt{AOBB}_t\texttt{+SVO}$ and $\texttt{AOBF}_t\texttt{+SVO}$ respectively) and therefore the gain of best-first over depth-first Branch-and-Bound search was due to the optimal cost bound, as before.

5.3 Uncapacitated Warehouse Location Problems

In the **uncapacitated warehouse location problem** (UWLP) a company considers opening m warehouses at some candidate locations in order to supply its n existing stores. The objective is to determine which warehouse to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimized. Each store must be supplied by exactly one warehouse. We used the 0/1 ILP formulation from [6].

Table 3 displays the results obtained on 6 randomly generated UWLP problem instances[4] with 50 warehouses and 200 stores. The warehouse opening and store supply costs were chosen uniformly randomly between 0 and 1000. These are large problems with 10,050 variables and 10,500 constraints, but having relatively shallow pseudo-trees with depths of 123. We can see that $\texttt{AOBF}_t\texttt{+PVO}$ dominates in all test cases,

[4] Problem generator from http://www.mpi-sb.mpg.de/units/ag1/projects/benchmarks/UflLib/

Table 3. Results for uncapacitated warehouse location problem instances

uwlp	n	w^*		BB	AOBB$_t$	AOBF$_t$	AOBB$_g$	AOBF$_g$	AOBB$_t$	AOBF$_t$
50x200	c	h		(lp_solve)	SVO	SVO	SVO	SVO	PVO	PVO
uwlp001	10,050	50	time	48.61	69.55	44.39	69.53	42.70	25.63	**20.22**
	10,500	123	nodes	86	62	20	62	20	20	7
uwlp004	10,050	50	time	61.08	46.39	37.58	46.42	36.27	17.47	**15.49**
	10,500	123	nodes	142	46	24	46	24	10	3
uwlp013	10,050	50	time	13693.76	116.19	111.28	116.25	105.72	78.86	**74.53**
	10,500	123	nodes	14,846	44	26	44	26	24	13
uwlp018	10,050	50	time	1477.74	161.03	54.58	161.05	52.41	59.52	**32.33**
	10,500	123	nodes	2,666	146	21	146	21	37	8
uwlp020	10,050	50	time	2179.39	190.77	87.58	190.81	83.70	68.91	**48.33**
	10,500	123	nodes	3,668	138	33	138	33	36	10
uwlp024	10,050	50	time	2177.67	125.85	86.64	125.86	82.27	28.19	**25.89**
	10,500	123	nodes	3,288	71	31	71	31	16	4

Table 4. Results for `pret` MAX-SAT problem instances

pret	n	w^*		MaxSolver	toolbar3	PBS	BB	AOBB$_t$	AOBF$_t$	AOBB$_g$	AOBF$_g$	AOBB$_t$	AOBF$_t$
	c	h					(lp_solve)	SVO	SVO	SVO	SVO	PVO	PVO
pret60-40	60	6	time	9.47	53.89	**0.00**	27208.09	7.88	7.56	7.38	**3.58**	8.41	8.70
	160	13	nodes		7,297,773	565	4,194,302	1,255	1,202	1,216	568	1,216	1,326
pret60-60	60	6	time	9.48	53.66	**0.00**	27628.52	8.56	8.08	7.30	**3.56**	8.70	8.31
	160	13	nodes		7,297,773	495	4,194,302	1,259	1,184	1,140	538	1,247	1,206
pret60-75	60	6	time	9.37	53.52	**0.00**	26990.70	6.97	7.38	6.34	**3.08**	6.80	8.42
	160	13	nodes		7,297,773	543	4,194,302	1,124	1,145	1,067	506	1,089	1,149
pret150-40	150	6	time	-	-	**0.02**	-	95.11	101.78	75.19	**19.70**	108.84	101.97
	400	15	nodes			2,592		6,625	6,535	5,625	1,379	7,152	6,246
pret150-60	150	6	time	-	-	**0.01**	-	98.88	106.36	78.25	**19.75**	112.64	102.28
	400	15	nodes			2,873		6,851	6,723	5,813	1,393	7,347	6,375
pret150-75	150	6	time	-	-	**0.02**	-	108.14	98.95	84.97	**20.95**	115.16	103.03
	400	15	nodes			2,898		7,311	6,282	6,114	1,430	7,452	6,394

outperforming the classic `BB` with several orders of magnitude in terms of both running time and size of the search space explored. In `uwlp013` for example, one of the hardest instances, AOBF$_t$+PVO causes a speed-up of 186 over the classic OR Branch-and-Bound algorithm, exploring a search tree 1,142 times smaller. When comparing the best-first AND/OR search algorithms with the depth-first AND/OR Branch-and-Bound algorithms we observe only minor savings in running time. This is because the corresponding AND/OR search spaces are already small enough and the savings in number of nodes caused by the best-first AND/OR search algorithms do not translate into time savings as well. Notice that for this problem class the context minimal AND/OR search graph explored by the AOBB$_g$+SVO and AOBF$_g$+SVO algorithms is in fact a tree and therefore all cache entries are dead.

5.4 MAX-SAT Problems

Given a set of Boolean variables the goal of **maximum satisfiability** (MAX-SAT) is to find a truth assignment to the variables that violates the least number of clauses. The

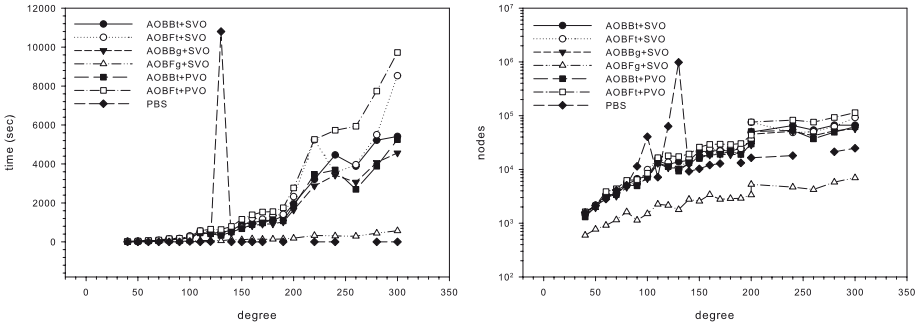


Fig. 3. Results for dubois MAX-SAT problem instances

MAX-SAT problem can be formulated as a 0/1 ILP as described in [15]. We experimented with problem classes pret and dubois from the SATLIB[5] library, which were previously shown to be difficult for 0/1 ILP solvers (e.g., CPLEX 8.1) [20].

Table 4 shows the results for experiments with 6 pret instances. These are unsatisfiable instances of graph 2-coloring with parity constraints. The size of these problems is relatively small (60 variables with 160 clauses for pret60 and 150 variables with 400 clauses for pret150, respectively). We observe that, for this problem class, AOBF$_g$+SVO is the best performing algorithm amongst the 0/1 ILP solvers. For example, on pret150-75, the hardest instance, AOBF$_g$+SVO is 4 times faster than AOBB$_g$+SVO and the search space explored is 6 times smaller. This is due to the problem structure which is partially captured by a very small context with size 6 and a shallow pseudo-tree with depth 13. Overall, PBS offers the best performance on this dataset. However, the search space explored by AOBF$_g$+SVO appears to be the smallest. This indicates that the computational overhead of AOBF$_g$+SVO is due to evaluating its guiding lower bound (i.e., solving the linear relaxation of the current subproblem via SIMPLEX). Notice that BB, MaxSolver and toolbar3 solvers were not able to solve any of the pret150 instances within a 10 hour time limit.

Figure 3 displays the results for experiments with random dubois instances with increasing number of variables. These are 3-SAT instances with $3 \times \mathit{degree}$ variables and $8 \times \mathit{degree}$ clauses, each of them having 3 literals. As in the previous test case, the dubois instances have very small contexts of size 6 and shallow pseudo-trees with depths ranging from 10 to 20. We can see that AOBF$_g$+SVO takes full advantage of the relatively small context-minimal AND/OR search graph and, on some of the larger instances, it outperforms its 0/1 ILP competitors with up to one order of magnitude in terms of both running time and number of nodes expanded. PBS is again the overall best-performing algorithm, however it fails to solve 4 test instances: on instance dubois130 it exceeds the 3 hour time limit, whereas on instances dubois180, dubois200 and dubois260 the clause/pseudo-boolean constraint learning mechanism causes the solver to run out of memory. We observe that in this domain also AOBF$_g$+SVO explores the smallest search space as compared to PBS, but its

[5] http://www.satlib.org/

computational overhead does not pay off in terms of running time. `BB`, `MaxSolver` and `toolbar3` performed very poorly on this dataset and they were not able to solve any of test instances within a 3 hour time limit.

6 Conclusion

In this paper we introduced a best-first AND/OR search algorithm which extends the classic `AO*` algorithm and traverses a context-minimal AND/OR search graph for solving 0/1 ILPs. Since variable selection can influence dramatically the search performance, we also proposed a best-first search algorithm that explores an AND/OR search tree, rather than a graph, and incorporates dynamic variable ordering heuristics. Our empirical evaluation demonstrated on a variety of 0/1 ILP benchmark problems that the best-first AND/OR search algorithms outperform the depth-first OR and AND/OR Branch-and-Bound algorithms sometimes by several orders of magnitude in terms of both running time and size of the search space explored.

Our best-first AND/OR search approach leaves room for future improvements, which are likely to make it more efficient in practice. For instance, it can be modified to incorporate *cutting planes* to tighten the linear relaxation of the current subproblem. The space required by the best-first AND/OR search can be enormous, due to the fact that all the nodes generated by the algorithm have to be saved prior to termination. Therefore, the algorithm can be extended to incorporate a memory bounding scheme similar to the one suggested in [22]. Finally, we can incorporate good initial upper bound techniques (using incomplete schemes), which in some cases can allow a best-first performance using depth-first AND/OR Branch-and-Bound algorithms.

Acknowledgments

We would like to thank the anonymous reviewers for commenting on an earlier version of the paper. This work has been partially supported by the NSF grant IIS-0412854.

References

1. R. Marinescu and R. Dechter. Memory intensive branch-and-bound search for graphical models. *In National Conference on Artificial Intelligence (AAAI'06)*, 2006.
2. G. Nemhauser and L. Wolsey. *Integer and combinatorial optimization.* Wiley, 1988.
3. E. Lawler and D. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, 1966.
4. R. Dechter and R. Mateescu. And/or search spaces for graphical models. *Artificial Intelligence*, 2006.
5. R. Marinescu and R. Dechter. And/or branch-and-bound for graphical models. *In International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 224–229, 2005.
6. R. Marinescu and R. Dechter. And/or branch-and-bound search for pure 0/1 integer linear programming problems. *In International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization (CPAIOR'06)*, pages 152–166, 2006.

7. R. Marinescu and R. Dechter. Dynamic orderings for and/or branch-and-bound search in graphical models. *In European Conference on Artificial Intelligence (ECAI'06)*, pages 138–142, 2006.
8. R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of a*. *In Journal of ACM*, 32(3):505–536, 1985.
9. G.B. Dantzig. Maximization of a linear function of variables subject to linear inequalities. *Activity Analysis of Production and Allocation*, 1951.
10. M. Vasquez and J. Hao. A hybrid approach for the 0/1 multidimensional knapsack approach. *In International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 328–333, 2001.
11. W. Shih. A branch-and-bound method for the multiconstraint 0/1 knapsack problem. *Journal of the Operational Research Society*, 30:369–378, 1979.
12. B. Gavish and H. Pirkul. Allocation of data bases and processors in a distributed computing system. *Management of Distributed Data Processing*, 31:215–231, 1982.
13. T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. *In International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 542–547, 1999.
14. K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. *In ACM Electronic Commerce*, pages 66–76, 2000.
15. S. Joy, J. Mitchell, and B. Borchers. A branch and cut algorithm for max-sat and weighted max-sat. *In Satisfiability Problem: Theory and Applications*, pages 519–536, 1997.
16. Z. Xing and W. Zhang. Efficient strategies for (weighted) maximum satisfiability. *In Constraint Programming (CP'04)*, pages 660–705, 2004.
17. E. Freuder and M. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. *In International Joint Conference on Artificial Intelligence (IJCAI'85)*, pages 1076–1078, 1985.
18. A. Darwiche. Recursive conditioning. *Artificial Intelligence*, 126(1-2):5–41, 2001.
19. K. Nillson. *Principles of Artificial Intelligence.* Tioga, 1980.
20. S. de Givry, J. Larrosa, and T. Schiex. Solving max-sat as weighted csp. *In Constraint Programming (CP'03)*, 2003.
21. F. Aloul, A. Ramani, I. Markov, and K. Sakallah. Pbs: A backtrack search pseudo-boolean solver. *In Symposium on the Theory and Applications of Satisfiability Testing (SAT'02)*, 2002.
22. P. Chakrabati, S. Ghose, A. Acharya, and S. de Sarkar. Heuristic search in restricted memory. *In Artificial Intelligence*, 3(41):197–221, 1989.

A Position-Based Propagator for the Open-Shop Problem

Jean-Noël Monette, Yves Deville, and Pierre Dupont

Department of Computing Sciences and Engineering
Université catholique de Louvain
{jmonette,yde,pdupont}@info.ucl.ac.be

Abstract. The Open-Shop Problem is a hard problem that can be solved using Constraint Programming or Operation Research methods. Existing techniques are efficient at reducing the search tree but they usually do not consider the absolute ordering of the tasks. In this work, we develop a new propagator for the One-Machine Non-Preemptive Problem, the basic constraint for the Open-Shop Problem. This propagator takes this additional information into account allowing, in most cases, a reduction of the search tree. The underlying principle is to use shaving on the positions. Our propagator applies on one machine or one job and its time complexity is in $\mathcal{O}(N^2 \log N)$, where N is either the number of jobs or machines. Experiments on the Open-Shop Problem show that the propagator adds pruning to state-of-the-art constraint satisfaction techniques to solve this problem.

1 Introduction

Open-Shop Problems (OSP) are disjunctive scheduling problems known to be really hard to solve. Up to now, some problems with less than 50 tasks remain unsolved, although several powerful techniques and algorithms mentioned below have been designed to reduce efficiently the search.

The Open-Shop Problem aims at finding the order in which a set of tasks is executed such as to minimize the makespan, *i.e.* the ending time of the latest tasks. Each task must be executed on a particular machine for a given duration and without interruption. A machine cannot process two tasks at the same time. Additionally, tasks are part of jobs and two tasks of the same job cannot be processed at the same time. There is no predefined ordering between tasks.

This problem fits very well in the framework of Constraints Programming (CP). Propagators have been developed to remove inconsistent values from the domains as early as possible in order to reduce the size of the search tree. The more prominent techniques are Edge-Finding (EF) and Not-First-Not-Last (NFNL). Edge-Finding [1,2,3,4] consists in testing whether a particular task must start before or after a set of tasks. It can be implemented with a time complexity of $\mathcal{O}(N \log N)$ where N is the number of tasks on one machine or one job. Not-First-Not-Last [5,6,7] checks if a task can be the first or the last among a set of tasks. Its best time complexity is $\mathcal{O}(N \log N)$.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 186–199, 2007.

Shaving [5,6,8] is an orthogonal technique that performs well in practice for solving the OSP. It consists in iteratively assigning to a variable its possible values and checking if this assignment leads to inconsistency. In that case, the value is removed from the domain of the variable. Every constraint can be propagated to check consistency until the fixpoint is observed. Since it can be costly to reach this fixpoint, simpler propagators are used. For instance, in [6], only Edge-Finding is used to look for inconsistencies. Even so, shaving is costly because the size of the domain of the starting time variables can be huge. For this reason, shaving in OSP usually considers only the bounds of the domain.

We propose here a new propagator for the One-Machine Non-preemptive Problem that exploits information given by the position of the tasks. This idea has already been used successfully in [9], [10] and [11]. The first work uses the positions as permutation variables in a sorting constraint. An extension of Edge-Finding is also applied. Secondly, [10] proposes a possible way to decide if a task can start at some position looking at the number of other tasks that can come before and after this task. Finally, [11] extends this idea proposing tighter bounds with an algorithm running in $\mathcal{O}(N^3)$

In this paper, we present an alternative way to use the position of the tasks based on the idea of shaving. For each possible position of a task, lower and upper bounds on the possible starting time of the task are computed using the duration and the domain of the variables of the tasks in the same job or machine. The resulting propagator is applied on the tasks that are part of the same job or machine with a time complexity of $\mathcal{O}(N^2 \log N)$, where N is the number of tasks that are part of the job or that must be processed on the machine. This propagator permits additional pruning that is not performed by NFNL and EF and permits to detect about 14 % extra inconsistent nodes of the search tree on a standard benchmark [12].

The next section explains the problem under interest and its mapping in CP. Section 3 presents the new propagator and Section 4 describes experimental results assessing the pruning efficiency of the technique. In the last section, conclusions are drawn as well as directions for future work.

2 The One Machine Non-preemptive Problem

The OSP is an optimization problem that can be solved using branch-and-bound techniques. The goal of the optimization is to minimize the makespan, *i.e.* the ending time of the latest task. Branch-and-bound consists in solving successive feasibility versions of our problem. The feasibility version consists in determining if there exists a solution with a makespan smaller than a fixed value. Each time a solution with makespan m is found, another solution is searched with the constraint that its makespan is at most $m - 1$. When no more solution exists, the last solution found is an optimal one.

The feasibility version of the OSP can be stated as the conjunction of smaller problems called One Machine Non-preemptive Problem (1NP). This problem aims at scheduling a set of tasks on a machine such that there is only one not

interruptible task processed at a time. Each task is given a duration, an earliest and a latest starting times. To model the OSP, it is sufficient to define a 1NP for every machine and every job (and to link them with the makespan). Indeed, jobs and machines in the OSP have the same behavior: No two tasks associated with a same job or a same machine can be processed simultaneously. The 1NP is also the basis for other disjunctive problems such as the Job-Shop Problem for instance.

Formally, the 1NP is defined as follows: T is the set of tasks that must be processed and N is its cardinality. For each task $t \in T$, $d(t)$, initial $est(t)$ and initial $lst(t)$ are given and denote respectively the duration, earliest and latest starting times of the task t. The problem is to find for each task t, the value $S(t)$ of its starting time such that $est(t) \leq S(t) \leq lst(t)$ and without task overlap: $\forall t_1, t_2 \in T, S(t_1) + d(t_1) \leq S(t_2) \vee S(t_2) + d(t_2) \leq S(t_1)$.

2.1 Problem Modelling in CP

To model the 1NP, several variables are defined for each task $t \in T$. An integer variable $S(t)$ represents the starting time of the task t. Its domain ranges from the earliest starting time to the latest starting time of the task ($dom(S(t)) = [est(t), lst(t)]$). To model the relative order between the tasks, a set variable $B(t)$ represents the set of tasks that come before the task t. Its initial domain is $dom(B(t)) = [\emptyset, \{u | u \in T, u \neq t\}]$. Indeed, initially no task is known to come before t and all the tasks might come before t. The symbol $\overline{B}(t)$ (resp. $\underline{B}(t)$) represents the upper (resp. lower) bound of the variable $B(t)$. Furthermore, an additional variable $P(t)$ represents explicitly the absolute order (or the position) of the tasks in the machine. The domain of this variable ranges from 0 to $N-1$ with N being the number of tasks to be processed. The link between the relative and absolute orders of the tasks is that $P(t)$ represents the size of $B(t)$.

The starting time and the relative ordering between tasks are commonly used in the modelling of disjunctive scheduling. The use of an absolute order comes from [9] where the author solves the Job-Shop Problem fixing the permutations of task orders. In their proposed formulation, a variable is defined for the starting time of each task, a variable for the starting time of the task in each position and a variable for the position of each task. Those three sets of variables are linked together with a sorting constraint and various reduction rules are then defined. As an initial approach, we chose here for simplicity not to use the variables for the starting time of the task in each position.

2.2 Constraints

With three complementary representations, the 1NP can be equivalently expressed using anyone of the three following sets of constraints stating that two tasks cannot be processed at the same time.

1. $\forall t_1, t_2 \in T, \quad (S(t_1) + d(t_1) \leq S(t_2)) \quad \vee \quad (S(t_2) + d(t_2) \leq S(t_1))$
2. $\forall t_1, t_2 \in T, \quad (t_1 \in B(t_2)) \quad \vee \quad (t_2 \in B(t_1))$
3. $\forall t_1, t_2 \in T, \quad (P(t_1) < P(t_2)) \quad \vee \quad (P(t_2) < P(t_1))$

Our model uses the three sets of constraints to speed-up propagation. Additionally, the following channeling constraints ensure the consistency between variables of each representation. The position of a task t is the number of tasks that come before t ($|B(t)| = P(t)$). Also, a task t_1 ends before another task t_2 starts if and only if the position of t_1 is less than the position of t_2 ($S(t_1) + d(t_1) \leq S(t_2) \Leftrightarrow t_1 \in B(t_2) \Leftrightarrow P(t_1) < P(t_2)$).

In addition to these basic constraints, other redundant constraints can be defined. First, if t_1 comes before t_2, every task that comes before t_1 comes also before t_2 ($t_1 \in B(t_2) \Leftrightarrow B(t_1) \subset B(t_2)$). An AllDifferent constraint is also defined on the position variables ($alldiff(\{P(t) : t \in T\})$), because two tasks cannot have the same order of execution.

This last constraint is a first example of global constraint. Global constraints take into account more than two tasks at a time. NFNL and EF are also such global propagators that allow a much better pruning than the basic constraints. However, NFNL and EF do not use the information given by the position of the tasks to derive their information. This work shows how to use this additional information.

3 The Propagator

The main idea of the new propagator is to apply shaving on the position variables. Commonly, shaving is applied on the starting time variables and only on their bounds because of the size of their domains. On the contrary, the domain of the position variables is rather small and could be shaved in a reasonable time. To test if the task can be scheduled in a particular position, we compute bounds on its earliest and latest starting time under this assumption. If the resulting range does not intersect the domain of $S(t)$, the task cannot be scheduled in that position. Furthermore, shaving $P(t)$ permits also to reduce the domain of $S(t)$ to the union of the ranges computed for every position. Following this scheme, two issues need to be addressed. Firstly, the way to use the bounds on the task starting time to reduce the domains of the variables (Section 3.1). Secondly, the approximations used to compute ranges as tight as possible (Section 3.2). Notice that our approach of shaving is local to this propagator.

Let us first introduce some additional notations. As $est(t)$ represents the earliest starting time of a task t, $ect(t)$ will denote its earliest completion time. Those values are linked by $ect(t) = est(t) + d(t)$. The same quantities can be defined for set of tasks. If U is a non-empty subset of T, $d(U)$ is the sum of the durations of the tasks in U and $est(U)$ is the earliest starting time of the set of tasks U. It is equal to the earliest starting time of any tasks in U ($est(U) = \min_{t \in U} est(t)$). The dual quantity $ect(U)$ is the earliest completion time of the set U, the time when every task in U is finished. This last quantity cannot be computed easily but several lower bounds are known. Especially, the maximum, among every subset U' of U, of the sum of the earliest starting time of U' and the duration of U' will be used in this work to approximate $ect(U)$ (Equation 1). This is only

a bound because it does not take into account the latest starting time of the tasks. We denote it $b\_ect(U)$.

$$b\_ect(U) = \max_{\emptyset \neq U' \subseteq U} (est(U') + d(U')) \tag{1}$$

3.1 Shaving on Position Variables

Shaving enumerates every possible value of $P(t)$. Under the assumption that the position $P(t)$ of a task t takes a particular value p of its domain, the possible starting time of t belongs to an interval $[est(t,p), lst(t,p)]$ where $est(t,p)$ and $lst(t,p)$ denote respectively the earliest and latest possible starting times when t is in position p.

The value $est(t,p)$ is related with $ect(B(t),p)$ that is the earliest time when p tasks among those in $\overline{B}(t)$ have been processed and when all the tasks in $\underline{B}(t)$ have been processed. Indeed, in position p, the task t cannot start before that p tasks among those that can come before t have been processed. Furthermore, t cannot start before the tasks that must come before are completed. This leads to the relation

$$est(t,p) = \max\left(ect(B(t),p), est(t)\right).$$

In this formula, $ect(B(t),p)$ cannot be computed exactly with a reasonable complexity. We propose however to compute a lower bound as tight as possible. Section 3.2 details how to approximate the value of $ect(B(t),p)$. A very similar reasoning, not detailed here, can be made to approximate $lst(t,p)$.

Once the ranges $[est(t,p), lst(t,p)]$ have been computed for each $p \in P(t)$, the domain of $P(t)$ and $S(t)$ can be reduced with two simple rules:

$$\forall p \in dom(P(t)) : ([est(t,p), lst(t,p)] \cap dom(S(t)) = \emptyset) \Rightarrow P(t) \neq p \tag{2}$$

$$dom(S(t)) := dom(S(t)) \cap (\cup_{p \in dom(P(t))} [est(t,p), lst(t,p)]) \tag{3}$$

The first rule removes from the domain of $P(t)$ the values p for which there is no valid starting time, i.e. when the range $[est(t,p), lst(t,p)]$ is empty or when it does not intersect with the domain of $S(t)$. The second rule restricts the domain of $S(t)$ to be included in the union of the computed ranges. Alternatively rule (3) could only reduce the bounds of the domain of $S(t)$, while ensuring that $S(t)$ remains a single interval. The latter is standard in scheduling. $S(t)$ must then be greater than the least value among the $est(t,p)$ for valid p's and less than the greatest value among the $lst(t,p)$ for valid p's.

$$dom(S(t)) := [\min_{p \in dom(P(t))} (est(t,p)), \max_{p \in dom(P(t))} (lst(t,p))] \tag{4}$$

Experiments will consider the two versions of the reduction of $S(t)$. The reduction of $S(t)$ (using rule (4)) and $P(t)$ can be done with a time complexity of $\mathcal{O}(N)$ where N is the number of tasks to be processed on the machine, thus an upper bound on the size of the domain of $P(t)$.

3.2 Bounding the Earliest Completion Time of a Task Subset

This section presents the approximation of $ect(B(t), p)$ that is useful to evaluate $est(t, p)$. The algorithm to compute $lst(t, p)$ is similar but is not exposed. In order to compute a lower bound of $ect(B(t), p)$, we compute the minimum of the earliest completion time over all the sets U of cardinality p that are superset of $\underline{B}(t)$ and subset of $\overline{B}(t)$. In the following, $b\_ect(B(t), p)$ will denote the lower bound of $ect(B(t), p)$. This is a lower bound because it makes use of $b\_ect(U)$ which is a lower bound itself.

$$b\_ect(B(t), p) = \min_{U} (b\_ect(U)) \tag{5}$$

$$\text{where } |U| \geq p \text{ and } \underline{B}(t) \subseteq U \subseteq \overline{B}(t)$$

Interestingly, this lower bound can be computed efficiently using rules similar to the ones in the Jackson Preemptive Schedule [13] for computing the earliest ending time of a set of task supposing preemption. Our algorithm also allows preemption for the tasks but does not take into account the latest starting time of the tasks. Instead, the duration of the tasks is considered to schedule a subset of tasks of fixed size as soon as possible in a preemptive way. It is done respecting the following precedence rules:

- Whenever a task t is available and the machine is free, process t.
- When a task t_1 becomes available during the processing of another task t_2 and the remaining processing time of t_1 is less than the remaining processing time of t_2, stop t_2 and start processing t_1.
- When a task t_1 becomes available during the processing of another task t_2, such that $t_1 \in \underline{B}(t)$ and $t_2 \notin \underline{B}(t)$, stop t_2 and start t_1.

The value $b\_ect(B(t), p)$ is obtained when every tasks in $\underline{B}(t)$ have been processed and at least p tasks in $\overline{B}(t)$ have been processed.

An important property is that, although the algorithm supposes the tasks to be interruptible, the resulting quantities correspond exactly to the ones given by equation (5) where no preemption is supposed. Indeed, it is possible to merge the different parts of the completed tasks following the order of their starting times. The result is a non-preemptive schedule of the set of tasks. Preemption is not used here as a relaxation but just as a way to ease the computation. The computed value of $b\_ect(B(t), p)$ is however a relaxation of the exact value because the latest possible starting times of the tasks are not considered.

Moreover a single run of the above algorithm gives the value $b\_ect(B(t), p)$ for every p. Indeed, it suffices to remember the successive times when a task ends to have the $b\_ect(B(t), p)$ value for the successive values of p.

A pseudo-code of the algorithm is presented in Algorithm 1. The algorithm uses two priority queues. The first (Q1) sorts the tasks in order of earliest starting time. It permits to put in the second priority queue (Q2) only the available tasks at a particular time (lines 9-14). Q2 sorts the tasks in ascending order of remaining duration. When a task is popped from Q2, two situations arise.

Algorithm 1. Simplified Algorithm to Compute $b\_ect(B(t), p)$

Input: B : the set of tasks
D : vector of the duration of the tasks
EST : vector of the est of the tasks
Output: ECT : vector of the $b\_ect(B(t), p)$ for each position p

```
1  Q1 := new PriorityQueue()
2  Q2 := new PriorityQueue()
3  time := 0
4  p := 0
5  forall t ∈ B do
6  |  RD(t) := D(t) //RD is the remaining duration
7  |  Q1.put(t,EST(t))
8  while not Q1.empty() do
9  |  t := Q1.pop()
10 |  time := EST(t)
11 |  Q2.put(t,RD(t))
12 |  while not Q1.empty() and EST(Q1.top()) = time do
13 |  |  t := Q1.pop()
14 |  |  Q2.put(t,RD(t))
15 |  while not Q2.empty() and
16 |  (Q1.empty() or time + RD(Q2.top()) < EST(Q1.top()) ) do
17 |  |  t := Q2.pop()
18 |  |  time := time +RD(t)
19 |  |  RD(t) := 0
20 |  |  p := p+1
21 |  |  ECT(p) := time
22 |  if not Q2.empty() then
23 |  |  t := Q2.pop()
24 |  |  RD(t) := RD(t) + time - EST(Q1.top())
25 |  |  Q2.push(t,RD(t))
26 |  |  time := EST(Q1.top())
27 |
28 return ECT
```

Either it can be processed before a new task is available and the time when it ends is recorded (lines 15-21). Or the task must be interrupted to check if a newly available task could not end earlier (lines 23-26).

For simplicity, the outlined algorithm is a shortened version where the fact that some tasks are part of $\underline{B}(t)$ is not considered. Taking it into account can be done simply using a penalty in the second priority queue to ensure that those tasks are chosen first. Two parallel queues can also be used and the one containing the tasks in $\underline{B}(t)$ is emptied first. Additionally a counter must be used to record when all mandatory tasks have been processed.

The time complexity of the algorithm is $\mathcal{O}(n \log n)$ with $n = |\overline{B}(t)|$ which in the worst case is equal to $N - 1$ (N is the number of tasks that must be processed). Indeed, the operation put() and pop() of the priority queues can be implemented in $\mathcal{O}(\log n)$. There are exactly n tasks that are put in Q1 (lines

5-7) and at most $2n$ tasks that are put in Q2 because there are exactly n tasks that can be extracted from Q1 (lines 9-14) and at most n reinsertions of task due to interruption (lines 22-26).

Example 1. To show the computation of $b\_ect(B(t), p)$, let us suppose the following tasks:

- t_0 which is the task under consideration; $dom(B(t_0)) = [\{t_4\}, \{t_1, t_2, t_3, t_4\}]$ and $dom(P(t_0)) = [1, 4]$
- t_1 with $est(t_1) = 0$ and $d(t_1) = 5$.
- t_2 with $est(t_2) = 1$ and $d(t_2) = 3$.
- t_3 with $est(t_3) = 2$ and $d(t_3) = 1$.
- t_4 with $est(t_4) = 3$ and $d(t_4) = 3$.

Following a chronological order, t_1 is scheduled first, starting at the time 0. On time 1, t_2 is available and as its duration ($d(t_2) = 2$) is shorter than the remaining duration of t_1 ($5 - 1 = 4$), t_1 is stopped and t_2 is started. On time 2, t_2 is interrupted to let process t_3 whose duration is shorter than its remaining duration ($3 - 1 = 2 > 1$). On time 3, t_3 has been fully processed. Tasks t_1, t_2 and t_4 are available but t_4 is chosen as it is the only mandatory task among them. Indeed, by definition of $dom(B(t))$, t_4 is the only task which must be performed before t_0. This task is run for 3 units of time. When it is finished, t_2 is run before t_1 as its remaining duration is less than the remaining duration of t_1. After two more units of time, t_2 is fully processed and t_1 is processed until time 12. The next table gives the processing times of each task in a preemptive way.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Task	t_1	t_2	t_3	t_4			t_2		t_1				

Recording the values when tasks are fully processed, we obtain the following values:

- $b\_ect(B(t_0), 1) = b\_ect(B(t_0), 2) = 6$. Indeed, the mandatory task (t_4) was only finished in second position.
- $b\_ect(B(t_0), 3) = 8$
- $b\_ect(B(t_0), 4) = 12$

Although the computation interrupts several tasks, the obtained bounds correspond to non-preemptive schedules (as expected by equation (5)). The table below shows the reordering for each position.

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
$p = 1$				t_4									
$p = 2$			t_3	t_4									
$p = 3$		t_2			t_3	t_4							
$p = 4$	t_1						t_2		t_3	t_4			

For instance, with the 4 tasks being scheduled, it is possible to run t_1 from time 0 until time 5 where t_2 is run until time 8. At time 8, t_3 is started for 1

time unit and afterward t_4 is being run until the time 12 which corresponds to the computed value. □

Example 2. Figure 1 presents a small example where the new propagator permits to remove inconsistent values. In this example, there are five tasks to be processed. Their respective domains and duration are the following.

- $d(t_1) = 3$ and $dom(S(t_1)) = [8, 17]$
- $d(t_2) = 5$ and $dom(S(t_2)) = [0, 15]$
- $d(t_3) = 4$ and $dom(S(t_3)) = [5, 16]$
- $d(t_4) = 4$ and $dom(S(t_4)) = [1, 16]$
- $d(t_5) = 2$ and $dom(S(t_5)) = [7, 18]$

Applying NFNL or EF on this set of tasks does not reduce any domain of the starting time variables. However, our propagator allows to remove the value 8 from the domain of $S(t_1)$. Using the algorithm to compute the earliest and latest possible starting time of t_1 in each position, the obtained values are given next.

- $est(t_1, 0) = 8$ and $lst(t_1, 0) = 2$
- $est(t_1, 1) = 8$ and $lst(t_1, 1) = 7$
- $est(t_1, 2) = 9$ and $lst(t_1, 2) = 11$
- $est(t_1, 3) = 11$ and $lst(t_1, 3) = 15$
- $est(t_1, 4) = 15$ and $lst(t_1, 4) = 17$

From those values, it can be derived that t_1 cannot be processed in position 0 or 1. Thus the domain of its starting time can be reduced to the union of the ranges defined in position 2, 3 and 4, resulting in $dom(S(t_1)) = [9, 17]$. In comparison with the initial domain, the value 8 has been removed. □

The computing of $est(t, p)$ and $lst(t, p)$ for each $p \in dom(P(t))$ is done in $\mathcal{O}(N \log N)$ with N the number of tasks and the reduction of the domains can be done in $\mathcal{O}(N)$. The time complexity of the whole reduction algorithm for a

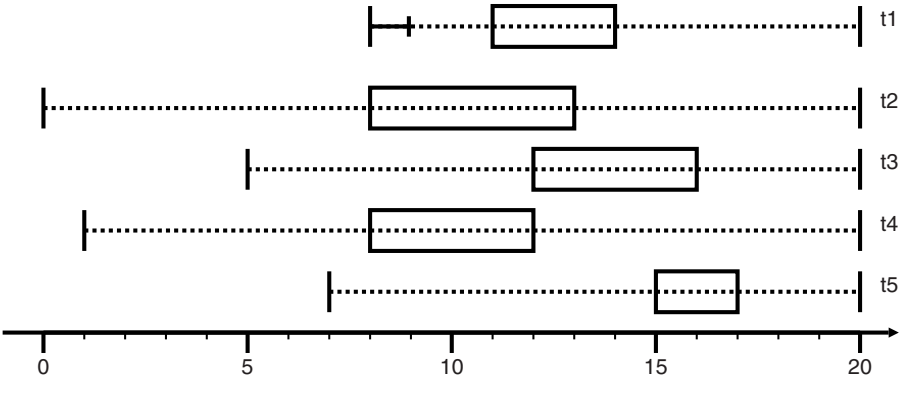


Fig. 1. Example of reduction, see Example 2 for details

task t is thus $\mathcal{O}(N \log N)$. This yields a total complexity of $\mathcal{O}(N^2 \log N)$ for one pass of our reduction algorithm, as there are N tasks to consider. In comparison, the well-known techniques NFNL and EF can be both implemented to run with a time complexity of $\mathcal{O}(N \log N)$.

4 Experiments

To assess the practical usefulness of the new propagator, we implemented it in the open constraint environment Gecode [14]. Two versions of the propagator have been written. The first that we will refer to as PS (standing for Position Shaving) may remove values inside the domains of the starting time variables, while the second, PSB (for Position Shaving with Bounds reduction), is limited to reduce the bounds of the starting time variables. We implemented also the NFNL and EF techniques following the algorithms described in [15]. Note that the implementations of EF and NFNL described in that book run in $\mathcal{O}(N^2)$ but they use much simpler data structures than the theoretically most efficient algorithm described respectively in [3] and [7]. Finally, we modeled the Open-Shop Problem as described in the first section with the NFNL, EF and PS or PSB propagators and the AllDifferent constraint. PS and PSB are never used together as they are two versions of the same propagator. Concerning the branching, we applied a simple heuristic that uses the position variables. It orders the tasks in the machine before ordering them in the jobs. Among the tasks whose position is not fixed, it chooses the task for which there is the smallest number of remaining possible positions. In case of tie, the shortest task is chosen. The value-heuristic chooses the smallest value in the position variable.

Our tests have been run using the instances of the Guéret and Prins benchmark [12]. It is composed of 80 square problems, i.e. the number of jobs and machines are equal. There are 10 instances for each size ranging from 3x3 tasks to 10x10 tasks. Every runs have been performed on an Intel Xeon 3 Ghz with 512 KB of cache.

The first experiment consists in observing the total runtime and the size of the search tree to solve each instance of the benchmark, using different combinations of propagators. The running time is limited to one hour for each instance. The results are presented in Tables 1 and 2. Table 1 gives the number of solved instances and the average number of nodes in the search tree. The mean is computed over the instances commonly solved whenever the number of solved instances differs (only for problem size 7x7). In table 2, the same scheme is used but the mean running time is presented instead of the size of the search tree. The running time is given in seconds.

In the two tables, columns 2 and 3 present the results when only PS is used but nor EF neither NFNL. Columns 4 and 5 presents the results when only PSB is used. In the third setting (columns 6-7), NFNL and EF are activated but not PS, nor PSB. In the columns 8-9 and 10-11, NFNL and EF are used in conjunction respectively with PS and with PSB.

Whenever NFNL and EF are used, the same total number of instances are solved with or without our new propagator. However, the solved instances are not

Table 1. Number of solved instances and mean size of the search tree

	PS		PSB		NFNL+EF		PS+NFNL+EF		PSB+NFNL+EF	
Size	Solved	Nodes	Solved	Nodes	Solved	Nodes	Solved	Nodes	Solved	Nodes
3x3	10	39	10	38	10	38	10	39	10	38
4x4	10	128	10	127	10	134	10	127	10	126
5x5	10	451	10	456	10	371	10	369	10	373
6x6	10	3483	10	3896	10	2612	10	3402	10	3816
7x7	3	-	3	-	7	280914	8	208571	8	208582
8x8	0	-	0	-	1	120156	1	12953	1	12929
9x9	0	-	0	-	1	747146	0	-	0	-
10x10	0	-	0	-	0	-	0	-	0	-
Tot	43		43		49		49		49	

Table 2. Number of solved instances and mean running time in seconds

	PS		PSB		NFNL+EF		PS+NFNL+EF		PSB+NFNL+EF	
Size	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time
3x3	10	0.008	10	0.008	10	0.006	10	0.01	10	0.008
4x4	10	0.075	10	0.047	10	0.054	10	0.069	10	0.068
5x5	10	0.38	10	0.32	10	0.19	10	0.36	10	0.32
6x6	10	3.9	10	3.5	10	1.9	10	4.3	10	3.9
7x7	3	-	3	-	7	338	8	496	8	432
8x8	0	-	0	-	1	106	1	43	1	37
9x9	0	-	0	-	1	1708	0	-	0	-
10x10	0	-	0	-	0	-	0	-	0	-
Tot	43		43		49		49		49	

always the same. From the two first settings, it can be concluded that the new propagator is not able to solve hard problems alone. In conjunction with NFNL and EF, Table 1 shows that PS and PSB are able to reduce the size of search tree, sometimes substantially, as it is the case for the unique solved instance of size 8x8. Concerning the size 6x6, surprisingly the mean size is greater when using PS and PSB. Looking at the detail for each instance of this size, it appears that only the first instance(GP06-01) has a greater search tree when using the new propagators. For GP06-01, the search tree is ten times bigger when using PS or PSB while it is on average 30% smaller for the nine other instances of size 6x6.

When the running times are considered, Table 2 shows that it is always greater when using PS or PSB than without them, except for the solved instance of size 8 where the time is 2 to 3 times smaller, while the search tree size was almost 9 times smaller.

Note that the reported times are much longer than those presented in [15] because we did not use an environment dedicated to scheduling but a general purpose constraint engine. However, implementing our new propagator in a dedicated environment would be beneficial.

The next experiment (Table 3) compares the mean runtime to reach the fixpoint when NFNL, EF and PS(B) are activated with the mean runtime when PS(B) is not used. This comparison is performed on the search tree obtained when NFNL, EF and PS(B) are activated with a maximum number of backtracks of 300,000. For each instance, the runtimes to reach the fixpoints are summed along every states in the search tree.

At the same time, the pruning is also compared. As for the runtime, this pruning is computed along the search tree obtained when every propagators are activated. The number of failed states with and without PS(B) activated are counted. Additionally in each state the supplementary reduction performed after adding PS(B) is counted for each type of variables ($S(t)$, $B(t)$ and $P(t)$) and these quantities are summed upon the whole search tree. The reduction is computed as the difference between the size of the domains of the variables in the initial state in a node of the search tree and their size after performing propagation until the fixpoint in the same node. If a failure is detected, the node is not taken into account for the reduction counts.

Table 3. Additional Pruning and time spent with PS and PSB(in %)

	Red. S(t)		Red. B(t)		Red. P(t)		Fails		Time	
Size	PS	PSB	PS	PSB	PS	PSB	PS	PSB	PS	PSB
3	7.6	1.5	0.3	0.3	5.7	3.6	0	0	-	-
4	13.6	5.6	7.0	7.4	14.2	12.7	2.1	2.1	184.0	165.7
5	14.1	5.6	4.8	4.9	11.2	9.8	0.7	0.8	192.1	182.2
6	27.3	14.9	9.0	9.2	15.6	14.1	8.0	8.2	241.3	181.0
7	108.7	58.3	21.3	21.8	42.5	42.7	13.9	14.2	333.2	325.3
8	116.9	34.9	17.4	16.7	30.1	26.1	13.3	13.9	281.1	254.0
9	78.9	25.4	13.9	13.2	20.5	18.0	37.7	36.3	291.5	272.0
10	64.6	17.9	9.9	10.3	20.9	19.5	37.4	37.3	155.5	196.5
Mean	54.0	20.5	10.4	10.5	20.1	18.3	14.1	14.1	239.8	225.3

Table 3 presents the results averaged by size. The three first pairs of columns presents the additional pruning of the variables $S(t)$, $B(t)$ and $P(t)$. The next two columns shows the additional failures detected and the last columns reports the additional time spent to reach those improvements. Two cells are empty because the running time was to short to compute them accurately.

It can be seen that the results are quite similar between PS and PSB, except in the columns of the starting time variables, since PS may prune inside the domain of $S(t)$ while PSB cannot. However, this difference does not influence the other variables nor the failures. Indeed, no other constraint considers forbidden values inside domains. Concerning the increase of the running time to reach a fixpoint, it is smaller with PSB because less values are removed by PSB. Taking into account the pruning potential and the used time, we can conclude that PSB is more efficient than PS. Furthermore, there are about 14% more failures detected with either version of our propagator. When no failure is detected, the domains of the variables are also substantially reduced.

Looking at the evolution of the results in function of the size of the problem, the amount of reduction of the domains increases until problems of size 7 and then decreases. The time spent follows the same scheme while the number of failures keeps increasing. Because from size 7 the search trees may be not full (because the search is cut) and the explored part is smaller for increasing size, we can suppose that our propagator detect more failures early in the search but reduces more domains at the end or in the middle of the search than in the first steps. Observing the failures for the smallest sizes, it can also be seen that PS and PSB do not reduce further the small search trees of these instances. When size grows (≥ 6) and complexity increases, PS and PSB prove their usefulness.

In conclusion, the experiments show that although the introduction of PS or PSB does not increase the number of solved instances, the addition of such a propagator substantially improves the pruning at the nodes of the search tree, as well as the number of detection of inconsistencies.

5 Conclusion

This work addresses the Open-Shop Problem by Constraint Programming. It presents a new propagator, in two versions, that uses the absolute position of the tasks to detect new inconsistencies not discovered by standard algorithms, known as Not-First-Not-Last or Edge-Finding. Based on the principle of shaving, this propagator prunes the variables for the starting time of tasks and for the position of tasks. In its first version, holes in the starting time variable are allowed while this is not the case in the second version that only reduces the bounds of the domains.

Experiments on a standard benchmark show that the new propagator helps efficiently in the reduction of the domains and may detect about 14% more inconsistent states in the search tree but at a higher cost. Concerning the size of the search tree, the reduction of the domains is not always reflected by a reduction of the tree size. The search can be up to 10 time smaller but for the majority of the problems the reduction is not as important. In a few cases, the search tree is even increased with the new constraint.

Another observation comes from the comparison between the two versions of the propagators. Making holes in the domain of the starting time variables is not rewarding regarding the reduction of the other variables and of the search tree. The cause is that no other constraint makes use of this additional information.

In conclusion, we argue that our propagator would be especially useful in conjunction with other constraints that take into account the position of tasks or the holes in the domains. This may be a good way to improve resolution of hard disjunctive scheduling problems. The branching heuristic should also be adapted in order to avoid an augmentation of the size of the search tree when the filtering is strengthened.

Possible future work includes the definition of tighter bounds for the earliest completion time of a subset of tasks of fixed size. It also covers the definition of better branching heuristics and additional position-based constraints.

Acknowledgment

The authors wish to thank the anonymous reviewers for their constructive comments. This research is supported by the Walloon Region, project TransMaze (516207).

References

1. J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. Management Science, 35(2):164-176, 1989.
2. D. Applegate and B. Cook. A computational study of the job shop scheduling problem. ORSA J. Comput., 3(2):149-156, 1991.
3. J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. European J. Oper. Res.,78:146-161, 1994
4. Y. Caseau and F. Laburthe. Improved CLP scheduling with task intervals. Proc. 11th Intl. Conf. on Logic Programming, 1994.
5. J. Carlier and E. Pinson. A practical use of Jackson's preemptive schedule for solving the job-shop problem. Ann. Oper. Res.,26:269-287, 1990.
6. U. Dorndorf, E. Pesch and T. Phan-Huy. Solving the open shop scheduling problem. J. Scheduling, 4:157-174, 2001.
7. P. Vilim. $\mathcal{O}(n \log n)$ filtering algorithms for unary resource constraint. Proc. CPAIOR 2004, LNCS 3011, 335-347, 2004.
8. P. Martin and D. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. Proc. 5th Conf. Integer Programming and Combinatorial Optimization, 1996.
9. Jianyang Zhou. A permutation-based approach for solving the job-shop problem. Constraints, 2:185-213, 1997.
10. W. Nuijten and C. Le Pape. Constraint-based job shop scheduling with ILOG Scheduler. J. Heuristics, 3:271-286, 1998.
11. A. Wolf. Better propagation for non-preemptive single-ressource constraint problems. Proc. of CSCLP 2004, LNAI 3419, 201-215, 2005.
12. C. Guéret and C. Prins. A new lower bound for the open-shop problem. Annals of Operation Research, 92:165-183, 1999.
13. J. R. Jackson. An extension of Johnson's results on job lot scheduling. Naval Research Logistics Quarterly, 3:201-203, 1956.
14. http://www.gecode.org
15. P. Baptiste, C. Le Pape and W. Nuijten. Constraint-based scheduling. Kluwer Academics Publisher, 2001.

Directional Interchangeability for Enhancing CSP Solving

Wady Naanaa

Faculté des Sciences de Monastir, 5019 Monastir, Tunisia;
naanaa.wady@planet.tn

Abstract. This paper introduces directional interchangeability, a weak form of neighborhood interchangeability [6]. The basic idea is that although two values of a variable may not be neighborhood interchangeable if we consider the whole neighborhood of the variable, they could be neighborhood interchangeable if we restrict the neighborhood to a subset of neighboring variables induced by a variable ordering.

In spite of the fact that the proposed concept cannot be used to remove redundant values while preserving problem satisfiability, it provides a mean to partition value domains into subsets of directionally interchangeable values that can be attempted simultaneously by a tree search.

Several experiments carried out on various binary CSPs, clearly indicate that variations of the Forward-Checking algorithm and the Maintaining Arc-Consistency algorithm that exploit directional interchangeability often outperform the original algorithms.

1 Introduction

Constraint Satisfaction Problems (CSPs) provide a general framework for modeling and solving numerous combinatorial problems. Basically, a CSP consists of a set of variables, each of which can take a value chosen among a set of potential values called its *domain*. The constraints express restrictions on combinations of domain values. The problem is to find an assignment of values to variables, from their respective domains, such that all the constraints are satisfied.

CSPs, which are known to be NP-complete problems, can model problems in various domains. For that reason, many solving algorithms have been developed for them. In this article we are interested in complete methods which have the advantage of finding at least one solution to a problem if such a solution exists. *Forward Checking* (FC) [7] and *Maintaining Arc Consistency* (MAC) [12] are two widely studied complete algorithms. Each of them enforces during search a form of local consistency to prune the search tree and therefore to fasten problem solving.

However, for large CSPs, finding a solution remains a time consuming task. Moreover, the emergence of new concrete problems, reinforces the need for increasingly powerful algorithms.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 200–213, 2007.

For the purpose of enhancing complete algorithms, E. C. Freuder proposed neighborhood interchangeability of domain values, a particular form of symmetry, and used it to reduce the domains of variables by removing redundant values [6]. Because in a subset of neighborhood interchangeable values, we can keep only one value of the subset in the domain of the variable while not affecting the satisfiability of the original problem. This may lead to the exploration of a significantly smaller search tree provided that neighborhood interchangeability occurs frequently in a problem. However, for many problems little interchangeability occurs and the overhead due to determining redundant values could offset the benefit. In an attempt to exploit interchangeability more intensively, many local versions of interchangeability have been proposed [2,9,11,13,16].

This paper introduces directional interchangeability, a variation of neighborhood interchangeability which is inspired by the concept of directional arc-consistency [5]. The proposed concept differs from the original one in that it assumes a variable ordering and uses it to focus only on a subset of the neighboring variables. That is, when computing directional interchangeability for the domain of a given variable, the neighborhood of this variable is limited to variables coming earlier in the ordering. This restriction yields directional interchangeability to occur more often than neighborhood interchangeability, but in compensation it does not allow the removal of redundant values. The reason is that directionally interchangeable values are not equivalent with regard to the domains of all the neighboring variables and then one cannot keep a single value from each class of directionally interchangeable values while preserving problem satisfiability. Nonetheless, directional interchangeability can be used to partition the value domains into subsets of directionally interchangeable values in such a way that values within the same subset could be attempted simultaneously by a tree search. In other words, directional interchangeability can be exploited by a tree search to assign to each variable a specific subset of its domain at each branch of search instead of attempting a single value per branch as it is the case in a classical search.

The paper is organized as follows: the next section introduces some definitions and preliminaries. Section 3 is devoted to formally defining directional interchangeability and describing and proving the correctness of the algorithm used to compute directionally interchangeable values. In Sect. 4, we describe a search algorithm which exploits directional interchangeability. An experimental study carried on various binary CSPs is reported in Sect. 5. Section 6 discusses related work and Sect. 7 presents some conclusions and future works.

2 Definitions and Notations

Definition 1. *A constraint satisfaction problem (CSP) is given by a triple* (X, D, C) *where:*

1. $X = \{x_1, \ldots, x_n\}$ *is a finite set of variables.*
2. $D = \{D_1, \ldots, D_n\}$ *is a sequence of value domains so that* D_k *is the domain of* x_k*.*

3. *$C = \{C_1, \ldots, C_m\}$ is a set of constraints. Each constraint C_k applies on a list of variables $Var(C_k) = (x_{k_1}, \ldots, x_{k_r})$ called the scope of C_k and is defined by a r-ary relation $Rel(C_k) \subseteq D_{k_1} \times D_{k_2} \ldots \times D_{k_r}$. $Rel(C_k)$ determines the r-tuples of values accepted by C_k.*

The arity of a constraint is the size of its scope. The arity of a problem is the maximum arity over its constraints. In this paper, we focus on binary CSPs, i.e. CSPs with binary and unary constraints only. It must be emphasized however that the proposed concept can be easily extended to non-binary CSPs. Two variables x_i and x_j connected by a binary constraint denoted by $C_{i,j}$ are said to be neighbors. A value $a \in D_i$ is compatible with $b \in D_j$ if $(a, b) \in Rel(C_{i,j})$. In this case, a is called a support of b. Given a binary CSP, its constraint graph associates each variable with a node and links any oriented pair of nodes (x_i, x_j) such that $C_{i,j} \in C$. The notion of arc-consistency, a widely studied consistency level, is defined from the constraint graph as follows: an arc (x_i, x_j) in the constraint graph is arc-consistent iff every value in D_i has a support in D_j.

The support set of a value a of a variable x_i is defined by:

$$N(x_i, a) = \{b \mid (a, b) \in Rel(C_{i,j})\} \ .$$

We define the preceding support set of a value a of a variable x_i with respect to an ordering $\prec$ of the variables as follows:

$$\vec{N}(x_i, a) = \{b \mid (a, b) \in Rel(C_{i,j}) \text{ and } x_j \prec x_i\} \ .$$

3 Directional Interchangeability (DI)

First, let us remind the definition of the concept from which DI is derived, namely neighborhood interchangeability (NI) [6]. Given a binary CSP, two values a and b of a variable x_i are neighborhood interchangeable iff they have the same support set, i.e., $N(x_i, a) = N(x_i, b)$. NI identifies equivalent values in the domain of a variable x_i by considering all the neighbors of x_i. In contrast, DI supposes a variable ordering and focuses exclusively on the neighboring variables preceding x_i in the ordering.

Formally, DI is defined as follows:

Definition 2. *Let P be a binary CSP and let a and b be two values in the domain of a variable x_i. a and b are said to be directionally interchangeable (DI) with respect to a variable ordering of the variables iff they have the same preceding support set, i.e., $\vec{N}(x_i, a) = \vec{N}(x_i, b)$.*

Note that the DI condition is weaker than the condition imposed by NI. As a consequence, DI may occur more frequently than NI. As it is the case for NI, DI defines an equivalence relation on the domains of each variable. The domain D_i of a variable x_i can be split into a set of sub-domains $\Delta_i = \{\Delta_{i,1}, \ldots, \Delta_{i,s}\}$ such that the elements of each $\Delta_{i,k}, k : 1, \ldots, s$ are pairwise DI.

In [6], the author proposed an algorithm for partitioning the domain of a variable into subsets of NI values. Practically, the algorithm builds a discrimination tree in which the nodes represent variable-value pairs from the neighborhood of the considered variable except for the root which is a fictive node. Each path of the tree from the root to a leaf is annotated by a computed subset of NI values. All the variable-value pairs that appear in a path are supports of the values appearing in the annotation of that path. The complexity of the algorithm is $O(nd^2)$, where n is the number of variable and d the size of the largest domain.

In practice, the construction of an n-ary tree, a dynamic data structure, at each choice point significantly slows down the solving process. For this reason we propose a new algorithm (see Function 3) which avoids this drawback while keeping the same time and space complexity.

The algorithm iterates over the values of the neighbors of x_i for the purpose of partitioning the values of D_i. Note that only the values of the variables preceding x_i are considered (see line 3). As it will be shown below, the discrimination between non DI values can be guaranteed by computing, at each iteration, a class number for each $a \in D_i$ (line 1 thru 16). At the two last loops of the algorithm, the partition Δ_i is deduced from the class numbers obtained at the last iteration of the loop starting at line 3.

Function 3. DI–Partition(x_i, X, D, C)

1. **for** each $a \in D_i$ **do** $c(a) \longleftarrow 0$
2. $c_{max} \longleftarrow 0$
3. **for** each $C_{i,j} \in C$ such that $j \prec i$ **do**
4. **for** each $b \in D_j$ **do**
5. **for** each $a \in D_i$ **do**
6. $score(a) \longleftarrow 2 * c(a) + comp(a, b)$
7. **for** s from 0 to $2 * c_{max} + 1$**do**
8. $freq(s) \longleftarrow 0$
9. **for** each $a \in D_i$ **do**
10. $freq(score(a)) \longleftarrow freq(score(a)) + 1$
11. $rank(0) \longleftarrow 0$
12. **for** s from 1 to $2 * c_{max} + 1$**do**
13. $rank(s) \longleftarrow rank(s - 1) + freq(s - 1)$
14. **for** each $a \in D_i$ **do**
15. $c(a) \longleftarrow rank(score(a))$
16. $c_{max} \longleftarrow \max(c)$
17. **for** s from 0 to c_{max} **do**
18. $\Delta_{i,s} \longleftarrow \emptyset$
19. **for** each $a \in D_i$ **do**
20 $\Delta_{i,c(a)} \longleftarrow \Delta_{i,c(a)} \cup \{a\}$
21. return(Δ_i)

To establish the correctness proof of algorithm DI-Partition, we introduce the following notations. The values of the variables preceding x_i according to the ordering $\prec$ are denoted by $b_1, b_2, \ldots, b_m$, where $m = \sum_{C_{i,j} \in C, x_j \prec x_i} card(D_j)$.

The algorithm is based on the sequence $(c_k)_{0\leq k\leq m}$ which is defined for all $a \in D_i$ as follows

$$c_0(a) = 0 \tag{1}$$
$$c_k(a) = card(\{a' \in D_i \mid 2c_{k-1}(a') + \kappa(a', b_k) < 2c_{k-1}(a) + \kappa(a, b_k)\}) \tag{2}$$

where the $\kappa(a, b)$ function is defined as:

$$\kappa(a,b) = \begin{cases} 1 & \text{if } a \text{ and } b \text{ are compatible} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

As specified by (1), at the beginning of the process (line 1), all the values in D_i are assumed to be directionally interchangeable, that is $c_0(a) = 0$ for all $a \in D_i$. At iteration k, the algorithm computes c_k according to (2). The value obtained by setting $c_k(a)$ to $rank(score(a))$ (see line 15) is precisely the value specified by (2).

Lemma 4. *if* $c_k(a) < c_k(a')$ *then* $c_{k'}(a) < c_{k'}(a')$ *for all* $k' > k$.

Proof. Suppose we have $c_k(a) < c_k(a')$ for some k and there exists $k' > k$ such that $c_{k'}(a) \geq c_{k'}(a')$. Assume k' is the smallest integer which verifies the latter assumption. This means that $c_{k'-1}(a) < c_{k'-1}(a')$ and then that $2c_{k'-1}(a)+1 < 2c_{k'-1}(a')$. By (3), we get

$$2c_{k'-1}(a) + \kappa(a, b_{k'}) < 2c_{k'-1}(a') + \kappa(a', b_{k'}) \ . \tag{4}$$

It follows that

$$\begin{aligned} c_{k'}(a) &= card(\{a'' \in D_i \mid 2c_{k'-1}(a'') + \kappa(a'', b_{k'}) < 2c_{k'-1}(a) + \kappa(a, b_{k'})\}) \\ &< card(\{a'' \in D_i \mid 2c_{k'-1}(a'') + \kappa(a'', b_{k'}) < 2c_{k'-1}(a) + \kappa(a, b_{k'})\} \cup \{a\}) \\ &\leq card(\{a'' \in D_i \mid 2c_{k'-1}(a'') + \kappa(a'', b_{k'}) < 2c_{k'-1}(a') + \kappa(a', b_{k'})\}) \\ &= c_{k'}(a') \end{aligned}$$

Hence, $c_{k'}(a) < c_{k'}(a')$ which contradicts the hypothesis. □

Theorem 5. $c_m(a) = c_m(a')$ *iff* $\vec{N}(x_i, a) = \vec{N}(x_i, a')$.

Proof. Assume that $c_m(a) = c_m(a')$ but $\vec{N}(x_i, a) \neq \vec{N}(x_i, a')$ and proceed to deduce a contradiction. $\vec{N}(x_i, a) \neq \vec{N}(x_i, a')$ implies that $\exists k, 1 \leq k \leq m$ such that $\kappa(a, b_k) = 0$ and $\kappa(a', b_k) = 1$ or vice versa. We distinguish two cases

1. $c_{k-1}(a) = c_{k-1}(a')$
   Suppose, without loss of generality that $\kappa(a, b_k) = 0$ and $\kappa(a', b_k) = 1$. It follows that $2c_{k-1}(a) + \kappa(a, b_k) < 2c_{k-1}(a') + \kappa(a', b_k)$. By proceeding as in the proof of lemma 4 (at the level of (4)), we get $c_k(a) < c_k(a')$ and by lemma 4, we deduce that $c_m(a) < c_m(a')$ which contradicts the hypothesis.
2. $c_{k-1}(a) \neq c_{k-1}(a')$
   By lemma 4, we get either $c_m(a) < c_m(a')$ or $c_m(a) > c_m(a')$ which contradicts the hypothesis.

The remaining part of the proof is obvious, since $\vec{N}(x_i, a) = \vec{N}(x_i, a')$ implies that $\kappa(a, b_k) = \kappa(a', b_k)$ for all k in the range $1 \leq k \leq m$. Thus, starting from $c_0(a) = c_0(a') = 0$, we get $c_m(a) = c_m(a')$. □

As can be seen from the pseudo-code of Function 3, there are at most three nested loops. The number of times these loops are executed is bounded by $O(nd^2)$. Indeed, each variable can be constrained by at most $n-1$ constraints, the size of the largest value domain is d and c_{max} is less or equal to d since, by definition, the c_k's map into $\{0, \ldots, d-1\}$. As supplementary storage units, we used arrays *score*, *freq* and *rank* which have all a size of $O(d)$.

4 Exploiting Directional Interchangeability

In this section, we present FC-DI a variation of the Forward-Checking algorithm which exploits directional interchangeability. We do not describe MAC-DI which is also used in the experiments because it only differs from FC-DI by its look-ahead schema which consists in maintaining arc-consistency during search.

4.1 The Search

FC-DI is typically a 2-way-branching depth first search except that it branches over subsets of DI values. At each choice point, FC-DI selects a non instantiated variable x_i, and computes the partition $\Delta_i = \{\Delta_{i,1}, \ldots, \Delta_{i,s}\}$ of D_i based on DI. During search, the order in which the variables are considered suffices to compute Δ_i because the variables that precede x_i are already known when x_i is processed. Hence, the method do not fix a predefined variable ordering what will allow the use of dynamic variable ordering. Note that the computation of the $\Delta_i, i : 1, \ldots, n$ is carried out dynamically at each choice point in order to increase the opportunities of DI.

The algorithm selects a subset $\Delta_{i,k}$ in the domain partition Δ_i and reduces D_i to $\Delta_{i,k}$ (see line 7). Then function Propagate is called. At this point, x_i is considered as an instantiated variable and is discarded from the current sub-problem even if its domain is not reduced to a singleton. If no empty domain is encountered, then the algorithm performs a recursive call to process the future variables. Else, it considers the other subsets in Δ_i within the second recursive call. If the two recursive calls return no solution, the algorithm backtracks to the immediately preceding variable. We show in the next paragraph that if FC-DI succeeds to consider all the variables then a solution bundle [10] is found.

Function 6. FC–DI(X, D, C)

1. **if** $X = \emptyset$ **then**
2. return(D)
3. **else**
4. $x_i \longleftarrow$ Select(X)
5. $\Delta_i \longleftarrow$ DI–Partition(x_i, X, D, C)

6 $\Delta_{i,k} \longleftarrow Select(\Delta_i)$
7. $D_i \longleftarrow \Delta_{i,k}$
8. Propagate(x_i, X, D, C)
9. **if** no empty domain **then**
10. $I \longleftarrow$ FC–DI$(X - \{x_i\}, D, C)$
11. **if** $I \neq \emptyset$ **then**
12. return(I)
13. Restore(D)
14. return(FC–DI$(X, D - \Delta_{i,k}, C)$)

4.2 Constraint Propagation

The potential of DI as a means of improving search efficiency may be even greater if some level of local consistency is maintained during search. For simplicity, we chose to show how to exploit DI within the Forward-Checking (FC) [7]. To specify the look-ahead scheme performed by FC-DI, we need to distinguish past and future variable. A past variable is a variable which has already been assigned a specific subset of its domain. A future variable is a variable that has not been instantiated yet. The invariant to be maintained by FC-DI during search is the same as in FC, that is every arc of the constraint graph whose target node represents a past variable must be arc-consistent. This is ensured by procedure Propagate which is depicted below. For the sake of simplicity, we assume that the variables are selected following the natural order of their indices.

Let x_i be the current variable. Procedure propagate proceeds in two stages. In the first stage, all the arcs (x_k, x_i) are made arc-consistent. Next, Propagate processes all the arcs (x_k, x_j) such that x_k is a future variable and x_j is a past variable whose domain has been modified in the first stage. It uses *ReviseDomain* which is a standard procedure for restoring the arc-consistency of a given arc.

Procedure 7. Propagate(x_i, X, D, C)

1. $Q \longleftarrow \emptyset$
2. **for** each (x_j, x_i) **do**
3. **if** ReviseDomain$((x_j, x_i), X, D, C)$ **then**
4. $Q \longleftarrow Q \cup \{(x_k, x_j) \mid j < i < k\}$
5. **for** each $(x_k, x_j) \in Q$ **do**
6. ReviseDomain$((x_k, x_j), X, D, C)$)

The following two propositions ensure that FC-DI maintains the invariant specified above.

Lemma 8. *Let x_i and x_{i+k} be two neighbor variables of a binary CSP such that all the values in D_{i+k} are DI with respect to the natural order. If (x_i, x_{i+k}) is arc-consistent then all the tuples of $D_i \times D_{i+k}$ are compatible.*

Proof. Suppose there exists $(a, b) \in D_i \times D_{i+k}$ such that a and b are not compatible. Then, since all the elements of D_{i+k} are DI with respect to the natural

order and that $i < i+k$, all the elements of D_{i+k} would be incompatible with a. This implies that $a \in D_i$ has no support in D_{i+k} and therefore that (x_i, x_{i+k}) is not arc-consistent which contradicts the hypothesis. □

Lemma 9. *All arcs whose target node represents a past variable are maintained arc-consistent by FC–DI.*

Proof. To begin with, denote by D_j (resp. D'_j) the value domain of x_j before (resp. after) instantiating the current variable x_i. We proceed by induction on the past variable set. After instantiating x_1, all arcs (x_k, x_1) are processed by procedure Propagate in order to restore the arc-consistency of each of them. At that stage, the invariant holds again since x_1 is the only past variable.

Now assume that $x_j,\ j : 1, \ldots, i-1$ are all past variables and then that all the arcs (x_k, x_j) are arc-consistent before instantiating x_i and proceed to prove that after instantiating x_i and executing procedure Propagate, all the arcs $(x_k, x_j),\ j : 1, \ldots, i$, will be arc-consistent.

By instantiating x_i, this latter becomes a past variable. Then, for the invariant to hold, all the arcs ending at x_i must be arc-consistent. This is ensured by the first loop of Propagate. Among these arcs, one can have an (x_j, x_i) such that x_j is a past variable, (i.e. $j < i$). Enforcing the arc-consistency of such an arc may cause the reduction of D_j. As a consequence, if there exists an arc (x_k, x_j), this latter may lose its arc-consistency, thereby altering the invariant. We prove in the following that this can happen only if x_k is a future variable (i.e., $k > i$).

Indeed, suppose x_k is a past variable, that is $k \leq i$. We distinguish two cases:

1. $k = i$
   In that case, (x_j, x_k), is arc-consistent after executing the first loop of Propagate. Moreover, since all the elements of D'_k are DI. we can apply lemma 8 to (x_j, x_k) and deduce that all the pairs of $D'_j \times D'_k$ are compatible.
2. $k < i$
   Assume without loss of generality that $k < j$. Since $j < i$, according to the induction hypothesis, (x_k, x_j) is arc-consistent before instantiating x_i. Moreover, since all the elements of D_j are DI, we can apply lemma 8 to (x_k, x_j) and deduce that all value pairs in $D_k \times D_j$ are compatible. It follows that, since $D'_k \subseteq D_k$ and $D'_j \subseteq D_j$, all the pairs of $D'_k \times D'_j$ are compatible too.

In both cases, (x_k, x_j) is already arc-consistent before executing the second loop of Propagate unless D'_j is empty. Furthermore, (x_k, x_j) cannot lose its arc-consistency since there is no disallowed tuple in $D'_k \times D'_j$. Hence, there is no need to process any arc whose source node represents a past variable in the second loop of Propagate. As a result, the second loop cannot reduce the domain of any past variable. Then, it cannot cause the loss of arc-consistency for an arc whose end-point is a past variable. Thus, to restore the invariant, the second loop processes only once the arcs (x_k, x_j) such that x_k a future variable and x_j a past variable whose domains has been modified in the first loop. □

In fact, at the end of each complete path constructed by FC–DI, we get in D a solution bundle [10], that is, a set of solution expressed as a Cartesian product of subsets of the value domain of each variable.

Theorem 10. *Each complete path built by FC–DI yields a solution bundle, i.e., all the elements of* $D_1 \times \ldots \times D_n$ *are solutions.*

Proof. When FC–DI succeeds to build a complete path, all the variables are past variables. Then, according to lemma 9, all $D_i, i : 1, \ldots, n$ are filtered such that all the arcs of the constraint graph are arc-consistent. From lemma 8 it follows that, since each arc (x_i, x_j) such that $i < j$ is arc-consistent and all the elements of D_j are DI, all the elements of $D_i \times D_j$ are compatible This holds for all $1 \leq i < j \leq n$. As a result, all the elements of $D_1 \times \ldots \times D_n$ are solutions. □

4.3 An Example

Let us consider the CSP depicted in Fig. 1. All variables have the same value domain which is equal to $\{0, 1, 2, 3, 4\}$. The application of FC-DI to this example gives the steps detailed in Fig. 2. For simplicity, we did not use any variable or value ordering heuristic. Note that in step 1 all the values in D_1 are DI since x_1 is the first variable in the ordering and then $\vec{N}(x_1, a) = \emptyset$ for all $a \in D_1$. In the next step, the DI based partition of D_2 gives $\Delta_2 = \{\{0, 2, 4\}, \{1, 3\}\}$. In a first attempt, D_2 is reduced to $\{0, 2, 4\}$ which yields the removal of $0, 2$ and 4 from D_1, D_3 and D_4. All paths explored under this branch lead to an empty domain. Hence, the algorithm backtracks to x_2 and sets D_2 to the remaining element in Δ_2 which is $\{1, 3\}$. By constraint propagation, the value domains of D_1, D_3 and D_4 are reduced to $\{0, 2, 4\}$. Then the algorithm processes x_3 whose value domain is trivially partionned into $\Delta_3 = \{\{0\}, \{2\}, \{4\}\}$ because of the difference constraint between x_3 and x_1. The choice of value 0 for x_3 leads to a bundle containing two solutions (see the last step in Fig. 2).

5 Experimental Results

The problems investigated in our experiments are random binary CSPs generated according to model B, radio link frequency assignment problems (RLFAP), modified RLFAP and job-shop problems. We compared FC-DI and MAC-DI with FC and MAC. The arc-consistency algorithm underlying MAC and MAC-DI is AC-3. The variable ordering heuristic used by all algorithms is *min-domain/wdeg*. For value ordering, we used the min-conflict-first heuristic. The evaluation criteria are the number of expanded nodes and CPU time in second. All algorithms were implemented in C++. They were run in a Windows xp 1.7 GHZ PC having 256 Mb of RAM.

Random Binary CSPs: for random binary CSPs, we experimented on problems involving $n = 30$ variables, a uniform domain size of $d = 25$. The problem density p_1, (i.e. the ratio of the number of constraints present in the constraint

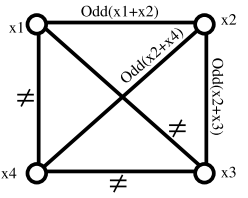


Fig. 1. A binary CSP

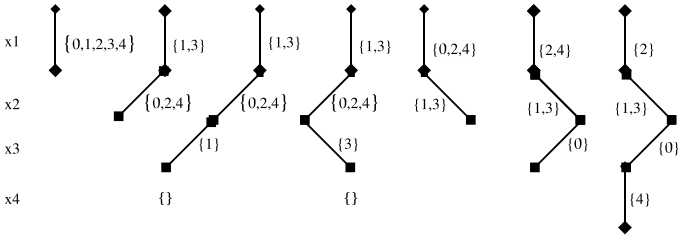


Fig. 2. Paths explored by FC-DI when applied to the CSP of Fig. 1

graph over that of all possible constraints), is varied from 0.4 to 0.6 with 0.1 increment step. The constraint tightness p_2, (i.e. the ratio of the number of disallowed tuples over that of all possible tuples), is varied so that we obtain instances around the peak of complexity, except for problems with $p_1 = 0.6$, we restricted the tests to under-constrained problems because the execution times were prohibitively long around the peak of complexity. We used the model B generator developed by Frost et *al.* which is available at [1]. According to [15], we started from $p_1 = 0.4$ in order to avoid flawed instances. It must be emphasized, however, that around the peak of complexity, model B instances are much more difficult to solve than those obtained following model RB which is a flawless version of model B. The size of samples is 100 problem instances for each data point. We compared FC against FC-DI which seem to be the best algorithms on these problems.

For Fig. 3 which reports results expressed as average values of number of expanded nodes and CPU time, the horizontal axis denotes various tightness. The results indicate that FC-DI outperforms FC. This advantage is less and less significant as we move toward dense problems. For the densest problems ($p_1 = 0.6$), both algorithms are very close. Nonetheless, FC-DI remains almost always faster that FC.

RLFAP and Modified RLFAP: We considered the scen11 instance and instances obtained from scen11 by removing the highest frequencies denotes by scen11-f2 to scen11-f10 [3]. We compared MAC against FC-DI on these instances because on the one hand MAC outperformed FC and on the other hand FC-DI gave better results than MAC-DI.

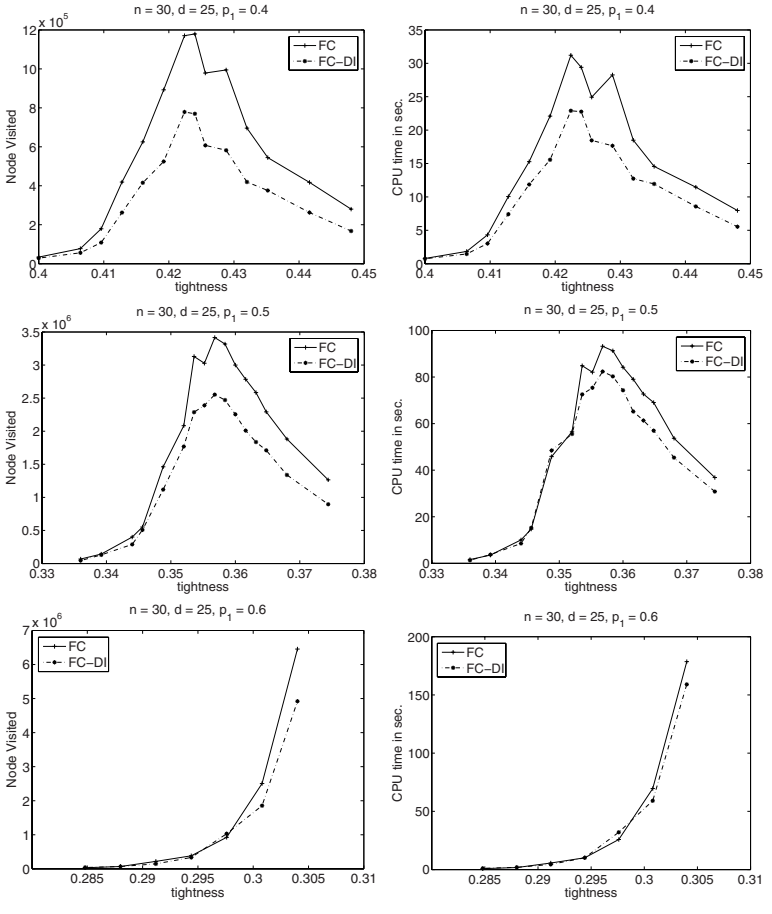


Fig. 3. Average search effort for FC and FC-DI. Mean number of expended nodes and mean CPU time is reported.

As can be seen in Fig. 4, FC-DI is clearly better than MAC regarding CPU time, although the reverse is true if we consider the number of expanded nodes. But the latter criterion is less significant since we compared FC-DI against MAC which spends much more time at each node of the search tree.

Job-shop Problems: For the job-shop problems, we investigated the js-taillard-15-by-15-105 instances [4]. The experiment involves ten satisfiable instances generated according to the Taillard model [14]. In the resulting CSPs, the variables represent the tasks to be scheduled, the domain values represent the possible start-times of the different tasks and the constraints are precedence and resource constraints. To cope with the large domain values, (more than one thousand values), which caracterise these instances, we resorted to a limited discrepancy search [8]. Hence, we immersed the MAC and the MAC-DI algorithms in a loop

	expended nodes		time (in sec.)	
instance	MAC	FC-DI	MAC	FC-DI
scen11 (sat)	1.2K	7.3K	1	1
scen11-f10	1.3K	1.9K	1	1
scen11-f9	22K	40K	11	5
scen11-f8	38K	43K	20	6
scen11-f7	284K	482K	99	49
scen11-f6	394K	580K	141	64
scen11-f5	2022K	3150K	713	313
scen11-f4	12M	18M	3564	1682
scen11-f3	38M	69M	11571	6726
scen11-f2	–	221M	>30K	16.7K

Fig. 4. Search effort for MAC and FC-DI in terms of thousands of expended nodes and execution time (in second) obtained on RLFAP and modified RLFAP instances

	expended nodes		time (in sec.)	
instance	MAC	FC-DI	MAC	FC-DI
tai-15-by-15-105-0	–	25907	–	286
tai-15-by-15-105-1	18859	9710	183	92
tai-15-by-15-105-2	–	171615	–	1563
tai-15-by-15-105-3	–	93180	–	948
tai-15-by-15-105-4	–	26394	–	271
tai-15-by-15-105-5	–	176110	–	1532
tai-15-by-15-105-6	27073	644429	213	6183
tai-15-by-15-105-7	–	–	–	–
tai-15-by-15-105-8	1183674	248104	7261	2332
tai-15-by-15-105-9	830122	937	6308	13

Fig. 5. Search effort for LDS-MAC and LDS-MAC-DI. Number of expended nodes and CPU time are reported. A dash indicates a timeout after 3 hours.

which incrementally varies the number of allowed discrepancies. Discrepancies are defined with regard to the min-conflit-first value ordering heuristic. The resulting searches are denoted by LDS-MAC and LDS-MAC-DI.

On these instances, the results are somewhat chaotic. However, LDS-MAC-DI is clearly more efficient than LDS-MAC as it can be seen in Fig. 5. Indeed, within three hours per instance, LDS-MAC solved only four instances over ten, while LDS-MAC-DI solved nine instances. Moreover, on instances solved by both algorithms, LDS-MAC-DI is faster on three instances over four.

6 Related Work

Many variations of neighborhood interchangeability have been studied. In [9], Haselböck proposed a weak form of interchangeability which is limited to a single constraint and showed that it can be useful for enhancing filtering algorithms.

In [16], the authors proposed conditional interchangeability, which is intended to strengthen the pruning ability of NI. A condition is a restriction on the domain of neighboring variables whose role is to capture interchangeably in a limited situation that cannot be detected by NI.

Bowen and Likitvivatanavong introduced domain transmutation a concept which is closely related to interchangeability [2]. This approach consists in splitting domain values into several "sub-values" or merging values so that interchangeability is more intensively exploited. The authors reported that their approach is particularly advantageous in finding all solutions.

The concept of neighborhood interchangeability was also applied to non-binary CSPs [11]. The authors proposed an algorithm for computing NI values which takes into account non-binary constraints. They also described how to interleave their algorithm with search in order to obtain a search which performs dynamic bundling.

While the above approaches consider on all the neighboring variables to determine interchangeable values, our approach examines past variables only. It can therefore be classified as a look-back scheme for exploiting interchangeability. From the practical point of view, directional interchangeability do not enable the removal of redundant values but in compensation it allow to attempt more than one value at a single branch of the search tree.

7 Conclusion

This paper presented directional interchangeability (DI), a weak form of neighborhood interchangeability that assumes a variable ordering and uses this order to focuses on a subset of the neighboring variables. We presented an effective algorithm for determining DI values and described how to integrate it into the Forward-checking algorithm to get a solving algorithm (FC-DI) that assigns to variables subsets of their respective domains at each branch of search.

An experimental study carried on numerous binary CSPs showed that directional interchangeability is a concrete technique that repays its overhead. Indeed, FC-DI defeats the standard FC and MAC on many random and structured problems experimentally proving that exploiting DI values is worth the effort.

A natural extension of this work is to consider directional substitutability. This extension of the present work is motivated by the fact that directional substitutability may occur more frequently than directional interchangeability. Another direction consists in exploiting directional substitutability in the framework of non-binary CSPs. In the case of a k-ary CSP, determining directionally substitutable values amounts to testing inclusions between sets containing $(k-1)$-tuples which is not a prohibitive task.

References

1. Frost, D., Bessière, C., Dechter, R., Régin, J.C.: Random uniform CSP generators. http://www.lirmm.fr/~bessiere/generator.html.
2. Bowen J., Likitvivatanavong, C.: Domain transmutation in constraint satisfaction problems. Proceedings of AAAI-04 2004.

3. Cabon, B., De Givry, S., Lobjois, L. Schiex, T., Warners, J. P.: Radio link frequency assignement. Constraints **4** (1) 79–89.
4. http://www.cril.univ-artois.fr/ lecoutre/research/benchmarks/benchmarks.html.
5. Dechter, R., Pearl, J.: Network-based heuristics for constraint satisfaction problems. Artificial Intelligence **34** (1988) 1–38.
6. Freuder, E. C.: Eliminating interchangeable values in constraint satisfaction problems. Proceedings of AAAI-91 (1991) 227–233.
7. Haralick, R., Elliot, G.: Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence **14** (1980) 263–313.
8. Harvey, W., Ginsberg, M.: Limited discrepancy search. In Proceedings of the 14^{th} International Joint Conference On Artificial Intelligence (1995) 607–613.
9. Haselböck, A.: Exploiting interchangeability in constraint satisfaction problems. In Proceedings of IJCAI-93 (1993) 282–297.
10. Ginsberg, M.L., Parkes, A. J., Roy, A.: Super-model and robustness. Proceeding of AAAI-98 Madison Wisconsin 1998.
11. Lal, A., Choueiry, B.Y., Freuder, E.C.: Neighborhood interchangeability and dynamic bundling for non-binary finite CSPs. Proceedings of AAAI-05 (2005) 397–404.
12. Sabin, D., Freuder, E.C: Contradicting conventional wisdom in constraint satisfaction. Proceedings of the 11th European Conference on Artificial Intelligence (1994).
13. Weigel R., Faltings, B. V.: Compiling constraint satisfaction problems. Artificial Intelligence **115** (2) 257–287.
14. Taillard, E.: Benchmarks for basic scheduling problems. European Journal of Operational Research **64** (1993) 278–285.
15. Xu, K., Li, W.: Exact phase transitions in random constraint satisfaction problems. Journal of Artificial Intelligence Research **12** (2000) 93–103.
16. Zhang, Y., Freuder, E.C.: Conditional interchangeability and substitutability. Proceedings of Fourth International Workshop on Symmetry and Constraint Satisfaction Problems (SymCon04) Toronto 2004.

A Continuous Multi-resources *cumulative* Constraint with Positive-Negative Resource Consumption-Production

Nicolas Beldiceanu and Emmanuel Poder

École des Mines de Nantes, LINA FRE CNRS 2729,
4 rue Alfred Kastler, La Chantrerie, BP 20722, 44307 Nantes Cedex 3, France.
{Nicolas.Beldiceanu,Emmanuel.Poder}@emn.fr

Abstract. This article first introduces an extension of the classical *cumulative* constraint: each task is no more a rectangle but rather a sequence of contiguous trapezoid sub-tasks with variable duration and heights. The resource function is no more constant but is a positive or negative piecewise linear function of time. Finally, a task is no more pre-assigned to one resource, but to a task corresponds a set of possible resource assignments. In this context, this article provides an $O(p \cdot (\log p + q))$ for computing all the cumulated resource profiles where q is the number of resources and p is the total number of trapezoid sub-tasks of all the tasks.

1 Introduction

Most of the work in resource-constrained scheduling is dedicated to problems dealing with tasks that use a constant amount of resource during their execution [3]. However, application fields handling continuously divisible resources like water, oil or electric power, prompt at considering more complex resource consumption or production profiles [8], [6], [11] and [7].

A first contribution of this article is a task model that unifies the two tasks models introduced in [1] and [9]. Each task is no more a rectangle but rather a sequence of contiguous trapezoid sub-tasks. The duration and heights of a trapezoid sub-task are variables over intervals and express a positive or a negative linear resource function of time. Hence, a trapezoid sub-task has positive or negative variable heights and a positive variable duration. A task is non–preemptive, needs for its execution to be assigned to exactly one resource, and to each task corresponds a set of possible resource assignments. Finally, a task must start during a given time interval (due, for instance, to a release date, or to the availability of the resource). Symmetrically, it must also finish during a given time interval. Finally, The duration of each sub-task belongs to a specified intervals.

The main contribution is, for a given resource, an effective algorithm in $O(p \cdot (\log p + q))$ for computing a lower and upper estimation of its final resource profile where q is the number of resources and p is the total number of trapezoid

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 214–228, 2007.

sub-tasks of all the tasks. These two estimations will be respectively called the *minimum* and *maximum cumulated resource's profiles*. The minimum resource utilisation profile provides an easy way for checking that a given partial schedule (i.e., a schedule where the tasks and sub-tasks variables are not yet all fixed) is not feasible according to the fact that one should not exceed an overall given resource limit. Similarly, the maximum resource utilisation profile allows to check that one can effectively reach, at specific time intervals, a given level of minimum resource utilisation. Moreover, these two profiles coincide when all the tasks are completely fixed. The minimum resource utilisation profile is made up from the sum of all minimum task's profiles, where the *minimum task's profile* represents, for each time point, the smallest possible contribution of the task to a resource profile.[1] For a given task T including possibly positive and negative heights, the computation of its minimum task profile is more complex than just determining a compulsory part [5]. In fact, we first fix the heights of each sub-tasks of T to their minimum value and then consider separately its positive sub-tasks T^+ and its negative sub-tasks T^-. For the former, we compute its *compulsory part* [9] (i.e., the intersection of all feasible schedules of T^+) and for the latter its *envelope* (i.e., the union of all feasible schedules of T^-) and recombine them. Note that the method relies on the fact property the compulsory part of T^+ and envelope of T^- don't overlap each other over time.[2]

The article is organised as follows: Section 2 introduces the piecewise linear *cumulatives_pwl* constraint and the associated new task model. Section 3 presents different possible contributions of a task to the utilisation of a resource: the compulsory part, the envelope, and finally, the minimum and maximum task's profiles. Section 4 first defines, for each resource, its minimum and maximum cumulated resource's profiles. Then, it describes a sweep algorithm for computing them from all minimum and maximum task's profiles. All derived algorithms are polynomial in the total number of trapezoid sub-tasks.

2 The Piecewise Linear *cumulatives_pwl* Constraint

We consider a set of q renewable resources where the k^{th} resource has a maximum capacity $C_k \geq 0$ and a set of n non–preemptive tasks. Each task needs for its execution to be assigned to exactly one resource within a given subset (that depends of the considered task) of the q resources. Finally, each task is composed of a sequence of contiguous trapezoid sub-tasks which expresses a piecewise linear function of resource requirement. Within this context, this section introduces the task model as well as the corresponding constraint.

2.1 Task Model

After introducing the task model used throughout this article, this section presents the notion of feasible instance of a task as well as the notion of resource function.

[1] The maximum resource utilisation profile and the maximum task's profile are defined in a similar way.

[2] The computation of the maximum task's profile is similar.

Definition 1. *A task* T_i *is defined by a quintuple* $(s_{T_i}, td_{T_i}, e_{T_i}, Seq_{T_i}, a_{T_i})$ *where:*

- *The variables*[3] s_{T_i}, td_{T_i} *and* e_{T_i} *represent respectively the* start, *the* duration *and the* end *of task* T_i.
- Seq_{T_i} *is a sequence of contiguous trapezoid of* p_i *sub-tasks* $\langle T_i^1, T_i^2, \cdots, T_i^{p_i} \rangle$ *where the trapezoid sub-task* T_i^j $(j = 1..p_i)$ *is defined by a triple* $(sh_{T_i^j}, d_{T_i^j}, eh_{T_i^j})$ *where the variables* $sh_{T_i^j}$, $d_{T_i^j}$ *and* $eh_{T_i^j}$ *respectively represent the* start height *of* T_i^j *(i.e., the resource requirement at its start), the* duration *of* T_i^j *and the* end height *of* T_i^j *(i.e., the resource requirement at its end). Moreover, we assume that* $(\underline{sh}_{T_i^j} \geq 0 \vee \overline{sh}_{T_i^j} \leq 0)$ *and* $(\underline{eh}_{T_i^j} \geq 0 \vee \overline{eh}_{T_i^j} \leq 0)$[4] *and finally that* $\underline{d}_{T_i^j} \geq 0$.
- *The variable* a_{T_i} *represents the* resource assignment *of task* T_i *and takes its value in a finite set of integer values* $dom(a)$[5].

$\underline{s}_{T_i}$ is called the *release date* (earliest starting time) while $\overline{e}_{T_i}$ is the *due date* (latest ending time) of T_i. $\overline{s}_{T_i}$ and $\underline{e}_{T_i}$ are respectively the *latest starting time* and the *earliest finishing time* of T_i.

Example 1. The fixed task T, defined by the quintuple $(s, td, e, Seq, a) = (1, 7, 8, \langle (2,3,3), (3,1,1), (0,1,-1), (0,1,0), (3,1,0) \rangle, 1)$, starts at instant 1, has a total duration of 7 and ends at instant 8 and consists of 5 contiguous trapezoid sub-tasks. Moreover, T is assigned to resource 1 and its piecewise linear resource function (resource requirement) h_T is defined by: $h_T(t) = 1/3 \cdot (t-1) + 2$ for $t \in [1,4[$, $h_T(t) = -2 \cdot (t-4) + 3$ for $t \in [4,5[$, $h_T(t) = -(t-5)$ for $t \in [5,6[$, $h_T(t) = 0$ for $t \in [6,7[$ and $h_T(t) = -2 \cdot (t-7) + 2$ for $t \in [7,8[$.

Definition 3 introduces the notion of fixed feasible task which is a fixed task's instance verifying the constraint $\sum_{j=1}^{p_i} d_{T_i^j} = td \in [\underline{td}_{T_i} .. \overline{td}_{T_i}]$. For this purpose we first introduce the notion of fixed feasible trapezoid sub-task.

Definition 2. *Given a task* T_i *defined by* $(s_{T_i}, td_{T_i}, e_{T_i}, Seq_{T_i}, a_{T_i})$ *and a given total duration* $td \in [\underline{td}_{T_i} .. \overline{td}_{T_i}]$, *a* fixed feasible trapezoid sub-tasks sequence of the task T_i with total duration td *is such that all the variables* $sh_{T_i^j}$, $d_{T_i^j}$, $eh_{T_i^j}$ $(j = 1..p)$ *are fixed within their respective range and satisfy* $\sum_{j=1}^{p_i} d_{T_i^j} = td$.

Definition 3. *Given a task* T_i *defined by* $(s_{T_i}, td_{T_i}, e_{T_i}, Seq_{T_i}, a_{T_i})$ *and given* Seq, *a fixed feasible trapezoid sub-tasks sequence of* T_i, *a* fixed feasible instance *of* T_i *is such that* s_{T_i} *and* e_{T_i} *are fixed within their respective possible values and* Seq_{T_i} *is fixed to* Seq. *We note* Φ_{T_i} *the set of all the fixed feasible tasks of* T_i.

[3] A variable v ranges over the interval of consecutive integer values $[\underline{v}, \overline{v}]$.

[4] The signs of $sh_{T_i^j}$ and $sh_{T_i^j}$ are initially known.

[5] Without loss of generality all variables, except the resource assignment, could be continuous variables.

Example 2. Throughout this article, we will consider the following example where $Tasks$ is the collection of the four tasks T_1, T_2, T_3 and T_4 defined as follows:

$$\begin{array}{llll} T_1 = (1..2, & 4..5, & 5..6, & \langle (1..2, 2..3, 2), (-1, 2, -1)\rangle, \quad \{1, 2\}) \\ T_2 = (1..2, & 6, & 7..8, & \langle (3, 2, 2), (-2, 2, -1), (1, 2, 1)\rangle, 1 \quad) \\ T_3 = (0..3, & 6, & 6..9, & \langle (1, 2, 2), (1, 2, 1), (1, 2, 0)\rangle, \quad 1 \quad) \\ T_4 = (1..6, & 2, & 3..8, & \langle (-1, 2, -1)\rangle, \quad \{1, 2\}) \end{array}$$

All the starts and ends of tasks T_1 to T_4 are not fixed as well as the first trapezoid sub-task of T_1. T_1 and T_4 are not yet assigned. Figure 1 provides all the fixed feasible trapezoid sub-task sequences associated to tasks T_1 to T_4. Observe that task T_1 has four feasible sequences (two of duration 4 ($Seq_{1,1}$ and $Seq_{1,3}$) and two of duration 5 ($Seq_{1,2}$ and $Seq_{1,4}$)) while the other tasks have each only one feasible sequence. Finally, Figure 2 provides the six fixed feasible task's instances ($I_{1,k}$ $(k = 1..6)$) of T_1.

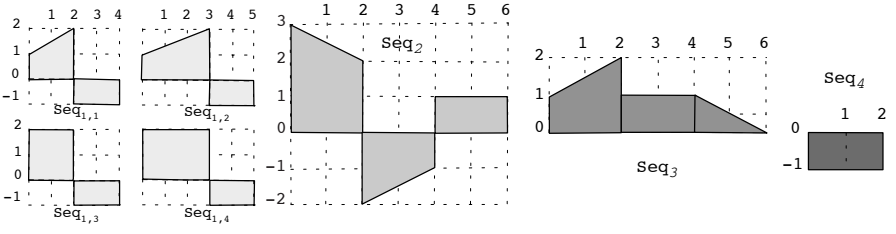


Fig. 1. Feasible trapezoid sub-tasks sequences of the tasks used throughout the article

Definition 4. *We note* $st_{T_i^j}$ *the start of* T_i^j $(st_{T_i^1} = s_{T_i})$. *To any* $I \in \Phi_{T_i}$ *is associated a* resource function $h_I(t)$ *defined on* $[s_{T_i}, e_{T_i}[$ *by:* $\forall\ t \in [s_{T_i}, e_{T_i}[, \exists! j \in \{1, 2, ..., p_i\}$ *such that* $t \in [st_{T_i^j}, st_{T_i^j} + d_{T_i^j}[$. *Then* $h_I(t) = \frac{eh_{T_i^j} - sh_{T_i^j}}{d_{T_i^j}} \left(t - st_{T_i^j}\right) + sh_{T_i^j}$ *and* $sl_{T_i^j} = \frac{eh_{T_i^j} - sh_{T_i^j}}{d_{T_i^j}}$ *is the* slope value *of the trapezoid sub-task* T_i^j. *If* Φ_{T_i} *is reduced to one instance (i.e.,* T_i *is fixed) then we note* h_{T_i} *the resource function of* T_i.

Example 3. (continuation of Example 2) The feasible task's instance $I_{1,1}$ (see Figure 2) associated with T_1 is $(1, 4, 5, \langle (1, 2, 2), (-1, 2, -1)\rangle, \{1, 2\}) = (1, 4, 5, Seq_{1,1}, \{1, 2\})$. Its resource function is defined by $h_I(t) = 1/2 \cdot (t - 1) + 1$ for $t \in [1, 3[$, $h_T(t) = -1$ for $t \in [3, 5[$.

The next definition introduces the notion of non–negative and non–positive task as well as the notion of positive, negative and null trapezoid sub-tasks.

Definition 5. *A task* T_i *is* non–negative *if the heights of each trapezoid sub-task are non–negative (i.e.,* $\forall\ j = 1..p_i, \underline{sh}_{T_i^j} \geq 0$ *and* $\underline{eh}_{T_i^j} \geq 0$*) and is* non–positive *if the heights of each trapezoid sub-tasks are non–positive (i.e.,* $\forall\ j = 1..p, \overline{sh}_{T_i^j} \leq 0$ *and* $\overline{eh}_{T_i^j} \leq 0$*). A trapezoid sub-task is* positive *if its two heights are non–negative*

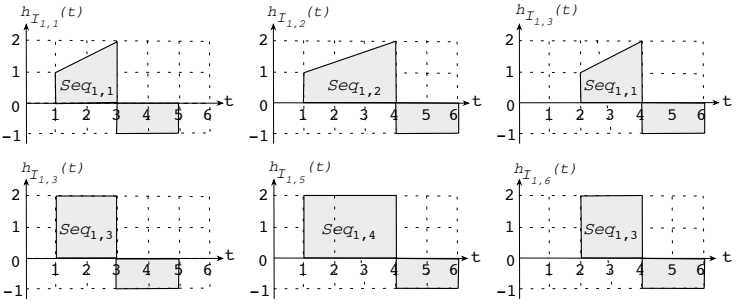


Fig. 2. Fixed feasible instances of T_1

and not both null and is negative *if its two heights are non–positive and not both null. Otherwise the trapezoid is* null *(*$\underline{sh}_{T_i^j} = \overline{sh}_{T_i^j} = \underline{eh}_{T_i^j} = \overline{eh}_{T_i^j} = 0$*).*

Example 4. (continuation of Example 3) T_3 is a non–negative task while T_4 is a non–positive task. T_1^1 is a positive trapezoid sub-task while T_1^2 is a negative trapezoid sub-task.

2.2 The Piecewise Linear Cumulative *cumulatives_pwl*

The *piecewise linear cumulative* constraint has the form *cumulatives_pwl*(*Tasks, Resources, Constraint*) where:

- *Tasks* is a collection of tasks $\langle T_1, T_2, \cdots, T_n \rangle$ where the task T_i $(i = 1..n)$ is defined by a quintuple $(s_{T_i}, td_{T_i}, e_{T_i}, Seq_{T_i}, a_{T_i})$.
- *Resources* is a set of q integers $C_1, C_2, \cdots, C_q$ where $C_k \geq 0$ $(k = 1..q)$ is the resource capacity of the k^{th} resource if $Constraint = '\leq'$ or the minimum level of the k^{th} resource if $Constraint = '\geq'$.
- *Constraint*[6] is the less or equal (i.e., $\leq$) or the greater or equal (i.e., $\geq$) constraint.

The constraint *cumulatives_pwl* holds if the following conditions are true:

1. $\forall\ i = 1..n$, $s_{T_i} + td_{T_i} = e_{T_i}$ (i.e., the end of a task is equal to the sum of its start and its duration),[7]
2. $\forall\ i = 1..n$, $\sum_{j=1}^{p_i} d_j = td_{T_i}$ (i.e., the total duration of a task is equal to the sum of the durations of its trapezoid sub-tasks),
3. Case $Constraint = '\leq'$: $(\forall\ k = 1..q)$, $(\forall\ t \in \mathbb{R})$, $\sum_{i/a_{T_i}=\{k\}} h_T(t) \leq C_k$ (i.e., for each resource k and for each instant t, the sum of the values taken at t by the resource functions that are assigned to the resource k, is less than or equal to the capacity C_k of the resource k).

[6] Observe that, currently, multiple execution modes for a task cannot be directly defined within the *cumulatives_pwl* constraint.

[7] When s_{T_i}, td_{T_i} and e_{T_i} belong to specific intervals, none of them is redundant.

Case $Constraint = ' \geq ' : (\forall\ k = 1..q), (\forall\ t \in \mathbb{R}$ such that $\exists T_i / t \in [s_{T_i}, e_{T_i}[), \sum_{i/a_{T_i}=\{k\}} h_T(t) \leq C_k$ (i.e., for each resource k and for each instant t such that at least one task is executed at t, the sum of the values taken at t by the resource functions of the tasks that are assigned to the resource k, is greater than or equal to the minimum level C_k of the resource k).

Example 5. (continuation of Example 2) The constraint *cumulatives_pwl*($\langle$ $(2, 4, 6, \langle(1,2,2),(-1,2,-1)\rangle, 2), (1, 6, 7, \langle(3,2,2),(-2,2,-1),(1,2,1)\rangle, 1)$ $(3, 6, 9, \langle(1,2,2),(1,2,1),(1,2,0)\rangle, 1), (1, 2, 3, \langle(-1,2,-1)\rangle, 1)$ $\rangle$ $\langle 2, 2\rangle$, $\leq$) holds.

3 Minimum and Maximum Cumulated Resource's Profiles

Given a resource r, this section shows how to compute its minimum (respectively maximum) cumulated resource's profiles for r. They are in fact determined by cumulating all minimum (maximum) task's profiles. For this purpose, we first remind (3.1 and 3.2) some results, proved by [9], for computing the compulsory part of a task and then we extend them to the computation of the envelope (3.3) [8]. Then, in 3.4, we explain how to use simultaneously the notions of compulsory part and envelope to compute the minimum and maximum task's profiles. Finally, in 4.1, we use the previous notions in order to compute the minimum and maximum resource's profiles for a given resource.

To compute the minimum task's profiles of a given task T, we first need to introduce two specific instances of T that are its earliest and its latest schedules.

3.1 Earliest and Latest Schedules of a Task T

Let T^{min} and T^{max} denote the earliest and the latest schedules of T and $\underline{st}_{T^j}$ and $\overline{st}_{T^j}$ $(j = 1..p)$ respectively denote the starts of the trapezoid sub-task T^j in the schedules T^{min} and T^{max}. Poder *et al.* have shown in [9] how to compute $\underline{st}_{T^j}$ and $\overline{st}_{T^j}$ with a complexity of $O(p)$. To obtain T^{min} the task T is started at its earliest date $\underline{s}_T$; then, $st_{T^1}, st_{T^2}, \cdots, st_{T^p}$ are successively fixed as small as possible with respect to the feasibility of the end of the task. The following recursive formulae computes $\underline{st}_{T^j}$ $(j = 1..p+1)$:

$\underline{st}_{T^1} = \underline{s}_T$ and $\forall\ j = 1..p, \underline{st}_{T^{j+1}} = \max\left(\underline{st}_{T^j} + \underline{d}_{T^j}, \underline{e}_T - \sum_{k=j+1}^{p} \overline{d}_{T^k}\right)$.

T^{max} is obtained in a similar way i.e.,:

$\overline{st}_{T^1} = \overline{s}_T$ and $\forall\ j = 1..p, \overline{st}_{T^{j+1}} = \min\left(\overline{st}_{T^j} + \overline{d}_{T^j}, \overline{e}_T - \sum_{k=j+1}^{p} \underline{d}_{T^k}\right)$.

Observe that $\underline{st}_{T^{p+1}} = \underline{e}_T$ and $\overline{st}_{T^{p+1}} = \overline{e}_T$.

[8] Within the context of a non–negative task, its compulsory part and its envelope can be respectively interpreted as the lower and upper bounds of the task (i.e., at each instant they provide the minimum and maximum heights of the task).

3.2 Compulsory Part of a Non–negative Task

The *compulsory part* was initially introduced by Lahrichi [5], for a rectangle task as the intersection of all feasible schedules of the task. As the domains of the variables of the task get more and more restricted, the compulsory part will increase until becoming a schedule of the task. Within rectangle tasks, [4] use the notion of compulsory part to tighten lower bounds for resource-constrained scheduling problems, while [2] use the compulsory part in a set of propagation rules to solve cumulative constraints.

Here, we consider a non–negative task T. Nevertheless, observe that computing the compulsory part of a non–positive task is symmetric: in fact, if T is defined, for any t, by $h_T(t) = -h_{T'}(t)$ then, for any t, $h_{CP(T')}(t) = -h_{CP(T)}(t)$.

Note that it is sufficient, in the computation of the compulsory part of T, to consider only feasible instances of T where heights are minimum. So in the following, a task T has its heights fixed at their minimum.

Definition 6. *The compulsory part $CP(T)$ of a task T is not empty if and only if $\overline{s}_T < \underline{e}_T$. Its resource function $h_{CP(T)}$ satisfies for any $t \in [\overline{s}_T, \underline{e}_T[$ $h_{CP(T)}(t) = inf_{I \in \Phi_T}(h_I(t))$ and $h_{CP(T)}(t) = 0$ elsewhere.*

In order to compute the compulsory part of a task, we first need to introduce the notion of valley in a sequence of trapezoid sub-tasks and the associated task named the Level Valley Task.

Definition 7. *Let $H_T^{min} = h_1 h_2 \cdots h_{2p} = \underline{sh}_{T^1} \underline{eh}_{T^1} \underline{sh}_{T^2} \underline{eh}_{T^2} \cdots \underline{sh}_{T^p} \underline{eh}_{T^p}$ be the sequence of all start and end minimum heights of trapezoid sub-tasks of T. An height $h_k \in H_T^{min}$ $(1 < k < 2p)$ defines* an end of valley *if and only if $\exists\, j$ $(1 < j \leq k)$ such that $h_{j-1} > h_j$ and $h_j = h_{j+1} = \cdots = h_k$ and $h_k < h_{k+1}$ (a strict decrease in the resource function is followed by a strict increase).*

Result 1. *Let $h_{j_1} h_{j_2} \cdots h_{j_v}$ be the possibly empty sub-sequence of H_T^{min} of all heights of trapezoid sub-tasks of T that correspond to the ends of the v valleys. Let $LVT(T)$, called the* Level Valley Task *of T, be the task defined for any $t \in [\overline{s}_T, \underline{e}_T[$ by $h_{LVT(T)}(t) = \min\left(+\infty, \left\{h_{j_k} \text{ such that } t \in \left[\underline{st}_{T^{\lfloor \frac{j_k}{2} \rfloor + 1}}, \overline{st}_{T^{\lfloor \frac{j_k}{2} \rfloor + 1}}\right[\right\}\right)$*

Then, $CP(T) = T^{min} \bigcap T^{max} \bigcap LVT(T)$ i.e., for any $t \in [\overline{s}_T, \underline{e}_T[$, $h_{CP(T)}(t) = \min\left(h_{T^{min}}(t), h_{T^{max}}(t), h_{LVT(T)}(t)\right)$ and is null elsewhere. If T has no valley ($v = 0$) then, for any $t \in [\overline{s}_T, \underline{e}_T[$ $h_{CP(T)}(t) = \min\left(h_{T^{min}}(t), h_{T^{max}}(t)\right)$.

Example 6. The left part of Figure 3 illustrates the computation of $CP(T)$ where T has two valleys (see Part (A)). They correspond to the start st_{T^2} of T^2 ($h_1 = \underline{sh}_{T^2}$) and the end st_{T^4} of T^3 ($h_2 = \underline{eh}_{T^3}$). Parts (A) and (B) give respectively T^{min} and T^{max}. Then, Part (C) shows $LVT(T)$ which is defined by $h_{LVT(T)}(t) = +\infty$ for $[\overline{s}_T, \underline{st}_{T^2}[$, $h_{LVT(T)}(t) = h_1$ for $t \in [\underline{st}_{T^2}, \underline{st}_{T^4}[$, and $h_{LVT(T)}(t) = h_2$ for $t \in [\underline{st}_{T^4}, \overline{st}_{T^2}[\bigcup [\overline{st}_{T^2}, \underline{e}_T[$ (as $h_1 > h_2$). Finally, Part (D) shows the computation of $CP(T)$ as the intersection of T^{min}, T^{max} and $LVT(T)$.

Complexity for computing the compulsory part. Within [9], Poder *et al.* provide an algorithm for computing the compulsory part of a task, made of p trapezoid sub-tasks and $v > 0$ valleys, in $O(p+v \cdot log v)$. In fact, T^{min} and T^{max} are computed in $O(p)$ (i.e., the complexity for computing the two sequences $\underline{st}^j$ and $\overline{st}^j (j = 1..p+1)$), $LVT(T)$ is computed using a heap structure in $O(v \cdot \log v)$. Then, the intersection of T^{min}, T^{max} and $LTT(T)$ is obtained, by scanning them in parallel, in $O(p+v)$. Hence, a complexity of $O(p+v \cdot \log v)$ that is $O(p \log p)$. If $v = 0$ the complexity is $O(p)$.

3.3 Envelope of a Non–negative Task

The *envelope* of a task is the union of all feasible schedules of the task. As the domains of the variables of the task get more and more restricted, the envelope will decrease until it becomes a schedule of the task. In the context of multiple resources, a task has the same envelope on each resource where it may be potentially assigned and an empty envelope elsewhere. Here, we consider a non–negative task T and we compute its envelope $Env(T)$. Nevertheless, observe that computing the envelope of a task where the heights of each trapezoid sub-task are non–positive is symmetric. in fact, if T is defined, for any t, by $h_T(t) = -h_{T'}(t)$ then, for any t, $h_{Env(T')}(t) = -h_{Env(T)}(t)$.

Definition 8. *The envelope $Env(T)$ of a task T is such that its resource function $h_{Env(T)}$ satisfies $h_{Env(T)} = sup_{I \in \Phi_T}(h_I(t))$ for any $t \in [\underline{s}_T, \overline{e}_T[$ and $h_{Env(T)}(t) = 0$ elsewhere.*

Note that it is sufficient, in the computation of the envelope of T, to consider only feasible schedules of T where the heights of each trapezoid sub-task are fixed at their maximum. So, in this section, a task T has its heights fixed at their maximum.

As we have introduced, for computing the compulsory part of a task, the notion of valley in a sequence of trapezoid, we introduce now, for computing the envelope, the notion of top and the associated task named the Level Top Task.

Definition 9. *Let $H_T^{max} = h_1 h_2 \cdots h_{2p} = \overline{sh}_{T^1}\overline{eh}_{T^1}\overline{sh}_{T^2}\overline{eh}_{T^2} \cdots \overline{sh}_{T^p}\overline{eh}_{T^p}$ be the sequence of all start and end maximum heights of trapezoid sub-tasks of T. A height $h_k \in H_T^{max} (1 \leq k \leq 2p)$ defines an* end of a top *if and only if $\exists\, j (1 \leq j \leq k)$ such that $h_{j-1} < h_j$ and $h_j = h_{j+1} = \cdots = h_k$ and $h_k > h_{k+1}$ with the convention that $h_0 = h_{2p+1} = 0$.*

Result 2. *Let $h_{j_1} h_{j_2} \cdots h_{j_w}$ be the sub-sequence of HT^{max} of all heights of T that define a top. The* Level Top Task *$LTT(T)$ of T is the task defined by:*
$\forall\, t \in [\underline{s}_T, \overline{e}_T[, h_{LVT(T)}(t) = \max\left(0, \left\{h_{j_k} \text{ such that } t \in \left[\underline{st}_{T^{\lfloor \frac{j_k}{2} \rfloor + 1}}, \overline{st}_{T^{\lfloor \frac{j_k}{2} \rfloor + 1}}\right[\right\}\right)$
(with the convention that $s_{T_p+1} = e$). The envelope *of T is obtained by computing the union of T^{min}, T^{max} and $LTT(T)$ i.e., $h_{Env(T)}$ satisfies $h_{Env(T)}(t) = sup(h_{T^{min}}(t), h_{T^{max}}(t), h_{LTT(T)}(t))$ for any $t \in [\underline{s}_T, \overline{e}_T[$ and 0 elsewhere.*

The demonstration of Result 2 is similar to the proof of Result 1 so is omitted.

Example 7. The right part of Figure 3 illustrates the computation of the envelope of a task T that has three tops with heights $h_1 = \overline{sh}_{T^1}$, $h_2 = \overline{eh}_{T^2}$ and $h_3 = \overline{eh}_{T^4}$ (see part (E)). They correspond to the start $s_T = st_{T^1}$ of the task, the end st_{T^3} of the second sub-task and the end $e_T = st_{T^4}$ of the task T. Parts (E) and (F) give respectively T^{min} and T^{max}. Then, Part (G) shows $LTT(T)$ that is defined by $h_{LTT(T)}(t) = h_1$ for $t \in [\underline{s}_T, \overline{s}_T[$, $h_{LVT(T)}(t) = h_2$ for $t \in [\underline{st}_{T^3}, \underline{e}_T[$ (as $h_2 < h_3$ on $[\underline{e}_T, \overline{st}_{T^3}[$), and $h_{LVT(T)}(t) = h_3$ for $t \in [\underline{e}_T, \overline{e}_T[$. Finally, part (H) shows the computation of $Env(T)$ as the union of T^{min}, T^{max} and $LTT(T)$.

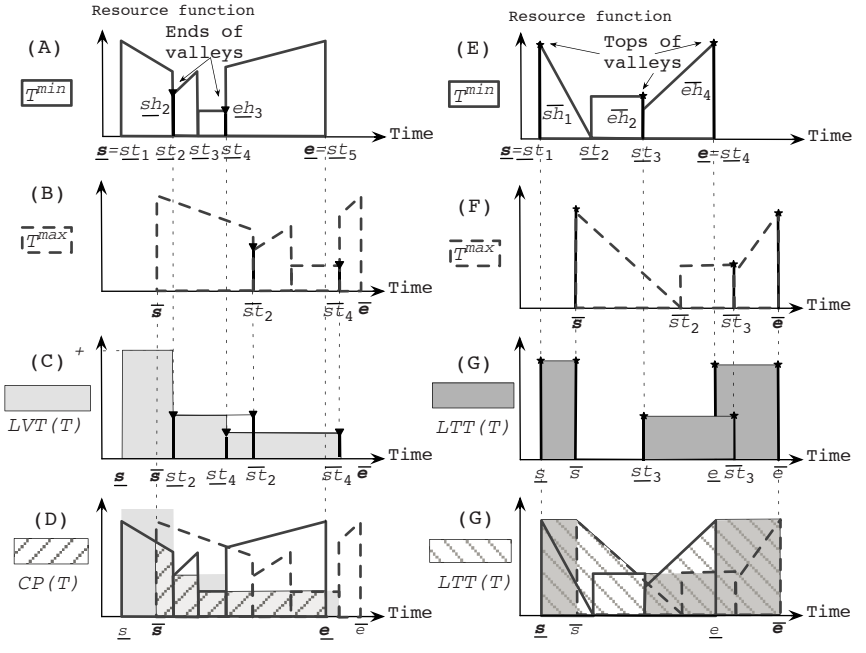


Fig. 3. Computation of the compulsory part and the envelope of a task

Complexity for computing the envelope of a task. A task T has always at least one top except if T satisfies $\forall\, t, h_T(t) = 0$. The algorithm to compute the envelope is similar to the one given for the computation of the compulsory part. So it has a complexity of $O(p + w \cdot \log w)$ that is $O(p \log p)$.

3.4 Minimum and Maximum Task's Profiles of Any Task T

We now explain how to compute the minimum and maximum task's profiles of a task T. Let us first introduce this two notions formally.

Definition 10. *Let $mP(T)$ (resp. $MP(T)$) denote the* minimum *(resp.* maximum*)* task's profile *of the task T. Its resource function $h_{mP(T)}$ (resp. $h_{MP(T)}$ satisfies, for any $t \in [\underline{s}_T, \overline{e}_T[$, $h_{mP(T)}(t) = inf_{I \in\, \Phi_T}(h_I(t))$ (resp. $h_{MP(T)}(t) = sup_{I \in\, \Phi_T}(h_I(t))$.*

Note that it is enough, in the computation of $mP(T)$ (resp. $MP(T)$) to consider only feasible schedules where all trapezoid's heights of T are fixed at their minimum (resp. maximum).

Assume now that heights of T are fixed. Then, let T^+ (resp. T^-) denote the task made from T by replacing each negative (resp. positive) trapezoid sub-task T^j of T by a null trapezoid sub-task and same duration as the replaced trapezoid sub-task (we merge consecutive null sub-tasks in a single sub-task).

The next result expresses the resource function of the minimum (resp. maximum) task's profile of a task T according to the resource functions of the compulsory part of T^+ (resp. T^-) and of the enveloppe of T^- (resp. T^+).

Result 3. *Let t be a given time point.*
If T is assigned to a resource then, on that resource

$$\begin{cases} h_{mP(T)}(t) = h_{CP(T^+)}(t) + h_{Env(T^-)}(t), \\ h_{MP(T)}(t) = h_{CP(T^-)}(t) + h_{Env(T^+)}(t). \end{cases}$$

else, on each resource where T may be assigned

$$\begin{cases} h_{mP(T)}(t) = h_{Env(T^-)}(t), \\ h_{MP(T)}(t) = h_{Env(T^+)}(t). \end{cases}$$

Moreover, for any t, we can't have neither both $h_{CP(T^+)}(t) \neq 0$ and $h_{Env(T^-)}(t) \neq 0$ nor both $h_{Env(T^+)}(t) \neq 0$ and $h_{CP(T^-)}(t) \neq 0$.

Proof. of Result 3

As the definitions of $mP(T)$ and $MP(T)$ are symetric, we only prove the result for $mP(T)$. Note that, by definition of T^+ and T^-, for any instance $I \in \Phi_T$, the associated instances I^+ and I^- are such that $h_{I^+}(t) = \max(0, h_I(t))$ and $h_{I^-}(t) = \min(0, h_I(t))$. Therefore $h_{CP(T^+)}(t) = \inf_{I \in \Phi_T}(h_{I^+}(t))$ and $h_{Env(T^-)}(t) = \inf_{I \in \Phi_T}(h_{I^-}(t))$.

We distinguish two cases: $\forall\ I \in \Phi_T, h_I(t) > 0$ (1) and its contrary $\exists\ I \in \Phi_T / h_I(t) \leq 0$ (2). In the former case, $\forall\ I \in \Phi_T, h_I(t) = h_{I^+}(t)$. Then $\inf_{I \in \Phi_T}(h_I(t)) = \inf_{I \in \Phi_T}(h_{I^+}(t)) > 0$. i.e., $h_{mP(T)}(t) = h_{CP(T^+)}(t) > 0$. Moreover, $\forall\ I \in \Phi_T, h_{I^-}(t) = 0$ so $h_{Env(T^-)}(t) = 0$. In the latter case $\inf_{I \in \Phi_T}(h_I(t)) = \inf_{I \in \Phi_T}(h_{I^-}(t)) > 0$. i.e., $h_{mP(T)}(t) = h_{Env(T^-)}(t)$. Moreover, $\inf_{I \in \Phi_T}(h_{I^+}(t)) = 0$ i.e $h_{CP(T^+)}(t) = 0$.

We have proved that we can't have both $h_{CP(T^+)}(t) \neq 0$ and $h_{Env(T^-)}(t) \neq 0$ (by definition $h_{CP(T^+)}(t) \geq 0$ and $h_{Env(T^-)}(t) \leq 0$) and that if $h_{CP(T^+)}(t) > 0$ then $h_{mP(T)}(t) = h_{CP(T^+)}(t)$ else $h_{mP(T)}(t) = h_{Env(T^-)}(t)$. □

Complexity for computing the minimum and maximum profiles. From Result 3 and as the complexity for computing a compulsory part or an envelope is $O(p \log p)$ then the complexity for computing the minimum or the maximum profile is also $O(p \log p)$.

Example 8. (continuation of Example 2) As T_1 and T_4 may be assigned to resources 1 and 2 ($a_{T_1} = a_{T_4} = \{1, 2\}$), only envelopes are taken into account so

$h_{mP(T_1)} = h_{Env(T_1^-)}$ and $h_{mP(T_4)} = h_{Env(T_4)}$ ($T_4^- = T_4$ as T_4 is non–positive). As T_2 and T_3 are assigned to resource 1 ($a_{T_2} = a_{T_3} = \{1\}$), compulsory parts and envelopes are taken into account so $h_{mP(T_2)} = h_{CP(T_2^+)} + h_{Env(T_2^-)}$ and $h_{mP(T_3)} = h_{CP(T_3)}$ ($T_3^+ = T_3$ and $T_3^- = \emptyset$ as T_3 is non–negative). Parts (A) to (D) of Figure 4 respectively give $mP(T_1)$, $mP(T_2)$, $mP(T_3)$ and $mP(T_4)$. Finally, the following table summarises the different profiles (a trapezoid sub-task is encoded by $\langle start, startheight, end, endheight\rangle$).

$T_i; a_{T_i}$	$CP(T_i^+)$	$Env(T_i^-)$	$mP(T_i)$
$T_1; a_{T_1} = \{1,2\}$	$\langle 2,1,3,3/2\rangle$	$\langle 3,-1,6,-1\rangle$	$Env(T_1^-)$
$T_2; a_{T_2} = 1$	$\langle 2,5/2,3,2\rangle$, $\langle 6,1,7,1\rangle$	$\langle 3,-2,4,-2\rangle$, $\langle 4,-2,6,-1\rangle$	$CP(T_2^+) \cup Env(T_2^-)$
$T_3; a_{T_3} = 1$	$\langle 3,1,4,1\rangle$,$\langle 4,1,2,6,0\rangle$	$\emptyset$	$CP(T_3)$
$T_4; a_{T_4} = \{1,2\}$	$\emptyset$	$\langle 1,-1,8,-1\rangle$	$Env(T_4)$

4 Computing All the Minimum and Maximum Cumulated Resource's Profiles Using a Sweep Algorithm

This section first defines the notions of minimum and maximum cumulated resource's profiles. Then it describes a polynomial (according to the number of sub-tasks and of resources) sweep algorithm to compute, for each resource r its minimum cumulated resource's profile $mcrP(r)$. These profile is computed from all minimum task's profiles $mP(T_i)$ $i = 1..n$ of the *cumulatives_pwl* constraints.

4.1 Minimum and Maximum Cumulated Resource's Profiles

We now show how to compute the minimum and maximum cumulated resource's profiles from the minimum and the maximum task's profiles.

Definition 11. *Let r be a resource and $mcrP(r)$ (resp. $McrP(r)$) denotes the* minimum (resp. maximum) cumulated profile of resource r *by all the tasks. Then, for any t, $h_{mcrP(r)}(t) = \sum_{T_i / r \in a_{T_i}} h_{mP(T)}(t)$ (resp. $h_{McrP(r)}(t) = \sum_{T_i, r \in a_{T_i}} h_{MP(T)}(t)$).*

As the domains of the variables of all tasks are progressively reduced (until becoming completely fixed), $mcrP(r)$ and $McrP(r)$ respectively increases and decreases (until becoming a same single profile). From now on, as $mcrP(r)$ and $McrP(r)$ have similar definitions, we focus on the computation of $mcrP(r)$

Example 9. (continuation of Example 8) Parts (E) and (F) of Figure 4 and the following table provide $mcrP(1)$ and $mcrP(2)$ ($h_{mcrP(1)} = \sum_{i=1}^{4} h_{mP(T_i)}$ and $h_{mcrP(2)} = h_{mP(T_1)} + h_{mP(T_4)}$)

r	Trapezoid sub-tasks of $mrP(r)$
1	$\langle 1,-1,2,-1\rangle$, $\langle 2,3/2,3,1\rangle$, $\langle 3,-3,4,-3\rangle$, $\langle 4,-3,6,-3\rangle$, $\langle 6,0,7,0\rangle$, $\langle 7,-1,8,-1\rangle$
2	$\langle 1,-1,3,-1\rangle$, $\langle 3,-2,6,-2\rangle$, $\langle 6,-1,8,-1\rangle$

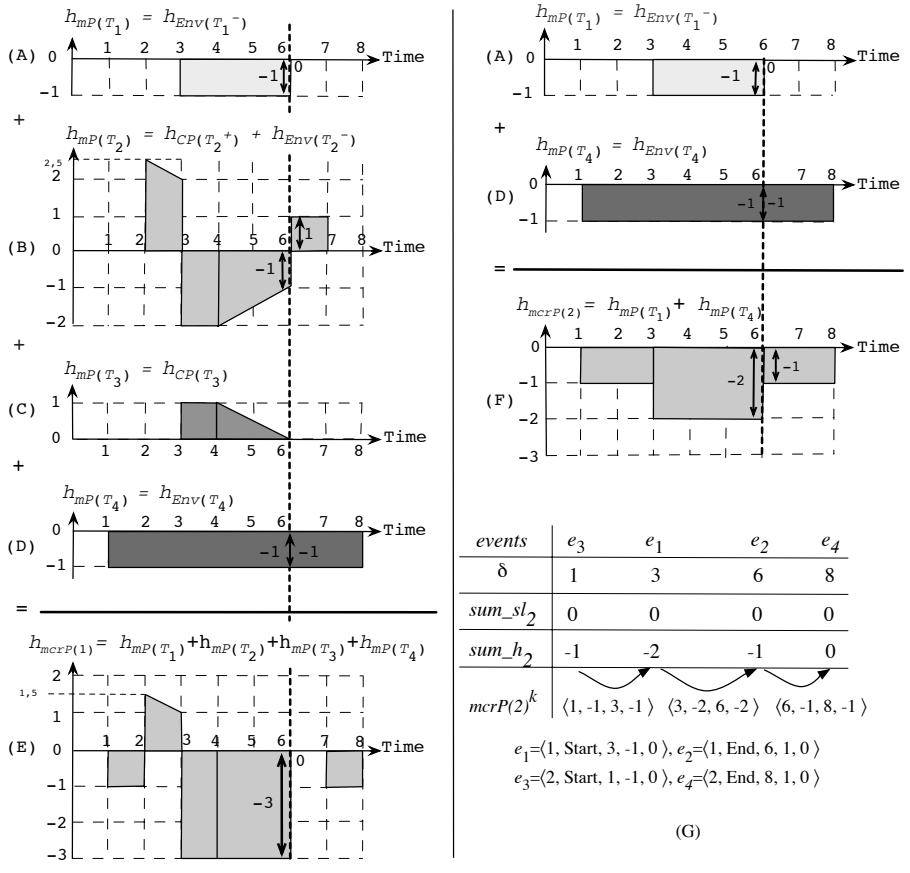


Fig. 4. Computation of $mcrP(1)$ and $mcrP(2)$

4.2 Sweep Algorithm

To compute $mcrP(r)$, for a given resource r, we have extended the sweep algorithm presented in [1] in order to handle trapezoid sub-tasks. A general introduction on sweep is given in [10] and an example of use of sweep, within the context of disjunctive scheduling, is provided in [12].

The extended sweep algorithm moves a vertical line Δ from one event to the next event - in our context, an event corresponds to a start or an end of a trapezoid sub-task of $mP(T_i)$ $(i = 1..n; r \in a_{T_i})$ - on the time axis and builds $mcrP(r)$ incrementally. It uses two data structures:

- The first one, called *the sweep line status*, contains for the resource r some information related to the current position δ of the vertical line Δ: sum_h_r and sum_sl_r, respectively the height and the slope of $mcrP(r)$ at position δ.
- The second data structure, called *the event points series*, contains the events associated with the trapezoid sub-tasks used for building $mcrP(r)$ (namely

the trapezoid sub-tasks of $mP(T_i)$ such that T_i may be assigned to r (see Result 3 and Definition 11). We encode these events by using the following fields: $\langle task, kind, date, height, slope\rangle$, where:

- *task* indicates which task generates the event.
- *kind* expresses whether the event corresponds to the start ($kind = \text{Start}$) or to the end ($kind = \text{End}$) of trapezoid that has generated this event.
- *date* specifies the location in time of the event i.e., contains, if $kind = \text{Start}$, the start of the trapezoid otherwise its end.
- *height* gives the quantity to add to sum_h_r. It contains, if $kind = \text{Start}$, the height of the trapezoid at its start otherwise it contains the opposite of the height at its ends.
- *slope* gives the quantity to add to sum_sl_r. It contains, if $kind = \text{Start}$, the slope of the trapezoid otherwise it contains the opposite of its slope.

Each non–null sub-task $mP(T_i)^j$ [9] of $mP(T_i)$ generates the pair of events:
$\langle i, \text{Start}, st_{mP(T_i)^j}, sh_{mP(T_i)^j}, sl_{mP(T_i)^j}\rangle$
$\langle i, \text{End}, et_{mP(T_i)^j}, -eh_{mP(T_i)^j}, -sl_{mP(T_i)^j}\rangle$
where $sl_{mP(T_i)^j} = \frac{eh_{mP(T_i)^j} - sh_{mP(T_i)^j}}{d_{mP(T_i)^j}}$.

All the events are initially computed and sorted (by Main_Algorithm), in the list L_{events}, in increasing lexicographically order according to the pair $\langle kind, date\rangle$ where $kind = end$ is considered to be less than $kind = start$. Observe that one event participates to the construction of all $mcrP(r)$ such that $r \in a_{T_i}$.

sum_h_r and sum_sl_r are initially set to 0 (line 1 of SA) and δ to the date of the first event, associated to a task T_i such that $r \in a_{T_i}$, on the time axis (line 3 of SA). When the current position of Δ changes (line 5) from δ_k to δ_{k+1}, the sweep algorithm:

- First computes the k^{th} trapezoid sub-task $mcrP(r)^k$ of $mcrP(r)$ (line 6).
- Then (line 7) verifies that $mcrP(r)^k$ do not exceed the resource capacity C_r, otherwise the constraint has no solution.
- Then sum_h_r takes the value of the end height of the previous returned trapezoid sub-task ($mcrP(r)^k$) and δ is updated to δ_{k+1} (line 9).

In any case all the contributions (heights and slopes) of sub-tasks that start or end at δ_{k+1} are taken into account in sum_h_r and sum_sl_r (line 10).

Example 10. (continuation of Example 8) The sub-task $\langle 3, -1, 6, -1\rangle$ of $mP(T_1)$ generates the pair of events ($e_1 = \langle 1, \text{Start}, 3, -1, 0\rangle$, $e_2 = \langle 1, \text{End}, 6, 1, 0\rangle$) while the trapezoid sub-task $\langle 1, -1, 8, -1\rangle$ of $mP(T_4)$ generates the pair of events ($e_3 = \langle 4, \text{Start}, 1, -1, 0\rangle, e_4 = \langle 4, \text{End}, 8, 1, 0\rangle$). The sorted relevant events of L_{events} for computing $mcrP(2)$ from $mP(T_1)$ and $mP(T_4)$ are $\langle e_3, e_1, e_2, e_4\rangle$. When e_1 becomes the current event then, the sweep line moves from position 1 to 3: the algorithm first computes the 1^{st} trapezoid sub-task $(1, -1, 3, -1)$ of

[9] $mP(T_i)^j = (st_{mP(T_i)^j}, sh_{mP(T_i)^j}, et_{mP(T_i)^j}, eh_{mP(T_i)^j})$ (start, startheight, end, end-height) denote the j^{th} trapezoid sub-task of the profiles $mP(T_i)$.

$mcrP(2)$ (see Figure 4 Parts (F) and (G)). Then the sweep line status is updated: $sum\_h_r \leftarrow -1$ and $\delta \leftarrow 3$. Finally, the contribution of e_1 is added to sum_h_r which decreases from -1 to to -2 and to sum_sl_r which doesn't change.

Sweep_Algorithm (SA) - Compute $mcrP(r)$ for a given resource r
In: The list L_{events} of all sorted events $\langle task, kind, date, height, slope\rangle$ and a resource r.
Out: Fail if $mcrP(r)$ exceed the resource capacity C_r; Otherwise Delay.

```
1: k ← 1; sum_h_r ← 0; sum_sl_r ← 0;
2: Extract, if it exists, the first event e from L_events such that r ∈ a_{T_{e.task}};
3: δ ← e.date;
4: while e is defined do
5: |  if e.date ≠ δ then /* Δ has just move: Compute the k^th trapezoid of mcrP(r) */
6: |  |  st ← δ; sh ← sum_h_r; d ← e.date − δ; eh ← sum_sl_r * (e.date − δ) + sum_h_r;
7: |  |  if sh > C_r ∨ eh > C_r then return Fail; /* mcrP(r)^k exceeds C_r */
8: |  |  mcrP(r)^k ← (st, sh, e.date, eh); k ← k + 1;
9: |  |  sum_h_r ← eh; δ ← e.date; /* Update sum_h_r and δ */
10: |  sum_h_r ← sum_h_r + e.height; sum_sl_r ← sum_sl_r + e.slope;
11: |  Extract, if it exists, the next event e from L_events such that r ∈ a_{T_{e.task}};
12: return Delay;
```

Main_Algorithm (MA) - Compute all $mcrP$
In: The tasks T_i $(i = 1..n)$; **Out:** Fail if there is no solution otherwise return Delay.

```
1: L_events ← ∅;
2: for i = 1 to n do /* For all the tasks */
3: |  Compute mP(T_i);
4: |  for j = 1 to ‖mP(T_i)‖ do /* Generate mP-events */
5: |  |  if (sh_{mP(T_i)^j} ≠ 0 ∨ eh_{mP(T_i)^j} ≠ 0) then /* i.e., not a null sub-task */
6: |  |  |  sl = (eh_{mP(T_i)^j} − sh_{mP(T_i)^j})/d_{mP(T_i)^j};
7: |  |  |  Add ⟨i, Start, st_{mP(T_i)^j}, sh_{mP(T_i)^j}, sl⟩ to L_events;
8: |  |  |  Add ⟨i, End, et_{mP(T_i)^j}, −eh_{mP(T_i)^j}, −sl⟩ to L_events;
9: Sort L_events in increasing lexicographically order according to ⟨date, kind⟩;
10: for r = 1 to q do if Sweep_Algorithm(L_events, r) = Fail then return Fail;
11: return Delay;
```

The next result gives the complexities for computing all the minimum cumulated resource's profiles using a sweep algorithm.

Result 4. *The complexities for computing all the minimum cumulated resource's profiles using a sweep algorithm is* $O(p \cdot (\log p + q))$ *where* q *is the number of resources,* $p = \sum_{i=1}^{n}(p_i)$ *is the total number of trapezoid sub-tasks of the* n *tasks.*

Proof. of Result 4. The complexity for computing (see 3.4) all the $mP(T_i)$ and generating all the events is of $O\left(\sum_{i=1}^{n}(p_i \log p_i)\right)$. The complexity for sorting the at most $2 \cdot p + n$ events, using a sort algorithm with a complexity in $O(p \log p)$, is $O(p \log p)$. Finally, for each resource r, the Sweep_Algorithm computes $mcrP(r)$ in $O(p)$ (in the worst case). Therefore, we obtain a complexity in the worst case of $O\left(\sum_{i=1}^{n}(p_i \log p_i) + p \cdot \log p + \sum_{i=1}^{q} p\right)$ i.e., $O(p \cdot (\log p + q))$. □

5 Conclusion

We have introduced a new task model that takes advantages of two models: in the first one, several resources could be handled, but only a constant consumption or production of resource could be expressed. In the second one, only positive piecewise linear resource functions and a single resource were considered. For this new task model, we came up with a polynomial algorithm to compute, for each resource, its minimum and maximum cumulated resource's profiles.

References

1. Beldiceanu, N. and Carlsson, M. (2002). A New Multi-Resource cumulatives Constraint with negative heights. In: P. Van Hentenryck, ed. 8*th Int. Conf. on Principles and Practice of Constraint Programming - CP'2002*, Ithaca, NY, USA, Sept. 8-13, 2002. Springer-Verlag: LNCS 2470, pp 63–79.
2. Caseau Y, Laburthe F (1996) Cumulative Scheduling with Task Intervals. In Proceedings of *the Joint International Conference and Symposium on Logic Programming*, The MIT Press.
3. Herroelen, W., Demeulemeester, E. and De Reyck, B. (1998). A Classification Scheme for Project Scheduling Problems. In: Weglarz J, ed., *Project Scheduling Recent Models, Algorithms and Applications*. Kluwer Academic Publishers, pp 1–26.
4. Klein R, Scholl A (1999) Computing lower bounds by destructive improvement: An application to resource-constrained project scheduling, *European Journal of Operational Research*, Vol 112, pp 322-346.
5. Lahrichi, A. (1982). Scheduling: the Notions of Hump, Compulsory Parts and their Use in Cumulative Problems. *C. R. Acad. Sci. Paris*, t. 294, pp 209–211.
6. Laborie, P. (2003). Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, Vol 143, pp 151–188.
7. Maravelias, C.T. and I. E. Grossmann, (2004). A Hybrid MILP/CP Decomposition Approach for the Continuous Time Scheduling of Multipurpose Batch Plants, *Computers & chemical engineering*, vol. 28 (10), pp. 1921–1949.
8. Muscettola, N. (2002). Computing the Envelope for Stepwise-Constant Resource Allocations. In: P. Van Hentenryck, ed. 8*th Int. Conf. on Principles and Practice of Constraint Programming - CP'2002*, Ithaca, NY, USA, Sept. 8-13, 2002. Springer-Verlag: LNCS 2470, pp 139–153.
9. Poder, E., Beldiceanu, N., Sanlaville E. (2004). Computing a lower approximation of the compulsory part of a task with varying duration and varying resource consumption. *European Journal of Operational Research*, Vol. 153, pp 239–254.
10. Preparata, F. P. and Ian Shamos, M. (1995). *Computational Geometry, An introduction*. Texts and Monographs in Computer Science. Springer Verlag, New-York.
11. Sourd F and Rogerie J (2002) Continuous Filling and Emptying of Storage Systems in Constraint-Based Scheduling. *8th International Workshop on Project Management and Scheduling - MPS 200*2, Valencia, Spain, April 3-5.
12. Wolf, A. (2003). Filtering while Sweeping over Task Intervals. In: F. Ross, ed. 9*th Int. Conf. on Principles and Practice of Constraint Programming - CP 2003*, Kinsale, Ireland. Sept./Oct. 2003. Springer-Verlag: LNCS 2833, pp 739–753.

Replenishment Planning for Stochastic Inventory Systems with Shortage Cost

Roberto Rossi[1], S. Armagan Tarim[2], Brahim Hnich[3], and Steven Prestwich[1]

[1] Cork Constraint Computation Centre, University College, Cork, Ireland
{r.rossi,s.prestwich}@4c.ucc.ie
[2] Department of Management, Hacettepe University, Turkey
armagan.tarim@hacettepe.edu.tr
[3] Faculty of Computer Science, Izmir University of Economics, Turkey
brahim.hnich@ieu.edu.tr

Abstract. One of the most important policies adopted in inventory control is the (R,S) policy (also known as the "replenishment cycle" policy). Under the non-stationary demand assumption the (R,S) policy takes the form (R_n,S_n) where R_n denotes the length of the n^{th} replenishment cycle, and S_n the corresponding order-up-to-level. Such a policy provides an effective means of damping planning instability and coping with demand uncertainty. In this paper we develop a CP approach able to compute optimal (R_n,S_n) policy parameters under stochastic demand, ordering, holding and shortage costs. The convexity of the cost-function is exploited during the search to compute bounds. We use the optimal solutions to analyze the quality of the solutions provided by an approximate MIP approach that exploits a piecewise linear approximation for the cost function.

1 Introduction

Much of the inventory control literature concerns the computation of optimal replenishment policies under demand uncertainty. One of the most important policies adopted is the (R,S) policy (also known as the *replenishment cycle* policy). In this policy a replenishment is placed every R periods to raise the inventory position to the order-up-to-level S. This provides an effective means of damping planning instability (deviations in planned orders, also known as *nervousness*) and coping with demand uncertainty. As pointed out by Silver et al. ([8], pp. 236–237), (R,S) is particularly appealing when items are ordered from the same supplier or require resource sharing. In these cases all items in a coordinated group can be given the same replenishment period. Periodic review also allows a reasonable prediction of the level of the workload on the staff involved, and is particularly suitable for advanced planning environments. For these reasons (R,S) is a popular inventory policy.

An important class of stochastic production/inventory control problems assumes a non-stationary demand process. Under this assumption the (R,S) policy takes the non-stationary form (R_n,S_n) where R_n denotes the length of the n^{th}

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 229–243, 2007.

replenishment cycle and S_n the corresponding order-up-to-level. To compute the *near* optimal policy parameters for (R_n,S_n), Tarim and Kingsman [4] propose a mixed integer programming (MIP) formulation using a piecewise linear approximation to a complex cost function.

This paper focuses on the work of Tarim and Kingsman, in which a finite-horizon, single-installation, single-item (R_n,S_n) policy is addressed. They assume a fixed procurement cost each time a replenishment order is placed, whatever the size of the order, and a linear holding cost on any unit carried over in inventory from one period to the next. Instead of employing a service level constraint — the probability that at the end of every time period the net inventory will not be negative is at least a certain value (see Tarim and Kingsman [3] for (R_n,S_n) under a service level constraint) — their model employs a penalty cost scheme. They propose a certainty-equivalent formulation of the above problem in the form of a MIP model. So far no CP approach has been proposed for (R_n,S_n) under a penalty cost. In fact, as shown in [4], the cost structure is complex in this case and it differs significantly from the one under a service level constraint. In [2] the authors proposed a CP model under a service level constraint. In this paper it was shown that not only CP is able to provide a more compact formulation than the MIP one, but that it is also able to perform faster and to take advantage of dedicated pre-processing techniques that reduce the size of decision variable domains. Moreover dedicated cost-based filtering techniques were proposed in [1] for the same model, these techniques are able to improve performances of several orders of magnitude.

In this paper, we give an *exact* formulation of the (R_n,S_n) inventory control problem via constraint programming, instead of employing a piecewise linear approximation to the total expected cost function. This exact CP formulation provides an optimal solution to (R,S) policy. Our contribution is two-fold: we can now obtain provably optimal solutions, and we can gauge the accuracy of the piecewise linear approximation proposed by Tarim and Kingsman.

2 Problem Definition and (R_n, S_n) Policy

The demand d_t in period t is considered to be a normally distributed random variable with known probability density function (PDF) $g_t(d_t)$, and is assumed to occur instantaneously at the beginning of each period. The mean rate of demand may vary from period to period. Demands in different time periods are assumed to be independent. A fixed holding cost h is incurred on any unit carried over in inventory from one period to the next. Demands occurring when the system is out of stock are assumed to be back-ordered and satisfied as soon as the next replenishment order arrives. A fixed shortage cost s is incurred for each unit of demand that is back-ordered. A fixed procurement (ordering or set-up) cost a is incurred each time a replenishment order is placed, whatever the size of the order. In addition to the fixed ordering cost, a proportional direct item cost v is incurred. For convenience, and without loss of generality, the initial inventory level is set to zero and the delivery lead-time is not incorporated. It is assumed

that negative orders are not allowed, so that if the actual stock exceeds the order-up-to-level for that review, this excess stock is carried forward and does not return to the supply source. However, such occurrences are regarded as rare events and accordingly the cost of carrying the excess stock is ignored. The above assumptions hold for the rest of this paper.

The general multi-period production/inventory problem with stochastic demands can be formulated as finding the timing of the stock reviews and the size of non-negative replenishment orders, X_t in period t, minimizing the expected total cost over a finite planning horizon of N periods:

$$\min \; E\{TC\} = \int_{d_1}\int_{d_2}\dots\int_{d_N}\sum_{t=1}^{N}\left(a\delta_t + vX_t + hI_t^+ + sI_t^-\right) g_1(d_1)\dots g_N(d_N)\mathrm{d}(d_1)\dots\mathrm{d}(d_N) \tag{1}$$

subject to

$$X_t > 0 \Rightarrow \delta_t = 1 \tag{2}$$

$$I_t = \sum_{i=1}^{t}(X_i - d_i) \tag{3}$$

$$I_t^+ = \max(0, I_t) \tag{4}$$

$$I_t^- = -\min(0, I_t) \tag{5}$$

$$X_t, I_t^+, I_t^- \in \mathbb{Z}^+ \cup \{0\}, \quad I_t \in \mathbb{Z}, \quad \delta_t \in \{0,1\} \tag{6}$$

for $t = 1 \dots N$, where

d_t : the demand in period t, a normal random variable with PDF $g_t(d_t)$,
a : the fixed ordering cost,
v : the proportional direct item cost,
h : the proportional stock holding cost,
s : the proportional shortage cost,
δ_t : a {0,1} variable that takes the value of 1 if a replenishment occurs in period t and 0 otherwise,
I_t : the inventory level at the end of period t, $-\infty < I_t < +\infty$, $I_0 = 0$
I_t^+ : the excess inventory at the end of period t carried over to the next period, $0 \le I_t^+$,
I_t^- : the shortages at the end of period t, or magnitude of negative inventory $0 \le I_t^-$,
X_t : the replenishment order placed and received in period t, $X_t \ge 0$.

The proposed non-stationary (R,S) policy consists of a series of review times and associated order-up-to-levels. Consider a review schedule which has m reviews over the N period planning horizon with orders arriving at $\{T_1, T_2, \dots, T_m\}$, $T_j > T_{j-1}$. For convenience $T_1 = 1$ is defined as the start of the planning horizon and $T_{m+1} = N + 1$ the period immediately after the end of the horizon.

In [3], the decision variable X_{T_i} is expressed in terms of a new variable $S_t \in \mathbb{Z}$, where S_t may be interpreted as the opening stock level for period t, if there is no replenishment in this period (i.e. $t \neq T_i$ and $X_t = 0$) and the order-up-to-level for the i-th review period T_i if there is a replenishment (i.e. $t = T_i$ and $X_t > 0$). According to this transformation the expected cost function, Eq. (1), is written as the summation of m intervals, T_i to T_{i+1} for $i = 1, \ldots, m$, defining $D_{t_1,t_2} = \sum_{j=t_1}^{t_2} d_j$:

$$\begin{aligned} \min\ E\{TC\} = \sum_{i=1}^{m} \left(a\delta_{T_i} + \sum_{t=T_i}^{T_{i+1}-1} E\{C_{T_i,t}\} \right) + \\ vI_N + v \int_{D_{1,N}} D_{1,N} \times g(D_{1,N}) \mathrm{d}(D_{1,N}), \end{aligned} \tag{7}$$

The term $v \int_{D_{1,N}} D_{1,N} \times g(D_{1,N}) \mathrm{d}(D_{1,N})$ is constant and can therefore be ignored in the optimization model. $E\{C_{T_i,t}\}$ of Eq. (7) is defined as:

$$\int_{-\infty}^{S_{T_i}} h\left(S_{T_i} - D_{T_i,t}\right) g(D_{T_i,t}) \mathrm{d}(D_{T_i,t}) - \int_{S_{T_i}}^{\infty} s\left(S_{T_i} - D_{T_i,t}\right) g(D_{T_i,t}) \mathrm{d}(D_{T_i,t}). \tag{8}$$

As stated in [4], $E\{C_{T_i,t}\}$ is the expected cost function of a single-period inventory problem where the single-period demand is $D_{T_i,t}$. Since S_{T_i} may be interpreted as the order-up-to-level for the i-th review period T_i and $S_{T_i} - D_{T_i,t}$ is the end of period inventory for the "single-period" with demand $D_{T_i,t}$, the expected total subcosts $E\{C_{T_i,t}\}$ are the sums of single-period inventory costs where the demands are the cumulative demands over increasing periods. By dropping the T_i and t subscripts in Eq. (8) we obtain the following well-known expression for the expected total cost of a single-period newsvendor problem:

$$E\{TC\} = h \int_{-\infty}^{S} (S - D) g(D) \mathrm{d}(D) - s \int_{S}^{\infty} (S - D) g(D) \mathrm{d}(D) \tag{9}$$

where we consider two cost components: holding cost on the positive end of period inventory and shortage cost for any back-ordered demand. Let $G(\cdot)$ be the cumulative distribution function of the demand in our single-period newsvendor problem. A known result in inventory theory (see [17]) is convexity of Eq. (9). The so-called *Critical Ratio*, $\frac{s}{s+h}$, can be seen as the service level β (i.e. probability that at the end of the period the inventory level is non-negative) provided when we fix the order-up-to-level S to the optimal value S^* that minimizes expected holding and shortage costs (Eq. (9)). By assuming $G(\cdot)$ to be strictly increasing, we can compute the optimal order-up-to-level as $S^* = G^{-1}\left(\frac{s}{s+h}\right)$.

2.1 Stochastic Cost Component in Single-Period Newsvendor

We now aim to characterize the cost of the policy that orders S^* units to meet the demand in our single-period newsvendor problem. Since the demand

D is assumed to be normal with mean μ and standard deviation σ, then we can write $D = \mu + \sigma Z$, where Z is a standard normal random variable. Let $\Phi(z) = \Pr(Z \leq z)$ be the cumulative distribution function of the standard normal random variable. Since $\Phi(\cdot)$ is strictly increasing, $\Phi^{-1}(\cdot)$ is uniquely defined. Let $z_\beta = \Phi^{-1}(\beta)$, since $\Pr(D \leq \mu + z_\beta\sigma) = \Phi(z_\beta) = \beta$, it follows that $S^* = \mu + z_\beta\sigma$. The quantity z_β is known as the safety factor and $S^* - \mu = z_\beta\sigma$ is known as the safety stock. It can be shown [17] that

$$\int_{S^*}^{\infty} (S^* - D)g(D)\mathrm{d}(D) = E\{D - S^*\}^+ = \sigma E\{Z - z_\beta\}^+ = \sigma[\phi(z_\beta) - (1-\beta)z_\beta] \tag{10}$$

where $\phi(\cdot)$ is the PDF of the standard normal random variable. Let $E\{S^* - D\}^+ = \int_{-\infty}^{S}(S - D)g(D)\mathrm{d}(D)$, it follows

$$\begin{aligned} E\{TC(S^*)\} = h \cdot E\{S^* - D\}^+ + s \cdot E\{D - S^*\}^+ = \\ h \cdot (S^* - \mu) + (h+s)E\{D - S^*\}^+ = \\ hz_\beta\sigma + (h+s)\sigma E\{Z - z_\beta\}^+ = \\ hz_\beta\sigma + (h+s)\sigma[\phi(z_\beta) - (1-\beta)z_\beta] = \\ (h+s)\sigma\phi(z_\beta) \end{aligned} \tag{11}$$

The last expression $(h+s)\sigma\phi(z_\beta)$ holds only for the optimal order-up-to-level S^* that provides the service level $\beta = \left(\frac{s}{s+h}\right)$ computed from the *critical ratio* (CR). Instead, expression

$$hz_\alpha\sigma + (h+s)\sigma[\phi(z_\alpha) - (1-\alpha)z_\alpha] \tag{12}$$

can be used to compute the expected total cost for any given level S such that $\alpha = \Phi\left(\frac{S-\mu}{\sigma}\right)$. In Fig. 1 we plot this cost for a particular instance as a function of the opening inventory level S.

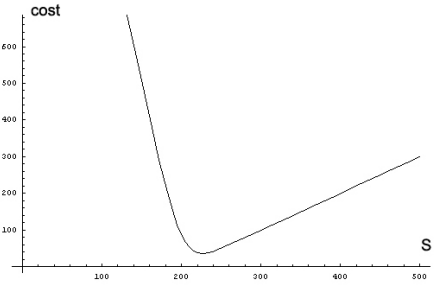


Fig. 1. Single-period holding and shortage cost as a function of the opening inventory level S. The demand is normally distributed with mean 200 and standard deviation 20. Holding cost is 1, shortage cost is 10.

2.2 Stochastic Cost Component in Multiple-Period Newsvendor

The considerations in the former sections refer to a single-period problem, but they can be easily extended to a replenishment cycle $R(i,j)$ that covers the period span $i,\ldots,j$. The demand in each period is normally distributed with PDF $g_i(d_j),\ldots,g_j(d_j)$. The cost for the multiple periods' replenishment cycle, when ordering costs are neglected, can be expressed as

$$E\{TC\} = \sum_{k=i}^{j}\left(h\int_{-\infty}^{S}(S-d_{i,k})g_{i,k}(d_{i,k})\mathrm{d}(d_{i,k}) - s\int_{S}^{\infty}(S-d_{i,k})g_{i,k}(d_{i,k})\mathrm{d}(d_{i,k})\right) \tag{13}$$

Since demands are independent and normally distributed in each period, the term $g_{i,j}(d_{i,j})$ (that is the p.d.f. for the overall demand over the period span $\{i,\ldots,j\}$) can be easily computed (see [12]) once the demand in each period $d_i,\ldots,d_j$ are known. It is easy to apply the same rule as before and compute the second derivative of this expression:

$$\frac{\mathrm{d}^2}{\mathrm{d}S^2}E\{TC\} = \sum_{k=i}^{j}(h\cdot g_{i,k}(S) + s\cdot g_{i,k}(S)) \tag{14}$$

which is again a positive function of S, since $g_{i,k}(S)$ are PDFs and both holding and shortage cost are assumed to be positive. The expected cost of a single replenishment cycle therefore remains convex in S regardless of the periods covered. Unfortunately it is not possible to compute the CR as before, using a simple algebraic expression to obtain the optimal S^* which minimizes the expected cost. But since the cost function is convex, it is still possible to compute S^* efficiently. Eq. (12) can be extended in the following way to compute the cost for the replenishment cycle $R(i,j)$ as a function of the opening inventory level S:

$$\sum_{k=i}^{j}\left(hz_{\alpha(i,k)}\sigma_{i,k} + (h+s)\sigma_{i,k}[\phi(z_{\alpha(i,k)}) - (1-\alpha(i,k))z_{\alpha(i,k)}]\right) \tag{15}$$

where $G_{i,k}(S) = \alpha(i,k)$ and $z_{\alpha(i,k)} = \Phi^{-1}(\alpha(i,k))$. Therefore we have $j-i+1$ cost components: the holding and shortage cost at the end of period $i, i+1,\ldots,j$. In Fig. 2 we plot this cost for a particular instance as a function of the opening inventory level S. For each possible replenishment cycle we can efficiently compute the optimal S^* that minimizes such a cost function, using gradient based methods for convex optimization such as Newton's method. Notice that the complete expression for the cost of replenishment cycles that start in period $i \in \{1,...,N\}$ and end in period N is

$$\sum_{k=i}^{N}\left(hz_{\alpha(i,k)}\sigma_{i,k} + (h+s)\sigma_{i,k}[\phi(z_{\alpha(i,k)}) - (1-\alpha(i,k))z_{\alpha(i,k)}]\right) + v\left(S - \sum_{k=i}^{N}d_k\right) \tag{16}$$

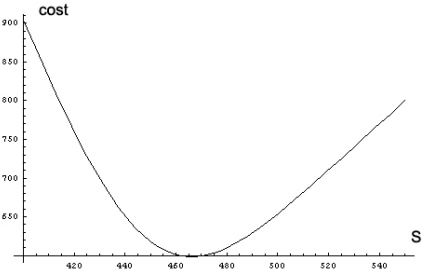


Fig. 2. Three periods holding and shortage cost as a function of the opening inventory level S. The demand is normally distributed in each period with mean respectively 150, 100, 200, the coefficient of variation is 0.1. Holding cost is 1, shortage cost is 10.

In fact for this set of replenishment cycles we must also consider the unit cost component. Once S^* is known, by subtracting the expected demand over the replenishment cycle we obtain the optimal expected buffer stock level $b(i,j)$ required for such a replenishment cycle in order to minimize holding and shortage cost. Notice that every other choice for buffer stock level will produce a higher expected total cost for $R(i,j)$.

An *upper bound* for the value of the opening inventory level in each period $t \in \{1,...,N\}$ can be computed by considering the buffer stock $b(1,N)$ required to optimize the convex cost of a single replenishment cycle $R(1,N)$ that covers the whole planning horizon. Then for each period $t \in \{1,...,N\}$, $\max(S_t) = \sum_t^N \tilde{d}_t + b(1,N)$. A *lower bound* for the value of the expected closing inventory level in each period $t \in \{1,...,N\}$, i.e. opening inventory level minus expected demand, can be computed by considering every possible buffer stock $b(i,j)$ required to optimize the convex cost of a single replenishment cycle $R(i,j)$, independently of the other cycles that are planned. The lower bound will be the minimum value among all these possible buffer values for $j \in \{1,...,N\}$ and $i \in \{1,...,j\}$.

3 Deterministic Equivalent CP Formulation

Building on the considerations above it is easy to construct a *deterministic equivalent* CP formulation for the non-stationary (R_n, S_n) policy under stochastic demand, ordering cost, holding and shortage cost. (For a detailed discussion on deterministic equivalent modeling in stochastic programming see [14]).

In order to correctly compute the expected total cost for a replenishment cycle $R(i,j)$ with opening inventory level S_i, we must build a special-purpose constraint $objConstraint(\cdot)$ that dynamically computes such a cost by means of an extended version of Eq. (15)

$$C(S_i,i,j) = a + \sum_{k=i}^{j} \left(h z_{\alpha(i,k)} \sigma_{i,k} + (h+s)\sigma_{i,k}[\phi(z_{\alpha(i,k)}) - (1-\alpha(i,k)) z_{\alpha(i,k)}] \right) \tag{17}$$

that considers the ordering cost. Then the expected total cost for a certain replenishment plan will be computed as the sum of all the expected total costs for replenishment cycles in the solution, plus the respective ordering costs. $objConstraint(\cdot)$ also computes the optimal expected buffer stock level $b(i,j)$ for every replenishment cycle $R(i,j)$ identified by a partial assignment for $\delta_{k\in\{1,\ldots,N\}}$ variables. A *deterministic equivalent* CP formulation is

$$\min \; E\{TC\} = C \tag{18}$$

subject to

$$\text{objConstraint}\left(C, \tilde{I}_1, \ldots, \tilde{I}_N, \delta_1, \ldots, \delta_N, d_1, \ldots, d_N, a, h, s\right) \tag{19}$$

and for $t = 1 \ldots N$

$$\tilde{I}_t + \tilde{d}_t - \tilde{I}_{t-1} \geq 0 \tag{20}$$

$$\tilde{I}_t + \tilde{d}_t - \tilde{I}_{t-1} > 0 \Rightarrow \delta_t = 1 \tag{21}$$

$$\tilde{I}_t \in \mathbf{Z}, \quad \delta_t \in \{0,1\} \tag{22}$$

Each decision variable $\tilde{I}_t$ represents the expected closing inventory level at the end of period t; bounds for the domains of these variables can be computed as explained above. Each $\tilde{d}_t$ represents the expected value of the demand in a given period t according to its PDF $g_t(d_t)$. The binary decision variables δ_t state whether a replenishment is fixed for period t ($\delta_t = 1$) or not ($\delta_t = 0$).

Eq. (20) enforces a no-buy-back condition, which means that received goods cannot be returned to the supplier. As a consequence of this the expected inventory level at the end of period t must be no less than the expected inventory level at the end of period $t-1$ minus the expected demand in period t. Eq. (21) expresses the replenishment condition. We have a replenishment if the expected inventory level at the end of period t is greater than the expected inventory level at the end of period $t-1$ minus the expected demand in period t. This means that we received some extra goods as a consequence of an order.

The objective function (18) minimizes the expected total cost over the given planning horizon. $objConstraint(\cdot)$ dynamically computes buffer stocks and it assigns to C the expected total cost related to a given assignment for replenishment decisions, depending on the demand distribution in each period and on the given combination for problem parameters a, h, s. In order to propagate this constraint we wait for a partial assignment involving δ_t, $t = 1, \ldots, N$ variables. In particular we look for an assignment where there exists some i s.t. $\delta_i = 1$, some $j > i$ s.t. $\delta_{j+1} = 1$ and for every k, $i < k \leq j$, $\delta_k = 0$. This will uniquely identify a replenishment cycle $R(i,j)$ (Fig. 3). There may be more replenishment cycles associated to a partial assignment. If we consider each $R(i,j)$ identified by the current assignment, it is easy to minimize the convex cost function already discussed, and to find the optimal expected buffer stock $b(i,j)$ for this particular replenishment cycle independently on the others. By doing this for every replenishment cycle identified, two possible situations may arise: the

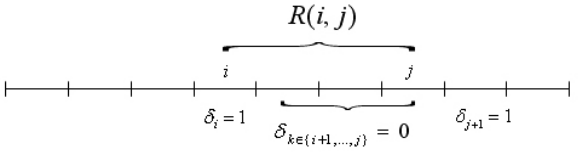


Fig. 3. A replenishment cycle $R(i, j)$ is identified by the current partial assignment for δ_i variables

buffer stock configuration obtained satisfies every inventory conservation constraint (Eq. (20)), or for some couple of subsequent replenishment cycles this constraint is violated (Fig. 4). Therefore we observe an expected negative order quantity. If the latter situation arises we can adopt a fast convex optimization

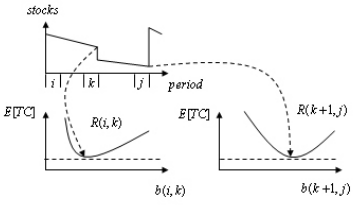


Fig. 4. The expected total cost of both replenishment cycles is minimized, but the inventory conservation constraint is violated between $R(i, k)$ and $R(k + 1, j)$

procedure to compute a feasible buffer stock configuration with minimum cost. The key idea is to identify two possible limit situations: we increase the opening inventory level of the second cycle, thus incurring a higher overall cost for it, to preserve optimality of the first cycle (Fig. 5 - a). Or we decrease the buffer stock of the first replenishment cycle, thus incurring a higher overall cost for it, to preserve optimality of the second cycle cost (Fig. 5 - b). A key observation is that, when negative order quantity scenarios arise, at optimality the closing inventory levels of the first and the second cycle lie in the interval delimited by the two situations described. This directly follows from the convexity of both the cost functions. Moreover the closing inventory level of the first cycle must

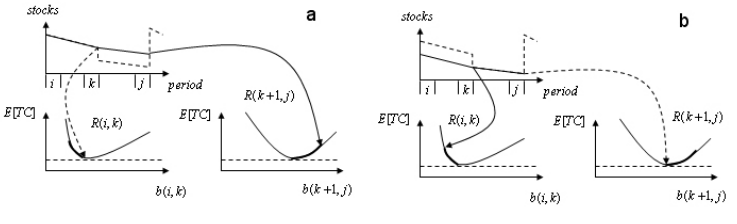


Fig. 5. Feasible limit situations when negative order quantity scenarios arise

be equal to the opening inventory level of the second cycle. In fact, if this does not hold, then either the first cycle has a closing inventory level higher than the opening inventory level of the second cycle and the solution is not feasible (Fig. 6 - a), or the first cycle has a closing inventory level smaller than the opening inventory level of the second cycle. In the latter case we can obviously decrease the overall cost by choosing a smaller opening inventory level for the second cycle (Fig. 6 - b). The algorithm for computing optimal buffer stock

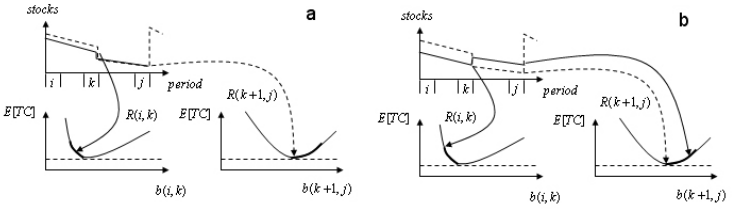


Fig. 6. Infeasible (a) and suboptimal (b) plans realized when the opening inventory level of the second cycle doesn't equate the closing inventory level of the first cycle

configurations in presence of negative order quantity scenarios simply exploits the linear dependency between opening inventory level of the second cycle and closing inventory level of the first cycle. Due to this dependency the overall cost is still convex in $b(i,k)$ (or equivalently in $b(k+1,j)$, since they are linearly dependent) and we can apply any convex optimization technique to find the optimal buffer stock configuration. Notice that this reasoning still holds in a recursive process. Therefore we can optimize buffer stock for two subsequent replenishment cycles, then we can treat these as a new single replenishment cycle, since their buffer stocks are linearly dependent, and repeat the process in order to consider the next replenishment cycle if a negative order quantity scenario arises.

Once buffer stocks are known we can apply Eq. (17) to the opening inventory level $S_i = \tilde{d}_i + \ldots + \tilde{d}_j + b(i,j)$ and compute the cost $C(S_i,i,j)$ associated to a given replenishment cycle. Since the cost function in Eq. (17) is convex and we handle negative order quantity scenarios, a lower bound for the expected total cost associated to the current partial assignment for δ_t, $t = 1,\ldots,N$ variables is now given by the sum of all the cost components $C(S_i,i,j)$, for each replenishment cycle $R(i,j)$ identified by the assignment. Furthermore this bound is tight if all the δ_t variables have been assigned. $objConstraint(\cdot)$ exploits this property in order to incrementally compute a lower bound for the cost of the current partial assignment for δ_t variables. When every δ_t variable is ground, since such a lower bound becomes tight, buffer stocks computed for each replenishment cycle identified can be assigned to the respective I_t variables. Finally, in order to consider the unit variable cost v we must add the term $v \cdot I_N$ to the cycle cost $C(S_i,i,N)$ for $i \in \{1,...,N\}$. Therefore the complete expression for the cost of replenishment cycles that start in period $i \in \{1,...,N\}$ and end in period N is:

Table 1. Expected demand values

Period	1	2	3	4	5	6	7	8
$\tilde{d}_t$	200	100	70	200	300	120	50	100

$$C(S_i, i, N) = a + \sum_{k=i}^{N} \left(h z_{\alpha(i,k)} \sigma_{i,k} + (h+s)\sigma_{i,k}[\phi(z_{\alpha(i,k)}) - (1-\alpha(i,k)) z_{\alpha(i,k)}] \right) + v\left(S_i - \sum_{k=i}^{N} d_k \right) \tag{23}$$

4 Comparison of the CP and MIP Approaches

In [4] Tarim and Kingsman proposed a piecewise linear approximation of the cost function for the single-period newsvendor type model under holding and shortage costs, which we analyzed above. Thus they were able to build a MIP model approximating an optimal solution for the multi-period stochastic lot-sizing under fixed ordering, holding and shortage costs. They gave a few examples to show the effect of higher noise levels (uncertainty in the demand forecasts) on the order schedule. Using the same examples we shall compare the policies obtained using our exact CP approach with their approximation. Depending on the number of segments used in the piecewise approximation, the quality of the solutions obtained can be improved. We shall consider approximations with two and seven segments. The forecast of demand in each period are given in Table 1. We assume that the demand in each period is normally distributed about the forecast value with the same coefficient of variation τ. Thus the standard deviation of demand in period t is $\sigma_t = \tau \cdot \tilde{d}_t$. In all cases, initial inventory levels, delivery lead-times and salvage values are set to zero.

In Fig. 7-11 optimal replenishment policies obtained with our CP approach are compared for four different instances, with respect to τ, v, a and s, with the policies provided by the 2-segment (PW-2) and 7-segment (PW-7) approximations. For each instance we compare the expected total cost provided by the exact method with the expected total cost provided by the policies found using approximate MIP models. Since the cost provided by PW-2 and PW-7 is an approximation, it often differs significantly from the real expected total cost related to policy parameters found by these models. It is therefore not meaningful to compare the cost provided by the MIP model with that of the optimal policy obtained with our CP model. To obtain a meaningful comparison we computed the real expected total cost by applying the exact cost function (Eqs. 17, 23) discussed above to the (R^n,S^n) policy parameters obtained through PW-2 and PW-7. It is then possible to assess the accuracy of approximations in [4]. Fig. 7 shows the optimal replenishment policy for the deterministic case ($\tau = 0.0$). The direct item cost (v) is taken as zero. Four replenishment cycles are planned.

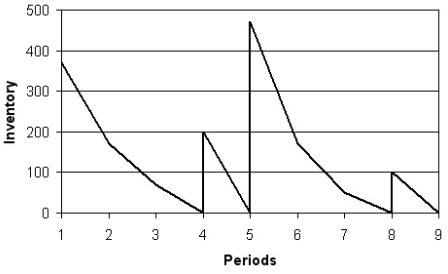


Fig. 7. $h = 1$, $a = 250$, $s = 10$, $v = 0$, $\tau = 0.0$

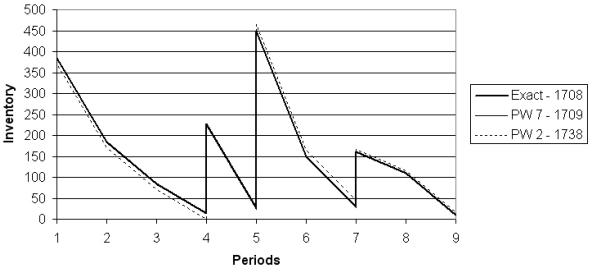


Fig. 8. $h = 1$, $a = 250$, $s = 10$, $v = 0$, $\tau = 0.1$

The (R^n,S^n) policy parameters are $R = [3, 1, 3, 1]$ and $S = [370, 200, 470, 100]$. The total cost for this policy is 1460. Fig. 8 shows an instance where we consider low levels of forecast uncertainty ($\tau = 0.1$). In this case both PW-2 and PW-7 perform well compared to our exact CP solutions. Since forecast uncertainty must be considered, all the models introduce buffer stocks. The optimal (R^n,S^n) policy parameters found by our CP approach are $R = [3, 1, 2, 2]$ and $S = [384, 227, 449, 160]$. The PW-2 solution is 1.75% more costly than the exact solution, while the PW-7 solution is slightly more costly than the exact solution.

Fig. 9 shows that as the level of forecast uncertainty increases ($\tau = 0.2$), the quality of the PW-2 solution deteriorates, in fact it is now 3.62% more costly than the exact solution. The optimal (R^n,S^n) policy parameters found by our CP approach are $R = [3, 1, 2, 2]$ and $S = [401, 253, 479, 170]$. In contrast the PW-7 solution is still only slightly more costly than the exact solution. As noted in [4] the quality of the approximation decreases for high ratios s/h. In Fig. 10 we consider $s/h = 50$ and a different demand pattern. The forecast of demand in each period are given in Table 2. Now the PW-2 solution is 6.66% more costly than the exact approach, while the PW-7 solution is 1.03% more costly. The optimal (R^n,S^n) policy parameters found by our CP approach are $R = [3, 1, 2, 1, 1]$ and $S = [483, 324, 592, 324, 486]$. In Fig. 11 we consider the same instance but a direct item cost is now incurred ($v = 15$). The buffer stock held in the last replenishment cycle is affected by this parameter, and is decreased

Table 2. Expected demand values

Period	1	2	3	4	5	6	7	8
d_t	200	100	70	200	300	120	200	300

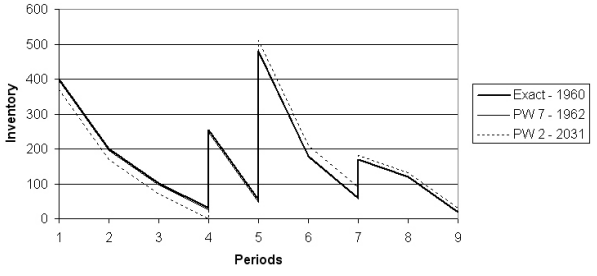


Fig. 9. $h = 1$, $a = 250$, $s = 10$, $v = 0$, $\tau = 0.2$

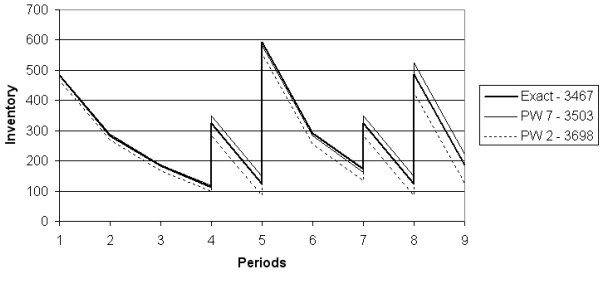


Fig. 10. $h = 1$, $a = 350$, $s = 50$, $v = 0$, $\tau = 0.3$

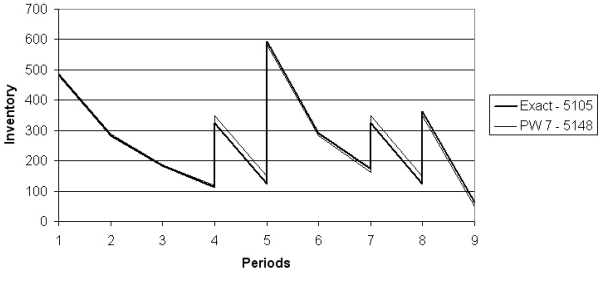


Fig. 11. $h = 1$, $a = 350$, $s = 50$, $v = 15$, $\tau = 0.3$

from 186 to 63. The PW-7 policy is now 0.84% more costly than the exact one. For these instances seven segments usually provides a solution with a cost reasonably close to optimal. In terms of running times, for all these instances both the MIP approximations and the CP model perform very quickly. In our experiments we used ILOG OPL Studio 3.7 to solve the MIP models of [4], and Choco [16] (an open source solver written in Java) to implement our CP model. All experiments were performed on an Intel Centrino 1.5 GHz with 500Mb

RAM. Since the planning horizon is short (8 periods), we were able to solve any instance in less than a second. As the planning horizon length increases the pure CP model becomes slower than the MIP one. This is due both to the size of decision variable domains and to the lack of good bounds in the search. We do not discuss efficiency issues in this paper, but we emphasise that a significant reduction in decision variable domain sizes can be achieved in a way similar to the one discussed in [2]. Furthermore it is possible to incorporate in our CP model dedicated cost-based filtering methods [15] based on a *dynamic programming relaxation* [5] that is able to generate good bounds during the search. Such a technique has been already employed under a service level constraint [1] and preliminary results in this direction under a penalty cost suggest that our exact CP model, when enhanced with these dedicated filtering techniques, is able to produce an optimal solution for instances up to 50 periods and more in a few seconds.

5 Conclusions

We presented a CP approach that finds optimal (R_n,S_n) policies under non-stationary demands. Using our approach it is now possible to evaluate the quality of a previously published MIP-based approximation method, which is typically faster than the pure CP approach. Using a set of problem instances we showed that a piecewise approximation with seven segments usually provides good quality solutions, while using only two segments can yield solutions that differ significantly from the optimal. In future work we will aim to develop domain reduction techniques and cost-based filtering methods to enhance the performance of our exact CP approach.

Acknowledgements. this work was supported by Science Foundation Ireland under Grant No. 03/CE3/I405 as part of the Centre for Telecommunications Value-Chain-Driven Research (CTVR) and Grant No. 00/PI.1/C075.

References

1. S. A. Tarim, B. Hnich, R. Rossi, S. Prestwich. Cost-Based Filtering for Stochastic Inventory Control. *Lecture Notes in Computer Science*, Springer-Verlag, 2007, to appear.
2. S. A. Tarim, B. Smith. Constraint Programming for Computing Non-Stationary (R,S) Inventory Policies. *European Journal of Operational Research*. to appear.
3. S. A. Tarim, B. G. Kingsman. The Stochastic Dynamic Production/Inventory Lot-Sizing Problem With Service-Level Constraints. *International Journal of Production Economics* 88:105–119, 2004.
4. S. A. Tarim, B. G. Kingsman. Modelling and Computing (R^n,S^n) Policies for Inventory Systems with Non-Stationary Stochastic Demand. *European Journal of Operational Research* 174:581–599, 2006.
5. S. A. Tarim. Dynamic Lotsizing Models for Stochastic Demand in Single and Multi-Echelon Inventory Systems. PhD Thesis, Lancaster University, 1996.

6. J. H. Bookbinder, J. Y. Tan. Strategies for the Probabilistic Lot-Sizing Problem With Service-Level Constraints. *Management Science* 34:1096–1108, 1988.
7. H. M. Wagner, T. M. Whitin. Dynamic Version of the Economic Lot Size Model. *Management Science* 5:89–96, 1958.
8. E. A. Silver, D. F. Pyke, R. Peterson. Inventory Management and Production Planning and Scheduling. John Wiley and Sons, New York, 1998.
9. E. L. Porteus. Foundations of Stochastic Inventory Theory. Stanford University Press, Stanford, CA, 2002.
10. K. Apt. Principles of Constraint Programming. Cambridge University Press, Cambridge, UK, 2003.
11. A. Charnes, W. W. Cooper. Chance-Constrainted Programming. *Management Science* 6(1):73–79, 1959.
12. L. Fortuin. Five Popular Probability Density Functions: a Comparison in the Field of Stock-Control Models. *Journal of the Operational Research Society* 31(10):937–942, 1980.
13. I. J. Lustig, J.-F. Puget. Program Does Not Equal Program: Constraint Programming and its Relationship to Mathematical Programming. *Interfaces* 31:29–53, 2001.
14. J. R. Birge, F. Louveaux. Introduction to Stochastic Programming. Springer Verlag, New York , 1997.
15. F. Focacci, A. Lodi, M. Milano. Cost-Based Domain Filtering. Fifth International Conference on the Principles and Practice of Constraint Programming, *Lecture Notes in Computer Science* 1713, Springer Verlag, 1999, pp. 189–203.
16. F. Laburthe and the OCRE project team. Choco: Implementing a CP Kernel. Bouygues e-Lab, France.
17. G. Hadley, T. M. Whitin. Analysis of Inventory Systems. Prentice Hall, 1964.

Preprocessing Expression-Based Constraint Satisfaction Problems for Stochastic Local Search

Sivan Sabato and Yehuda Naveh

IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel
{sivans,naveh}@il.ibm.com

Abstract. This work presents methods for processing a constraint satisfaction problem (CSP) formulated by an expression-based language, before the CSP is presented to a stochastic local search solver. The architecture we use to implement the methods allows the extension of the expression language by user-defined operators, while still benefiting from the processing methods. Results from various domains, including industrial processor verification problems, show the strength of the methods. As one of our test cases, we introduce the concept of random-expression CSPs as a new form of random CSPs. We believe this form emulates many real-world CSPs more closely than other forms of random CSPs. We also observe a satisfiability phase transition in this type of problem ensemble.

1 Introduction

The most important aspect of constraint programming (CP) over other variable-assignment paradigms (e.g., Satisfiability or integer linear programming) is its ease of modeling. CP allows users to describe the problem at hand in a way that is close to the problem's domain, as opposed to a formal language derived from the solution scheme in other methods. One generic way to allow this natural modeling is to define an expression-based language with simple operators that have a wide range of semantics. Operators may be arithmetic (e.g., +,-), logical (e.g., `and`, `or`), set operators (e.g., `member-of`), or problem-domain specific. They may be binary (e.g., `equal-to`, `less-than`) or global (e.g., `all-different`, `equal-sum`). One example of a generic expression language is OPL [1], supported by ILOG.

Some classes of constraint satisfaction problems (CSPs) are recognized as not easily solved by systematic methods, and stochastic local search (SLS) methods need to be called upon [2]. In this paper we show that the ease with which SLS methods can solve a CSP model written in an expression language highly depends on the way the model is processed and analyzed before it is presented to the SLS solver. We present generic processing algorithms that transform the input CSP model into an SLS model that is easier to solve in many cases. As in [3], the processing algorithms use interfaces (or abstract methods) to support

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 244–259, 2007.

the addition of any user-defined operator. However, while previous works focus on abstracting methods related to the search phase [4,3], we are interested in processing the CSP model itself, *before* entering the search phase.

One of the main tasks in developing an SLS solver involves enhancing its ability to escape local minima in the topography defined by the cost function of all complete assignments. Therefore, SLS solvers (e.g., Walksat [5] or COMET [3]), can incorporate many types of meta-heuristics, such as simulated annealing [6], min-conflicts [7], Tabu Search [8], or variable-neighborhood search [9], designed to escape local minima. Still, for all these methods, it is highly beneficial to have the CSP mapped into a topography with fewer local minima and plateaus.

The abstract interfaces and concrete methods we present aim to achieve this goal in a generic way. More specifically, real world CSPs are sometimes composed of differently structured constraints, written independently from each other. The resulting problem of 'bad models' is known [10], and processing algorithms aimed at improving the models exist for non-SLS search schemes (e.g., in Satisfiability [10] and arc-consistency methods [11]). Here we extend these methods to SLS.

This paper is outlined as follows. Section 2 defines an expression language we use to model the input CSP. Section 3 describes the architecture of the SLS solver and its use of operators in the expression language. Section 4, which forms the main part of the paper, refines this architecture to support the processing methods and presents the processing algorithms. Experimental results are shown in Section 5, where the concept of random expression CSP is also presented.

2 Expression-Based CSPs

The formal grammar of the example language we use as input throughout the paper is listed in Figure 1. This language captures many of the basic constructs expected from a general-purpose expression-based CSP language. While additional generic or domain-specific operators may be added, this language is powerful enough to easily model many of the problems in CSPLib (`http://www.csplib.org/`), as well as a large number of real world problems.

In Figure 1, words in upper case stand for non-terminals and underlined words are reserved. The entire CSP is generated from the non-terminal P. A CSP description is comprised of a list of declarations of integer variables and their domains, and a list of constraints created from logical, arithmetic, and other operators. Domains of variables are defined as ranges of integers. An example for the use of the input language is given in Figure 2. We will sometimes formulate CSP descriptions more loosely for ease of presentation, when it is obvious how one would translate them to a valid description using our grammar.

3 System Architecture

3.1 Internal CSP Representation

An expression-based CSP is represented by a rooted tree as exemplified in Figure 3. The leaves of the tree are mapped to variables or constants and the

P ⟶ VARDECL constraints CONSDECL
VARDECL ⟶ VD ; VARDECL | ϵ
CONSDECL ⟶ CONS; CONSDECL | ϵ
VD ⟶ VAR RANGES
RANGES ⟶ RANGES, [NUM, NUM]
RANGES ⟶ [NUM, NUM]
CONS ⟶ (CONS) OP (CONS)
CONS ⟶ `not` (CONS)
CONS ⟶ ((EXP) COMPARE (EXP))
CONS ⟶ `all-diff`(VARLIST)
CONS ⟶ `some-equal`(VARLIST)
EXP ⟶ ((EXP) EXP_OP (EXP))
EXP ⟶ NUM
EXP ⟶ VAR
VARLIST ⟶ VAR, VARLIST | ϵ
OP ⟶ `and` | `or` | `implies` | `iff`
COMPARE ⟶ = | $\neq$ | $\geq$ | $\leq$ | < | >
EXP_OP ⟶ + | − | × | /
NUM ⟶ [0-9]+
VAR ⟶ [A-Za-z_][A-Za-z0-9_]+

Fig. 1. Input language grammar

Bread [0,2], [4,5];
Milk [0,2];
Cheese [0,2];
Butter [0,2];
Apple [0,3], [6,6];
Payment [0,1000];
constraints
(Bread = Cheese + Butter);
(Apple < 2) `implies` (Bread > 1);
(Cheese + Butter < 3);
((Apple = 1) `or` (Bread = 0));
(`all-diff`(Milk, Cheese, Butter));
(Payment = Milk×3 + Cheese×4);

Fig. 2. Input language example

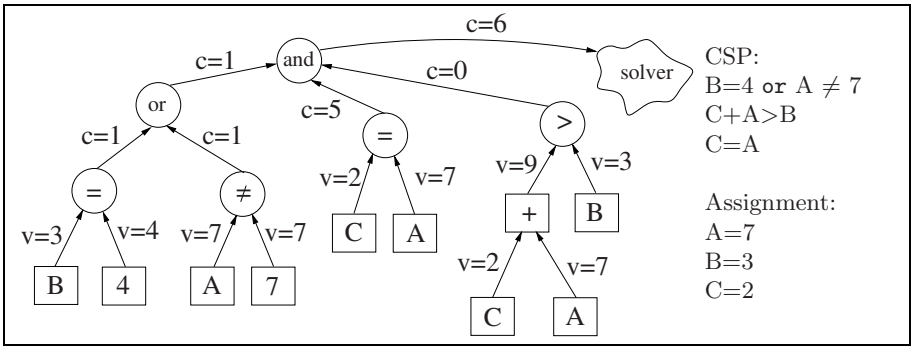


Fig. 3. A CSP tree and the flow of information when calculating an assignment's cost, where 'c' stands for constraint cost (or penalty) and 'v' stands for expression value. Natural cost functions were chosen for the various operators.

nodes to operators. Nodes that are mapped to arithmetic operators represent *expressions*, while nodes that are mapped to comparison operators or to logical operators represent *sub-constraints*. The root node is mapped to an *and* operator and its children are the individual constraints in the input CSP.

A sub-constraint node is defined by its operator and by its children. We refer to sub-constraints with comparison operators as *atomic*, since their children are expressions and not other sub-constraints. We refer to sub-constraints with

logical operators as *compound*. Global constraints such as `all-different` can be implemented either as atomic operators or as compound expressions.

Nodes are implemented as objects that are sub-classed from an abstract sub-constraint object or from an abstract expression object. Sub-constraint objects implement (among other) the usual interface:

`CalculateCost(Assignment)`

which returns the cost that the sub-constraint assigns to `Assignment`. Expression objects implement the interface:

`CalculateValue(Assignment)`

This interface returns the value of the expression in `Assignment`. The solver only has access to the interfaces of the root of the constraint tree. The computational cost of calculating the cost of the CSP tree for any single assignment is linear in the number of nodes in the tree.[1]

The cost function implemented for any sub-constraint is zero if and only if the sub-constraint is satisfied; otherwise it is positive. In addition, this cost function should exhibit the best possible fitness-distance correlation (FDC) [12]: The further the state is from a solution, the higher the cost of the state should be. There are no conceptual problems with the definition of a cost function for atomic sub-constraints, whether simple expressions or global constraints [13]. However, implementing the cost of a compound sub-constraint is not as straightforward, since it should depend only on the cost of its child sub-constraints. This is because the same sub-constraint (and hence the same implementation of its cost function) may span very different sub-trees of sub-constraints and variables. Some operators may benefit from the fact that there exists a cost function whose topography is related to the topographies of the child constraints. This is the case for `and` and `or` operators with the natural costs of `sum` and `min` of children, respectively. However, for other operators, it may be impossible to implement a cost function that reflects the topographies of the child constraints. Notable examples are `not`, `implies` and `iff`. We address this problem in detail in Section 4.1.

3.2 The Search Scheme

A typical scheme of a greedy SLS algorithm is outlined in Algorithm 1. The algorithm starts with an initial assignment. In each iteration, it generates a set of steps from the current assignment to a set of new assignments, and calculates the cost of each of the resulting assignments. If at least one step results in an assignment with a lower cost, this assignment becomes the current one. Otherwise, the topography of the problem is modified by giving a larger weight to constraints that are not satisfied by the current assignment. The algorithm

[1] Powerful heuristics that exploit the fact that each step in the search space usually changes the cost of only a few of the CSP tree nodes are also applied, but are similar to those reported elsewhere [3].

Algorithm 1. General Scheme of Search Algorithm

```
Initialize A to an initial complete assignment
repeat
  repeat
    if cost(A) = 0 then
      Return
    end if
    Initialize S to a set of possible steps
    Calculate cost(A + s) for all s ∈ S
    s_1 ← argmin_{s∈S} cost(A + s)
    if cost(A + s_1) < cost(A) then
      A ← A + s_1
    end if
  until cost(A + s_1) ≥ cost(A)
  Set constraint weights such that unsatisfied constraints get a larger weight
until Timeout is reached
```

stops when a zero-cost assignment is reached or at timeout. One of the main differentiators between solvers that use this scheme is the neighborhood function that generates S.

3.3 The Search Space

In our implementation, we follow Algorithm 1 and define the neighborhood of an assignment to be the assignments in which up to M bits are changed in the 2's complement bit-representation of the integer variables of the CSP, where M is given, and calculated by dynamic heuristics [14]. In this representation, an additional unary constraint is added for each variable whose domain is not a power of two, to enforce the domain requirement. This simple approach is favorable for some constraint types (e.g., `less-than` and `greater-than`). It is also particularly useful in hardware verification where many constraints are defined on bit-ranges [15]. However, most methods presented below can be generalized to other representation schemes and more sophisticated neighborhood functions.

4 Model Processing for SLS

In this section we present several methods for processing the input CSP model. The processing methods rely on implementing specific interfaces for sub-constraints and expressions. Unlike the interfaces `CalculateCost()` and `CalculateValue()` that define the *semantics* of the objects, the interfaces presented below are used only by the processing algorithms and do not change the semantics of the CSP. Hence, it is not necessary to implement all interfaces for all operators in the language.

We demonstrate our methods on the grammar of Figure 1, but the methods can be applied to grammars that use other operators by implementing the required abstract interfaces for each operator. The modeler of a CSP may thus

experiment with different types of newly-defined operators, without changing the processing or search algorithms. This extends the regular generic interface of the search phase to the pre-search phase. To demonstrate the operation of the processing methods, we present the following simple CSP example.

V1,V2,V3,V4,V5,V6 [0,5000];
constraints
1. (((V3 ≠ V5+3) `or` (V5 > 10)) `implies` (V2 ≠ V3));
2. ((V4 > 11) `and` (V2 = V3));
3. ((V1 < 5) `or` ((V1 < 12) `and` (V1 > V3−4)));

This problem exemplifies a mixture of logical and arithmetic operators and a diverse structure of constraints often found in real world problems. We define the variable domains in this example to be relatively large, since the problem is small for didactic reasons and we want to keep the search space large. In the 2's complement representation, each variable is represented by 13 bits and constraints of the form $V_n \leq 5000$ are added.

4.1 Transformation to Negation Normal Form

In Section 3.1, we mentioned `sum` and `min` as reasonable cost functions for the Boolean operators `and` and `or`. We now show that other Boolean operators may present an inherent problem to the tree structure of the cost function.

Consider the unary operator `not`. Let us look at a sub-constraint node $C = not(C')$. We need to implement a cost function f_{not} such that on any complete assignment A, $cost(C, A) = f_{not}(cost(C', A))$. For f_{not} to be a legal cost function, it must output zero if C' is not satisfied by A (i.e., if $cost(C', A) > 0$) and non-zero if C is satisfied (i.e., if $cost(C', A) = 0$). The only functions that obey these limitations are of the following form, for some $k > 0$:

$$f_{not}(c) = \begin{cases} k & \text{for} \quad c = 0 \\ 0 & \text{otherwise} \end{cases}$$

This implies that f_{not} has zero gradient when it is unsatisfied, leaving no possibility of finding a satisfying assignment using gradient descent ('greedy') methods. In other words, the `not` operator 'hides' information on the location of minima in its child sub-constraint cost function. A similar problem is encountered with the logical operators `implies` and `iff`. All these operators are, however, a basic part of any natural expression language. Our first processing method therefore transforms the model to negation normal form (NNF), which substitutes the ill-behaved operators with the better-behaved `and` and `or`.

The *NNF transformation* is applied in the regular manner to compound sub-constraints realizing the above operators. Atomic sub-constraint operators need to implement the following interface in order to take advantage of this method:

`GetNegatedOperator`().

For example, the implementation of the $=$ operator would return the operator $\neq$, the operator $>$ would return the operator $\leq$, and the global operator `all-different` would return its negation `some-equal`. The transformation is applied to the input CSP recursively from top to bottom. Its time-complexity is linear in the size of the CSP tree and the resulting tree is about the same size as the original one. In our example CSP, the NNF transformation changes constraint No. 1 to: $(((\text{V3} = \text{V5} + 3)$ `and` $(\text{V5} \leq 10))$ `or` $(\text{V2} \neq \text{V3}))$;

4.2 Reducing the Search Space Size

Two processing methods presented here perform low-cost inferences that enable the pruning of large parts of the search space for which search is useless. These inferences are special and simple cases of domain reductions that could have also been achieved using propagators in an arc-consistency algorithm. While there are many ways to combine arc-consistency with local search (see [16] for an early example), the overall search may be prohibitive in problems that are not suitable for arc-consistency methods. In contrast, here we limit our processing methods to ones whose processing cost is linear in the size of the CSP tree, and we apply the methods only on the initial CSP model before starting SLS. Hence, our search is dominated by SLS and the inference cost is usually negligible.

The two methods presented in this section rely on the dimensions defining the search space. In a bit-representation (sub-section 3.3), each dimension is defined by a single bit. In other representations, each dimension may correspond to a single CSP variable or to any combination of variables' values.

If the cost function does not depend on the dimension's value in any assignment, the solver does not need to change this value during search. We term such a dimension *unimportant*. For example, in the bit-representation, the least-significant-bit of a variable X in the constraint "X $>$ 5" is unimportant. Alternatively, if we can infer in advance that the value in a given dimension is the same for all solutions, the solver can set the value to this fixed value and remove the dimension from the search space. We refer to such a dimension as *predetermined*. An example of a predetermined dimension in the bit-representation is the least-significant-bit of a variable V constrained by "V `mod` 2 $= 0$".

Finding Unimportant Dimensions. We find unimportant dimensions by having sub-constraint- and expression-objects implement the interface:

`GetDependentDimensions`()

which returns the list of dimensions that may affect the object's cost or value. Dimensions that do not appear in the list returned by the root node are unimportant. Note that finding all unimportant dimensions in a general CSP is NP-hard:

If the CSP includes one constraint that is a 3-CNF formula, deciding whether there are any important dimensions is tantamount to finding whether the formula is satisfiable.

In our CSP example, the variable V6 is not used by any constraint; Hence, all its bits are found to be unimportant. The same applies to the two least-significant-bits of V4. Additional unimportant bits of this CSP will be found after other processing methods are applied.

Finding Predetermined Dimensions. We find predetermined dimensions (PDs) using a recursive and iterative algorithm: Each sub-constraint node implements the interface

`InferPredeterminedDimensions(CurrentPredeterminedDimensions)`

which returns a set of PDs along with their predetermined value. For atomic sub-constraints, the interface uses the currently known PDs and tries to find new PDs according to its own semantics. For example, in a bit-representation, in the atomic constraint "X < 5", the bits higher than the 3 least-significant-bits in X are zero.

For a compound sub-constraint, the interface calls `InferPredetermined-Dimensions(CurrentPredeterminedDimensions)` for each of its child sub-constraints, and decides on the actual PDs according to its own semantics. For example, an `and` sub-constraint returns the union of the results of the child constraints, while an `or` sub-constraint returns the intersection of the results of the child constraints. Since new PDs are decided according to current ones, the process is iterative and stops when no more PDs are found. In our CSP example, we infer from the third constraint that the nine most-significant-bits of V1 are predetermined to be 0.

4.3 Dealiasing

The Dealiasing processing method finds and enforces *aliases*. An alias is a pair $(V, f(\mathcal{V}))$ of a variable and a function of other CSP variables, such that $V = f(\mathcal{V})$ in any solution to the CSP. Dealiasing limits the search space to assignments that satisfy the aliases.

V1,V2 $[0,M]$;
constraints
1. (V1 $= M$) `or` (V1 $= 0$);
2. (V2 $= M$) `or` (V2 $= 0$);
3. (V1 = V2)

An alias can be inferred from a sub-constraint node of the form "V = EXP", where V is a variable and EXP an expression,[2] but only if the sub-constraint's path to the root node is composed only of `and` (or equivalent) operators. After collecting all the aliases that can be identified in the CSP, all the references to the aliased variables are replaced by references to the corresponding functions. Before describing the Dealiasing algorithm, let

[2] An alias can also be inferred from a sub-constraint if it can be transformed to a functional form. For example V1 + V4 = 7 can be transformed to V1 = 7−V4.

us illustrate the criticality of aliasing for SLS[3]. Consider the simple CSP in the above box, for some positive number M. A natural cost derived from the `and` operator at the root of the CSP, and `or` operators of constraints 1 and 2 is:

$$\text{Cost} = \min\left(||M - \text{V1}||, ||\text{V1} - 0||\right) + \min\left(||M - \text{V2}||, ||\text{V2} - 0||\right) + ||\text{V1} - \text{V2}||$$

where $||A-B||$ is the distance between A and B according to some defined metric. For any choice of a reasonable linear metric (e.g., absolute-value of difference, or Hamming distance) this cost induces huge plateaus in the search space. For example, all states for which V1 is closer to M than to 0, while V2 is closer to 0 are plateau states. This renders the problem, as formulated, hard for SLS.

After Dealiasing, the CSP contains two copies of the second constraint and the cost becomes $\text{Cost} = 2\min\left(||M - \text{V2}||, ||\text{V2} - 0||\right)$. This cost has two global minima, no local minima, and no plateaus.

The Dealiasing algorithm uses two sub-constraint node interfaces:

`GetAliases`()

`ApplyAliases`()

`GetAliases()` returns a list of all the aliases found in the sub-constraint: For example, an `and` sub-constraint returns the union of the lists returned by its children, while an `or` sub-constraint returns no aliases. `ApplyAliases()` replaces all occurrences of the aliased variable with a reference to the function to which the variable is aliased (possibly turning the CSP tree into a DAG).

The Dealiasing processing method is implemented by calling `GetAliases()` for the root node of the CSP tree to get a set of aliases A, finding a consistent subset of aliases $A1 \subseteq A$ (in order to avoid cyclic definitions between the aliases), and calling `ApplyAliases()` for the root node with $A1$[4]. In our CSP example we now infer that V2 is aliased to V3 from the second constraint. We replace all occurrences of V2 by V3 accordingly. The reformulated problem is now:

1. ((V3 = V5+3) `and` (V5 ≤ 10)) or (V3 ≠ V3);
2. ((V4 > 11) `and` (V3 = V3));
3. ((V1 < 5) or ((V1 < 12) `and` (V1 > V3−4)));

4.4 Pruning – Removing Tautologies and Contradictions

In the *prune* processing method, we recursively remove sub-constraints that are identified as tautological or contradictory. Tautological sub-constraints are ones that would be satisfied in any assignment consistent with known predetermined

[3] In contrast, Dealiasing hardly helps reach a solution in MAC-based algorithms because an aliased constraint of the form $V = f(\mathcal{V})$ will just propagate from $\mathcal{V}$ to V.

[4] Finding a maximal set $A1$ is equivalent to the Directed Feedback Edge Set problem, which is NP-complete [17]. We therefore implement a heuristic algorithm that does not guarantee global maximality.

dimensions. Contradictory sub-constraints would be unsatisfied by any such assignment. We call both contradictory and tautological sub-constraints *redundant* sub-constraints. Removing redundant sub-constraints serves three purposes:

1. The cost function of the pruned CSP exhibits better FDC. For example, suppose that in a sub-constraint of the form "C1 `or` C2", C1 is contradictory. Then the natural cost function $\min(\text{Cost}(C1), \text{Cost}(C2))$ may exhibit a local minimum where Cost(C1) is minimized. Replacing "C1 or C2" by the equivalent "C2" immediately prevents this problem.
2. Creating more opportunities for inferences by other processing methods. In Section 4.5, we exemplify this effect on our CSP example.
3. Reducing the computational toll of calculating the cost function.

Though, in general, it has been shown that removing redundant constraints does not necessarily improve gradient solutions [18], in our experiments this has proved to be a vital step in complex expression-based problems. We attribute this to the combination of the three items listed above. These items may be less relevant to simple and well-structured CSPs. (Item 2 is only relevant to solvers applying our other processing methods.)

The following interface is implemented for any sub-constraint node type:

`Prune`()

To run the pruning method, `Prune()` is called for the CSP tree root node. `Prune()` for an atomic sub-constraint may identify two kinds of redundant sub-constraints. First, it may identify patterns syntactically recognized as redundant, for instance "A > A", "A = A". (These patterns may exist in real-world CSPs that were generated automatically.) Second, it may identify constraints that are redundant due to constants and predetermined dimensions. The `Prune()` method for a compound sub-constraint may call `Prune()` for each of its child constraints and operate according to its own semantics. For example, the sub-constraint `and` would remove a tautological child node and would report itself as contradictory if one of its child nodes is contradictory. In our CSP example, the pruning method finds a contradiction and a tautology, resulting in:

1. (V3 = V5+3) `and` (V5 ≤ 10);
2. (V4 > 11);
3. (V1 < 5) or ((V1 < 12) `and` (V1 > V3−4)));

4.5 Combining Processing Methods

We apply an iterative algorithm to make full use of the interaction between the processing methods, stopping when no more changes occur.

Transform to NNF
repeat
Find Predetermined Dimensions
Apply Dealiasing
Prune
until No changes have occurred in the last iteration
Find Unimportant Dimensions

This algorithm runs for two iterations on our CSP example. The first iteration results in the form given in Section 4.4. The second iteration then finds more predetermined dimensions in V5 and results in the alias (V3,V5+3). This allows another round of successful pruning. The final CSP is:

1. (V5 $\leq$ 10)
2. (V4 $>$ 11);
3. (V1 $<$ 5) or ((V1 $<$ 12) `and` (V1 $>$ (V5+3)$-$4)));
The 9 most-significant-bits of V1 and of V5 are predetermined to be 0.

This formulation is invariant to all processing methods. The unimportant dimensions are now inferred to be all bits of variables V2, V3 and V6, and the two unimportant bits of V4 found earlier.

5 Experimental Results

Experimental results were obtained using a tool called Stocs, which implements Simulated Variable Range Hopping (SVRH) [14]. Run-times reported include both the preprocessing and search phases, though the latter dominates in all cases not solved by preprocessing alone. The experiments were run on a single-core Intel (TM) 3GHz PC running Red-Hat Linux.

5.1 Artificial Example

Results for the CSP example are presented in the following table for several configurations of the processing methods. In the first column, results of using all processing methods are shown. In each of the other columns, one of the methods was disabled. Each scenario was run 10 times, with different starting states and random seeds. The timeout was 10 seconds.

	All Methods	No Predetermined Dimensions	No NNF transformation	No Dealiasing
Solved	**10**	**10**	0	9
Avg. Time (sec)	**0.10**	0.26	N/A	2.08
Min. Time (sec)	**0.08**	0.10	N/A	0.33
Max. Time (sec)	**0.12**	0.43	N/A	5.97

5.2 'Still Life' CSP

The table on the right lists results for the Still Life problem (prob0032 in CSPLib). The problem was modeled in a straightforward manner using the input language of Figure 1. To change it from an optimization problem to a CSP, a constraint requiring a minimum number of live cells was added. Each Life CSP was run 20 times with and without our processing methods, starting from different initial states and random seeds. Times are in seconds. The timeout was 500 seconds. The main processing method to affect the Still Life problem is the NNF transformation described in Section 4.1.

Board Size	Live Cells	Solved with Processing	Avg. Time	Solved w/o Processing	Avg. Time
6X6	14	100%	21	70%	188
6X6	15	95%	33	35%	245
6X6	16	100%	54	40%	241
6X6	17	100%	42	10%	294
6X6	18	100%	57	25%	290
7X7	24	95%	60	15%	355
7X7	25	75%	112	5%	356
7X7	26	70%	126	0	N/A
7X7	27	65%	139	0	N/A
7X7	28	65%	219	0	N/A
8X8	30	95%	78	0	N/A
8X8	32	75%	167	0	N/A
9X9	35	90%	152	0	N/A
9X9	37	70%	161	0	N/A
9X9	39	80%	138	0	N/A

5.3 Processor Verification

Most CSPLib problems are not natural candidates for testing our processing methods, as they have a simple recurring structure that can be easily modeled without requiring automatic processing to improve modeling. Industrial problems, on the other hand, can be very large and irregular, thus making it hard to manually model them in a way that would not hinder the results of an SLS solver. Moreover, the model is sometimes generated in a distributed fashion, making it even harder to take into account global considerations such as removing redundancies when modeling the CSP. Hardware verification [15] is one example of a domain where such problems are abundant. Our processing methods were applied to an industrial processor verification problem, generated automatically from user-interfaces for modeling the micro-architecture of the processor [19]. Typical problems consisted of 1,500 variables and 15,000 constraints. Details of the results of this work are beyond the scope of the paper, however, we can report that our processing methods reduced the number of variable-bits in the search space from about 100,000 to about 33,000, and enabled the SLS solver to reach significantly deeper local minima at a much shorter run-time.

5.4 Random Expression-Based CSPs

In order to test our algorithm in a less controlled environment, we generated a new type of random CSPs. Given a formal grammar of an input language such as the one shown in Figure 1, we consider ensembles of problems created by

probabilistic formal grammar rules that generate a subset of the input language. A random expression-based problem is defined by $< N, D, M, G, \boldsymbol{p} >$, where N is the number of variables, D is the domains, M is the number of constraints, G is the formal grammar, and $\boldsymbol{p}$ is a probability-vector for the probabilistic rules. Like many real-world problems, random problems generated using this scheme exhibit little regularity despite the small number of parameters that control their creation, with constraints of varying tree-depth and complexity. [5]

Table 1. Probabilistic Grammar for Random CSP Generation

CONS $\longrightarrow$ (0.9) PC | (0.1) (not PC)
EXP $\longrightarrow$ (0.1) (EXP ARITH EXP) | (0.72) VAR | (0.08) NUM
PC $\longrightarrow$ (0.3) (CONS OP CONS) | (0.7) (EXP COMP EXP)
OP $\longrightarrow$ (0.25) and | (0.25) or | (0.25) implies | (0.25) iff
ARITH $\longrightarrow$ (0.33) + | (0.66) *
COMP $\longrightarrow$ (0.35) $>$ | (0.35) $<$ | (0.15) $=$ | (0.15) $\neq$
VAR $\longrightarrow$ Uniform probability over variables
NUM $\longrightarrow$ Uniform probability over domain range

We generated 300 random CSPs according to the probabilistic grammar of Table 1. Each CSP had 50 variables with domains $[0, 1000]$, and 20 constraints generated by the non-terminal CONS. Relatively large variable domains were chosen because such domains are characteristic of many industrial problems, such as verification [15] and workforce management [20]. Additionally, the problem is much harder to solve with larger domains. Making it harder by increasing the number of variables and constraints would make it cumbersome to analyze. The structure of the random expressions created by this particular grammar is reminiscent in many ways of the processor verification problems discussed above.

The solver was run 20 times on each CSP in 4 configurations. A total of 156 CSPs were solved at least once. For these, we compared solve-times between a configuration that involves no processing methods, a configuration that includes all methods, and configurations that disable one method. We consider only statistically significant differences in solve-times[6]. Figure 4 shows the improvements in solve-times achieved for each CSP in the different configurations, compared to solving the CSP with no preprocessing. The following table summarizes the results, again comparing solve-times when using some preprocessing methods to using no preprocessing.

	Applying All Methods	No Predetermined Dimensions	No NNF transformation	No Dealiasing
Improvement	**79% (123)**	**69% (107)**	**39% (61)**	**40% (63)**
Deterioration	17% (26)	18% (28)	6% (9)	10% (16)
No Difference	4% (7)	13% (21)	55% (86)	50% (77)

[5] A repository of CSP instances generated according to this scheme and used in this and the next sub-section can be found in `http://www.haifa.il.ibm.com/projects/verification/octopus/random`.

[6] A paired t-test with significance level of 0.05 was used.

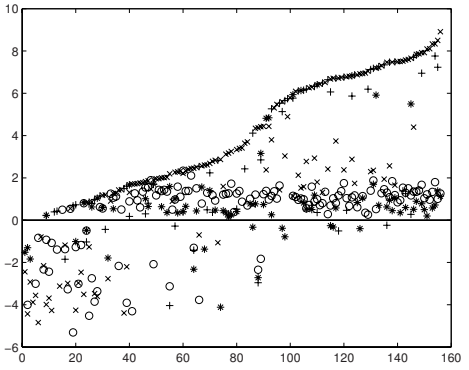


Fig. 4. Effect of the methods on random expression CSPs. The X axis corresponds to 156 CSPs, sorted by ascending ratio of improvement by our methods. The Y axis is log_2 of the ratio of time improvement by each configuration. Points above the zero line are ones in which improvement was achieved. Legend: circle = no predetermined dimensions, plus = no NNF-transformation, star = no Dealiasing, x = all methods applied.

5.5 Phase Transition in Random Expression-Based Problems

It is well documented that random tabular CSP and random SAT problems exhibit satisfiability phase transitions [21,22]. There exists a critical curve that describes the percentage of satisfiable instances of the random ensemble as a function of some parameter, e.g., the ratio between the number of constraints and the number of variables. Given a structure of the random problem, in the thermodynamic limit (i.e., with a large number of variables), the critical curve is universal — it does not depend on specifics such as the number of variables in the problem. Furthermore, instances with near-critical parameter values are found to be the hardest for systematic search.

We report a preliminary investigation of this phenomena on the random expression-based CSPs defined in the previous section. We used the grammar of Figure 1 with a probability-vector $\boldsymbol{p}$ to generate ensembles of 50 random problems for given numbers of variables N_V and constraints N_C. Figure 5 shows the percentage of problems solved by Stocs within a timeout of up to one minute.[7] A clear transition from solvable to unsolvable is observed as a universal function of N_C/N_V. We also find that the exact location of the transition depends on the probability-vector $\boldsymbol{p}$. As with systematic search, the hardest instances are around the critical area. Although the solver is not complete, it easily identifies unsatisfiable instances for CSPs far above the critical area during the application of its model-processing methods.

6 Summary

We presented a set of methods for processing a CSP model that is expressed using an expression language, in order to make this model more suitable for solving with an SLS solver. We used an architecture that allows definition of operators as part of an input expression language for SLS solvers. By implementing

[7] The timeout was set to a value much larger than the longest time for which a solution was ever found at a given combination of N_V and N_C.

the abstract interfaces for a new operator, the CSP defined by those operators automatically benefits from the model-processing methods we introduced. This extends the clear separation between the model and the search algorithm to the pre-search phase.

One part of the work that we only briefly investigated covers the random expressions introduced in sub-sections 5.4. We conjecture that this form of random CSPs resembles real world CSPs much better than random-table CSPs or random SAT instances. More specifically, by tuning the rules and parameters of the formal grammar, different ensembles may be generated, each possibly resembling a different application domain. Investigation of such ensembles may guide the design of algorithms and heuristics suitable for the particular domain.

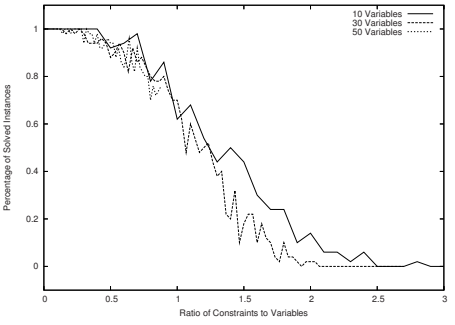


Fig. 5. Phase transition in expression based CSPs

Acknowledgments

We are grateful to Eyal Bin for presenting us with the processor verification problem and for defining much of the syntax of the expression language we used.

References

1. van Hentenryck, P.: The OPL optimization programming language. MIT Press, Cambridge, MA, USA (1999)
2. Hoos, H.H., Steutzle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, San Francisco (2004)
3. van Hentenryck, P., Michel, L.: Control abstractions for local search. In: CP 2003. (2003)
4. Nareyek, A.: Using global constraints for local search. In: Constraint Programming and Large Scale Discrete Optimization, DIMACS Vol. 57. (2001) 9–28
5. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 26. (1996)
6. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science **220** (1983) 671–680
7. Minton, S., Johnston, M., Phillips, A., Laird, P.: Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In: AAAI-90. (1990) 17–24
8. Glover, F., Laguna, M.: Tabu Search. Kluwer (1997)
9. Hansen, P., Mladenovic, N.: Introduction to variable neighbourhood search. In: Metaheuristics: Advances and Trends in Local Search Procedures for Optimization. (1999) 433–458

10. Bacchus, F.: Enhancing Davis Putnam with extended binary clause reasoning. In: AAAI-02. (2002)
11. Bacchus, F., Walsh, T.: Propagating logical combinations of constraints. In: IJCAI-05. (2005) 35–40
12. Boese, K.D., Kahng, A.B., Muddu, S.: A new adaptive multi-start technique for combinatorial global optimizations. Operations Res. Lett. **16**(3) (1994) 101–113
13. Bohlin, M.: Improving cost calculations for global constraints in local search. In: CP 2002. (2002) 772
14. Naveh, Y.: Stochastic solver for constraint satisfaction problems with learning of high-level characteristics of the problem topography. In: Local Search Techniques in Constraint Satisfaction (LSCS-04). (2004)
15. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. AI Magazine (2007)
16. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. Lecture Notes in Computer Science **1520** (1998) 417
17. Garey, M., Johnson, D.: Computers and Intractability: a Guide to Theory of NP-completeness. W.H.Freeman (1979)
18. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. The MIT Press (2005)
19. Adir, A., Bin, E., Peled, O., Ziv, A.: Piparazzi: A test program generator for micro-architecture flow verification. In: Eighth IEEE International High-Level Design Validation and Test Workshop, HLDVT-03. (2003) 23–28
20. Naveh, Y., Richter, Y., Altshuler, Y., Gresh, D.L., Connors, D.P.: Workforce optimization: Identification and assignment of professional workers using constraint programming. IBM Journal or Research and Development (2007)
21. Prosser, P.: An empirical study of phase transition in binary constraint satisfaction problems. Artificial Intelligence **81** (1996) 81–109
22. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Ttroyansky, L.: Determining computational complexity from characteristic 'phase transition'. Nature **400** (1999) 133–137

The Deviation Constraint

Pierre Schaus[1], Yves Deville[1], Pierre Dupont[1], and Jean-Charles Régin[2]

[1] Université catholique de Louvain, Belgium
{pschaus,yde,pdupont}@info.ucl.ac.be
[2] ILOG, France
regin@ilog.fr

Abstract. This paper introduces `DEVIATION`, a soft global constraint to obtain balanced solutions. A violation measure of the perfect balance can be defined as the L_p norm of the vector variables minus their mean. `SPREAD` constraints the sum of square deviations to the mean [5,7] *i.e.* the L_2 norm. The L_1 norm is considered here. Neither criterion subsumes the other but the design of a propagator for L_1 is simpler. We also show that a propagator for `DEVIATION` runs in $\mathcal{O}(n)$ (with respect to the number of variables) against $\mathcal{O}(n^2)$ for `SPREAD`.

1 Introduction

We consider the Balanced Academic Curriculum Problem (BACP) [1] as a motivating example. The goal is to assign a period to each course in a way that the prerequisite relationships are satisfied and *the academic load of each period is balanced*. This last constraint makes BACP a Constraint Optimization Problem where the objective is to maximize the balancing property.

In BACP the mean m of a solution is a constant of the problem since the load of each course and the number of periods are given. A hard balancing constraint would impose all periods to take a same load m. This often results in an over-constrained problem without solution. For a set of variables $\mathcal{X} = \{X_1, X_2, ..., X_n\}$ and a given fixed mean m, a violation measure of the perfect balance property can be defined as the L_p-norm of the vector $[\mathbf{X} - \mathbf{m}]$ with $\mathbf{X} = [X_1, X_2, ..., X_n]$, $\mathbf{m} = [m, m, ..., m]$ such that $\sum_{i=1}^{n} X_i = n.m$. The L_p-norm of $[\mathbf{X} - \mathbf{m}]$ is defined as $(\sum_{i=1}^{n} |X_i - m|^p)^{\frac{1}{p}}$ with $p \geq 0$.

Following the scheme proposed by Régin et al. [6] to soften global constraints, we define a violation of the perfect balance constraint as a cost variable L_p in the global balance constraint: `soft-balance`$(\mathcal{X}, m, L_p)$ constraint holds if and only if L_p-norm$([\mathbf{X} - \mathbf{m}]) = L_p$ and $\sum_{i=1}^{n} X_i = n.m$.

The interpretation of the violation to the mean for some specific norms is given below.

- L_0: $|\{X \in \mathcal{X} | X \neq m\}|$ is the number of values different from the mean.
- L_1: $\sum_{X \in \mathcal{X}} |X - m|$ is the sum of deviations from the mean.
- L_2: $\sum_{X \in \mathcal{X}} (X - m)^2$ is the sum of square deviations from the mean.
- L_∞: $\max_{X \in \mathcal{X}} |X - m|$ is the maximum deviation from the mean.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 260–274, 2007.

Note that none of these balance criteria subsumes the others. For instance, the minimization of L_1 does not imply in general a minimization of criterion L_2. This is illustrated on the following example. Assume a constraint problem with four solutions given in Table 1. The most balanced solution depends on the chosen norm. Each solution exhibits a mean of 100 but each one optimizes a different norm.

Table 1. Illustration showing that no balance criterion defined by the norm L_0, L_1, L_2 or L_∞ subsumes the others. The smallest norm is indicated in bold character. For example, solution 2 is the most balanced according to L_1.

sol. num.	solution	L_0	L_1	L_2	L_∞
1	100 100 100 100 30 170	**2**	140	9800	70
2	60 80 100 100 120 140	4	**120**	4000	40
3	70 70 90 110 130 130	6	140	**3800**	30
4	71 71 71 129 129 129	6	174	5046	**29**

The norm L_∞ has already been used in two previous works [2,4] to solve BACP. `SPREAD` is a constraint for L_2 [5,7]. A constraint for L_0 can easily be implemented using an $\texttt{ATLEAST}(i, [X_1, ..., X_n], m)$ constraint for $|\{X \in \mathcal{X} | X \neq m\}| \leq i$ and a $\texttt{SUM}([X_1, ..., X_n], n.m)$ constraint to ensure a mean of m. In this paper, a global constraint and its propagators for the L_1 norm with fixed mean is presented. This constraint is formulated in the following definition:

Definition 1. *A set of finite domain integer variables $\mathcal{X} = \{X_1, X_2, ..., X_n\}$, one mean value m and one interval variable D are given.*

The constraint $\texttt{DEVIATION}(\mathcal{X}, m, D)$ states that the collection of values taken by the variables of $\mathcal{X}$ exhibits an arithmetic mean m and a sum of deviations to m of D. More formally, $\texttt{DEVIATION}(\mathcal{X}, m, D)$ holds if and only if

$$n.m = \sum_{i=1}^{n} X_i \quad \text{and} \quad D = \sum_{i=1}^{n} |X_i - m|.$$

For the constraint to be consistent, $n.m$ must be an integer. As a consequence $n.D$ is also an integer.

Outline of the Paper

Section 2 mainly reviews preliminaries notions relative to constraint programming such as filtering, domain-consistency and bound-consistency. We also define some useful notations. Section 3 motivates the need of a global filtering algorithm for `DEVIATION` in terms of filtering. Section 4 explains the propagators narrowing the domain of D. This filtering makes use of the minimization and maximization of the sum of deviations. The minimization is solved in linear

time. The maximization is proved to be $\mathcal{NP}$-complete; however, an approximated upper bound can be calculated in linear time as well. Section 5 describes the filtering algorithm from m and D to the variables $\mathcal{X}$. The idea is similar to a bound consistency filtering algorithm for a `SUM` constraint but including the sum of deviations constraint. Section 6 shows that our propagators do not achieve bound-consistency. Section 7 gives a relaxation of `SPREAD` with `DEVIATION`. Finally, Section 8 evaluates the efficiency of the presented propagators in terms of filtering on randomly generated instances.

2 Background and Notations

Basic constraint programming concepts largely inspired from Section 2 of [8] are introduced.

Let X be a finite-domain (discrete) *variable*. The *domain* of X is a set of ordered values that can be assigned to X and is denoted by $Dom(X)$. The minimum (resp. maximum) value of the domain is denoted by $X^{\min} = \min(Dom(X))$ (resp. $X^{\max} = \max(Dom(X))$. Let $\mathcal{X} = \{X_1, X_2, ..., X_k\}$ be a sequence of variables. A *constraint* C on $\mathcal{X}$ is defined as a subset of the Cartesian product of the domains of the variables in $\mathcal{X}$: $C \subseteq Dom(X_1) \times Dom(X_2) \times ... \times Dom(X_k)$. A tuple $(v_1, ..., v_k) \in C$ is called a *solution* to C. A value $v \in Dom(X_i)$ for some $i = 1, ..., k$ is *inconsistent* with respect to C if it does not belong to a tuple of C, otherwise it is *consistent*. C is inconsistent if it does not contain a solution. Otherwise, C is called consistent.

A constraint satisfaction problem, or a CSP, is defined by a finite sequence of variables $\mathcal{X} = \{X_1, X_2, ..., X_n\}$, together with a finite set of constraint $\mathcal{C}$ each on a subset of $\mathcal{X}$. The goal is to find an *assignment* $X_i := v$ with $v \in Dom(X_i)$ for $i = 1, ..., n$, such that all constraints are satisfied. This assignment is called a solution to the CSP.

The solution process of constraint programming interleaves constraint *propagation*, or propagation in short, and search. The search process essentially consists of enumeration all possible variable-value combinations, until a solution is found or it is proved that none exists. We say that this process constructs a search tree. To reduce the exponential number of combinations, constraint propagation is applied to each node of the search tree: Given the current domains and a constraint C, the propagator for C removes domain values that do not belong to a solution to C. This is repeated for all constraints until no more domain value can be removed. The removal of inconsistent domain values is called *filtering*.

In order to be effective, filtering algorithms should be efficient, because they are applied many times during the search process. They should furthermore remove as many inconsistent values as possible. If a filtering algorithm for a constraint C removes all inconsistent values from the domains with respect to C, we say that it makes C *domain-consistent*. It is possible to achieve domain-consistency in polynomial time for some constraints such as `AllDiff` but for other constraints such as `SUM` this would be too costly. In such cases a weaker

notion of consistency called *bounds-consistency* (also called interval-consistency) appears to be highly cost-effective. A constraint C is bound consistent if the bounds of the domain of each variable implied in C belongs to at least one solution of C. The idea it to bound the domain of each variable by an interval and make sure that the end-points of the intervals obey the domain-consistency requirement. If not, the upper and lower bounds of the intervals can be tightened until bounds-consistency is achieved.

Proposition 1 states that achieving domain-consistency for `DEVIATION` is not more difficult than for arithmetic constraints in general.

Proposition 1. *Achieving domain-consistency for* `DEVIATION` *is* $\mathcal{NP}$*-Complete.*

Proof. The constraint $\texttt{SUM}(X_1...X_n, S)$ states that $\sum_{i=1}^{n} X_i = S$. It is well known that achieving domain consistency for `SUM` is $\mathcal{NP}$-Complete (The *subset sum* problem [3] can easily be reduced to achieving domain-consistency for `SUM`). In the particular case where $Dom(D) = [0, +\infty]$ and $m = S/n$, achieving domain-consistency for `SUM` constraint reduces to achieving domain-consistency for $\texttt{DEVIATION}(\mathcal{X}, m, D)$. ■

Domains of variables are considered as full and can be described by the interval $Dom(X) = [X^{\min}..X^{\max}]$.

Definition 2 introduces some useful notations.A numerical example is also given in Example 1.

Definition 2. *For a variable X and a given value m, the upper bounds on the right and left deviation are respectively*

- $\overline{rd}(X, m) = \max(0, X^{\max} - m)$ *and*
- $\overline{ld}(X, m) = \max(0, m - X^{\min})$.

The sum of these values over $\mathcal{X}$ are respectively

- $\overline{RD}(\mathcal{X}, m) = \sum_{X \in \mathcal{X}} \overline{rd}(X, m)$ *and*
- $\overline{LD}(\mathcal{X}, m) = \sum_{X \in \mathcal{X}} \overline{ld}(X, m)$.

The same idea holds for the lower bounds on the deviations:

- $\underline{rd}(X, m) = \max(0, X^{\min} - m)$.
- $\underline{ld}(X, m) = \max(0, m - X^{\max})$.
- $\underline{RD}(\mathcal{X}, m) = \sum_{X \in \mathcal{X}} \underline{rd}(X, m)$.
- $\underline{LD}(\mathcal{X}, m) = \sum_{X \in \mathcal{X}} \underline{ld}(X, m)$.

For a variable $X_i \in \mathcal{X}$ we define:

- $\overline{LD}_i(\mathcal{X}, m) = \overline{LD}(\mathcal{X}, m) - \overline{ld}(X_i, m)$ *and*
- $\overline{RD}_i(\mathcal{X}, m) = \overline{RD}(\mathcal{X}, m) - \overline{rd}(X_i, m)$.

To alleviate notations, $(\mathcal{X}, m)$ are sometimes omitted. For example $\overline{LD}(\mathcal{X}, m)$ is simply written $\overline{LD}$.

Example 1. Let $\mathcal{X}$=$\{X_1, X_2, X_3, X_4\}$ be four variables with domains $Dom(X_1)$=$[8, 10]$, $Dom(X_2) = [4, 7]$, $Dom(X_3) = [1, 5]$ and $Dom(X_4) = [3, 4]$. The following table exhibits the quantities introduced in Definition 2.

i	$\overline{rd}(X_i, 5)$	$\overline{ld}(X_i, 5)$	$\underline{rd}(X_i, 5)$	$\underline{ld}(X_i, 5)$
1	5	0	3	0
2	2	1	0	0
3	0	4	0	0
4	0	2	0	1
$\sum_i$	**7**	**7**	**3**	**1**
	$\overline{RD}_i(\mathcal{X}, 5)$	$\overline{LD}_i(\mathcal{X}, 5)$	$\underline{RD}_i(\mathcal{X}, 5)$	$\underline{LD}_i(\mathcal{X}, 5)$
1	2	7	0	1
2	5	6	3	1
3	7	3	3	1
4	7	5	3	0

The filtering for `DEVIATION` is based on the next theorem stating that the sum of deviations above and under the mean are equal.

Lemma 1. *Let $\mathcal{X} = \{X_1, ..., X_n\}$. The equality $n.m = \sum_{X \in \mathcal{X}} X$ holds if and only if $\sum_{X>m}(X - m) = \sum_{X<m}(m - X)$.*

Proof. $n.m = \sum_{X \in \mathcal{X}} X$ can be rewritten $0 = \sum_{X \in \mathcal{X}} X - n.m = \sum_{X>m}(X - m) + \sum_{X<m}(X - m) + \sum_{X=m}(X - m) = \sum_{X>m}(X - m) - \sum_{X<m}(m - X)$. ■

Property. 1 *Let $\mathcal{X} = \{X_1, ..., X_n\}$. An assignment on $\mathcal{X}$ satisfies:*

- $\sum_{X>m}(X - m) \in [\underline{RD}(\mathcal{X}, m), \overline{RD}(\mathcal{X}, m)]$ *and*
- $\sum_{X<m}(m - X) \in [\underline{LD}(\mathcal{X}, m), \overline{LD}(\mathcal{X}, m)]$.

Theorem 1. *`DEVIATION`$(\mathcal{X}, m, D)$ is consistent only if the following conditions are satisfied:*

1. $\underline{RD}(\mathcal{X}, m) \leq \frac{D^{\max}}{2}$
2. $\underline{LD}(\mathcal{X}, m) \leq \frac{D^{\max}}{2}$
3. $\overline{RD}(\mathcal{X}, m) \geq \frac{D^{\min}}{2}$
4. $\overline{LD}(\mathcal{X}, m) \geq \frac{D^{\min}}{2}$
5. $\left[\underline{LD}(\mathcal{X}, m), \overline{LD}(\mathcal{X}, m)\right] \cap \left[\underline{RD}(\mathcal{X}, m), \overline{RD}(\mathcal{X}, m)\right] \neq \phi$

Proof. 1. If $\underline{RD}(\mathcal{X}, m) > \frac{D^{\max}}{2}$ then $\sum_{X>m}(X - m) > \frac{D^{\max}}{2}$ (Property 1). Hence $\sum_{i=1}^{n} |X_i - m| > D^{\max}$ (by Lemma 1).
2., 3. and 4. similar to 1.
5. Direct consequence of Lemma 1 and Property 1. ■

3 Naive Implementation

This section explains why a naive implementation of DEVIATION by decomposition into more elementary constraints is not optimal in terms of filtering.

As stated in Definition 1, DEVIATION$(\mathcal{X}, m, D)$ holds if and only if $n.m = \sum_{i=1}^{n} X_i$ and $D = \sum_{i=1}^{n} |X_i - m|$. This suggests a natural implementation of the constraint by decomposing it into two SUM constraints. Figure 1 illustrates that the filtering obtained with the decomposition is not optimal.

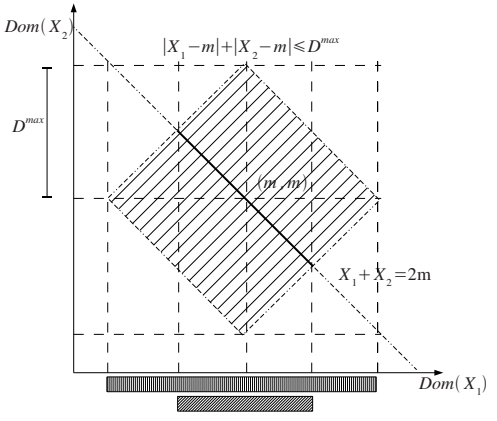


Fig. 1. Filtering of X_1 with decomposition and with DEVIATION

Assume two variables X_1, X_2 with unbounded finite domains and the constraint

$$\texttt{DEVIATION}(\{X_1, X_2\}, m, D \in [0, D^{\max}]).$$

The diagonally shaded square (see Figure 1) delimits the set of points such that $|X_1 - m| + |X_2 - m| \leq D^{\max}$. The diagonal line is the set of points such that $X_1 + X_2 = 2.m$. The unbounded domains for X_1 and X_2 are bound-consistent for the mean constraint. The vertically shaded rectangle defines the domain of X_1 after a bound-consistent filtering for $|X_1 - m| + |X_2 - m| \leq D^{\max}$. The set of solutions for DEVIATION is the bold diagonal segment obtained by intersecting the square surface and the diagonal line. It can be seen on the figure that more filtering is possible. Bound consistent filtering on the bold diagonal segment leads to a domain of X_1 defined by the diagonally shaded rectangle. In conclusion a bound consistent filtering for the decomposition leads to $Dom(X_1) = Dom(X_2) = [m - D^{\max}, m + D^{\max}]$ while a bound consistent filtering for DEVIATION$(\{X_1, X_2\}, m, D \in [0, D^{\max}])$ leads to domains two times smaller $Dom(X_1) = Dom(X_2) = [m - \frac{D^{\max}}{2}, m + \frac{D^{\max}}{2}]$.

4 Filtering of D

The filtering of D to achieve bound consistency requires to solve two optimization problems: the minimization and maximization of the sum of deviations from a given mean m. Definition 3 defines a relaxation of these problems by allowing rational assignments.

Definition 3. *$\underline{D}$ and $\overline{D}$ denote the optimal values to problems:*

$$\underline{D} = \min \sum_{i=1}^{n} |X_i - m| \quad \text{and} \quad \overline{D} = \max \sum_{i=1}^{n} |X_i - m|$$

$$\text{such that : } \sum_{i=1}^{n} X_i = n.m$$
$$X_i^{\min} \leq X_i \leq X_i^{\max},\ 1 \leq i \leq n$$
$$X_i \in \mathbb{Q},\ 1 \leq i \leq n.$$

Values $\underline{D}$ and $\overline{D}$ can be used to filter the domain of D:

$$Dom(D) \longleftarrow Dom(D) \cap [\underline{D}, \overline{D}].$$

The remaining of this section mainly explains how $\underline{D}$ can be computed in linear time with respect to the number of variables $n = |\mathcal{X}|$. Unfortunately finding $\overline{D}$ is an $\mathcal{NP}$-complete problem and the best we can do is to design a good upper bound for it as explained at the end of this section.

Definition 4 characterizes an optimal solution to the problem of finding $\underline{D}$.

Definition 4 (up and down centered assignment). *Let $\mathcal{X} = \{X_1, ..., X_n\}$. Let $A : X \rightarrow Dom(X)$ be an assignment of $X \in \mathcal{X}$. The quantity $s(A)$ denotes the sum of assigned values: $s(A) = \sum_{X \in \mathcal{X}} A(X)$.*

*An assignment A is said to be **up-centered** when:*

$$A(X) \begin{cases} = X^{\min} & \text{if } X^{\min} \geq s(A)/n \\ \leq s(A)/n & \text{otherwise} \end{cases}$$

In other words, each variable with minimum domain value larger than the mean of the assigned values takes its minimum domain value and the other variables take values smaller than the mean of the assigned values.

*An assignment A is said to be **down-centered** when:*

$$A(X) \begin{cases} = X^{\max} & \text{if } X^{\max} \leq s(A)/n \\ \geq s(A)/n & \text{otherwise} \end{cases}$$

In other words, each variable with maximum domain value smaller than the mean of the assigned values takes its maximum domain value and the other variables take values larger than the mean of the assigned values.

Example 2. Considering the variables and domains of Example 1, the following assignment is up-centered with mean $17/4$*:*

$$A(X_1) = 8, A(X_2) = 4, A(X_3) = 2, A(X_4) = 3.$$

Theorem 2. *An assignment is an optimal solution to the problem of finding* $\underline{D}$ *if and only if it is a down-centered assignment or an up-centered assignment of mean* m*.*

Proof. (if) Given an assignment A of mean m *i.e.* $s(A) = n.m$. The only way to decrease the sum of deviations while conserving the mean m is to find a pair of variables X_i, X_j such that $A(X_i) > m, A(X_i) > X_i^{\min}, A(X_j) < m, A(X_j) < X_j^{\max}$ and to decrease $A(X_i)$ and increase $A(X_j)$ by the same quantity to make them closer to m. By definition of a left and down centered assignment, it is impossible to find such a pair X_i, X_j. Hence, up-centered and a down-centered assignments are optimal solutions.

(only if) Assume an assignment A is neither down-centered nor up-centered such that $s(A) = n.m$. It is possible to find at least two variables $X_i, X_j \in \mathcal{X}$. One with $A(X_i) > m$ and $A(X_i) > X_i^{\min}$ (violation of up-centered) and one with $A(X_j) < m$ and $A(X_j) < X_j^{\max}$ (violation of down-centered). Let us define $\delta = \min\big(A(X_i) - \max(X_i^{\min}, m), \min(X_j^{\max}, m) - A(X_j)\big)$. The assignment $A(X)$ is not optimal since the sum of deviations can be decreased by 2δ by modifying the assignment on X_i and X_j: $A'(X_i) = A(X_i) - \delta$ and $A'(X_j) = A(X_j) + \delta$. ■

Theorem 3. *If* `DEVIATION` *is consistent then*

$$\underline{D} = 2.\max(\underline{LD}\,(\mathcal{X}, m), \underline{RD}(\mathcal{X}, m))\,.$$

Proof. Assume $\underline{LD} \geq \underline{RD}$, then it is possible to build a down-centered assignment A of mean m and which is optimal by Theorem 2. For this assignment $\sum_{A(X)<m}(m - A(X)) = \underline{LD}$ (by Definition 2 of $\underline{LD}$). Since $\sum_{X>m}(X - m) = \sum_{X<m}(m - X)$ (by Lemma 1), the sum of deviations for this down-centered assignment is $\sum_{X\in\mathcal{X}} |A(X) - m| = 2.\underline{LD}$.

The case $\underline{LD} \leq \underline{RD}$ is similar. The assignment is up-centered instead of down-centered. ■

Example 3. The variables and domains considered here are the same as those in Example 1. A mean $m = 5$ is considered. Using the computed values $\underline{LD}(\mathcal{X}, 5) = 1$ and $\underline{RD}(\mathcal{X}, 5) = 3$ from Example 1, it can be deduced that $\underline{D} = 2.\max(1, 3) = 6$.

Consequently, filtering on D for `DEVIATION`$(\mathcal{X}=\{X_1, X_2, X_3, X_4\}, m = 5, D \in [0, 7])$ leads to $Dom(D) = [6, 7]$.

Theorem 4. *Computing* $\overline{D}$ *is* $\mathcal{NP}$*-complete.*

Proof. It is possible to reduce the subset sum problem [3] to the problem of computing $\overline{D}$ (see Definition 3). This problem is not more difficult than the particular case where $m = 0$:

$$\overline{D} = \max \sum_{i=1}^{n} |X_i|$$
$$\text{such that : } \sum_{i=1}^{n} X_i = 0$$
$$X_i^{\min} \leq X_i \leq X_i^{\max},\ 1 \leq i \leq n.$$

Given a set of n positive values $\{b_1, ..., b_{n-1}, T\}$, the subset sum problem consists in finding if there exists a set of binary values $\{y_1, ..., y_{n-1}\}, y_i \in \{0, 1\}, 1 \leq i < n$ such that $\sum_{i=1}^{n-1} y_i.b_i = T$. The reduction is the following:

- $X_i^{\min} = -\frac{b_i}{2}$ and $X_i^{\max} = \frac{b_i}{2}$ for $1 \leq i < n$.
- $X_n = \frac{\sum_{i=1}^{n-1} b_i}{2} - T$.
- There is a solution to the subset sum problem if and only if $\overline{D} \geq \frac{\sum_{i=1}^{n-1} b_i}{2} + \left|\frac{\sum_{i=1}^{n-1} b_i}{2} - T\right|$. This constraint on the optimal value ensures that the optimal solution is such that $X_i \in \{-\frac{b_i}{2}, \frac{b_i}{2}\}$. The solution to the subset sum problem is then given by $y_i = 1$ if $X_i = \frac{b_i}{2}$ and $y_i = 0$ if $X_i = \frac{-b_i}{2}$. ■

Unless $\mathcal{P} = \mathcal{NP}$, the problem of computing $\overline{D}$ is exponential (Theorem 4). As explained in Section 2, in order to be effective, filtering algorithms should be as efficient as possible because they are applied many times during the search process. This is why we prefer to find efficiently a good upper bound $\overline{D}^{\uparrow}$ for $\overline{D}$ than to find its exact value. An upper bound which can be computed in $\mathcal{O}(n)$ is

$$\overline{D}^{\uparrow} = \sum_{i=1}^{n} \max\left(|X_i^{\max} - m|, |X_i^{\min} - m|\right).$$

The filtering on $Dom(D)$ becomes:

$$Dom(D) \longleftarrow Dom(D) \cap [\underline{D}, \overline{D}^{\uparrow}]$$

5 Filtering on $\mathcal{X}$

Two propagators could be considered to filter the domain of $\mathcal{X}$:

1. from $D^{\min}$ and m to $\mathcal{X}$ and
2. from $D^{\max}$ and m to $\mathcal{X}$.

Achieving bound-consistency for the first propagator is $\mathcal{NP}$-complete. Indeed, checking the consistency of one value requires to maximize the sum of deviations which is an $\mathcal{NP}$-complete problem (Theorem 4). The decomposition of `DEVIATION` presented in Section 3 can however be used to realize bound-consistency on constraint $\sum_{X \in \mathcal{X}} |X - m| \geq D^{\min}$. In any case, this filtering is useless if one seeks a balanced solution on $\mathcal{X}$. Hence, the remaining of this section focuses on a linear time filtering algorithm for the second propagator.

The filtering is based on the computation of the values $\overline{X}_i$ and $\underline{X}_i$ introduced in Definition 5.

Definition 5. *$\overline{X}_i$ and $\underline{X}_i$ are the optimal values to the following problems:*

$$\begin{aligned} & \overline{X}_i = \max(X_i) \quad \text{and} \quad \underline{X}_i = \min(X_i) \\ \text{such that} : & \sum_{j=1}^{n} X_j = n.m & (1) \\ & \sum_{j=1}^{n} |X_j - m| \leq D^{\max} & (2) \\ & X_j^{\min} \leq X_j \leq X_j^{\max}, 1 \leq j \leq n, j \neq i \\ & X_j \in \mathbb{Q}, 1 \leq j \leq n. \end{aligned}$$

The filtering rule on the domain of X_i can be simply written:

$$Dom(X_i) \longleftarrow Dom(X_i) \cap [\underline{X}_i, \overline{X}_i] \tag{3}$$

Theorem 5. *For a variable X_i, assuming the constraint is consistent, the following equalities hold:*

$$\overline{X}_i = \min\left(\frac{D^{\max}}{2}, \overline{LD}_i(\mathcal{X}, m)\right) - \underline{RD}_i(\mathcal{X}, m) + m.$$

$$\underline{X}_i = -\min\left(\frac{D^{\max}}{2}, \overline{RD}_i(\mathcal{X}, m)\right) + \underline{LD}_i(\mathcal{X}, m) + m.$$

Proof. Only $\overline{X}_i$ is considered because the proof for $\underline{X}_i$ is symmetrical with respect to m. Two cases can be considered:

- $\overline{LD}_i \leq \frac{D^{\max}}{2}$: By Lemma 1 the deviation above the mean and under the mean must be equal. Hence the optimal solution is such that $\overline{X}_i - m + \underline{RD}_i = \overline{LD}_i$. Constraint (2) is not tight in this case.
- $\overline{LD}_i > \frac{D^{\max}}{2}$: By Lemma 1 the constraint (1) means that the deviation above the mean and under the mean must be equal. The conjunction of constraint (1) with constraint (2) means that the deviation above and under the mean are equal and at most $D^{\max}/2$. Hence the optimal solution is such that $\overline{X}_i - m + \underline{RD}_i = \frac{D^{\max}}{2}$. Constraint (2) is tight in this case.

If both cases are considered together, equality $\overline{X}_i - m + \underline{RD}_i = \min(\frac{D^{\max}}{2}, \overline{LD}_i)$ holds at the optimal solution. ■

The filtering procedure on $\mathcal{X}$ applies rule (3) once on each $X_i \in \mathcal{X}$. This can be achieved in linear time with respect to the number of variables.

Example 4. Variables and domains considered are the same as in Example 1. The constraint considered is $\texttt{DEVIATION}(\mathcal{X} = \{X_1, X_2, X_3, X_4\}, m = 5, D \in [0, 7])$.

Values $\overline{X}_i$ and $\underline{X}_i$ are: $\overline{X}_1 = \min(3.5, 7) - 0 + 5 = 8.5$, $\overline{X}_2 = \min(3.5, 6) - 3 + 5 = 5.5$, $\overline{X}_3 = \min(3.5, 3) - 3 + 5 = 5$, $\overline{X}_4 = \min(3.5, 5) - 3 + 5 = 5.5$, $\underline{X}_1 = -\min(3.5, 2) + 1 + 5 = 4$, $\underline{X}_2 = -\min(3.5, 5) + 1 + 5 = 2.5$, $\underline{X}_3 = -\min(3.5, 7) + 1 + 5 = 2.5$ and $\underline{X}_4 = -\min(3.5, 7) + 0 + 5 = 1.5$. Hence filtering rule (3) leads to filtered domains: $Dom(X_1) = [8, 8]$, $Dom(X_2) = [4, 5]$, $Dom(X_3) = [3, 5]$ and $Dom(X_4) = [3, 4]$.

6 Bound Consistency for DEVIATION

The total filtering is achieved in $\mathcal{O}(n)$ as follows.

1. Filtering from $D^{\max}$ and m to $\mathcal{X}$: $\forall X \in \mathcal{X}, Dom(X) \longleftarrow Dom(X) \cap [\underline{X}, \overline{X}]$.
2. Filtering from $\mathcal{X}$ and m to D: $Dom(D) \longleftarrow Dom(D) \cap [\underline{D}, \overline{D}^{\uparrow}]$.

Even if it was possible to compute $\overline{D}$ efficiently, bound consistency is not necessarily obtained. The reason is that values $\overline{X}$, $\underline{X}$ (Definition 5) and $\underline{D}$ (Definition 3) are computed making the assumption that interval domains are defined on rational numbers $\mathbb{Q}$ rather than on integers $\mathbb{Z}$. As the next example shows, this can lead to miss some possible filtering.

Example 5. Assume a set of 10 variables $\mathcal{X}$ with domain $[0, 1]$ and a mean of $m = 0.5$. Theorem 3 gives a value $\underline{D} = 0$ because every domain overlaps m. In fact the only way to obtain an assignment respecting the mean constraint is to have five variables assigned to 0 and five to 1. For such an assignment, the minimal sum of deviations from the mean is 5 and not 0. Consequently, the constraint DEVIATION$(\mathcal{X}, m = 0.5, D \in [0, 3])$ is inconsistent but such an inconsistency will not be detected by our propagator.

Example 5 shows that all inconsistencies are not detected by the propagator. This occurs when the mean is not an integer but a rational number and when the domains of some variables include the mean. When the mean is an integer, such a problem does not occur and the propagator is bound-consistent.

7 Relation Between SPREAD and DEVIATION

This section shows that DEVIATION can be used as a relaxation of SPREAD. This relaxation might be useful since the propagator of the former runs in $\mathcal{O}(n)$ against $\mathcal{O}(n^2)$ for the latter. Furthermore, DEVIATION is easier to implement. The relaxation is illustrated graphically and the parameters given to DEVIATION to obtain the strongest relaxation as possible are expressed as a function of SPREAD parameters.

SPREAD$(\mathcal{X}, m, \Delta^2)$ holds if the m value is the average over $\mathcal{X}$ and the sum of square deviation to m is Δ^2. More formally SPREAD holds if $\sum_{x \in \mathcal{X}} X = n.m$ and $\sum_{x \in \mathcal{X}} (X - m)^2 = \Delta^2$.

From a geometrical point of view, $\sum_{x \in \mathcal{X}} (X - m)^2 \leq (\Delta^{\max})^2$ defines an hyper-sphere centered on $[m, ..., m]$ of radius $\Delta^{\max}$. The set of points satisfying

$\sum_{x\in\mathcal{X}} |X - m| \leq D^{\max}$ lies on a regular polytope centered in $[m, ..., m]$ with 2^n faces in an n-dimensional space [1].

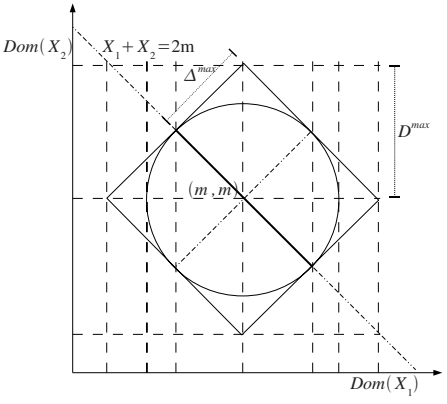


Fig. 2. Relation between SPREAD and DEVIATION for two variables

The idea is to relax SPREAD with deviation by finding the smallest $D^{\max}$ as possible such that the hyper-sphere is included in the polytope. For two variables X_1 and X_2, Figure 2 shows that the circle can be subsumed by the tangent outer-square. For $D^{\max} = \sqrt{2}\Delta$ the outer-square is tangent with the circle (Pythagore relation). For n variables the result is described in the following theorem.

Theorem 6. *$SPREAD(\mathcal{X}, m, [0, (\Delta^{\max})^2]) \subseteq DEVIATION(\mathcal{X}, m, [0, \sqrt{n}.\Delta^{\max}])$ and $\nexists$ $(D^{\max} < \sqrt{n}.\Delta)$ such that $SPREAD(\mathcal{X}, m, [0, (\Delta^{\max})^2]) \subseteq DEVIATION(\mathcal{X}, m, [0, D^{\max}])$.*

Proof. For simplicity we assume $m = 0$. Recall that, the set of points such that $\sum_{x\in\mathcal{X}} |X| \leq D^{\max}$ define a regular polytope centered in the origin with 2^n faces in an n-dimensional space. To find $D^{\max}$ such that the polyhedron is tangent with the hyper-sphere of radius Δ, it is easier to work in the positive orthant since others are symmetrical. In this orthant the problem is reduced to finding $D^{\max}$ such that the hyper-plan $X_1 + X_2 + ... + X_n = D^{\max}$ is tangent with the hyper-sphere $X_1^2 + X_2^2 + ... + X_n^2 = (\Delta^{\max})^2$. At the tangent point we have $X_1 = X_2 = ... = X_n$. Consequently at the tangent point $X_1 = X_2 = ... = X_n = \frac{\Delta^{\max}}{\sqrt{n}}$ and hence $D^{\max} = \frac{n}{\sqrt{n}}\Delta^{\max}$. ■

Note that the equality SPREAD$(\mathcal{X}, m, [0, (\Delta^{\max})^2]) =$DEVIATION$(\mathcal{X}, m, [0, \sqrt{n}.\Delta])$ is valid only when $n = 2$ (two variables). For three variables or more the strict inclusion holds. For instance, the tuple $t = \langle X_1 = \frac{\sqrt{3}}{2}\Delta^{\max}, X_2 = -\frac{\sqrt{3}}{2}\Delta^{\max}, X_3 = 0\rangle \in$ DEVIATION$(\mathcal{X}, m = 0, [0, \sqrt{3}.\Delta^{\max}])$ but $t \notin$ SPREAD$(\mathcal{X}, m = 0, [0, (\Delta^{\max})^2])$. Indeed, $\left(\frac{\sqrt{3}}{2}\Delta^{\max}\right)^2 + \left(\frac{\sqrt{3}}{2}\Delta^{\max}\right)^2 + 0^2 = \frac{3}{2}(\Delta^{\max})^2 > (\Delta^{\max})^2$.

[1] For $n = 2$ it is a square and for $n = 3$ it is an octahedron.

8 Experimental Results

The goal of this experiment is to compare the filtering of the DEVIATION propagators described in Section 4 and 5, with an implementation by decomposition as suggested in Section 3.

20,000 sets $\mathcal{X} = \{X_1, ..., X_{50}\}$ were generated. The domain of one variable X are all the integer values between the minimum and maximum of a generated pair of two uniform random integer values between -50 and 50. The mean constraint on each set is $m = 0.5$. The maximum sum of deviations $D^{\max}$ varies between 200 and 1000. This interval was found experimentally such that for $D^{\max} = 200$ (resp. 1000) all the sets are inconsistent (resp. consistent).

Statistics at the root node (no search was employed) are depicted on Figure 3. The number of inconsistent sets detected by both approaches are given on the left of Figure 3. Note that if an example is detected as inconsistent by decomposition, DEVIATION also detects it. The average percentage of filtering (*i.e.* the number of filtered values divided by the number of initial values in the domains) on consistent sets are plotted on the right of Figure 3.

The number of inconsistent sets detected is significantly larger with the presented propagator than with an implementation by decomposition. For instance, with $D^{\max} = 500$, DEVIATION detects 9,619 inconsistencies against 3,628 with the decomposition. On the 10,381 consistent sets, the pruning percentage obtained with DEVIATION is 11.8% against 0.9% with decomposition.

As shown in Section 6, all inconsistencies are not detected by our propagator if the mean is not integer. Since $m = 0.5$, some inconsistent sets can be undetected. Figure 4 shows a plot of the percentage of inconsistencies detected by decomposition and DEVIATION approaches. Almost all inconsistencies are detected by DEVIATION. The lowest percentage is 99.66% for $D^{\max} = 400$.

Section 7 introduces an approximation of SPREAD with DEVIATION. The left of Figure 5 shows the number of inconsistencies detected by SPREAD (as implemented in [7]) compared with the number of inconsistencies found by the approximation. Many inconsistencies remain undetected but, as shown on the

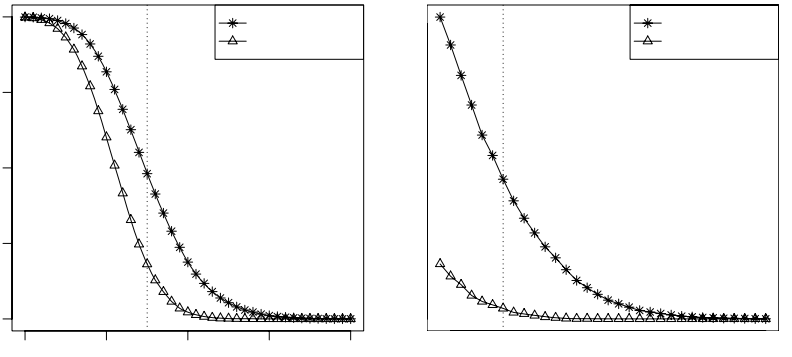


Fig. 3. Experimental Results: DEVIATION v.s. decomposition of Section 3

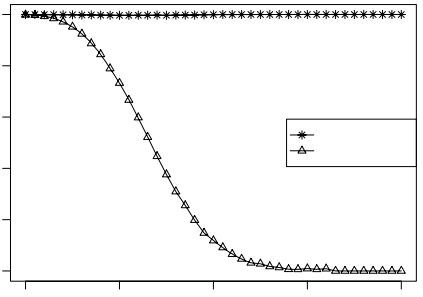


Fig. 4. Percentage of detected inconsistencies

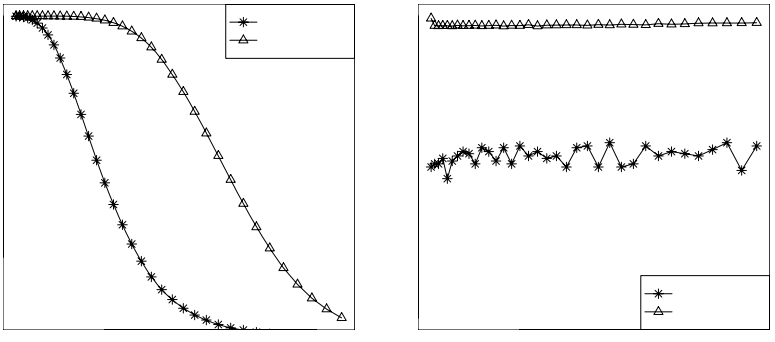


Fig. 5. Experimental Results: `SPREAD` v.s. approximation with `DEVIATION`

left of Figure 5, the propagation using `DEVIATION` is two orders of magnitude faster than with `SPREAD` on these 20.000 random instances.

9 Conclusion

This work presents `DEVIATION`, a new global constraint to balance a set of variables. This constraint is closely related to the `SPREAD` constraint [5,7]. While `SPREAD` constrains the L_2 norm to the mean, `DEVIATION` constrains the L_1 norm.

The filtering algorithms we introduce run in linear time with respect to the number of variables. Experiments evaluate the efficiency in terms of filtering of our propagators. A relaxation of `SPREAD` with `DEVIATION` is also introduced. Such a relaxation offers less filtering but significantly reduces the computation time.

Acknowledgment

The authors wish to thank Julien Hendrickx and Raphaël Jungers for their proof of Theorem 4 and also Jean-Noël Monette and Grégoire Dooms for various interesting discussions.

This research is supported by the Walloon Region, project Transmaze (516207).

References

1. *Problem 30 of CSPLIB (www.csplib.org).*
2. C. Castro and S. Manzano. Variable and value ordering when solving balanced academic curriculum problem. *Proc. of the ERCIM WG on constraints*, 2001.
3. M. R. Garey and David S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.
4. Brahim Hnich, Zeynep Kiziltan, and Toby Walsh. Modelling a balanced academic curriculum problem. *Proceedings of CP-AI-OR-2002*, 2002.
5. G. Pesant and J.C. Régin. Spread: A balancing constraint based on statistics. *Lecture Notes in Computer Science*, 3709:460–474, 2005.
6. J.C. Régin, T. Petit, C. Bessière, and J.-F. Puget. An original constraint based approach for solving over constrained problems. *Sixth International Conference on Principles and Practice of Constraint Programming (CP 2000)*, 1894, 2000.
7. P. Schaus, Y. Deville, P. Dupont, and J.C. Régin. Simplication and extension of the spread constraint. *Third International Workshop on Constraint Propagation And Implementation*, 2006.
8. W.J. van Hoeve, G. Pesant, L.M. Rousseau, and A. Sabharwal. Revisiting the sequence constraint. *Twelfth International Conference on Principles and Practice of Constraint Programming (CP 2006)*, 4204, 2006.

The Linear Programming Polytope of Binary Constraint Problems with Bounded Tree-Width

Meinolf Sellmann, Luc Mercier, and Daniel H. Leventhal

Brown University
Department of Computer Science
115 Waterman Street, P.O. Box 1910
Providence, RI 02912
sello,mercier@cs.brown.edu

Abstract. We show how to efficiently model binary constraint problems (BCP) as integer programs. After considering tree-structured BCPs first, we show that a Sherali-Adams-like procedure results in a polynomial-size linear programming description of the convex hull of all integer feasible solutions when the BCP that is given has bounded tree-width.

Keywords: constraint programming, integer programming, polyhedral combinatorics, cutting planes.

1 Introduction

When solving combinatorial problems, all competitive state-of-the-art solvers combine search with inference. Integer programming (IP) solvers like XpressMP or Cplex base inference on tight continuous linear programming (LP) relaxations. Satisfiability (SAT) solvers perform unit propagation and no-good learning. And constraint programming (CP) solvers make excessive use of constraint filtering techniques. The efficiency, i.e. the effectiveness over CPU-time, is the decisive performance measure for inference algorithms that are used within search.

In IP, effectiveness is best characterized by the gap between the value of the optimal solution and the bound that results from the continuous relaxation of the problem. For certain problems, we find that there is no gap at all. This is the case, for example, when the constraint matrix is totally unimodular and the right hand side vector is integer. In this case, the strength of the inference algorithm alone allows us to solve the corresponding problem in polynomial time.

Interestingly, such islands of tractability are not always found by the analysis of inference algorithms. For certain problems, we can also devise ways how to *search* more efficiently. This is for example the case when we solve a problem by means of dynamic programming. Another example are search algorithms that exploit problem decomposability, such as polynomial-time algorithms for problems on graphs with bounded tree-width [15]. A recent heuristic algorithm that exploits structure to speed-up search is given in [12].

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 275–287, 2007.

For practical reasons, a perfect inference algorithm is preferable over a specialized search routine. Consider for example the case where we hit an island of tractability as a subproblem within a general search. An inference algorithm automatically takes advantage of it, while a specialized search routine could only be invoked when the necessary conditions for its efficient application are detected. Moreover, consider the case where an IP has just a few additional constraints that compromise problem decomposability, thus preventing a specialized search algorithm from being applicable. Then, a tight description of the polytope that the majority of the constraints define still helps to tighten the LP/IP gap.

Consequently, researchers are interested in describing or at least approximating the convex hull of tractable problems. A recent example for such work is [3]. Given that Knapsack problems can be approximated efficiently, Vyve and Wolsey [17] had raised the question whether, for all $\varepsilon > 0$, the Knapsack polytope over n items can be approximated by at most a polynomial number (in n and $1/\varepsilon$) of cuts so that the LP relaxation value does not over-estimate the optimal IP value by more than a factor of 1+ε. This could be viewed as the mathematical programming analogue of a fully polynomial approximation scheme (FPTAS) for the Knapsack polytope. Bienstock provides the analogue of a polynomial approximation scheme (PTAS) by giving a lifted formulation of the Knapsack polytope with $O(n^{1+\lceil 1/\varepsilon \rceil}/\varepsilon)$ variables and $O(n^{2+\lceil 1/\varepsilon \rceil}/\varepsilon)$ constraints.

In this paper, we study the polytope of tree-structured binary constraint networks. It is well known that these problems can be solved in polynomial time by a specialized search based on problem decomposition. We bring this result to the realm of inference by providing a perfect characterization of the corresponding polytope. Particularly, we show that a certain set of linear constraints leads to continuous relaxations with no LP/IP gap. By introducing Sherali-Adams-like variables, we then generalize the result to problems with bounded tree-width.

2 Binary Constraint Satisfaction

We start our study by defining binary constraint satisfaction problems.

Definition 1 (Binary Constraint Satisfaction Problem)

- *A* binary constraint problem (BCP) *is a triplet* $\langle V, D, C \rangle$*, where* $V = \{X_1, \ldots, X_n\}$ *denotes the finite set of variables,* $D = \{D_1, \ldots, D_n\}$ *denotes a set of* n *finite sets of possible values for these variables (*D_i *is called the* domain *of variables* X_i*), and* $C = \{C_1, \ldots, C_m\}$ *is the set of* constraints*, where* $C_j : D_{j_1} \times D_{j_2} \rightarrow Bool$ *specifies which simultaneous assignments of values to the variables* X_{j_1} *and* X_{j_2} *are allowed. The set* $\{X_{j_1}, X_{j_2}\}$ *is called the* scope *of constraint* C_j*.*
- *An* assignment *for a BCP* $\mathcal{P} = \langle V, D, C \rangle$ *is a function* $\sigma : V \rightarrow \bigcup_{i \leq n} D_i$*. A* solution *to a BCP* $\mathcal{P} = \langle V, D, C \rangle$ *is an assignment* σ *such that* $\sigma(X_i) \in D_i$ *for all* $1 \leq i \leq n$ *and such that* $C_j(\sigma(X_{j_1}), \sigma(X_{j_2})) = \mathbf{true}$ *for all* $1 \leq j \leq m$*. The set of all solutions to a BCP* $\mathcal{P}$ *is denoted by* $Sol(\mathcal{P})$*.*

Note how, in contrast to the custom in integer programming, in CP the term “binary” is used to express that all *constraints* affect just two variables, while the size of the *domain*

of each variable is not limited! The fact that the arity of the constraints is limited to two allows us to state constraints simply as sets of allowed pairs $\mathcal{R}_{j_1,j_2} = \{(k,l) \mid X_{j_1} = k,\ X_{j_2} = l \text{ ok}\}$, or, alternatively, as sets of forbidden pairs $\overline{\mathcal{R}_{j_1,j_2}} = \{(k,l) \mid X_{j_1} = k,\ X_{j_2} = l \text{ forbidden}\}$.

It is easy to see that the general BCP is NP-hard. One simple way is to reduce from graph coloring where each node is modeled as a variable that must be assigned a color such that adjacent nodes are not colored identically (i.e., the corresponding constraint on each edge $\{i,j\}$ is a not-equal constraint $\overline{\mathcal{R}_{i,j}} = \{(k,k) \mid \forall\ k\}$). Conversely, every binary constraint problem can be visualized as a *constraint network* where each node corresponds to a variable and an edge connects two nodes iff there exists a constraint over the corresponding variables. Of course, the exact semantic of the constraints is lost in that visualization. However, it is a well-known fact that any BCP whose corresponding constraint network is a tree can be solved in polynomial time. Even more generally, a BCP is already tractable when its constraint network has bounded tree-width [6,7].

In the following, we consider ways to express BCPs by means of linear constraints. We will review a recently introduced IP model for BCPs and show that, for tree-structured BCPs, the model gives a perfect representation of the convex hull of all integer feasible solutions.

3 The Support Formulation

When designing constraints for a model, humans tend to think in terms of "what is forbidden." For BCPs, this leads to the common IP formulation in which we consider each constraint truth table over variables X_i and X_j and add an IP constraint $y_{ik} + y_{jl} \leq 1$ for each inconsistent pair $(k,l) \in \overline{\mathcal{R}_{i,j}}$, where $y_{pq} \in \{0,1\}$ is one iff $X_p = q$ in the solution to the BCP. Since each variable must take exactly one value, we also add constraints $\sum_k y_{ik} = 1$ for all $1 \leq i \leq n$. When the task is constraint optimization rather than constraint satisfaction as in CP, we are also given a linear objective function. The complete IP then reads:

Traditional IP model (T_{IP})

$$
\begin{array}{lll}
\max & \sum p_{ik} y_{ik} & \\
s.t. & y_{j_1 k} + y_{j_2 l} \leq 1 & \forall\ 1 \leq j \leq m, (k,l) \in \overline{\mathcal{R}_{j_1,j_2}} \qquad (1)\\
& \sum_{k \in D_i} y_{ik} = 1 & \forall\ i \in \{1,\ldots,n\} \qquad (2)\\
& y_{ik} \in \{0,1\} & \forall\ i \in \{1,\ldots,n\}, k \in D_i \qquad (3)
\end{array}
$$

In [2], we provided a different way of formulating a BCP as an IP by focussing on the allowed pairs in each constraint truth table. In essence, we use the linear constraints to specify that, when a variable X_{j_1} takes value k, variable X_{j_2} must take a value that is consistent with $X_{j_1} = k$. In that way, we enforce that each variable assignment is *supported* by a correct assignment to adjacent variables (by which we mean variables that share a constraint). The IP then reads:[1]

[1] Whereby, for simplicity in constraints (4), we assume that each constraint over variables i,j induces two truth tables $\mathcal{R}_{i,j}$ and $\mathcal{R}_{j,i}$.

Support IP model (S_{IP})

$$\begin{aligned}
\max \quad & \sum p_{ik} y_{ik} \\
s.t. \quad & y_{j_1 k} - \sum_{l:(k,l)\in\mathcal{R}_{j_1,j_2}} y_{j_2 l} \leq 0 && \forall\, 1 \leq j \leq m, k \in D_{j_1} && (4)\\
& \sum_{k\in D_i} y_{ik} = 1 && \forall\, i \in \{1,\ldots,n\} && (5)\\
& y_{ik} \in \{0,1\} && \forall\, i \in \{1,\ldots,n\}, k \in D_i && (6)
\end{aligned}$$

The fact that support encodings can lead to strong inference algorithms is not new. In [10,8], for example, it was shown that formulating a BCP as a SAT formula in this way has preferable propagation properties. In [2], we showed that the support IP formulation leads to stronger linear continuous relaxations than the Lagrangian relaxation proposed in [11]. Specifically, we showed that the support encoding of the Lagrangian subproblem studied in [2] is totally unimodular. Unfortunately, this property does not hold for the entire IP, even when the corresponding BCP has just two variables (consider for example the case of a two node/one edge graph coloring problem with three colors.) Despite this problem, we will now show that the linear continuous relaxation S_{LP} of S_{IP} (which is given by S_{IP} when replacing integrality constraints (6) with simple constraints on the bounds of the variables; in our case, the upper bounds are redundant and are therefore left out) provides a perfect characterization of the convex hull of all integer feasible solutions.

4 Tree-Structured Binary Constraint Programs

We state our result as follows:

Theorem 1. *If the BCP that is given has a tree-structured constraint network, then the programs S_{IP} and S_{LP} have the same value.*

Proof. Assume we are given a (potentially fractional) solution to S_{LP} (if there is no solution then the corresponding BCP is obviously infeasible, too). Denote this solution by y^0_{ik} with $1 \leq i \leq n$ and $k \in D_i$. Now consider the following sets: $D^0_i := \{k \mid y^0_{ik} > 0\} \subseteq D_i$. We make two important observations:

(a) First, none of the sets D^0_i is empty.
(b) And second, for each $1 \leq j \leq m$ and each value $k \in D^0_{j_1}$, there exists a value $l \in D^0_{j_2}$ such that setting $X_{j_1} = k$ and $X_{j_2} = l$ is allowed. Analogously, for each $1 \leq j \leq m$ and each value $l \in D^0_{j_2}$, there exists a value $k \in D^0_{j_1}$ such that setting $X_{j_1} = k$ and $X_{j_2} = l$ is allowed.

The first is a simple consequence of constraints (5), the latter follows from constraints (4) which enforce that at least one non-conflicting value must still be present in the adjacent variable's domain:

$$0 < y_{j_1 k} \leq \sum_{l:(k,l)\in\mathcal{R}_{j_1,j_2}} y_{j_2 l}.$$

The second property is known in CP as *arc-consistency*. Basically, arc-consistency just states that each value in a variable's domain has supporting values in each of the domains of adjacent variables. It is a well-known fact that a tree-structured BCP has a solution if properties (a) and (b) hold for the domains of the variables. This is easy to verify: Assume that the BCP that is given is arc-consistent. Take any variable and assign to it any value in its domain. Shrink the domains in the remaining BCP by removing all values without support until it is arc-consistent again. Since, for each value in each domain, there exists at least one supporting value in the domains of adjacent variables, no domain can be empty now. So we have properties (a) and (b) again, and we repeat. After at most n such steps all domains have become singletons and we can read out the integer feasible solution y^1.

Therefore, when we artificially shrink the domains of the variables in the given tree-structured BCP by replacing D_i with D_i^0, then the remaining constraint problem is still solvable.

It remains to show that any solution to the reduced BCP where each variable X_i takes a value in D_i^0 has the same objective value as our fractional solution. For this purpose, consider the dual optimal solution with variables $\pi_{jk} \geq 0$ for constraints (4) (and unsigned variables μ_i for constraints (5) that we will not use). Consider the relaxed problem where we soften constraints (4) and penalize their violation in the objective with the help of Lagrange multipliers π:

$$\begin{array}{lll} \max & \sum p_{ik} y_{ik} - \sum_{j,k \in D_j} \pi_{jk} y_{jk} + \sum_{j_1} \sum_{(k,l) \in \mathcal{R}_{j_1,j_2}} \pi_{jk} y_{j_2 l} & \\ s.t. & \sum_{k \in D_i} y_{ik} = 1 & \forall\, i \in \{1,\ldots,n\} \\ & y_{ik} \in \{0,1\} & \forall\, i \in \{1,\ldots,n\}, k \in D_i \end{array}$$

which can be simplified to

Relaxed IP model $(R_{IP}(\pi))$

$$\begin{array}{lll} \max & \sum_{s \leq n, t \in D_s} \overline{p_{st}}\, y_{st} & \\ s.t. & \sum_{k \in D_i} y_{ik} = 1 & \forall\, i \in \{1,\ldots,n\} \quad (7) \\ & y_{ik} \in \{0,1\} & \forall\, i \in \{1,\ldots,n\}, k \in D_i \quad (8) \end{array}$$

when setting $\overline{p_{st}} := p_{st} - \sum_{j_1 = s} \pi_{jt} + \sum_{j_2 = s, \exists\, u:\, (u,t) \in \mathcal{R}_{j_1,j_2}} \pi_{jt}$. From the theory of Lagrangian relaxation we then know the following facts [1, 14]:

- The Lagrangian subproblem exhibits the integrality property. Therefore, with optimal dual multipliers π, the optimal value of $R_{IP}(\pi)$, $R_{LP}(\pi)$, and S_{LP} are all the same.
- Then, the optimal fractional solution y^0 to S_{LP} is also optimal for $R_{LP}(\pi)$. Consequently, because of constraints (7), for all $1 \leq i \leq n$, it holds that $\overline{p_{ik}} = \overline{p_{il}}$ for all

$k, l \in \{t \mid y^0_{it} > 0\}$. That is, all domain values of variable i that received positive weight have the same reduced costs $\overline{p_{ik}}$.

Thus, our integer solution y^1 to S_{IP} for which $y^1_{ik} = 1$ implies $k \in D^0_i$ is feasible and optimal for $R_{IP}(\pi)$. And so, y^1 has the same objective value as y^0. Thus, for each fractional optimal solution there exists an integer optimal solution with the same objective value. □

Since Theorem 1 applies regardless of the objective function, what we have really shown is that the polytope described by S_{LP} is exactly the convex hull of the feasible solutions to S_{IP}. Note that there are certainly other ways to prove this result. However, this is the only proof we know which reveals a connection between the CP variable domains and the support (i.e. the set of those LP variables variables that have positive weight) of the LP relaxation.

So we can efficiently describe the convex hull of IPs modeling tree-structured BCPs. Since tree-structured problems are known to be polynomial-time solvable, this is not really surprising. However, the characterization of the convex hull has several advantages over specialized search procedures:

- In many cases, binary constraints constitute only a part of the constraint structure. Specialized search procedures that work for the BCP part do not generalize to the problem at hand. However, the tight description of the polytope of feasible solutions can still be exploited in a branch-and-bound procedure.
- Assume that the overall problem is not tree-structured, but becomes tree-structured after a couple of branching decisions. In practice, we do not want to put large overhead into the recognition of special cases that can be solved by a specialized search routine. When using the support formulation, special recognition is not necessary - the relaxation will automatically come out integer when using the simplex algorithm to solve the linear relaxation.

5 Exploiting Bounded Tree-Width

Obviously, we would like to generalize our results to BCPs with bounded tree-width which are also known to be polynomial-time tractable. We do this in two steps: First, we propose a set of constraints that define the convex hull of a BCP with bounded tree-width k when a tree-decomposition is known. In the second step, we then show how to achieve a perfect bound when the decomposition is not known.

Definition 2. *An undirected graph $G = (N, A)$ has* tree-width *k, iff there is a pair (S, T), where $S = \{S_1, \ldots, S_p\}$ is a family of subsets of N, and T is a tree whose nodes are the subsets S_i, such that:*

- $\bigcup_i S_i = N$,
- *for every edge $\{v, w\} \in A$, there is a node S_i that contains both v and w,*
- *if S_i and S_j both contain a vertex v, then all nodes S_z of the tree in the (unique) path between S_i and S_j contain v, and*
- *the size of all sets S_i is limited by $k + 1$, i.e., $|S_i| \leq k + 1$ for all i.*

A BCP has bounded tree-width k *when its constraint network has tree-width* k.

It is known that a BCP with bounded tree-width k can be transformed into a tree-structured BCP that contains the original variables as well as some additional auxiliary variables such that the size of the new BCP is polynomial in the original size (and exponential in the constant k). We can compute this new BCP by exploiting the tree-decomposition of the given problem. Note that, while computing the minimal tree-width of a graph is NP-hard, for fixed k we can efficiently check whether a graph has tree-width k and compute a corresponding tree-decomposition in linear time [5].

The construction of the new BCP is simple: the variables are the old variables $X_1, \ldots, X_n$ plus new variables $Y_1, \ldots, Y_p$, one for each set $S_i = \{X_{i_1}, \ldots, X_{i_{q(i)}}\}$ in the tree-decomposition (whereby $1 \leq q(i) \leq k+1$). The domain of variable Y_i are all tuples $(t_1, \ldots, t_{q(i)}) \in D_{i_1} \times \cdots \times D_{i_{q(i)}}$ that are *consistent with all binary constraints* over any two variables in S_i. The domains of the old variables are the same as before. Now, we add two sets of binary constraints to the problem:

- For each variable X_j there exists a set S_i such that $X_j \in S_i$. For one such set (ties can be broken arbitrarily), we add a constraint over X_j and Y_i that enforces that the value of X_j and the corresponding entry in the tuple assigned to Y_i are the same.
- For each pair of variables Y_h, Y_i such that $S_h \cap S_i \neq \emptyset$, we add a constraint that enforces that entries in the corresponding tuples that are associated with the same variable(s) X_j are identical.

Note how the restrictions that apply to a valid tree-decomposition ensure that the new BCP is connected, cycle-free, and polynomial in the size of the original problem when k is viewed as a constant. Most importantly, a valid tree-decomposition ensures that a solution to the original problem has a corresponding solution in the new BCP and vice versa: Given a solution σ to the original problem, we achieve a solution to the new one simply by assigning the same values to the old variables as before and by assigning the corresponding consistent tuples to the variables Y_i. Since σ obeys all original constraints, each such tuple is a member in the domain of Y_i. Conversely, assume we are given a solution τ to the new problem. Note that, for all X_j, the variables Y_i with $X_j \in S_i$ and X_j are all connected. Consequently, all tuples assigned to variables Y_i and the assignment to X_j agree on the value of X_j. Also, for each original constraint C_j on variables X_{j_1}, X_{j_2} there is at least one Y_i such that $X_{j_1}, X_{j_2} \in S_i$. Therefore, τ assigns a consistent pair of values to X_{j_1} and X_{j_2}. Consequently, when projecting τ on variables X_j, we achieve a feasible solution to the original problem.

Now, all that we need to do is formulate the new, tree-structured BCP by exploiting the support formulation as outlined in Section 3. Then, all extreme points of the new polytope give integer feasible solutions that are consistent with the original problem. It follows:

Theorem 2. *Given a constant* k*, any BCP with bounded tree-width* k *can be expressed as an integer program that exhibits the integrality property and which size is polynomial in the size of the given problem.*

Note that our formulation still contains all original variables which makes it easy to add the objective to the problem and which also allows us to augment the problem by adding

additional constraints on those original variables. Obviously, any additional constraints will, in general, compromise the integrality property of the feasible polytope. However, with the tight characterization of the convex hull of all solutions feasible for the BCP we can hope for a small LP/IP gap.

Now, in some cases we would prefer if we did not need to know the exact tree-decomposition of our problem. For example, it would be desirable if, during a branch-and-bound search, inference on a sub-problem encountered during search would simply turn out perfect whenever the sub-problem has bounded tree-width k, without the need of applying a special recognition algorithm for all sub-problems. That is, rather than analyzing a problem, determining its tree-width, and then choosing the formulation, it would be nicer if we could simply fix the parameter k (as an algorithm design choice) and then be sure that all subproblems of width lower than k that we may ever encounter will be solved perfectly by our inference algorithm. Fortunately, this can be achieved!

First, the following simple claim allows us to restrict our attention to a certain class of tree decompositions, that we call *saturated*.

Lemma 1. *Let $G = (N, A)$ be an undirected graph of tree-width less than or equal to k, with $|N| > k$. There exists a tree decomposition (S, T) of G such that $|s| = k + 1$ for all $s \in S$.*

Proof. First, we may assume there is no edge $\{S_i, S_j\} \in T$ such that $S_i \subseteq S_j$ since we can get a new tree decomposition (S', T') by contracting this edge and keeping only S_j. Now, if $|S| = 1$, then the tree consists in one node containing all $k + 1$ elements of N. So assume $|S| > 1$, and there is $S_i \in S$ such that $|S_i| \leq k$. Let S_j be adjacent to S_i. Since there exists $p \in S_j \setminus S_i$, we can augment the cardinality of S_i by adding p which gives a new valid tree-decomposition. By repetition, we achieve a decomposition where all nodes have cardinality $k + 1$. □

Now, we formulate a new BCP whose solutions closely correspond to solutions in the original problem. The construction is very easy: We add all original variables X_j to our problem, keeping their domains as before. Additionally, for each subset of exactly $k+1$ variables $\{X_{s_1}, \ldots, X_{s_{k+1}}\}$, we introduce a variable $Y_{s_1,\ldots,s_{k+1}}$. The domains of these variables are limited to tuples $(t_1, \ldots, t_{k+1}) \in D_{s_1} \times \cdots \times D_{s_{k+1}}$ which are *consistent with all binary constraints* over variables X_{j_1}, X_{j_2} with $j_1, j_2 \in \{s_1, \ldots, s_{k+1}\}$. We add the same binary constraints on the auxiliary variables as we did when the tree-decomposition was known. The original variables X_j are tied to our problem by adding a constraint between X_j and each variable $Y_{s_1,\ldots,s_{k+1}}$ with $j = s_r$ for some $1 \leq r \leq k + 1$, enforcing that the value assigned to X_j is the same as the entry with index s_r of the tuple assigned to $Y_{s_1,\ldots,s_{k+1}}$.

Let us denote the BCP that emerges in this way from a BCP $\mathcal{P}$ by $\mathcal{T}_k(\mathcal{P})$. Clearly, $\mathcal{T}_k(\mathcal{P})$ is not tree structured at all. However, it holds:

Theorem 3. *Given a constant k and a BCP $\mathcal{P}$ with tree-width lower or equal k, the support encoding of $\mathcal{T}_k(\mathcal{P})$ as an integer program is polynomial in the size of $\mathcal{P}$and it has the same value as its linear continuous relaxation.*

Proof. We consider the following integer programs and their linear continuous relaxations: IP_T, the integer programming formulation based on the saturated tree-

decomposition from Theorem 2; LP_T, the linear continuous relaxation of IP_T; IP_F, the support encoding of $\mathcal{T}_k(\mathcal{P})$ as an integer program; and LP_F, the linear continuous relaxation of the latter. By abuse of language, we identify the optimal value of the objective with the name of a problem. According to Theorem 1, it holds that $\mathrm{IP}_T = \mathrm{LP}_T$. Furthermore, we observe that LP_F is, in some sense, a lifted version of LP_T: LP_F operates on a super-set of variables of LP_T, but extra variables present in LP_F yield no additional profit in the objective function, and all original constraints are still present. Consequently, LP_F just contains some extra constraints, and it holds $\mathrm{LP}_T \geq \mathrm{LP}_F$. The same relation also holds for IP_F and IP_T. However, since all extra constraints present in IP_F are redundant for any integer solution in IP_T, we even have that $\mathrm{IP}_F = \mathrm{IP}_T$. But then: $\mathrm{LP}_T \geq \mathrm{LP}_F \geq \mathrm{IP}_F = \mathrm{IP}_T = \mathrm{LP}_T$. And thus, $\mathrm{LP}_F = \mathrm{IP}_F$. □

It is interesting to observe the analogies of our method to the work from Bienstock and Ozbay in [4]. They show that the polytope of packing problems whose matrices have a clique-graph with bounded tree-width can be described perfectly by adding a polynomial number (exponential in the tree-width) of Sherali-Adams variables and cuts [16]. This work does not apply directly to the matrices that we encounter (even when we ignore negative matrix entries, note that the size of BCP domains is usually in the same order as the number of variables which causes the clique-graphs to have large tree-width). However, it is interesting to see that in both their work and in our approach the introduction of variables that model subsets of the original variables leads to the desired result.

6 Numerical Results for Augmented BCPs with a Linear Objective

In [2], we showed that LP-inference is not worthwhile for pure BCPs. Here, we will experiment with BCPs that are augmented by some linear constraints and a linear objective function. For this purpose, we consider the following multi-knapsack problem: Given m knapsacks with capacities $C_1, \ldots, C_m$ and n items with knapsack-dependent weights $w_{1,1}, \ldots, w_{m,n} \geq 0$, we have to place each item in exactly one knapsack such that the sum of the weights of the items placed in each knapsack does not exceed its capacity. Items achieve knapsack-dependent profits $p_{1,1}, \ldots, p_{m,n} \geq 0$, and we try to maximize the total profit $\sum_{k=1}^{m} \sum_{i \text{ placed in } k} p_{k,i}$. We augment this problem by adding binary constraints that each forbid some specific simultaneous placements of two items. For instance, a constraint over items i and j could require that i and j cannot be placed in the same bin. Or that they must be placed in the same bin. Or generally, that placing item i in knapsack k_1 and placing item j in knapsack k_2 is not allowed for a set of tuples (k_1, k_2). For our experiments, we compare the following models.

$$
\begin{array}{lll}
(\mathrm{CP}): \max \sum_{i=1}^{n} p_{x_i,i} & & \\
(1) & \sum_{i \leq n,\ x_i = k} w_{k,i} \leq C_k & \forall\ 1 \leq k \leq m \\
(2) & F_p(x_{p_1}, x_{p_2}) = true & \forall\ 1 \leq p \leq q \\
(3) & x_i \in \{1, \ldots, m\} & \forall\ 1 \leq i \leq n
\end{array}
$$

where $F_1, \ldots, F_q$ denote the binary constraints and constraint F_p limits the simultaneous placement of items p_1 and p_2. The next model is based on the traditional LP formulation:

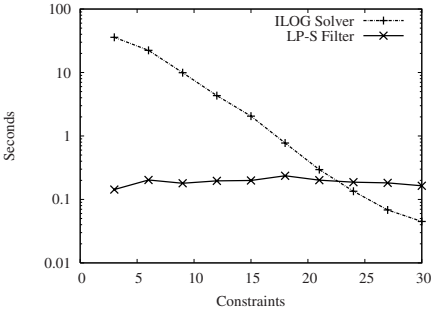


Fig. 1. Comparison of the average time [sec, log-scale] over 100 random instances to solve 5-knapsack instances with 12 items and varying number of binary constraints with a pure CP approach (Ilog Solver) and our LP-filtering approach with support formulation.

$$
\begin{array}{lll}
(\mathrm{LP_T}): \sum_{k=1}^{m} \sum_{i=1}^{n} p_{k,i}\ y_{k,i} & & \\
(1) & \sum_{i \leq n} w_{k,i}\ y_{k,i} \leq C_k & \forall\ 1 \leq k \leq m \\
(2) & y_{k,p_1} + y_{l,p_2} \leq 1 & \forall\ 1 \leq p \leq q,\ F_p(k,l) = false \\
(3) & \sum_k y_{k,i} = 1 & \forall\ 1 \leq i \leq n \\
(4) & y_{k,i} \in \{0,1\} & \forall\ 1 \leq k \leq m, 1 \leq i \leq n
\end{array}
$$

Our last model is based on the support LP formulation:

$$
\begin{array}{lll}
(\mathrm{LP_S}): \sum_{k=1}^{m} \sum_{i=1}^{n} p_{k,i}\ y_{k,i} & & \\
(1) & \sum_{i \leq n} w_{k,i}\ y_{k,i} \leq C_k & \forall\ 1 \leq k \leq m \\
(2a) & y_{k,p_1} \leq \sum_{l \mid F_p(k,l)=\mathrm{true}} y_{l,p_2} & \forall\ 1 \leq p \leq q,\ 1 \leq k \leq m \\
(2b) & y_{l,p_2} \leq \sum_{k \mid F_p(k,l)=\mathrm{true}} y_{k,p_1} & \forall\ 1 \leq p \leq q,\ 1 \leq l \leq m \\
(3) & \sum_k y_{k,i} = 1 & \forall\ 1 \leq i \leq n \\
(4) & Y_{k,i} \in \{0,1\} & \forall\ 1 \leq k \leq m,\ 1 \leq i \leq n
\end{array}
$$

In combination with the latter two models, we use two different methods for exploiting the corresponding LP relaxation. The first is called *LP-pruning* which uses it for pruning purposes (i.e. the early termination of search) only. The second is inspired by [11] and called *LP-filtering*: after computing the LP-solution to the problem, it chooses those assignments $X_i = k$ for which the continuous value of y_{ik} is lower than some threshold value $\varepsilon > 0$. Then, for each of the selected assignments, it sets up a new LP with the objective to maximize y_{ik}. If the relaxation gives a value lower than 1, k can be removed from D_i.

We generate random instances for given parameters m, n, and q by drawing weights $w_{k,i}$ uniformly at random between 1 and 100. Knapsack capacities are then set to $C_k := 2\ \frac{\sum_i w_{k,i}}{m}$. The profits $p_{k,i}$ were weakly correlated with the weights by drawing them uniformly at random in the intervals $[w_{k,i}-5, w_{k,i}+5]$. Binary constraints are generated randomly, whereby the number of allowed pairs of each constraint is set to $m^2/2$.

Figure 1 shows a comparison of a pure CP approach and our LP-filtering approach for random 5-knapsack instances with 12 items. We can see clearly how essential the use of a global bound is: even on this toy-example, in the low-constrained region solver needs, on average, more than 35 seconds while the LP-filtering approach takes less than

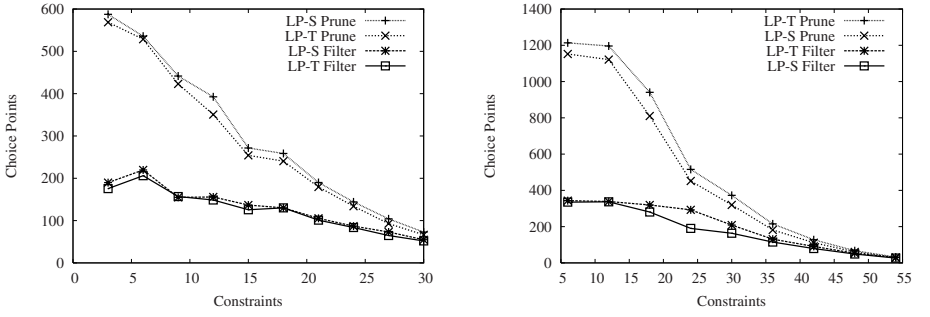


Fig. 2. Comparison of the average number of choice points over 100 random instances to solve 5-knapsack instances with 12 and 16 items and varying number of binary constraints

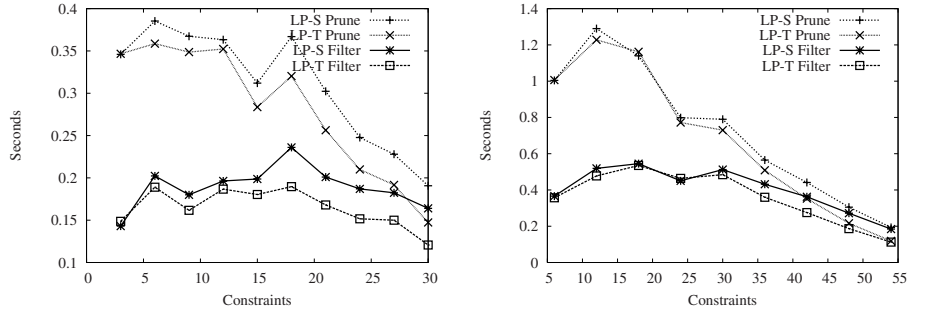


Fig. 3. Comparison of the average time over 100 random instances needed to solve 5-knapsack instances with 12 and 16 items and varying number of binary constraints

half a second. We also started runs on instances with 16 items, but the pure CP-approach took so much time that we had to cancel the experiment.

Regarding the effect of the support model and the traditional formulation of the binary constraints, in Figure 2 we compare the number of choice points visited by LP-pruning and filtering when based on the support or traditional formulation. We observe what was to be expected: LP filtering visits fewer choice points than LP-pruning, and for both pruning and filtering, the support formulation is stronger and results in smaller search trees than the traditional one.

Of course, LP-filtering incurs larger computational costs per choicepoint than LP-pruning. Whether or not the additional time needed to perform stronger inference will in general depend on the application. Figure 3 shows that LP-based filtering beats the approach that uses the LP-bound for pruning only. We observe that, on small multi-knapsack instances, using the support formulation does not pay off. This can be attributed to the fact that the traditional formulation leads to much sparser matrices and can therefore be solved much faster, which makes up for slightly worse relaxation values. Of course, as problem sizes and search spaces grow, the use of a stronger bound becomes more and more attractive as the larger costs per choice point can be paid for by

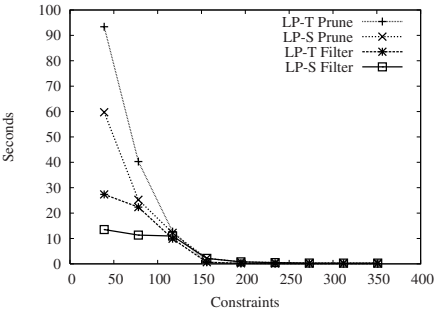


Fig. 4. Comparison of the average time over 100 random instances to solve 5-knapsack instances with 40 items and varying number of binary constraints

a much larger reduction in search costs. Consequently, in Figure 4, we see that already on moderately larger problem instances filtering based on the support LP-formulation becomes the method of choice.

7 Summary and Future Work

We have shown how to efficiently transform binary constraint problems into linear programs whose extreme points are integer whenever the tree-width of the initial BCP is bounded by some constant k.

Some questions remain open. When making the step from tree-structured BCPs to those with tree-width 2, the exponent in the number of CP variables jumps from 1 to 3. Can the exponents be controlled better than we did it here? Are there efficient ways to generate constraints in a lazy fashion, adding them only when the linear relaxation turns out to be fractional? What about linear programming polytopes of other islands of tractability, such as Horn formulas?

Moreover, our approach motivates a procedure for lifting arbitrary binary integer programs $\{x \in \{0,1\}^n \mid Ax \leq b\}$ that would be interesting to compare with the Sherali-Adams procedure.[2] To simplify the notation, all sets considered in the following are subsets of $\{1,\ldots,n\}$, and we write $X + Y$ for the union of sets X, Y that are disjoint. At level $k \geq 1$, to the original problem we add variables $w(Y,N) \geq 0$ for all $|Y + N| = k$ (with the idea that $x_j = 1$ for all $j \in Y$ and $x_h = 0$ for all $h \in N$). Moreover, we add the following sets of constraints:

- $\sum_{Y+N=S} w(Y,N) = 1$ for all $|S| = k$.
- $w(Y,N) \leq x_j$ and $w(Y,N) \leq x_h$ for all $|Y + N| = k$, $j \in Y$, $h \in N$.
- $x_j \leq \sum_{Y+N=S,\ j\in Y} w(Y,N)$ and $1 - x_j \leq \sum_{Y+N=S,\ j\in N} w(Y,N)$ for all $|S| = k$, $j \in S$.

[2] Whereby it is important to state explicitly that the lifting procedure that we sketch here, due to potentially large CP variable domains, would not have given the desired results when applied to S_{IP}. We really needed to exploit the original structure of the given BCP with bounded tree-width to achieve a good IP model.

- $w(X+Y, M+N) \leq \sum_{Z+O=S} w(X+Z, M+O)$ for all $1 \leq |X+M| < k$, $|S| = |Y+N|$, $S \cap (Y+N) = \emptyset$, and $|X+Y+M+N| = k$.
- $(\sum_{j \in Y} a_{ij} + \sum_{j \notin Y+N,\ a_{ij}<0} a_{ij} - b_i) w(Y,N) \leq 0$ for all $|Y+N| = k$.

Note that the variables that we add are semantically the same as the ones that are added by Sherali and Adams. However, the way we post the constraints is quite different so that it suffices to add variables for subsets of size equal to k only. As the results in this paper show, at level n, this lifting method gives a formulation that has integer extreme points only.

References

1. R.K. Ahuja, T.L. Magnati, J.B. Orlin. *Network Flows.* Prentice Hall, 1993.
2. I.D. Aron, D.H. Leventhal, M. Sellmann. A Totally Unimodular Description of the Consistent Value Polytope for Binary Constraint Programming. *CPAIOR*, LNCS 3990:16–28, 2006.
3. D. Bienstock. Approximate formulations for 0-1 knapsack sets. *CORC Report TR-2006-03*, Columbia University, 2006.
4. D. Bienstock and N. Ozbay. Tree-width and the Sherali-Adams operator. `citeseer.ist.psu.edu/bienstock03treewidth.html`, 2003.
5. H.L. Bodlaender. A Linear Time Algorithm for Finding Tree-decompositions of Small Treewidth. *SIAM Journal on Computing*, 25:1305–1317, 1996.
6. R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.
7. E.C. Freuder. Complexity of k-tree structured constraint satisfaction problems. *AAAI*, pp. 4–9, 1990.
8. I.P. Gent. Arc Consistency in SAT. *ECAI*, pp. 121–125, 2002.
9. J.N. Hooker. A hybrid method for planning and scheduling. *CP*, LNCS 3258:305–316, 2004.
10. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45:275–286, 1990.
11. M.O.I. Khemmoudj, H. Bennaceur, A. Nagih. Combining Arc-Consistency and Dual Lagrangean Relaxation for Filtering CSPs. *CPAIOR*, LNCS 3524:258–272, 2005.
12. R. Marinescu and R. Dechter. AND/OR Branch-and-Bound Search for 0-1 Integer Linear Programming. *CPAIOR*, LNCS 3990:152–166, 2006.
13. M. Milano. *Integration of Mathematical Programming and Constraint Programming for Combinatorial Optimization Problems*, *Tutorial at CP*, 2000.
14. G.L. Nemhauser and L.A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, 1988.
15. N. Robertson and P.D. Seymour. Graph minors - Algorithmic aspects of treewidth. *Algorithms*, 7:309–322, 1986.
16. H.D. Sherali and W.P. Adams. A hierarchy of relaxations between the continuous and convex hull representations for zero-one programming problems. *SIAM Journal on Discrete Mathematics*, 3:411–430, 1990.
17. M. Van Vyve, L.A. Wolsey. Approximate extended formulations. *Mathematical Programming*, 105(2–3):501–522, 2006.

On Boolean Functions Encodable as a Single Linear Pseudo-Boolean Constraint

Jan-Georg Smaus

Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 52, 79110 Freiburg im Breisgau, Germany
smaus@informatik.uni-freiburg.de

Abstract. A *linear pseudo-Boolean constraint* (LPB) is an expression of the form $a_1 \cdot \ell_1 + \ldots + a_m \cdot \ell_m \geq d$, where each ℓ_i is a *literal* (it assumes the value 1 or 0 depending on whether a propositional variable x_i is true or false) and $a_1, \ldots, a_m, d$ are natural numbers. An LPB is a generalisation of a propositional clause, on the other hand it is a restriction of *integer linear programming*. LPBs can be used to represent Boolean functions more compactly than the well-known *conjunctive* or *disjunctive* normal forms. In this paper, we address the question: *how much* more compactly? We compare the expressiveness of a single LPB to that of related formalisms, and give an algorithm for computing an LPB representation of a given formula if this is possible.

1 Introduction

A *linear pseudo-Boolean constraint* (LPB) [1,3,5,6,7,8] is an expression of the form $a_1\ell_1 + \ldots + a_m\ell_m \geq d$. Here each ℓ_i is a *literal* of the form x_i or $\bar{x}_i \equiv 1 - x_i$, i.e. x_i becomes 0 if x_i is false and 1 if x_i is true, and vice versa for $\bar{x}_i$. Moreover, $a_1, \ldots, a_m, d$ are natural numbers.

An LPB can be used to represent a Boolean[1] function; e.g. $x_1 + \bar{x}_2 + x_3 \geq 3$ represents the same function as the propositional formula $x_1 \wedge \neg x_2 \wedge x_3$ (we identify propositional formulae with functions). It has been observed that a function can be often represented more compactly as a set of LPBs than as a *conjunctive* or *disjunctive* normal form (CNF or DNF) [5,6,7,8]. E.g. the LPB $2x_1 + \bar{x}_2 + x_3 + x_4 \geq 2$ corresponds to the DNF $x_1 \vee (\neg x_2 \wedge x_3) \vee (\neg x_2 \wedge x_4) \vee (x_3 \wedge x_4)$.

Previous works on LPBs [1,5,6,7,8] have focused on generalising techniques applied in CNF-based propositional satisfiability solving [12,13,21] to LPBs, emphasising that this is beneficial because of the compactness of LPB representations. Dixon and Ginsberg show that since LPBs are a special case of *integer programming*, the *cutting planes* proof system, a standard technique in operations research (OR), can be applied to LPBs. Cutting planes is a generalisation of *resolution*, a standard technique in artificial intelligence (AI). Cutting planes proofs can be exponentially shorter than resolution proofs [6].

But where do the LPBs come from? One possibility is that for an application domain, one gives a direct representation of a problem as a *set* of LPBs (usually

[1] Whenever we say "function" we mean "Boolean function".

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 288–302, 2007.

interpreted as conjunction but also a disjunction is thinkable) and argues that the alternative representation as CNF would be less compact [1,6,8]. Another possibility is that one considers problem representations given as a CNF or DNF and transforms these into compact LPB representations. We are not aware that the latter has ever been proposed. In addition, except [8] (discussed in Subsec. 6.1), all the arguments that we found in favour of LPBs were not *strictly* about LPBs but about *cardinality constraints*, which are a subclass of LPBs. This raises the question: how can a propositional formula be transformed into an LPB representation that is as compact as possible? As a first but crucial step towards this aim, we believe that one should study the question which functions can be expressed by *a single* LPB, i.e. whether or not a given CNF or DNF represents a *threshold function* [15]. This is the topic of this paper.

In Sec. 3 we show that there is an inclusion chain from cardinality constraints to LPBs to the *monotone* functions (functions represented by a formula where each variable occurs only in one polarity). In Sec. 4 we recall the difficulty of determining the number of monotone functions, and give some results on the cardinality of classes of functions. We give an upper bound for the blowup of using a DNF instead of an LPB encoding. In Sec. 5, we show that if a DNF can be expressed by an LPB, then the dual CNF can be expressed by a very similar LPB, and vice versa. In Sec. 6 we give a theorem that states that ϕ can be represented as an LPB if and only if ϕ can be decomposed into several smaller formulae, each of which can be represented by an LPB, and all these LPBs are in a certain sense very similar. Based on this theorem we give an algorithm for converting a DNF ϕ to an LPB if possible. The proofs of all results of this paper can be found in [18].

2 Preliminaries

We assume the reader to be familiar with the basic notions of propositional logic.

An **m-dimensional Boolean function** f is a function $Bool^m \to Bool$. We say that f **properly depends** on the ith argument if there exist $\beta \in Bool^{i-1}$, $\beta' \in Bool^{m-i}$ with $f(\beta, 0, \beta') \neq f(\beta, 1, \beta')$.

We follow [5]. A **0-1 ILP constraint** is an inequality of the form

$$a_1 x_1 + \ldots + a_m x_m \geq d \qquad a_i, d \in \mathbb{R}, x_i \in Bool \ (Bool \equiv \{0,1\}). \tag{1}$$

We identify 0 with *false* and 1 with *true*. We call the a_i **coefficients** and d the **degree** [9]. Using the relation $\bar{x}_i \equiv 1 - x_i$ and noting that it is sufficient to consider integer coefficients, one can rewrite a 0-1 ILP constraint as a **linear pseudo-Boolean constraint** (LPB)

$$a_1 \ell_1 + \ldots + a_m \ell_m \geq d \qquad a_i \in \mathbb{N}, d \in \mathbb{Z}, \ell_i \in \{x_i, \bar{x}_i\}. \tag{2}$$

For example, $x_1 - 0.5x_2 - 0.5x_3 \geq 0$ can be written as $2x_1 + \bar{x}_2 + \bar{x}_3 \geq 2$. An occurrence of a **literal** x_i (resp., $\bar{x}_i$) is called an occurrence of x_i in **positive** (resp., **negative**) polarity. Note that if $d \leq 0$, then the LPB is a tautology. The reason for allowing for negative d will become apparent in Subsec. 6.2.

An LPB where $a_i = 1$ or $a_i = 0$ for all $i \in [1..m]$ is called a **cardinality constraint** (e.g. for $m = 4$: $1x_1 + 0x_2 + 1x_3 + 0x_4 \geq 1$, in short $x_1 + x_3 \geq 1$). Note that $\sum_{i \in J} \ell_i \geq 1$ (resp., $\sum_{i \in J} \ell_i \geq |J|$) corresponds to $\bigvee_{i \in J} \ell_i$ (resp., $\bigwedge_{i \in J} \ell_i$).

A **CNF** is a formula of the form $c_1 \wedge \ldots \wedge c_n$ where each **clause** c_j is a disjunction of literals. A **DNF** is defined dually; a conjunction of literals is called a **(dual) clause**. Formally, CNFs and DNFs are sets of sets of literals, i.e. the order of clauses and the order of literals within a clause are insignificant. For CNFs and DNFs, we assume without loss of generality that no clause is a subset of another clause (the latter clause would be redundant since it is *absorbed*). Given a CNF, the **dual** DNF is obtained by swapping $\wedge$ and $\vee$. Any Boolean function can be represented by a CNF or DNF [20].

An **assignment** σ is a mapping $\{x_1, \ldots, x_m\} \to \mathit{Bool}$. The notion '$\sigma$ **satisfies** an LPB I' is defined as expected [7].

3 Inclusion Results

The results of this section are not difficult but provide some useful insights into the expressiveness of an LPB or cardinality constraint.

Following [19], we define **monotone** functions as follows.

Definition 3.1. A function is **monotone** (or *unate* [5]) if it can be written as $\vee, \wedge$-combination of literals, where each variable occurs in only one polarity. A monotone function is **isotone** if all variables occur in positive polarity.

It turns out that the polarity of a particular variable is an issue that is orthogonal to the results of this section: each monotone function has 2^m variants obtained by modifying the polarity of each variable. Thus we assume here without loss of generality that each variable has positive polarity.

We say that assignment σ **minimally** satisfies the LPB I if σ satisfies I and any assignment obtained from σ by changing any variable occurring in I from *true* to *false* does not satisfy I. We say that a dual clause **corresponds** to an assignment if it consists of the variables assigned *true* by σ.

Proposition 3.2. An LPB I represents the DNF that consists of exactly those dual clauses that correspond to assignments that minimally fulfill I.

We now give an inclusion result between the functions representable as a single LPB and monotone functions.

Lemma 3.3. Every LPB represents a monotone function. For $m \geq 4$, there is at least one monotone function not represented by any LPB. For $m \leq 3$, each monotone function can be represented as LPB.

We now give an inclusion result between LPBs and cardinality constraints.

Lemma 3.4. Every cardinality constraint is an LPB. For $m \geq 3$, there is at least one LPB not expressible as cardinality constraint. For $m \leq 2$, each monotone function can be represented as a cardinality constraint.

Summarising, we have "cardinality constraints" $\subseteq$ "LPBs" $\subseteq$ "monotone functions", where these inclusions are strict except for very small dimensions.

4 Counting Boolean Functions

For comparing the expressiveness of formalisms for Boolean functions, it is of interest to compare the cardinalities of the function classes that can be expressed by the formalisms. Note, however, that from such comparisons we cannot infer how much blowup there is when translating from one formalism to another. We will come back to this point at the end of this section.

Proposition 4.1. There are $2^{(2^m)}$ m-dimensional Boolean functions [20].

The question of how many monotone functions there are is called *Dedekind's problem*, unsolved for more than a century. To be precise, Dedekind's problem is to determine the number of *iso*tone m-dimensional functions (*Dedekind numbers*). Confusingly, what we call *iso*tone is sometimes called *mono*tone, but we use the terminology of [19]. Nobody has found yet a closed form expression for the Dedekind numbers. In 1999, they have been calculated for up to $m = 8$, where the value is 56130437228687557907788. The Dedekind numbers are Sequence A000372 of [16]. Although the number of isotone functions is large, it is a small fraction of the number $2^{(2^m)}$ of Boolean functions [19]. The best known bound for the Dedekind numbers is given by [11].

Table 1. Some cardinalities

m	$\mathfrak{I}^{=}(m)$	$\mathfrak{I}^{\leq}(m)$	$\mathfrak{M}^{=}(m)$	$\mathfrak{M}^{\leq}(m)$
0	2	2	2	2
1	1	3	2	4
2	2	6	8	14
3	9	20	72	104
4	114	168	1824	2170

We show that the number of *mono*tone functions is related to the number of isotone functions, so that finding a closed form expression for the former cannot be easier than for the latter. We need the following notations.

Definition 4.2. We denote by $\mathfrak{I}^{\leq}(m)$ the number of m-dimensional isotone functions (Dedekind numbers); by $\mathfrak{I}^{=}(m)$ the number of isotone functions that *properly* depend on m variables; and by $\mathfrak{M}^{\leq}(m)$, $\mathfrak{M}^{=}(m)$ the corresponding numbers of *mono*tone functions.

Lemma 4.3. The following identities hold:

$$\mathfrak{I}^{\leq}(m) = \sum_{i=0}^{m} \binom{m}{i} \mathfrak{I}^{=}(i) \tag{3}$$

$$\mathfrak{M}^{=}(m) = 2^m \mathfrak{I}^{=}(m) \tag{4}$$

$$\mathfrak{M}^{\leq}(m) = \sum_{i=0}^{m} \binom{m}{i} \mathfrak{M}^{=}(i) \;=\; \sum_{i=0}^{m} \binom{m}{i} 2^i \mathfrak{I}^{=}(i) \tag{5}$$

Table 1 shows some of the values.

The number of LPBs describing distinct m-dimensional functions will probably not be easier to describe than the Dedekind numbers [14, p. 64] [15]. It is not difficult though to make a statement about cardinality constraints.

Lemma 4.4. There are $2+\sum_{k=1}^{m}\binom{m}{k}\cdot 2^k\cdot k$ cardinality constraints representing distinct m-dimensional functions.

Also it is not difficult to make a statement about (dual) clauses.

Proposition 4.5. There are 3^m m-dimensional functions expressible as clauses, and likewise for dual clauses.

In analogy to Prop. 4.5, one can give a loose upper bound $3^m\cdot m$ for the number of cardinality constraints, since the degree can be between 1 and m. So the number of cardinality constraints is at most a linear factor above that of usual clauses. However, encoding one cardinality constraint as CNF can entail an exponential blowup in formula size (not considering encodings involving auxiliary variables, encodings which are not equivalence preserving). More precisely, encoding $x_1+\ldots+x_m\geq k$ requires $\binom{m}{(m-k)+1}=\binom{m}{k-1}$ clauses of length $m-k+1$ as CNF [3] and $\binom{m}{k}$ dual clauses of length k as DNF (in [7], this is said for CNF but in fact it should be DNF). Note that $\binom{m}{\lfloor m/2\rfloor}\geq 2^{m/2}$.

The blowup when encoding an *LPB* as CNF or DNF is not worse however.

Lemma 4.6. Let $I\equiv\sum_{i=1}^{m}a_ix_i\geq d$ be an LPB. The DNF (CNF) ϕ represented by I has at most $\binom{m}{\lfloor m/2\rfloor}$ clauses.

Thus, an LPB can represent *more* DNFs than a cardinality constraint but not *bigger* DNFs. For example, $3x_1+2x_2+2x_3+x_4\geq 4$ represents a DNF of 4 dual clauses, while $2x_1+2x_2+2x_3+2x_4\geq 4$ (which is effectively a cardinality constraint) represents a DNF of 6 dual clauses.

Note that the CNF or DNF corresponding to an LPB must be distinguished from translations of an LPB that introduce additional variables [2].

5 Duality

We show that if a DNF can be represented as an LPB, then the dual CNF can also be represented as an LPB, and the two LPBs are closely related. As in Sec. 3, we assume that each variable has positive polarity.

Theorem 5.1. If a DNF is represented by an LPB $I\equiv\sum_{i=1}^{m}a_ix_i\geq d$, then the dual CNF is represented by $\sum_{i=1}^{m}a_ix_i\geq\sum_{i=1}^{m}a_i+1-d$, and vice versa.

Note the border cases: $\sum_{i=1}^{m}a_ix_i\geq\sum_{i=1}^{m}a_i$ represents a conjunction (of variables), $\sum_{i=1}^{m}a_ix_i\geq 1$ represents a disjunction.

Example 5.2. Consider $5x_1+2x_2+2x_3+2x_4\geq i$ for $i\in[1..11]$. Note first that for $i=1,2$ the represented function is the same, and the dual of that function is represented by setting $i=11,10$. Similarly one has $i=3,4$ vs. $i=9,8$. For $i=5$, the DNF is $x_1\vee(x_2\wedge x_3\wedge x_4)$, and the dual CNF $x_1\wedge(x_2\vee x_3\vee x_4)$ is represented by setting $i=7$. For $i=6$, the LPB represents $(x_1\wedge x_2)\vee(x_1\wedge x_3)\vee(x_1\wedge x_4)\vee(x_2\wedge x_3\wedge x_4)$. According to Thm. 5.1, since $12-6=6$, the dual CNF is represented by the same LPB, which means that the CNF is equivalent to its dual. This can easily be confirmed.

6 Representing a DNF as LPB

In this section we present an algorithm for the problem of converting a DNF to an equivalent LPB if possible.[2] Any results of this section can be applied to CNFs rather than DNFs using Sec. 5. In this section, by a *clause* we always mean a *dual* clause. As before, we assume that each variable has positive polarity.

6.1 Determining the Order of Coefficients

Given a DNF ϕ, one can determine a size order of the potential coefficients of an LPB representing ϕ. That is to say, if ϕ can be represented as an LPB at all, then the coefficients must respect this order.

The following notion is useful for reasoning about the structure of a formula.

Definition 6.1. Variables x and y are **symmetric** in ϕ if ϕ is equivalent to the formula obtained by exchanging x and y. A set of variables Y is **symmetric** in ϕ if each pair in Y is symmetric in ϕ.

Since the clause order and the order within a clause of a DNF or CNF is insignificant, symmetry is a straightforward syntactic property.

The following lemma relates symmetric variables to identical coefficients.

Lemma 6.2. Let $\sum_{i=1}^{m} a_i x_i \geq d$ be an LPB representing the DNF ϕ. For any $i, k \in [1..m]$, if $a_i = a_k$ then x_i, x_k are symmetric in ϕ; moreover, there exists an LPB $\sum_{i=1}^{m} a'_i x_i \geq d'$ representing ϕ such that if x_i, x_k are symmetric in ϕ then $a'_i = a'_k$.

For example, $x_1 \vee x_2$ can be represented by $2x_1 + x_2 \geq 1$ or $x_1 + x_2 \geq 1$.

We want to measure how often a variable occurs in a DNF, taking the length of the clauses into account. Intuitively, a variable is "important" if it occurs in *many* clauses and if it occurs in *short* clauses. To formalise this, we consider multisets of natural numbers. We represent multisets as strings of numbers in ascending order, written, e.g. $\{\!\{1, 1, 2\}\!\}$.

Definition 6.3. Let A, B be two multisets of numbers. We write $B \preceq A$ if B is obtained from A as follows: for each occurrence of a number n in A, either leave this occurrence in B, or replace it by an arbitrary (possibly 0) number of occurrences of numbers $> n$. We write $B \prec A$ if $B \preceq A$ and $A \not\preceq B$.

Example 6.4. We have $\{\!\{2, 2, 2, 2\}\!\} \succ \{\!\{2, 2, 2\}\!\} \succ \{\!\{2, 2, 3\}\!\} \succ \{\!\{2, 3\}\!\}$.

Note that $\{\!\{2, 2, 3\}\!\} \succ \{\!\{2, 3\}\!\}$ can be established in two ways: removing one 2 from $\{\!\{2, 2, 3\}\!\}$, or removing the 3 from $\{\!\{2, 2, 3\}\!\}$ and then replacing one 2 from $\{\!\{2, 2\}\!\}$ by one 3. $\preceq$ is a total order on multisets of natural numbers. In our representation, to determine whether $A \succeq B$, one must simply cut off the longest common prefix of A and B. If the remainder of A starts with a smaller number than that of B, or if the remainder of B is empty, then $A \succeq B$.

[2] By Prop. 3.2, there is of course a naïve semi-decision procedure for this problem, involving enumeration of all LPBs.

Definition 6.5. For a DNF ϕ, define $OP(\phi, x)$ as the multiset having one occurrence of n for each clause of length n in ϕ that contains x. We call $OP(\phi, x)$ the **occurrence pattern** of x.

Example 6.6. Consider $\phi \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_1 \wedge x_5) \vee (x_2 \wedge x_3) \vee (x_2 \wedge x_4) \vee (x_3 \wedge x_4 \wedge x_5)$. The occurrence patterns are $OP(\phi, x_1) = \{\!\{2, 2, 2, 2\}\!\}$, $OP(\phi, x_2) = \{\!\{2, 2, 2\}\!\}$, $OP(\phi, x_3) = OP(\phi, x_4) = \{\!\{2, 2, 3\}\!\}$, and $OP(\phi, x_5) = \{\!\{2, 3\}\!\}$. ϕ can be represented by $4x_1 + 3x_2 + 2x_3 + 2x_4 + x_5 \geq 5$.

To give some more intuition, compare x_1 and x_2, say. For clause $x_1 \wedge x_4$, replacing x_1 by x_2 yields another clause of ϕ, but for $x_1 \wedge x_5$ this is not the case. $OP(\phi, x_1)$ therefore has one more occurrence of 2 than $OP(\phi, x_2)$. The fact that replacing x_1 by x_2 in $x_1 \wedge x_5$ does not yield another clause of ϕ means that x_1 must have a bigger coefficient than x_2, in any LPB representing ϕ.

Computing the set of occurrence patterns for all variables in ϕ can be done in time linear in $|\phi|$. In fact, the number of elements of all occurrence patterns is exactly the number of literals in ϕ. Thus sorting the variables w.r.t. the occurrence patterns can be done in time polynomial in $|\phi|$.

The next lemma says that the coefficients of an LPB representing a DNF must correspond to the order given by the occurrence patterns. Thus, given a DNF, we know that *if* it can be represented as an LPB, then the coefficients of this LPB are ordered in a certain way.

Lemma 6.7. Let ϕ be a DNF represented by the LPB $\sum_{i=1}^{m} a_i x_i \geq d$. Then $a_i \geq a_k$ implies $OP(\phi, x_i) \succeq OP(\phi, x_k)$; moreover, there exists an LPB $\sum_{i=1}^{m} a'_i x_i \geq d'$ representing ϕ such that $OP(\phi, x_i) = OP(\phi, x_k)$ implies $a'_i = a'_k$.

The following is a corollary of Lemmas 6.2 and 6.7.

Corollary 6.9. If the DNF ϕ is represented by an LPB I, then x_i, x_k are symmetric in ϕ iff x_i, x_k have identical occurrence patterns.

The results so far can be used to make statements about which DNFs cannot be represented by a single LPB. For example, consider $\phi \equiv (x_1 \wedge x_2 \wedge x_5) \vee (x_1 \wedge x_4) \vee (x_3 \wedge x_4 \wedge x_5) \vee (x_2 \wedge x_3)$. We have $OP(\phi, x_i) = \{\!\{2, 3\}\!\}$ for $i \in [1..4]$, and yet $x_1, \ldots, x_4$ are not symmetric, and thus ϕ is not representable as LPB.

Also, it has been said that a single LPB can express an implication [7]. In [8], implications of the form $y \rightarrow (x_1 \wedge x_2)$ are expressed as LPB. However, the power of an LPB for expressing implications is very limited: an implication of the form $(x_1 \vee \ldots \vee x_m) \rightarrow (y_1 \wedge \ldots \wedge y_l)$, where $m, l \geq 2$, cannot be expressed by a single LPB [10].

6.2 Decomposing a DNF

We want to find an LPB representing ϕ if possible. Using Lemma 6.7, we can establish the order of the coefficients. Assume the numbering is such that we have $OP(\phi, x_1) \succeq \ldots \succeq OP(\phi, x_m)$. Consider now the maximal set $X = \{x_1, \ldots, x_l\}$

such that $OP(\phi, x_1) = \ldots = OP(\phi, x_l)$ (=: $OP(\phi, X)$). If X is not symmetric in ϕ, then by Cor. 6.9, ϕ cannot be represented by an LPB and we can stop. Otherwise, we partition ϕ according to how many variables from X each clause contains. We then remove the variables from X from each clause, which gives $l+1$ subproblems. Theorem 6.15 states under which conditions solutions to these subproblems can be combined to an LPB for ϕ. However, since the solutions have to be similar in a certain sense, it turns out that we cannot simply solve the subproblems independently and *then* combine the solutions, but we must solve the subproblems in parallel, as will be shown in Subsec. 6.3.

The following statements do not require X to be *maximal*, e.g. if $\{x_1, \ldots, x_5\}$ is the maximal set such that $OP(\phi, x_1) = \ldots = OP(\phi, x_5)$, then the statements will also hold for $X = \{x_1, x_2, x_3\}$.

Note that our formalism bears a certain resemblance with [2], where one considers LPBs obtained from a certain given LPB by removing some variables.

Definition 6.11. Let ϕ be a DNF and X a subset of its variables with $|X| = l$. If ϕ contains a clause $c \subseteq X$, then let $k_{\max}$ be the length of the longest such clause; otherwise let $k_{\max} := \infty$. For $0 \leq k \leq l$, we define $S(\phi, X, k)$ as the disjunction of clauses from ϕ containing exactly $\min\{k, k_{\max}\}$ variables from X, with those variables removed.

When constructing the $S(\phi, X, k)$ from ϕ, we say that we *split away* the variables in X from ϕ.

Example 6.12. Let $\phi \equiv (x_1) \vee (x_2) \vee (x_3 \wedge x_4)$ and $X = \{x_1, x_2\}$. We have $k_{\max} = 1$. Then $S(\phi, X, 0) = (x_3 \wedge x_4)$, $S(\phi, X, 1) = \mathit{true}$ (i.e. the disjunction of twice the empty conjunction), and $S(\phi, X, 2) = \mathit{true}$.

We must solve the $l+1$ subproblems in such a way that the resulting LPBs agree in all coefficients, and that the degree difference of neighbouring LPBs is always the same. Before giving the theorem, we give two examples for illustration.

Example 6.13. Consider $\phi \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_1 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_4)$ and $X = \{x_1\}$. Then $S(\phi, X, 0) = x_2 \wedge x_3 \wedge x_4$, represented by $x_2 + x_3 + x_4 \geq 3$. Moreover, $S(\phi, X, 1) = x_2 \vee x_3 \vee x_4$, represented by $x_2 + x_3 + x_4 \geq 1$.

Since the coefficients of the two LPBs agree, it turns out that ϕ can be represented by $2x_1 + x_2 + x_3 + x_4 \geq 3$. The coefficient of x_1 is given by the difference of the two degrees, i.e. $3 - 1$.

Example 6.14. Consider $\phi \equiv (x_1 \wedge x_2) \vee (x_1 \wedge x_3 \wedge x_4) \vee (x_2 \wedge x_3 \wedge x_4)$ and $X = \{x_1, x_2\}$. We have $S(\phi, X, 0) = \mathit{false}$, represented by $x_3 + x_4 \geq 4$, $S(\phi, X, 1) = x_3 \wedge x_4$, represented by $x_3 + x_4 \geq 2$, and $S(\phi, X, 2) = \mathit{true}$, represented by $x_3 + x_4 \geq 0$. The DNF ϕ is represented by $2x_1 + 2x_2 + x_3 + x_4 \geq 4$. The coefficient of x_1, x_2 is given by $4 - 2 = 2 - 0 = 2$ (the degrees are "equidistant").

Theorem 6.15. Let ϕ be a DNF in variables $x_1, \ldots, x_m$ and suppose $X = \{x_1, \ldots, x_l\}$ are symmetric variables such that $OP(\phi, X)$ is maximal w.r.t. $\preceq$ in ϕ. Then ϕ is represented by an LPB $\sum_{i=1}^{m} a_i x_i \geq d$, where $a_1 = \ldots = a_l$, iff for all $k \in [0..l]$, the DNF $S(\phi, X, k)$ is represented by $\sum_{i=l+1}^{m} a_i x_i \geq d - k \cdot a_1$.

The remaining problem is that a DNF might be represented by various LPBs, and so even if the LPBs computed recursively do not have agreeing coefficients and equidistant degrees, one might find alternative LPBs (such as the non-obvious LPB for *false* in Ex. 6.14) so that Thm. 6.15 can be applied.

Before addressing this problem, we generalise LPBs by recording to what extent degrees can be shifted without changing the meaning. To formulate this, we temporarily lift the restriction that coefficients and degrees must be integers. How to obtain integers in the end is explained at the end of Subsec. 6.3.

Definition 6.16. Given an LPB $I \equiv \sum_{i=1}^{m} a_i x_i \geq d$, we call s the **minimum degree** of I if s is the smallest number (possibly $-\infty$) such that for any $s' \in (s, d]$, the LPB $\sum_{i=1}^{m} a_i x_i \geq s'$ represents the same function as I. We call b the **maximum degree** if b is the biggest number (possibly ∞) such that $\sum_{i=1}^{m} a_i x_i \geq b$ represents the same function as I.

Note that the minimum degree of I is not a possible degree of I. Since the minimum and maximum degrees of an LPB are more informative than its actual degree, we introduce the notation $\sum_{i=1}^{m} a_i x_i \geq (s, b]$ for denoting an LPB with minimum degree s and maximum degree b.

The next lemma strengthens Thm. 6.15, stating that information about minimum and maximum degrees can be maintained with little overhead.

Lemma 6.17. Make the same assumptions as in Thm. 6.15, and assume that for all $k \in [0..l]$, the DNF $S(\phi, X, k)$ is represented by $I^k \equiv \sum_{i=l+1}^{m} a_i x_i \geq d - k \cdot a_1$. Moreover, for all $k \in [0..l]$, let s_k, b_k be minimum and maximum degrees of I^k, respectively. Then $s := \max_{k \in [0..l]}(s_k + k \cdot a_1)$, $b := \min_{k \in [0..l]}(b_k + k \cdot a_k)$ are the minimum and maximum degrees of $\sum_{i=1}^{m} a_i x_i \geq d$.

6.3 Composing LPBs

Theorem 6.15 suggests a recursive algorithm where, at least conceptually, in the base case we have at most 2^m trivial problems of determining an LPB, trivial since the formula for which we must find an LPB is either *true* or *false*.

Example 6.18. Consider Ex. 6.6. To find an LPB for ϕ, we must find LPBs for $S(\phi, \{x_1\}, 0)$ and $S(\phi, \{x_1\}, 1)$. To find an LPB for $S(\phi, \{x_1\}, 0)$, we must find LPBs for $S(S(\phi, \{x_1\}, 0), \{x_2\}, 0)$ and $S(S(\phi, \{x_1\}, 0), \{x_2\}, 1)$, and so forth. Table 2 gives all the formulae for which we must find LPBs. For a concise notation we use some abbreviations which we explain using $S(\cdot, x_{3..5}, 0) \equiv f$ in the top-right corner: it stands for $S((x_3 \wedge x_4 \wedge x_5), \{x_3, x_4, x_5\}, 0) \equiv \mathit{false}$, i.e. the '$\cdot$' stands for the nearest *non-shaded* formula to the left, here $(x_3 \wedge x_4 \wedge x_5)$. Note how we arranged the subproblem formulae in the table: e.g. $(x_3 \wedge x_4 \wedge x_5)$ has *three* symmetric variables that are split away to obtain the subproblems to be solved, so these subproblems are located *three* columns to the right of $(x_3 \wedge x_4 \wedge x_5)$. The two shaded boxes in between contain the subproblems obtained by splitting away only $\{x_3\}$, $\{x_3, x_4\}$, resp.

Table 2. The recursive problems of Ex. 6.18

ϕ					
$S(\cdot, x_1, 0)$ $\equiv (x_2 \wedge x_3) \vee$ $(x_2 \wedge x_4) \vee$ $(x_3 \wedge x_4 \wedge x_5)$	$S(\cdot, x_2, 0) \equiv$ $(x_3 \wedge x_4 \wedge x_5)$	$S(\cdot, x_3, 0) \equiv f$ $S(\cdot, x_3, 1)$ $\equiv (x_4 \wedge x_5)$	$S(\cdot, x_{3..4}, 0) \equiv f$ $S(\cdot, x_{3..4}, 1) \equiv f$ $S(\cdot, x_{3..4}, 2) \equiv x_5$	$S(\cdot, x_{3..5}, 0) \equiv f$ $S(\cdot, x_{3..5}, 1) \equiv f$ $S(\cdot, x_{3..5}, 2) \equiv f$ $S(\cdot, x_{3..5}, 3) \equiv t$
	$S(\cdot, x_2, 1) \equiv$ $x_3 \vee x_4$	$S(\cdot, x_3, 0) \equiv x_4$ $S(\cdot, x_3, 1) \equiv t$	$S(\cdot, x_{3..4}, 0) \equiv f$ $S(\cdot, x_{3..4}, 1) \equiv t$ $S(\cdot, x_{3..4}, 2) \equiv t$	
$S(\cdot, x_1, 1)$ $\equiv x_2 \vee x_3$ $\vee x_4 \vee x_5$	$S(\cdot, x_2, 0) \equiv$ $x_3 \vee x_4 \vee x_5$ $S(\cdot, x_2, 1) \equiv t$	$S(\cdot, x_3, 0) \equiv$ $x_4 \vee x_5$ $S(\cdot, x_3, 1) \equiv t$ $S(\cdot, x_3, 2) \equiv t$	$S(\cdot, x_{3..4}, 0) \equiv x_5$ $S(\cdot, x_{3..4}, 1) \equiv t$ $S(\cdot, x_{3..4}, 2) \equiv t$ $S(\cdot, x_{3..4}, 3) \equiv t$	$S(\cdot, x_{3..5}, 0) \equiv f$ $S(\cdot, x_{3..5}, 1) \equiv t$ $S(\cdot, x_{3..5}, 2) \equiv t$ $S(\cdot, x_{3..5}, 3) \equiv t$ $S(\cdot, x_{3..5}, 4) \equiv t$

The algorithm we propose is not a purely recursive one, since the subproblems at each level must be solved in parallel. Explained using the example, we first find LPBs for the formulae in the rightmost column, which have 0 variables and hence we must determine 0 coefficients. Next to the left, we have formulae that contain (at most) x_5, and we determine LPBs representing these, where we use the same a_5 for all formulae! Then we determine a_4, and so forth.

Taking $(x_3 \wedge x_4 \wedge x_5)$ in Table 2 as an example, Thm. 6.15 suggests that a_3, a_4, a_5 should be equal (x_3, x_4, x_5 are symmetric) and determined in one go. However, since a_3, a_4, a_5 also have to represent other subproblem formulae where x_3, x_4, x_5 are not necessarily symmetric, one cannot determine a_3, a_4, a_5 in one go, but rather first a_5, then a_4, then a_3. Therefore, it is necessary to define and interpret formulae obtained by splitting away fewer variables than one could split away, in the sense of Thm. 6.15. These are the shaded formulae.

We call the formulae in column $l+1$ the *l-successors*. Shaded formulae are called *auxiliary*, the others are called *main*. Formulae that have no further formulae to the right are called *final*. The following definition formalises these notions.

Definition 6.19. Let ϕ be a DNF in m variables. Then ϕ is the 0-**successor** of ϕ. Furthermore, ϕ is a **main** successor of ϕ. Moreover, if ϕ' is a main n-successor of ϕ, and l is maximal so that $x_{n+1}, \ldots, x_{n+l}$ are symmetric in ϕ', then for all l', k with $1 \leq l' \leq l$ and $0 \leq k \leq l'$, we say that $S(\phi', \{x_{n+1}, \ldots, x_{n+l'}\}, k)$ is an $(n+l')$-successor of ϕ. The $(n+l)$-successors are called **main**, and for $l' < l$, the $(n+l')$-successors are called **auxiliary**. If $x_{n+1}, \ldots, x_{n+l}$ are the only variables of ϕ', then we call the $(n+l)$-successors **final**.

Note in particular $x_3 \vee x_4$ in column 3 in Table 2. It does not contain x_5, and so we obtain final 4-successors in the last-*but-one* column. Clearly, a final successor of ϕ is either *true* or *false*.

Proposition 6.20. Assume ϕ, ϕ', n, l as in Def. 6.19. For $0 < l' < l$ and $0 \leq k \leq l'$, we have

$$S(S(\phi', \{x_{n+1}, \ldots, x_{n+l'}\}, k), \{x_{n+l'+1}\}, 0) \equiv S(\phi', \{x_{n+1}, \ldots, x_{n+l'+1}\}, k)$$
$$S(S(\phi', \{x_{n+1}, \ldots, x_{n+l'}\}, k), \{x_{n+l'+1}\}, 1) \equiv S(\phi', \{x_{n+1}, \ldots, x_{n+l'+1}\}, k+1)$$

For example, consider $S((x_3 \wedge x_4 \wedge x_5), \{x_3\}, 1) \equiv (x_4 \wedge x_5)$ in Table 2. We have $S((x_4 \wedge x_5), \{x_4\}, 0) \equiv S((x_3 \wedge x_4 \wedge x_5), \{x_3, x_4\}, 1)$ and $S((x_4 \wedge x_5), \{x_4\}, 1) \equiv S((x_3 \wedge x_4 \wedge x_5), \{x_3, x_4\}, 2)$. Generally, each non-final successor is associated with two formulae in the column right next to it, one slightly up and one slightly down, obtained by splitting away the variable with the smallest index. This is not surprising per se and corresponds to a naïve approach where we always split away one variable at a time (for applying Thm. 6.15), thereby constructing 2^m formulae in the rightmost column. The point of Prop. 6.20 is that we can usually construct fewer formulae since $S(S(\phi, \{x_{n+1}, \ldots, x_{n+l'}\}, k), \{x_{n+l'+1}\}, 1)$ and $S(S(\phi, \{x_{n+1}, \ldots, x_{n+l'}\}, k+1), \{x_{n+l'+1}\}, 0)$ coincide. In Table 2, we have 12 final formulae rather than $2^5 = 32$.

The following theorem states if and how one can find the next coefficient and degrees for representing all k-successors of ϕ provided one has coefficients and degrees for representing all $(k+1)$-successors.

Theorem 6.21. Assume ϕ as in Thm. 6.15 and some k with $0 \leq k \leq m-1$, and let Φ_k be the set of k-successors of ϕ. For every non-final $\phi' \in \Phi_k$, suppose we have two LPBs $\sum_{i=k+2}^{m} a_i x_i \geq (s_{\phi'0}, b_{\phi'0}]$ and $\sum_{i=k+2}^{m} a_i x_i \geq (s_{\phi'1}, b_{\phi'1}]$, representing $S(\phi', \{x_{k+1}\}, 0)$ and $S(\phi', \{x_{k+1}\}, 1)$, respectively.

If it is possible to choose a_{k+1} such that

$$\max_{\phi' \in \Phi_k} (s_{\phi'0} - b_{\phi'1}) < a_{k+1} < \min_{\phi' \in \Phi_k} (b_{\phi'0} - s_{\phi'1}), \tag{10}$$

then for all $\phi' \in \Phi_k$, the LPB $\sum_{i=k+1}^{m} a_i x_i \geq (s_{\phi'}, b_{\phi'}]$ represents ϕ', where

$$s_{\phi'} = \max\{s_{\phi'0}, s_{\phi'1} + a_{k+1}\},\ b'_{\phi} = \min\{b_{\phi'0}, b_{\phi'1} + a_{k+1}\} \text{ for non-final } \phi' \tag{11}$$
$$s_{\phi'} = -\infty,\ b_{\phi'} = 0 \text{ for } \phi' \equiv \mathit{true},\ s_{\phi'} = \textstyle\sum_{i=k+1}^{m} a_i,\ b_{\phi'} = \infty \text{ for } \phi' \equiv \mathit{false} \tag{12}$$

If $\max_{\phi' \in \Phi_k}(s_{\phi'0} - b_{\phi'1}) \geq \min_{\phi' \in \Phi_k}(b_{\phi'0} - s_{\phi'1})$, then no a_{k+1}, $s_{\phi'}$, $b_{\phi'}$ exist such that $\sum_{i=k+1}^{m} a_i x_i \geq (s_{\phi'}, b_{\phi'}]$ represents ϕ' for all $\phi' \in \Phi_k$.

The m-successors of ϕ are represented by LPBs with an empty sum as l.h.s.: $\sum_{i=m+1}^{m} a_i x_i \geq (0, \infty]$ for *false*, $\sum_{i=m+1}^{m} a_i x_i \geq (-\infty, 0]$ for *true*. Then we proceed using Thm. 6.21, in each step choosing an arbitrary a_{k+1} fulfilling (10).

Example 6.22. Consider again Ex. 6.18. Table 3 is arranged in strict correspondence to Table 2 and shows LPBs for all successors of Φ. In the top line we give the l.h.s. of the LPBs, which is of course the same for each LPB in a column. In the main table, we list the minimum and maximum degree of each formula.

In the first step, applying (10), we have to choose a_5 so that

$$\max\{0-\infty, 0-\infty, 0-0, \quad 0-0, -\infty-0, -\infty-0, -\infty-0\} < a_5 <$$
$$\min\{\infty-0, \infty-0, \infty--\infty, \quad \infty--\infty, 0--\infty, 0--\infty, 0--\infty\}.$$

Table 3. LPBs for Ex. 6.18

$4x_1+3x_2+2x_3+2x_4+x_5\geq\ldots$	$3x_2+2x_3+2x_4+x_5\geq\ldots$	$2x_3+2x_4+x_5\geq\ldots$	$2x_4+x_5\geq\ldots$	$x_5\geq\ldots$	$\sum_{i=6}^{5}a_ix_i\geq\ldots$
				$(1,\infty]$	$(0,\infty]$
			$(3,\infty]$	$(1,\infty]$	$(0,\infty]$
		$(4,5]$	$(2,3]$	$(0,1]$	$(0,\infty]$
	$(4,5]$				$(-\infty,0]$
				$(1,\infty]$	
			$(1,2]$	$(-\infty,0]$	
		$(1,2]$	$(-\infty,0]$	$(-\infty,0]$	
$(4,5]$					
				$(0,1]$	$(0,\infty]$
			$(0,1]$	$(-\infty,0]$	$(-\infty,0]$
	$(0,1]$	$(0,1]$	$(-\infty,0]$	$(-\infty,0]$	$(-\infty,0]$
		$(-\infty,0]$	$(-\infty,0]$	$(-\infty,0]$	$(-\infty,0]$
					$(-\infty,0]$

Choosing $a_5=1$ will do. The minimum and maximum degrees in column 5 are computed using (11); e.g. the topmost $(1,\infty]$ is $(\max\{0,0+1\},\min\{\infty,\infty+1\}]$.

In the next step, we have to choose a_4 so that

$$\max\{1-\infty,1-1,\quad 1-0,-\infty-0,\quad 0-0,-\infty-0,-\infty-0\}<a_4<\\ \min\{\infty-1,\infty-0,\quad \infty--\infty,0--\infty,\quad 1--\infty,0--\infty,0--\infty\}.$$

Choosing $a_4=2$ will do. Note that the bound $1-0<a_4$ comes from the middle box of the fifth column and thus ultimately from $x_3\vee x_4$. Our algorithm enforces that $a_4>a_5$, which must hold for an LPB representing $x_3\vee x_4$.

In the next step, a_3 can also be chosen to be any number >1 so we choose 2 again.[3] In the next step, $2<a_2<4$ must hold so we choose $a_2=3$. Finally, $3<a_1<5$ must hold so we choose $a_1=4$. We obtain the LPB $4x_1+3x_2+2x_3+2x_4+x_5\geq(4,5]$ given in Ex. 6.6.

We have seen in the example how our algorithm works. However, since the choice of a_{k+1} is not unique in general, one might be worried that a bad choice of a_{k+1} might later lead to non-applicability of Thm. 6.21. We have a lemma stating that this is not a problem [18].

However, there are some pragmatic choices. As stated in the example, to obtain an LPB with small coefficients, one might always choose a_{k+1} as the smallest possible integer value. It might also occur, though not in the above example, that a_{k+1} is forced to be between neighbouring integers, in which case it cannot be an integer itself. In this case, one can multiply all LPBs of the current system by 2 (this obviously preserves the meaning of the LPBs) before proceeding so that a_{k+1} can be chosen to be an integer.

[3] The algorithm could be improved by determining a_3 and a_4 in one go since x_3,x_4 are symmetric in Φ. We refrain from spelling this out to avoid further complication.

From the construction of the successors (see Table 2) it follows that all formulae in a column together have size less than all formulae in the column to the left of it, so that the entire table has size less than $|\phi| \cdot (m+1)$. One can thus show that the complexity of the algorithm is polynomial in the size of ϕ, while the size of ϕ itself can be exponential in m. In fact, this is the most interesting case, because in this case an LPB representation may yield an exponential saving.

A thorougher analysis of the complexity of the algorithm will be due once it is embedded into a more complete algorithm which converts an *arbitrary* DNF (or CNF) into a *set* of LPBs. It is clear that such an algorithm would first have to partition the DNF according to the polarity of each variable, which is straightforward. The next step would be to partition a DNF where each variable occurs in only one polarity into sub-DNFs each of which can be represented by a single LPB. This step is nontrivial and the main topic for future work.

7 Conclusion

Linear pseudo-Boolean constraints have attracted interest because they can often be used to represent Boolean functions more compactly than CNFs or DNFs, and because techniques applied in CNF-based propositional satisfiability solving can be generalised to LPBs, which can be more efficient than solving a problem based on a CNF representation [1,5,6,7,8]. This generalisation is essentially an application of a technique known from OR to the field of AI, or more specifically, propositional logic [6].

It is assumed here that the problems, as they arise in an application domain, have a natural encoding as LPB, and that the CNF encoding would be larger. Our work was initially motivated by three main issues, which were not addressed in previous works.

Firstly, several authors have emphasised that an LPB representation of a function can be exponentially more compact than a CNF representation [1,5,6,7,8]. However, it is shown in fact that *cardinality constraints* can be exponentially more compact than a CNF. Thus no evidence is given that the additional expressive power that LPBs have compared to cardinality constraints is useful.

Secondly, it has been noted *en passant* that a single LPB can be used to express an implication [7], but it remains unclear what kind of implications can or cannot be expressed. In fact, the power of an LPB for expressing implications is very limited.

Most importantly, since an LPB representation can be more compact than a CNF representation, one might use LPB encodings even in cases where they do not arise naturally from the application domain. That is, one might convert a CNF to a (small) set of LPBs and then apply LPB solving [1,5,6,7,8]. Here we see the potential for practical application of our work.

As a further comment on the first point, Barth [3] mentions that LPBs arise in AI applications [4]. Since he used a solver that could only deal with cardinality constraints, he proposes a transformation of LPBs to cardinality constraints. Note that this transformation goes in the opposite direction compared to ours, from a more concise to a less concise representation.

In [8], LPBs are used for bounded model checking. At one point, an LPB of the form $x_1 + x_2 + 2\bar{y} \geq 2$ (which is not a cardinality constraint) is used.

Apart from that, the above works say little about where the problem instances come from, and if anything, then these are in fact cardinality constraints rather than LPBs. In [1], problems Min-Cover, Max-SAT, and MAX-ONEs are mentioned. E.g. Max-SAT is the problem of finding a variable assignment that maximises the number of satisfied clauses of an unsatisfiable SAT instance. Furthermore, applications from design automation [5], the pigeonhole problem [6], and gate level netlists [7] are mentioned as applications.

However, we are not suggesting that our approach of converting a CNF or DNF to an LPB is the only way to go. If for a problem domain, there is a natural direct encoding as an LPB not going via CNF or DNF, then this should definitely be considered.

Hooker has proposed an algorithm for generating the strongest 0-1 ILP constraints, within a candidate set T, that are implied by a set S of 0-1 ILP constraints [9]. Letting T be the set of all LPBs, the algorithm can be used to transform a CNF to an LPB. However, the algorithm is practical only for certain restrictions of T. In the general case, which we need here, it is unclear if the algorithm is any better than enumerating and checking all LPBs. This is however an interesting topic for future work.

Complementary to this paper, we have also obtained results about Boolean functions that can definitely *not* be represented compactly as a set of LPBs. More precisely, there is a class of *monotone* functions for which the DNF representation is exponential and the LPB representation saves nothing [17].

We summarise our contributions to the understanding of LPBs. We demonstrated that the functions expressible as one LPB constraint are a strict subset of the monotone functions. We gave some results about the cardinality of various classes of Boolean functions, and showed that the blowup when encoding an LPB as CNF or DNF is not worse than when encoding a cardinality constraint. We showed that the problems of encoding a DNF or a CNF as LPB have a very simple duality. Finally and most importantly, we gave an algorithm for computing an LPB representation for a DNF whenever this is possible.

Acknowledgements. This work was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB TR14/AVACS). I would like to thank Markus Behle, Martin Fränzle, Marc Herbstritt, Christian Herde, Felix Klaedtke, Bernhard Nebel, and the other AVACS colleagues, for useful discussions.

References

1. Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Generic ILP versus specialized 0-1 ILP: an update. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, pages 450–457. ACM, 2002.

2. Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. A translation of pseudo Boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:191–200, 2006.
3. Peter Barth. Linear 0-1 inequalities and extended clauses. In Andrei Voronkov, editor, *Proceedings of the 4th International Conference on Logic Programming and Automated Reasoning*, volume 698 of *LNCS*, pages 40–51. Springer-Verlag, 1993.
4. Peter Barth and Alexander Bockmayr. Solving 0-1 problems in CLP(PB). In *Proceedings of the 9th Conference on Artificial Intelligence for Applications*. IEEE, 1993.
5. Donald Chai and Andreas Kuehlmann. A fast pseudo-Boolean constraint solver. In *Proceedings of the 40th Design Automation Conference*, pages 830–835. ACM, 2003.
6. Heidi E. Dixon and Matthew L. Ginsberg. Combining satisfiability techniques from AI and OR. *The Knowledge Engineering Review*, 15:31–45, 2000.
7. Martin Fränzle and Christian Herde. Efficient SAT engines for concise logics: Accelerating proof search for zero-one linear constraint systems. In Moshe Y. Vardi and Andrei Voronkov, editors, *Proceedings of the 10th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2850 of *LNCS*, pages 302–316. Springer-Verlag, 2003.
8. Martin Fränzle and Christian Herde. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design*, 2006. Online version, http://dx.doi.org/10.1007/s10703-006-0031-0. Print version in press.
9. John N. Hooker. Generalized resolution for 0-1 linear inequalities. *Annals of Mathematics and Artificial Intelligence*, 6(1-3):271–286, 1992.
10. John N. Hooker and Hong Yan. Tight representations of logical constraints as cardinality rules. *Mathematical Programming*, 85(2):363–377, 1999.
11. D. Kleitman and G. Markowsky. On Dedekind's problem: the number of isotone Boolean functions. II. *Transactions of the American Mathematical Society*, 213:373–390, 1975.
12. João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
13. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM, 2001.
14. Raúl Rojas. *Neural Networks. A Systematic Introduction*. Springer-Verlag, 1996.
15. Ching Lai Sheng. *Threshold Logic*. Academic Press, 1969.
16. Neil J. A. Sloane. On-line encyclopedia of integer sequences. http://www.research.att.com/~{}njas/sequences/Seis.html.
17. Jan-Georg Smaus. Representing Boolean functions as linear pseudo-Boolean constraints. In Youssef Hamadi, editor, *Proceedings of the CP 2006 Workshop on the Integration of SAT and CP techniques*, 2006.
18. Jan-Georg Smaus. On Boolean functions encodable as a single linear pseudo-Boolean constraint. Technical Report 230, Institut für Informatik, Universität Freiburg, 2007. Also available as TR No. 13 on http://www.avacs.org/.
19. Vetle Ingvald Torvik and E. Trintaphyllou. Inference of monotone Boolean functions. In Chris A. Floudas and Panos M. Pardalos, editors, *Encyclopedia of Optimization*, pages 472–480. Kluwer Academic Publishers, 2001.
20. Ingo Wegener. *The Complexity of Boolean Functions*. Wiley & Sons, 1987.
21. Hantao Zhang. SATO: An efficient propositional prover. In William McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction*, volume 1249 of *LNCS*, pages 272–275. Springer-Verlag, 1997.

Solving a Stochastic Queueing Control Problem with Constraint Programming

Daria Terekhov and J. Christopher Beck

Department of Mechanical and Industrial Engineering,
University of Toronto, Toronto, Ontario, Canada
{dterekho,jcb}@mie.utoronto.ca

Abstract. In a facility with front room and back room operations, it is useful to switch workers between the rooms in order to cope with changing customer demand. Assuming stochastic customer arrival and service times, we seek a policy for switching workers such that the expected customer waiting time is minimized while the expected back room staffing is sufficient to perform all work. Three novel constraint programming models and a shaving algorithm are presented. Experimental results show that the best constraint programming model, using shaving, is able to find and prove optimal solutions for almost all problem instances within a reasonable run-time, but that an existing heuristic algorithm performs better in terms of solution quality over time. A hybrid method combining the heuristic and the best constraint programming method is shown to perform better than either of these approaches separately. This is the first work of which we are aware that solves a queueing control problem with constraint programming.

1 Introduction

Many retail facilities, such as stores or banks, have back room and front room operations. In the front room, workers have to serve arriving customers, and customers form a queue and wait to be served when all workers are busy. In the back room, work is less time-sensitive, and may include such tasks as sorting or processing paperwork. All workers in the facility are cross-trained and are assumed to be able to perform back room tasks equally well and serve customers with the same service rate. Therefore, it makes sense for the managers of the facility to switch workers between the front room and the back room depending both on the number of customers in the front room and the amount of work that has to be performed in the back room. These managers are thus interested in finding a switching policy that minimizes the expected customer waiting time in the front room, subject to the constraint that the expected number of workers in the back room is sufficient to complete all required work. This queueing control problem has been studied in detail by Berman et al. [3], who propose a heuristic for solving it.

In this paper, a constraint programming (CP) approach is proposed for the problem. Thus, the contributions of this paper are twofold. Firstly, CP is, for the first time, used to solve a stochastic queueing control problem. Secondly, a complete approach for a problem for which only a heuristic algorithm existed previously is presented.

The paper is organized as follows. Section 2 presents a description of the problem and the work done by Berman et al. [3]. Section 3 presents three CP models for this

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 303–317, 2007.

problem. Section 4 describes a shaving procedure used for improving the efficiency of the CP models. In Section 5, the performance of the three CP models is compared and the best CP approach is contrasted with the Berman et al. heuristic. Based on these results, a hybrid method is proposed and evaluated in Section 6. In Section 7, a discussion of the results is presented. Section 8 describes related problems and states some directions for future work. Section 9 concludes the paper.

2 Problem Description

Let N denote the number of workers in the facility, and let S be the maximum number of customers allowed in the front room at any one time.[1] When there are S customers present, arriving customers will be blocked from joining the front room queue. Customers arrive according to a Poisson process with rate λ. Service times in the front room follow an exponential distribution with rate μ. The minimum expected number of workers that is required to be present in the back room in order to complete all of the necessary work is assumed to be known, and is denoted by B_l, where l stands for 'lower bound'. Only one worker is allowed to be switched at a time, and both switching time and switching cost are assumed to be negligible. The goal of the problem is to find an optimal approach to switching workers between the front room and the back room so as to minimize the expected customer waiting time, denoted W_q, while at the same time ensuring that the expected number of workers in the back room is at least B_l. Thus, a policy needs to be constructed that specifies how many workers should be in the front room and back room at a particular time, and when switches should occur.

2.1 Problem Definition

Berman et al. define a policy in terms of quantities k_i, for $i = 0, \ldots, N$. This policy states that there should be i workers in the front room whenever there are between $k_{i-1} + 1$ and k_i customers (inclusive) in the front room, for $i = 1, 2, \ldots, N$. As an illustration, consider the policy $(k_0, k_1, k_2, k_3) = (0, 2, 3, 6)$, with $N = 3$. This policy states that when there are $k_0 + 1 = 1$ or $k_1 = 2$ customers in the front room, there is one worker in the front room; when there are 3 customers, there are 2 workers; and when there are 4, 5, or 6 customers, all 3 workers are employed in the front. This definition of a policy forms the basis of the model proposed by Berman et al., with the switching points k_i, $i = 0, \ldots, N-1$, being the decision variables of the problem, and k_N being fixed to S, the capacity of the system. In this paper, we follow Berman et al. and define an optimal policy as a set of values for the switching points, k_i, which minimize the expected waiting time subject to the back room constraint.[2]

In order to determine the expected waiting time and the expected number of workers in the back room given a policy defined by particular values of k_i, Berman et al. first

[1] The notation used by Berman et al. [3] is adopted throughout this paper.

[2] The term *optimal policy* is used in queueing control literature [6] to mean both the optimal *type* of policy and the optimal parameter values for a given policy type. In particular, for our problem, it is possible that an alternative type of policy (e.g., one that allowed randomization in the switching decision) may lead to a smaller expected waiting time.

define a set of probabilities, $P(j)$, for $j = k_0, k_0 + 1, \ldots, S$. Each $P(j)$ with $j > k_0$ denotes the steady-state (long-run) probability of there being exactly j customers in the facility. Since k_0 may not necessarily be 0 in a particular policy, $P(k_0)$ has a different interpretation – it is the probability of having between 0 and k_0 (inclusive) customers in the front room. Berman et al. define a set of balance equations for the determination of these probabilities:

$$P(j)\lambda = P(j+1)i\mu \quad j = k_{i-1}, k_{i-1}+1, \ldots, k_i - 1 \quad i = 1, ..., N. \tag{1}$$

An additional constraint on the values of $P(j)$ is $\sum_{j=k_0}^{S} P(j) = 1$.

All quantities of interest can be expressed in terms of the probabilities $P(j)$. Expected number of workers in the front room is

$$F = \sum_{i=1}^{N} \sum_{j=k_{i-1}+1}^{k_i} iP(j) \tag{2}$$

while the expected number of workers in the back room is $B = N - F$. The expected number of customers in the front room is

$$L = \sum_{j=k_0}^{S} jP(j). \tag{3}$$

Expected waiting time in the queue can be expressed as

$$W_q = \frac{L}{\lambda(1 - P(S))} - \frac{1}{\mu}. \tag{4}$$

This expression is derived using Little's Laws [6,8] for a system of capacity $k_N = S$.

Given a family of switching policies $\mathbf{K} = \{K; K = \{k_0, k_1, ..., k_{N-1}, S\}$, k_i integer, $k_i - k_{i-1} \geq 1$, $k_0 \geq 0, k_{N-1} < S\}$, the problem can formally be stated as:

$$\begin{gathered} minimize\ W_q \\ s.t\ B \geq B_l \\ equations\ (1), (2), (3), (4). \end{gathered} \tag{5}$$

Note that B, F and L are expected values and can be real-valued. Consequently, the constraint $B \geq B_l$ states that the expected number of workers in the back room resulting from the realization of any policy should be greater than or equal to the minimum expected number of back room workers needed to complete all back room work. At any particular time point, there may, in fact, be fewer than B_l workers in the back room.

2.2 Berman et al.'s Heuristic

Berman et al. propose a heuristic method for the solution of this problem based on the following theorem, the details and proof of which are presented in [3].

Theorem 1 (Berman et al.). *Consider two policies K and K' which are equal in all but one k_i. In particular, suppose that the value of k'_J equals $k_J - 1$, for some J from the set $\{0, ..., N-1\}$ such that $k_J - k_{J-1} \geq 2$, while $k'_i = k_i$ for all $i \neq J$. Then (a) $W_q(K) \geq W_q(K')$, (b) $F(K) \leq F(K')$, (c) $B(K) \geq B(K')$.*

In addition, Berman's heuristic is based on two policies having special properties. Firstly, consider the policy $\hat{K} = \{k_0 = 0, k_1 = 1, \ldots, k_{N-1} = N-1, k_N = S\}$. This policy results in the largest possible F, and the smallest possible B and W_q. Because this policy yields the smallest possible expected waiting time, if it is feasible, then it is optimal. On the other hand, the smallest possible F and the largest possible W_q and B are obtained by applying the policy $\hat{\hat{K}} = \{k_0 = S-N, k_1 = S-N+1, \ldots, k_{N-1} = S-1, k_N = S\}$. Thus, if this policy is infeasible, the problem (5) is infeasible also.

Heuristic P_1 of Berman et al. starts with the policy $\hat{K}$. If this policy is feasible, then the switching point k_i with the smallest index i satisfying the condition $k_i - k_{i-1} > 1$ for $0 < i < N$ and $k_i > 0$ for $i = 0$ is decreased by 1. By Theorem 1, this results in a policy with a better W_q value but smaller B. The heuristic continues decreasing switching points with this property until the resulting policy becomes infeasible (or is the policy $\hat{K}$, in which case P_1 stops because this policy is optimal). Once infeasibility is reached, a switching point k_i having the smallest index and satisfying the condition $k_{i+1} - k_i > 1$, for $i < N$, is increased by 1. By Theorem 1, increasing a switching point with such properties allows the policy to become more feasible in terms of the back room constraint, but also increases W_q. Once a feasible policy is found again, the heuristic tries to find switching points to decrease. Thus, the heuristic alternates between trying to reach a policy with smaller W_q and a policy that is feasible with respect to the B_l constraint. Each time an infeasible policy is found, the set of switching points that can be increased or decreased at subsequent steps is reduced in order to prevent cycling. Assuming the problem is feasible, P_1 stops when it is unable to find any more switching points to decrease or increase, in which case it returns the best feasible policy that it has been able to find. The heuristic guarantees optimality only when the policy it returns is $\hat{K}$ or $\hat{\hat{K}}$.

Empirical results regarding the performance of heuristic P_1 are not presented in [3] and so the ability of P_1 to find good switching policies is not explicitly evaluated. In particular, it is not clear how close policies provided by P_1 are to the optimal policies.

3 Constraint Programming Models

Some work has been done on extending CP to stochastic problems [12,13,16]. Our problem is different from the problems addressed in these papers because all of the stochastic information can be explicitly encoded as constraints and expected values, and there is no need for either stochastic variables or scenarios. The major motivation for our work is to investigate whether CP can be successfully used to solve such problems. To this end, we investigate three CP models for our queue control problem:

– The *If-Then* model is a CP version of the formal definition of Berman et al.

- The *PSums* model uses some slightly different sets of variables, and some constraints are included which are based on closed-form expressions derived from the constraints that are used in the *If-Then* model.
- The *Dual* model includes a set of dual decision variables in addition to the variables used in the *If-Then* and *PSums* models. Most of the constraints of this model are expressed in terms of these dual variables.

Implementation of our models uses the predefined constraints available in standard CP solvers.

The proposed models have some similarities. Firstly, all of them have a set of decision variables k_i, $i = 0, 1, \ldots, N$, representing the switching policy. Each k_i with $i < N$ has the domain $[i, i+1, \ldots, S-N+i]$ (since $k_i < k_{i+1}$) and k_N is constrained to equal S. Secondly, a set of auxiliary variables is included in each model to represent the probabilities needed for the calculation of the quantities of interest. In addition, a constraint stating that $B \geq B_l$, a set of constraints $k_i < k_{i+1}$, for all i from 0 to $N-1$, (since the number of workers in the front room, i, increases only when the number of customers, k_i, increases) and constraint (6) are included in all three models. Constraint (6) ensures that an assignment of all decision variables leads to a unique solution of the balance equations.

$$P(k_0) \sum_{i=0}^{N} \beta Sum(k_i) = 1 \tag{6}$$

The calculation of $\sum_{i=0}^{N} \beta Sum(k_i)$ requires some auxiliary variables, which are defined in Equations (7) and (8). The derivation of these equations, based on expressions presented in [3], can be found in [14].

$$\beta Sum(k_i) = \begin{cases} X_i \left(\frac{\lambda}{\mu}\right)^{k_{i-1}-k_0+1} \left(\frac{1}{i}\right) \left[\frac{1-\left(\frac{\lambda}{i\mu}\right)^{k_i-k_{i-1}}}{1-\left(\frac{\lambda}{i\mu}\right)}\right] & \text{if } \frac{\lambda}{i\mu} \neq 1 \\ X_i \left(\frac{\lambda}{\mu}\right)^{k_{i-1}-k_0+1} \left(\frac{1}{i}\right) (k_i - k_{i-1}) \; otherwise. \end{cases} \tag{7}$$

$$X_i = \prod_{g=1}^{i-1} \left(\frac{1}{g}\right)^{k_g - k_{g-1}} \quad i = 1, \ldots, N; \quad (X_1 \equiv 1) \tag{8}$$

All models also include constraints for representing F, L and W_q. However, the expressions for F and L differ slightly depending on the model, as described below.

3.1 *If-Then* Model

The initial model is based directly on the formulation of Berman et al. The model includes the variables $P(j)$ for $j = k_0, k_0+1, \ldots, k_1, k_1+1, \ldots, S-1, S$, each representing the probability of there being j customers in the front room. These are floating point variables with domain $[0..1]$. The balance equations are represented by a set of

if-then constraints. For example, the first balance equation, $P(j)\lambda = P(j+1)\mu$ for $j = k_0, k_0+1, ..., k_1-1$, is represented by the constraint: $(k_0 \leq j \leq k_1 - 1) \rightarrow P(j)\lambda = P(j+1)\mu$. Thus, somewhat inelegantly, an if-then constraint of this kind has to be added for each j between 0 and $S-1$ in order to represent one balance equation. In order to represent the rest of these equations, this technique has to be applied for each pair of switching points k_i, k_{i+1} for i from 0 to $N-1$. In addition, such if-then constraints are used for Equation (2), due to the dependence of this constraint on sums of variables between two switching points.

3.2 *PSums* Model

In order to avoid the if-then constraints, closed-form expressions[3] for the sums of probabilities between two switching points were derived and used as the basis of the second model. The set of $P(j)$ variables from the formulation of Berman et al. is replaced by a set of $PSums(k_i)$ variables for $i = 0, ..., N-1$, together with a set of probabilities $P(k_i)$ for $i = 0, 1, 2, \ldots, N$. The $PSums(k_i)$ variable represents the sum of all probabilities between k_i and $k_{i+1} - 1$ and is defined in Equation (9). Equation (10) is a recursive formula for computing $P(k_i)$. $P(k_0)$ can be computed using Equation (6).

$$PSums(k_i) = \begin{cases} P(k_i)\dfrac{1 - \left[\dfrac{\lambda}{(i+1)\mu}\right]^{k_{i+1}-k_i}}{1 - \dfrac{\lambda}{(i+1)\mu}} & \text{if } \frac{\lambda}{(i+1)\mu} \neq 1 \\ P(k_i)(k_{i+1} - k_i) \quad otherwise. \end{cases} \tag{9}$$

$$P(k_{i+1}) = \left[\frac{\lambda}{(i+1)\mu}\right]^{k_{i+1}-k_i} P(k_i) \tag{10}$$

All quantities of interest can be expressed in terms of the $PSums(k_i)$ variables and the switching point probabilities, $P(k_i)$. In particular, the expected number of workers in the front room is

$$F = \sum_{i=1}^{N} i \left[PSums(k_{i-1}) - P(k_{i-1}) + P(k_i)\right]. \tag{11}$$

L, the expected number of customers in the front room, is

$$L = \sum_{i=0}^{N-1} L(k_i) + k_N P(k_N) \tag{12}$$

[3] The derivation of most of these expressions is based on expressing each $P(j)$ in terms of $P(k_i)$ for $k_i \leq j$ via the balance equations. In some derivations, such as that of Equation (9), the geometric series formula is used. Details can be found in [14].

where

$$L(k_i) = k_i PSums(k_i) + P(k_i)\left[\frac{(\frac{\lambda}{(i+1)\mu})^{k_{i+1}-k_i}(k_i-k_{i+1})+(\frac{\lambda}{(i+1)\mu})^{k_{i+1}-k_i+1}(k_{i+1}-k_i-1)+\frac{\lambda}{(i+1)\mu}}{\left[\frac{(i+1)\mu-\lambda}{(i+1)\mu}\right]^2}\right]. \quad (13)$$

3.3 *Dual* Model

The problem can be alternatively formulated using variables w_j, which represent the number of workers in the front room when there are j customers present. Several expressions and constraints of the above models can be simplified by using these variables. Firstly, the balance equations can be stated as

$$P(j)\lambda = P(j+1)w_{j+1}\mu \qquad j = 0, 1, \ldots, S-1. \quad (14)$$

This formulation of the balance equations avoids the inefficient if-then constraints.

Secondly, F, the expected number of workers in the front room, can be stated as

$$F = \sum_{j=0}^{S} w_j P(j). \quad (15)$$

The difficulty with this model arises from the fact the $P(j)$ variables should be defined only for $j \geq k_0$ (since $P(k_0)$ is the probability of having from 0 to k_0 customers in the front room). It is hard to express this condition without explicitly having the variable k_0 in the model. Because of this, and since it is known that adding redundant variables to a model may be beneficial [11], it was decided that both the k_i and the w_j variables would be included in this model. In order to use two sets of redundant variables, the following channelling constraints[4] have to be included:

$$w_j < w_{j+1} \;\leftrightarrow\; k_{w_j} = j \qquad j = 0, 1, \ldots, S-1, \quad (16)$$

$$w_j = w_{j+1} \;\leftrightarrow\; k_{w_j} \neq j \qquad j = 0, 1, \ldots, S-1, \quad (17)$$

$$w_j = i \;\leftrightarrow\; k_{i-1} + 1 \leq j \leq k_i \qquad j = 0, 1, \ldots, S, \;\; i = 1, \ldots, N. \quad (18)$$

Additional constraints on the worker variables that are included in the model are: $w_0 = 0$, $w_S = N$ and $w_j \leq w_{j+1}$ for all j from 0 to $S-1$.

Preliminary experiments with these models showed poor performance. As one might expect from a problem with few constraints between decision variables, there was little constraint propagation, and search was required to essentially investigate the entire branch-and-bound tree. As a consequence, we examine shaving [4,9].

[4] Constraints (16) and (17) are redundant given the constraint $w_j \leq w_{j+1}$. However, such redundancy can often lead to increased propagation [7]. In future work, we will examine the effect that removing one of these constraints may have on the performance of the program.

4 Shaving

Shaving is a procedure for enforcing consistency in CSPs. It is based on temporarily adding constraints to the problem, performing propagation and making inferences according to the resulting state of the problem [5, 15]. In our proposed shaving algorithm, *AlternatingSearchAndShaving*, two shaving procedures are run initially until they are no longer able to make domain reductions. Search is then performed until a better solution is found, at which point the shaving procedures are applied again. Subsequently, search and shaving alternate until one of them proves optimality of the best solution found. The first of the two shaving procedures makes inferences based on the feasibility of policies with respect to the B_l constraint, while the second one is based on the constraint $W_q \leq bestW_q$, where $bestW_q$ is the W_q value of the best policy found up to that point. In both shaving procedures, if the inferred constraint violates the current upper or lower bound of a k_i, then the best policy found up to that point is optimal.

B_l Shaving. Let $\min(k_i)$ and $\max(k_i)$ be, respectively, the smallest and largest values in the current domain of variable k_i. At each step of the B_l-based shaving procedure, $k_i = \min(k_i)$ or $k_i = \max(k_i)$ is temporarily added to the model for $i \in \{0, ..., N-1\}$. If $k_i = \min(k_i)$ is added, then all other switching points are assigned the maximum possible values subject to the condition that $k_n < k_{n+1}, \forall n \in \{0, ..., N-1\}$. If the resulting policy is infeasible, the constraint $k_i > \min(k_i)$ can be permanently added: if all variables except k_i are set to their maximum values, and the problem is infeasible (based on the B_l constraint), then, by Theorem 1, in any feasible policy k_i must be greater than $\min(k_i)$. If $k_i = \max(k_i)$ is added, all other switching points are assigned the minimum possible values. If the resulting policy is feasible, the constraint $k_i < \max(k_i)$ can be permanently added to the model. Since all variables except k_i are at the minimum values already, and k_i is at its maximum, it must be true, again by Theorem 1, that in any better solution the value of k_i has to be smaller than $\max(k_i)$. In both cases, after the resulting policy is checked for feasibility, the temporary constraint is removed.

W_q Shaving. The W_q-based shaving procedure makes inferences based strictly on the constraint $W_q \leq bestW_q$. The constraint $B \geq B_l$ is removed prior to running this procedure in order to eliminate the possibility of incorrect inferences. Similarly to the B_l-based shaving procedure, a constraint of the form $k_i = \max(k_i)$ is added and the smallest possible values are assigned to the rest of the variables. As the B_l constraint has been removed, the only reason why the policy could be infeasible is because it has a W_q value greater than the best W_q that has been encountered so far. Since all switching points except k_i are assigned their smallest possible values, this implies that in any solution with a better W_q, the value of k_i has to be strictly smaller than $\max(k_i)$.

5 Experimental Results

Several sets of experiments were performed in order to evaluate the efficiency of the proposed models and the shaving procedure, as well as to compare the performance of the best CP model with the performance of the heuristic proposed by Berman et al. All

CP models were implemented in ILOG Solver 6.2, while Berman et al.'s heuristic was implemented using C++. In all models, search assigns switching points in increasing index order and the smallest value in the domain of each variable is tried first.

The experimental results presented here are based only on the instances for which the optimal is between $\hat{K}$ and $\hat{\hat{K}}$, as instances in which either of these policies is optimal, or $\hat{\hat{K}}$ is infeasible, are easily solved both by the heuristic and the CP models with the elementary B_l-based shaving procedure. Preliminary experiments indicated that the value of S has a significant impact on the efficiency of the algorithms since higher values of S result in larger domains for the k_i variables for all models and also a higher number of w_j variables for the *Dual* model. Therefore, we considered instances for each value of S from the set $\{10, 20, \ldots, 100\}$ in order to gain an accurate understanding of the performance of the model and the heuristic.[5] Thirty feasible instances for which the optimal policy is neither $\hat{K}$ nor $\hat{\hat{K}}$ were generated for each S. A 10-minute time limit on the overall run-time of the program was enforced in the experiments. All experiments were performed on a Dual Core AMD 270 CPU with 1 MB cache, 4 GB of main memory, running Red Hat Enterprise Linux 4.

In order to perform comparisons between the CP models and the heuristic, we look at the number of instances in which the optimal solution was found and in which optimality was proved, and the mean relative error (MRE). MRE is a measure of solution quality that allows one to observe how quickly a particular algorithm is able to find a good solution. MRE is defined as $\frac{1}{|M|}\sum_{m\in M}\frac{c(a,m)-c^*(m)}{c^*(m)}$, where a is a particular algorithm used to solve the problem, M is the set of problem instances on which the algorithm is being tested, $c(a, m)$ is the cost of a solution found for instance m by algorithm a, and $c^*(m)$ is the best solution for instance m found during our experiments.

5.1 Comparison of Constraint Programming Models

Table 1 presents, for each model, the number of instances in which it finds the best solution (out of 300), the number of instances in which it finds the optimal solution (out of the 240 instances for which the optimal solution is known), and the number of times it proves optimality. It can be seen that all models find the optimal solution in the 240 instances for which it is known. However, the *PSums* model outperforms the other two models in the rest of the performance measures, proving optimality in 79% of all instances, and finding the best-known solution of any algorithm in 97.3% of all the instances considered.

Observations from Table 1 can be further confirmed by looking at Figure 1 (Left). The figure shows how MRE changes over the first 50 seconds of run-time for *If-Then*, *PSums* and *Dual* models with *AlternatingSearchAndShaving*, and for P_1 (we comment on the performance of P_1 in Section 5.2). It can be seen that *PSums* is, on average, able to find better solutions than the other two models given the same amount of run-time.

[5] For most instances with S greater than 100, neither our method nor Berman et al.'s heuristic P_1 may be used due to numerical instability. The maximum value of S used in the experiments of Berman et al. is also 100.

Table 1. Comparison of three CP models with *AlternatingSearchAndShaving* with Berman's Heuristic P_1. The Hybrid model is presented in Section 6.

	# best found (/300)	# optimal found (/240)	# optimal proved (/300)
PSums	292	240	238
If-Then	280	240	234
Dual	281	240	234
P_1	282	239	0
PSums-P_1 Hybrid	300	240	238

5.2 P_1 vs. the Best Constraint Programming Approach

It can be seen, from Table 1, that the heuristic performs extremely well, finding the best-known solution in only ten fewer instances than the *PSums* model, in two more instances than the *If-Then* model and in one more than the *Dual*. Moreover, it finds, but, of course, cannot prove, the optimal solution in 79.6% of all instances (239 out of the 240 instances for which the optimal is known). Its run-time is negligible, while the mean run-time of the best CP model, *PSums*, is approximately 130 seconds.

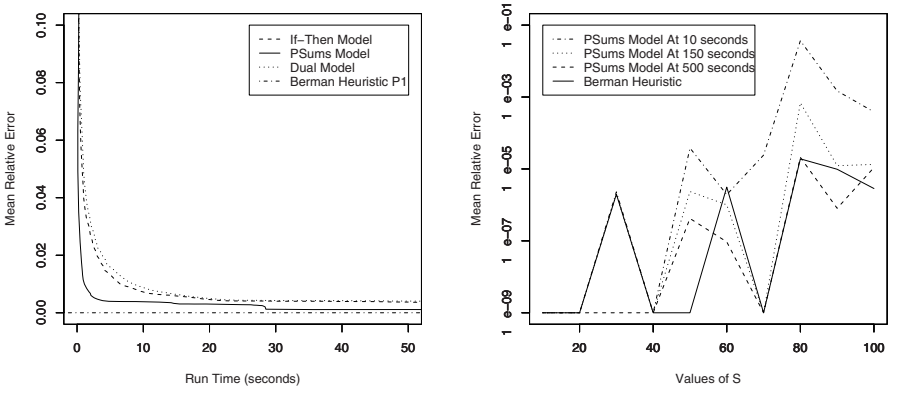


Fig. 1. Left: Comparison of MRE of three CP models with *AlternatingSearchAndShaving* with Berman's P_1 heuristic. Right: MRE for each value of S for P_1 and the $PSums$ model.

From Figure 1 (Left), it can be observed that the heuristic achieves a very small MRE in a negligible amount of time. After about 50 seconds of run-time, the MRE over 300 instances resulting from *PSums* with *AlternatingSearchAndShaving* becomes comparable to that of the heuristic MRE.

In Figure 1 (Right), the MRE for the 30 instances for each value of S is presented for P_1 and for *PSums* with *AlternatingSearchAndShaving* at 10, 150 and 500 seconds of run-time. After 10 seconds, the performance of *PSums* is comparable to that of the heuristic for $S \leq 40$, but the heuristic appears to be quite a bit better for higher values of S. At 150 seconds, the performance of *PSums* is comparable to that of the heuristic

except at S of 50 and 80. After 500 seconds, *PSums* has a smaller or equivalent MRE over the 300 instances and also a lower MRE for each value of S except 50 and 100.

Overall, these results indicate that P_1 performs extremely well—its run-time is negligible, it finds the optimal solution in 79.6% of the instances and the best-known solution in 94%. Moreover, it results in very low MRE. Although *PSums* with *AlternatingSearchAndShaving* is able to achieve better performance in both of these measures, it is clear that these improvements are quite small given that the *PSums* run-time is so much higher than the run-time of the heuristic.

6 *PSums*-P_1 Hybrid

Naturally, it is desirable to create a method that would be able to find a solution of high quality in a short amount of time, as does Berman's heuristic, and that would also have the same high rate of being able to prove optimality within a reasonable run-time as does *PSums* with *AlternatingSearchAndShaving*. It is therefore worthwhile to experiment with a *PSums*-P_1 Hybrid, which starts off by running P_1 and then, assuming the instance is feasible, uses the *PSums* model with *AlternatingSearchAndShaving* to find a better solution or prove the optimality of the solution found by P_1 (infeasibility of an instance is proven if the heuristic determines that policy $\hat{K}$ is infeasible).

Since P_1 is very fast, running it first incurs almost no overhead. We have also shown that P_1 provides solutions of very high quality (in 94% of instances used in the experiments, it found the best-known solution). Therefore, the first iteration of the W_q-based procedure should be able to significantly prune the domains of switching point variables because of the good quality solution found by the heuristic. Continuing by alternating the two shaving techniques and search, which has also been shown to be an effective approach, should result in at least as many instances for which optimality is proven as for the *PSums* model with *AlternatingSearchAndShaving*.

The proposed hybrid algorithm was tested on the same set of 300 instances that was used above. Results of the hybrid are presented in Table 1. The hybrid was able to find the best-known solution in all 300 cases while still being able to prove optimality in as many cases as *PSums*. Thus, in spite of the good quality solutions that are discovered quickly because of the heuristic, the domains of switching points in some instances are not reduced enough to increase the number of cases in which optimality is proved. The mean run-time for the hybrid is 130.18 seconds, which is essentially identical to the mean run-time of 129.98 seconds for *PSums* with *AlternatingSearchAndShaving*.

Thus, the hybrid is the best choice for solving this problem: it finds as good a solution as the heuristic in as little time (close to 0 seconds), it is able to prove optimality in as many instances as the best pure CP method, and it finds the best-known solution in all instances considered. Moreover, all these improvements are achieved with a negligible increase in the average run-time over the *PSums* model with shaving.

7 Discussion

In this section, we examine some of the reasons for the poor performance of the CP models without shaving and suggest reasons for the observed differences among them.

7.1 Lack of Back-Propagation

In our experiments, we have some instances for which even the *PSums*-P_1 hybrid with *AlternatingSearchAndShaving* is unable to find and prove optimality within the 10-minute time limit. Further analysis of the algorithm's behaviour suggests that this performance can be explained by the lack of back-propagation. Back-propagation refers to the pruning of the domains of the decision variables due to the addition of a constraint on the objective function: the objective constraint propagates "back" to the decision variables, removing domain values and so reducing search. In the CP models presented above, there is very little back-propagation. We illustrate this by focusing on the *PSums* model without shaving.

Throughout search, if a new best solution is found, the constraint $W_q \leq bestW_q$, where $bestW_q$ is the new objective value, is added to the model. However, the domains of the switching point variables are usually not reduced in any way after the addition of such a constraint. This can be illustrated by observing the amount of propagation that occurs in the model when W_q is constrained.

For example, consider an instance of the problem with $S = 6$, $N = 3$, $\lambda = 15$, $\mu = 3$, and $B_l = 0.32$. The initial domains of the switching point variables are $[0..3]$, $[1..4]$, $[2..5]$ and $[6]$. The initial domains of the probability variables $P(k_i)$ for each i, after the addition of W_q bounds provided by $\hat{K}$ and $\hat{\hat{K}}$, are listed in Table 2. The initial domain of W_q, also determined by the objective function values of $\hat{K}$ and $\hat{\hat{K}}$, is $[0.22225..0.425225]$. The initial domains of L and F, are $[2.8175e^{-7}..6]$ and $[0..2.68]$, respectively. Upon the addition of the constraint $W_q \leq 0.306323$, where 0.306323 is the known optimal value for this instance, the domain of W_q is reduced to $[0.22225..0.306323]$, the domain of L becomes $[1.68024..6]$ and the domain of F remains $[0..2.68]$. The domains of $P(k_i)$ after this addition are listed in Table 2. The domains of both types of probability variables are reduced by the addition of the new W_q constraint. However, the domains of the switching point variables remain unchanged. Therefore, even though all policies with value of W_q less than 0.306323 are infeasible, constraining W_q to be less than this value does not result in any reduction of the search space. It is still necessary to enumerate all remaining policies in order to show that no better feasible solution exists.

One of the reasons for the lack of pruning of the domains of the k_i variables due to the W_q constraint is likely the complexity of the expression for W_q. In particular, recall that W_q is expressed in all models as $W_q = \frac{L}{\lambda(1-P(S))} - \frac{1}{\mu}$. In the example above, when W_q is constrained to be less than or equal to 0.306323, we get the constraint $0.306323 \geq \frac{L}{15(1-P(S))} - \frac{1}{3}$, which implies that $9.594845(1 - P(S)) \geq L$. This explains why the domains of both L and $P(S)$ change upon this addition to the model. The domains of the rest of the $P(k_i)$ variables change because of the relationships between the $P(k_i)$s (Equation (10)) and because of the constraint that the sum of all probability variables has to be 1. Similarly, the domains of $PSums(k_i)$ change because these variables are expressed in terms of $P(k_i)$ (Equation (9)). However, because the actual k_i variables mostly occur as exponents in expressions for $PSums(k_i)$, $P(k_i)$, and $L(k_i)$, the minor changes in the domains of $PSums(k_i)$, $P(k_i)$, or $L(k_i)$ that happen due to the constraint on W_q have no effect on the domains of the k_i. This analysis suggests

Table 2. Domains of $P(j)$ and $PSums(j)$ variables for $j = k_0, k_1, k_2, k_3$, before and after the addition of the constraint $W_q \leq 0.306323$

	Before addition of $W_q \leq 0.306323$		After addition of $W_q \leq 0.306323$	
j	$P(j)$	$PSums(j)$	$P(j)$	$PSums(j)$
k_0	$[4.40235e^{-6}..0.979592]$	$[0..1]$	$[4.40235e^{-6}..0.979592]$	$[0..0.683666]$
k_1	$[1.76094e^{-7}..1]$	$[0..1]$	$[0.000929106..1]$	$[0..0.683666]$
k_2	$[2.8175e^{-8}..0.6]$	$[2.8175e^{-8}..1]$	$[0.0362932..0.578224]$	$[0.0362932..0.71996]$
k_3	$[4.6958e^{-8}..1]$	N/A	$[0.28004..0.963707]$	N/A

that it may be interesting to investigate a CP model based on log-probabilities rather than on the probabilities themselves. Such a model may lead to stronger propagation.

7.2 Differences in the Constraint Programming Models

Experimental results demonstrate that the best CP model of those proposed is *PSums*. In all models, the shaving procedures make the same number of domain reductions because shaving is based on the W_q and B_l constraints. However, the time that each shaving iteration takes is dependent on the model. Our empirical results show that each iteration of shaving takes a smaller amount of time with the *PSums* model than with the other two. This appears to be the primary reason for the *PSums* model finding good solutions faster than the other models, as shown in Figure 1 (Left). *PSums* is radically different from the other two models because it does not include an explicit representation of the balance equations. This model thus avoids the if-then constraints required in the *If-Then* model. Moreover, *PSums* has a smaller number of probability variables than the other two models, because it calculates sums of probabilities between two switching points rather than the probability of j customers being present in the front room for all j from 0 to S. This reduces the number of probability variables from $S + 1$ to $2N + 1$. In addition, the probability variables included in this model are more tightly linked by the closed-form expressions. Thus, because of the tighter links between variables, and a smaller number of variables, each iteration of shaving in *PSums* takes a smaller amount of time than in the other two models.

A comparison of the *If-Then* model with the *Dual* using Figure 1 shows that the *If-Then* model is usually able to find good solutions in a smaller amount of time. This is slightly surprising because the *Dual* model uses a much simpler representation of the balance equations and the expression for F, avoiding the use of if-then constraints. One possible explanation for the *Dual* sometimes taking more time to find a good solution is that, at each shaving iteration, it has to assign more variables (via propagation) than the other two models. In particular, in order to represent a switching policy, the *Dual* has to assign S w_j variables in addition to N k_i variables (usually, S is much larger than N).

On the other hand, within the given time limit, the *Dual* found the best solution in one more instance than the *If-Then* model. This may be due to an increase in the amount of propagation which results from the use of dual variables. In fact, an examination of the initial domains of the probability variables for the example instance of Section 7.1 shows that these domains are quite a bit smaller in the *Dual* model than in the *If-Then* model (e.g. the domain of $P(0)$ is $[0..1]$ in the *If-Then* model, while it

is $[0..0.00926208]$ in the *Dual*). This examination also shows that the initial domains of probability variables in the *If-Then* and *Dual* models are actually smaller than those in the *PSums* model. This implies that there exist some instances in which more initial propagation occurs in the *If-Then* model or the *Dual* model than in *PSums*.

8 Related Work and Possible Extensions

Several papers exist that deal with similar types of problems as the one considered here. For example, Berman & Larson [2] study a similar problem of switching workers between two rooms in a retail facility where the customers in the front room are divided into two categories, those "shopping" in the store and those at the checkout. Similarly, Palmer & Mitrani [10] consider the problem of switching computational servers between different types of jobs where the randomness of user demand may lead to unequal utilization of resources. Batta et al. [1] study the problem of assigning cross-trained customer service representatives to different types of calls in a call centre, depending on estimated demand patterns for each type of call. These three papers provide examples of problems for which CP could prove to be a useful approach. Investigating CP solutions to these problems is therefore one possible direction of future work. In particular, it may be interesting to look at problems with more complex constraints (e.g., on capacities or between workers) that may be naturally suitable for the CP approach. A complementary direction is to study the basic models of queueing theory in order to understand the applicability of CP.

9 Conclusions

In this paper, a constraint programming approach is proposed for the problem of finding the optimal times to switch workers between the front room and the back room of a retail facility under stochastic customer arrival and service times. This is the first work of which we are aware that examines solving such stochastic queueing control problems using constraint programming. The best pure CP method proposed is able to prove optimality in a large proportion of instances within a 10-minute time limit. Previously, there existed no non-heuristic solution to this problem aside from naive enumeration. As a result of our experiments, we hybridized the best pure CP model with the heuristic proposed for this problem in the literature. This hybrid technique is able to achieve performance that is equivalent to, or better than, that of each of the individual approaches alone: it is able to find very good solutions in a negligible amount of time due to the use of the heuristic, and is able to prove optimality in a large proportion of problem instances within 10 CPU minutes due to the CP model.

This work demonstrates for the first time that constraint programming can be a good approach for solving a stochastic optimization problem based on queueing theory.

References

1. R. Batta, O. Berman, and Q. Wang. Balancing staffing and switching costs in a call/service center. *European Journal of Operations Research*. To Appear. Available at: http://www.acsu.buffalo.edu/~batta/papers/Batta_et_al.pdf.

2. O. Berman and R. Larson. A queueing control model for retail services having back room operations and cross-trained workers. *Computers and Operations Research*, 31(2):201–222, 2004.
3. O. Berman, J. Wang, and K. P. Sapna. Optimal management of cross-trained workers in services with negligible switching costs. *European Journal of Operations Research*, 167(2):349–369, 2005.
4. Y. Caseau and F. Laburthe. Cumulative scheduling with task intervals. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 363–377. MIT Press, 1996.
5. S. Demassey, C. Artigues, and P. Michelon. Constraint-propagation-based cutting planes: An application to the resource-constrained project scheduling problem. *INFORMS Journal on Computing*, 17(1):52–65, 2005.
6. D. Gross and C. Harris. *Fundamentals of Queueing Theory*. John Wiley & Sons, Inc., 1998.
7. B. Hnich, B. Smith, and T. Walsh. Dual modelling of permutation and injection problems. *Journal of Artificial Intelligence Research*, 21:357–391, 2004.
8. J. D. C. Little. A proof of the queueing formula $L = \lambda W$. *Operations Research*, 9:383–387, 1961.
9. P. Martin and D. B. Shmoys. A new approach to computing optimal schedules for the job shop scheduling problem. In *Proceedings of the Fifth Conference on Integer Programming and Combinatorial Optimization*, pages 389–403, 1996.
10. J. Palmer and I. Mitrani. Optimal server allocation in reconfigurable clusters with multiple job types. In *Proceedings of the Computational Science and Its Applications International Conference*, pages 76–86, 2004.
11. B.M. Smith. Modelling for constraint programming. Lecture Notes for the First International Summer School on Constraint Programming, 2005. Available at: http://www.math.unipd.it/˜frossi/cp-school/.
12. S.A. Tarim, S. Manandhar, and T. Walsh. Stochastic constraint programming: A scenario-based approach. *Constraints*, 11(1):53–80, 2006.
13. S.A. Tarim and A. Miguel. A hybrid Benders' decomposition method for solving stochastic constraint programs with linear recourse. In *Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming*, pages 133–148, 2005.
14. D. Terekhov and J. C. Beck. Solving a stochastic queueing control problem with constraint programming. Technical Report MIE-OR-TR2006-06, Department of Mechanical and Industrial Engineering, University of Toronto, 2006. Available from http://www.mie.utoronto.ca/labs/ORTechReps/.
15. M.R.C. van Dongen. Beyond singleton arc consistency. In *Proceedings of the 17th European Conference on Artificial Intelligence*, pages 163–167, 2006.
16. T. Walsh. Stochastic constraint programming. In *Proceedings of the 15th European Conference on Artificial Intelligence*, pages 111–115, 2002.

Constrained Clustering Via Concavity Cuts

Yu Xia*

Center of Operations Research and Econometrics (CORE), Université catholique de Louvain, 34 Voie du Roman Pays, 1348 Louvain-la-Neuve, Belgium
xiay@optlab.mcmaster.ca.

Abstract. In this paper, we adapt Tuy's concave cutting plane method to the problem of finding an optimal grouping of semi-supervised clustering. We also give properties of local optimal solutions to the semi-supervised clustering. On test data sets with up to 1500 points, our algorithm typically find a solution with objective value around 2% smaller of the initial function value than that obtained by k-means algorithm within 4 seconds, although the run time is hundred times of that of the k-means algorithm.

Keywords: Partitional clustering, semi-supervised clustering, instance-level constraints, global optimization, concave cutting plane method.

1 Introduction

We are interested in efficient algorithms for partitional clustering, i.e. partitioning n points (or patterns, objectives), denoted as $\mathbf{a}_1, \ldots, \mathbf{a}_n \in \mathbb{R}^d$, into k clusters (or groups, classes) $C_1, \ldots, C_k$. Clustering is a fundamental tool in statistics, computer science, etc. And it has been widely used in areas such as finance, biology. See [7] for applications of it.

We use the k-means (square-error) criteria in this paper, as it is the most common clustering criterion. For the k-means criteria, the dissimilarity measure is the squared Euclidean distance; the task is to assign each point to its closest cluster centroid, denoted as $\mathbf{c}_i\,(i = 1, \ldots, k)$, which are representations of corresponding clusters. The objective function (the square-error loss) is the following

$$\sum_{j=1}^{k} \sum_{\mathbf{a}_i \text{ assigned to } C_j} \|\mathbf{a}_i - \mathbf{c}_j\|_2^2 \ .$$

The most widely used algorithm for partional clustering is the k-means algorithm. It works as follows. Starting from an initial partition, it assigns each pattern to its closed centroid. Then it re-calculate the centroids and re-assign each pattern. It repeats until there is no re-assignment. Unfortunately, this algorithm may not produce even a local optimal solution. An example in [8] shows

* Research was done when the author was a JSPS foreign research fellow at the Institute of Statistical Mathematics, Japan. The author thanks support from JSPS (Japan Society for the Promotion of Science). The author also thanks three anonymous referees for their helpful suggestions and comments.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 318–331, 2007.

that a solution of the k-means algorithm can have arbitrarily high approximation ratio. it is desirable that the clustering results be as accurate as possible. For instance, in categorizing the customers of a bank according to their credit risk, it is bad to assign a low credit risk customer to the high risk group; and it is worse to put a high risk customer to the low risk group.

Labeled data are good sources to improve the accuracy of clustering, but they may not be easy or cheap to obtain. In many applications, individual labels are not known a priori; however, some instance-level information about the relationship between some points may be available. Instance-level constraints include must-link constraints, i.e. some points should have the same, but unknown, label, and cannot-link constraints, i.e. two points should be in different, but unknown, groups. Wagstaff et al. in [13] show that incorporating instance-level constrains in k-means clustering can reduce the number of iterations and total runtime of clustering, provide a better initial solution, increase accuracy, generalize the constraint information to improve performance on the unconstrained instances as well. Another application of instance-level constraints in clustering is the learning of a distortion measure. It is observed empirically that groups obtained by k-means clustering tend to be equal-numbered and hyper-spherical-shaped [5]. In [15], Xing et al. show that with a Mahalanobis distance metric instead of the Euclidean distance, the k-means clustering can identify clusters of hyper-ellipsoidal shape. Their numerical experiments show that a Mahalanobis distance metric with the instance-level constraints can improve the k-means clustering. In [3], Bilenko et al. further incorporate metric learning in the seeding of k-means, and in learning individual Mahalanobis distance metric for each cluster. Clustering with instance-level constraints, also known as semi-supervised clustering, has been studied in [4], [5], [10], [9], [2], and some other papers as well. Their numerical results show that instance-level constraints do help improve the performance of k-means algorithm. However, the algorithms they use are some variants of the k-means or EM algorithm, which can not guarantee to find a global minimum.

In this paper, we use a global optimization approach to solve the semi-supervised clustering. Our numerical results show that this algorithm can get a better solution than k-means algorithm does.

The remaining of the paper is organized as follows. In § 2, we give our mathematical model of the semi-supervised clustering. In § 3, we discuss the properties of a local solution to the mathematical model. In § 4, we describe the concavity cutting plane method for the model. In § 5, we give some numerical examples.

2 The Mathematical Model

To describe our approach to the semi-supervised clustering, in this part, we give our mathematical model of the semi-supervised clustering problem.

The integer programming model. Let $X = (x_{ij})$ denote the cluster membership matrix, with $x_{ij} = 1$ (resp., 0) if the ith pattern belongs (resp., does not

belong) to cluster C_j $(i = 1, \ldots, n;\ j = 1, \ldots, k)$. Then the constrained k-means clustering problem can be modeled as the following:

$$\min_{x_{ij},\, \mathbf{c}_j} \quad \sum_{j=1}^{k} \sum_{i=1}^{n} x_{ij} \, \|\mathbf{a}_i - \mathbf{c}_j\|_{M_j}^2 \tag{1a}$$

$$\text{s.t.} \quad \sum_{j=1}^{k} x_{ij} = 1 \quad (i = 1, \ldots, n) \tag{1b}$$

$$x_{rj} = x_{sj} \quad (r - s \text{ must-linked};\ r, s = 1, \ldots, n;\ j = 1, \ldots, k) \tag{1c}$$

$$x_{pj} + x_{qj} \le 1 \quad (p - q \text{ cannot-linked};\ p, q = 1, \ldots, n;\ j = 1, \ldots, k) \tag{1d}$$

$$x_{ij} \in \{0, 1\} \quad (i = 1, \ldots, n;\ j = 1, \ldots, k)\ . \tag{1e}$$

The constraint (1b) ensures that each pattern $\mathbf{a}_i$ is assigned to one and only one cluster. The must-link constraint (1c) imposes patterns $\mathbf{a}_r$ and $\mathbf{a}_s$ be in a same cluster. The cannot-link constraint (1d) forces patterns $\mathbf{a}_p$ and $\mathbf{a}_q$ be assigned to different clusters. M_j is the distance metric for cluster C_j, which is symmetric positive semidefinite. The metric is Euclidean if $M_j = I$. The metric may be obtained a priori or learned from the data.

Observe that (1) is a nonconvex nonlinear integer programming model, which is hard to solve. Instead, we consider its continuous relaxation.

We first define some notations on the domain $X \ge 0$ (which means $x_{ij} \in \mathbb{R}^+$ and $x_{ij} \ge 0$ $(i = 1, \ldots, n;\ j = 1, \ldots, k)$) . Let $\mathbf{x}_j$ represent the jth column of X, which is the membership vector of cluster C_j. We denote the number of patterns in cluster C_j by $n_j \stackrel{\text{def}}{=} \sum_{i=1}^{n} x_{ij}$. When $n_j > 0$, under given $\mathbf{x}_j$, $\sum_{i=1}^{n} x_{ij} \|\mathbf{a}_i - \mathbf{c}_j\|_{M_j}^2$ is convex in $\mathbf{c}_j$ (a real-valued function $f(\mathbf{x})$ is convex on its domain D if $\forall\, \mathbf{x}_1, \mathbf{x}_2 \in D$ and $0 \le \lambda \le 1$, $f(\lambda \mathbf{x}_1) + f[(1-\lambda)\mathbf{x}_2] \le f(\lambda \mathbf{x}_1 + (1-\lambda)\mathbf{x}_2)$); so the minimum with regard to $\mathbf{c}_i$ is attended at $\mathbf{c}_j = \frac{\sum_{i=1}^{n} x_{ij} \mathbf{a}_i}{\sum_{i=1}^{n} x_{ij}}$, the centroid of C_j, where $\frac{\partial \left(\sum_{i=1}^{n} x_{ij} \|\mathbf{a}_i - \mathbf{c}_j\|_{M_j}^2 \right)}{\partial \mathbf{c}_j} = \mathbf{0}$. Note that $n_j = 0$ implies $\mathbf{x}_j = \mathbf{0}$. In this case, the minimum of the objective function is attained at $\mathbf{c}_j = \mathbf{0}$. Therefore, in the remaining of the paper, we set

$$\mathbf{c}_j = \begin{cases} \frac{\sum_{i=1}^{n} x_{ij} \mathbf{a}_i}{\sum_{i=1}^{n} x_{ij}} & n_j > 0\ , \\ \mathbf{0} & n_j = 0\ . \end{cases} \tag{2}$$

Denote the square-error, or within-cluster variation, of cluster C_j as

$$\mathrm{SSE}_j(\mathbf{x}_j) = \begin{cases} \sum_{i=1}^{n} x_{ij} \left\| \mathbf{a}_i - \frac{\sum_{i=1}^{n} x_{ij} \mathbf{a}_i}{\sum_{p=1}^{n} x_{pj}} \right\|_{M_j}^2 & n_j > 0\ , \\ 0 & n_j = 0\ . \end{cases}$$

Then the sum of square-error for the clustering is $\mathrm{SSE}(X) = \sum_{j=1}^{k} \mathrm{SSE}_j(\mathbf{x}_j)$. And the objective function (1a) is $\min_X \mathrm{SSE}(X)$.

The continuous relaxation. The continuous relaxation of (1) can then be written as:

$$\begin{array}{ll} \min_X & \mathrm{SSE}(X) \\ \text{s.t.} & \sum_{j=1}^{k} x_{ij} = 1 \quad (i = 1, \ldots, n) \\ & x_{rj} = x_{sj} \quad (r - s \text{ must-linked}; r, s = 1, \ldots, n;\ j = 1, \ldots, k) \\ & x_{pj} + x_{qj} \le 1 \quad (p - q \text{ cannot-linked}; p, q = 1, \ldots, n; j = 1, \ldots, k) \\ & x_{ij} \ge 0 \quad (i = 1, \ldots, n;\ j = 1, \ldots, k)\ . \end{array} \tag{3}$$

3 The Solutions

In this part, we study the characteristics of (1) and (3). Especially, we show that (3) is a concave program and describe its extreme points.

3.1 General Properties of the Mathematical Model

We first show that (3) is a concave program, i.e., $-\mathrm{SSE}(X)$ is convex and the feasible region is convex.

Lemma 1. *The function* $\mathrm{SSE}(X)$ *is concave and continuously differentiable over* $X \ge 0$.

Proof. We only need to show that the Hessian of $\mathrm{SSE}(X)$ exists and is negative semidefinite over $X \ge 0$, since it is sufficient for the assertion of Lemma 1. We first derive the gradient of $\mathrm{SSE}(X)$.

Let $\mathbf{e}_i$ denote the vector whose ith entry is 1 and the remaining entries are 0. When $n_j = 0$, by definition,

$$\frac{\partial \mathrm{SSE}(X)}{\partial x_{lj}} = \lim_{t \to 0} \frac{\mathrm{SSE}(X + t\mathbf{e}_l \mathbf{e}_j^T) - \mathrm{SSE}(X)}{t} = \lim_{t \to 0} \frac{t \left\| \mathbf{a}_l - \frac{t\mathbf{a}_l}{t} \right\|_{M_j}^2}{t} = 0\ .$$

When $n_j > 0$, by chain rule,

$$\begin{aligned} \frac{\partial \mathrm{SSE}(X)}{\partial x_{lj}} &= \left\| \mathbf{a}_l - \frac{\sum_{i=1}^{n} x_{ij}\mathbf{a}_i}{\sum_{i=1}^{n} x_{ij}} \right\|_{M_j}^2 \\ &\quad + 2 \sum_{i=1}^{n} x_{ij} \left(\frac{\sum_{p=1}^{n} x_{pj}\mathbf{a}_p}{\sum_{p=1}^{n} x_{pj}} - \mathbf{a}_i \right)^T M_j \left(\frac{\mathbf{a}_l}{\sum_{p=1}^{n} x_{pj}} - \frac{\sum_{p=1}^{n} x_{pj}\mathbf{a}_p}{\left(\sum_{p=1}^{n} x_{pj} \right)^2} \right)\ . \end{aligned}$$

Since $\sum_{i=1}^{n} x_{ij} \left(\frac{\sum_{p=1}^{n} x_{pj}\mathbf{a}_p}{\sum_{p=1}^{n} x_{pj}} - \mathbf{a}_i \right) = \mathbf{0}$ and $\left(\frac{\mathbf{a}_l}{\sum_{p=1}^{n} x_{pj}} - \frac{\sum_{p=1}^{n} x_{pj}\mathbf{a}_p}{\left(\sum_{p=1}^{n} x_{pj} \right)^2} \right)$ is independent of i, the second term in the above equality vanishes.

Therefore,

$$\frac{\partial \mathrm{SSE}(X)}{\partial x_{lj}} = \begin{cases} \left\| \mathbf{a}_l - \frac{\sum_{i=1}^{n} x_{ij}\mathbf{a}_i}{\sum_{i=1}^{n} x_{ij}} \right\|_{M_j}^2 & n_j > 0\ , \\ 0 & n_j = 0\ . \end{cases} \tag{4}$$

Let $\mathbf{v}_{lj} \in \mathbb{R}^d$ denote the difference of the lth pattern from the centroid of cluster C_j, i.e.,

$$\mathbf{v}_{lj} \stackrel{\text{def}}{=} \mathbf{a}_l - \mathbf{c}_j = \mathbf{a}_l - \frac{\sum_{i=1}^n x_{ij}\mathbf{a}_i}{\sum_{i=1}^n x_{ij}} .$$

Then from (4), we obtain that for any $l, g \in \{1, \ldots, n\}$ and $j, m \in \{1, \ldots, k\}$:

$$\frac{\partial^2 \mathrm{SSE}(X)}{\partial x_{lj} x_{gm}} = \begin{cases} -\frac{2}{n_j}\mathbf{v}_{lj}^T M_j \mathbf{v}_{gj} & j = m \text{ and } n_j > 0 \\ 0 & j \neq m \text{ or } n_j = 0 . \end{cases}$$

Let V_j denote the matrix whose lth row is the vector $\mathbf{v}_{lj}^T$. Then $\nabla^2 \mathrm{SSE}_j(X) = \begin{cases} -\frac{2}{n_j} V_j M_j V_j^T & n_j > 0 \\ 0 & n_j = 0 . \end{cases}$ The Hessian of $\mathrm{SSE}(X)$ is

$$\nabla^2 \mathrm{SSE}(X) = \begin{bmatrix} \nabla^2 \mathrm{SSE}_1(X) & & \\ & \ddots & \\ & & \nabla^2 \mathrm{SSE}_k(X) \end{bmatrix} ;$$

so it is negative semidefinite for $X \geq 0$.

This concludes our proof.

The feasible region of (3), denoted as D, is a polytope, i.e. it is bounded and is the intersection of a finite number of half spaces. It follows that D is a convex set.

Next we describe the extreme points of D, since the minimum of a concave function is achieved at some extreme points of its feasible region.

Proposition 1. *Any integer feasible solution to (3) is an extreme point (vertex) of D and any extreme point of D is an integer feasible solution to (3).*

Proof. We first prove that an integer feasible solution X of (3) is an extreme point of D, i.e., there do not exist $Y, Z \in D$, $Y \neq Z$, and $0 < \lambda < 1$, such that $X = \lambda Y + (1 - \lambda)Z$.

Assume $x_{ij} = 1$. from (1b), we have $x_{il} = 0\,(l \neq j)$. Then for any $0 < \lambda < 1$, the equality $x_{il} = \lambda y_{il} + (1 - \lambda) z_{il}$ and $y_{il} \geq 0$, $z_{il} \geq 0$ imply

$$y_{il} = z_{il} = 0 \quad (l \neq j). \tag{5}$$

From (5) and (1b), we have

$$y_{ij} = z_{ij} = 1.$$

Therefore, $Y = Z = X$. So X is an extreme point of D.

Next, we prove that any extreme point of D is an integer feasible solution to (3). We only need to show that any non-integer feasible solution to (3) is not an extreme point of D.

Let X be a non-integer feasible solution to (3), i.e. there exisits a must-link closure $\{\mathbf{a}_{m_p}\}_{p=1}^r$ whose membership $0 < x_{ij} < 1\,(i = m_1, \ldots, m_r)$. By (1b),

there exists $l \neq j$ such that $0 < x_{il} < 1\,(i = m_1, \ldots, m_r)$. In addition, we have $0 < x_{ij} + x_{il} \leq 1\,(i = m_1, \ldots, m_r)$. Let Y be a matrix with $y_{ij} = x_{ij} + x_{il}$, $y_{il} = 0$, $(i = m_1, \ldots, m_r)$ and other components the same as those of X. Let Z be a matrix with $z_{il} = x_{ij} + x_{il}$, $z_{ij} = 0$, $(i = m_1, \ldots, m_r)$ and other components being the same as those of X. Then $Y, Z \in D$. Let $\lambda = \frac{x_{ij}}{x_{ij}+x_{il}}\,(i = m_1, \ldots, m_r)$. We obtain $X = \lambda Y + (1 - \lambda)Z$ and $0 < \lambda < 1$. Therefore, X is not an extreme point of D.

We have proved that the set of extreme points of D is exactly the set of integer feasible solutions to (1).

The minimum of a concave function is attained on the facets of its feasible region. The extreme points of the feasible region of the continuous relaxation of (3) are integer. So Proposition 1 shows that SSE of a global solution to (3) is the same as that to (1).

For updating the centroid of each cluster, a must-link closure can be replaced by its centroid weighted by its number of patterns; however, a must-link closure cannot be replaced by any single point for local minimum search.

4 The Concave Cutting Algorithm

Based on the analysis in §3, we give a concave optimization algorithm for (3) in this section. A large number of approaches for concave minimization problems can be traced back to Tuy's cutting algorithm [12] for minimizing a concave function over a full dimensional polytope. We will briefly describe Tuy's cuts in the first part of this section for completeness. However, Tuy's cuts can't be applied directly to (3), because its feasible region doesn't have full dimension. In the second part of this section, we will show how to adapt Tuy's cutting algorithm to (3) and prove that this method can find a global minimum of (3) in finite steps.

4.1 Basic Ideas of Tuy's Cuts

For self-completeness, we sketch Tuy's cuts (also known as concavity cuts) below (see [6] for details). We assume $\mathbf{x} \in \mathbb{R}^n$ in this subsection.

Tuy's cuts are originally designed to find a global minimum of a concave function $f(\mathbf{x})$ over a polyhedron $D \in \mathbb{R}^n$. It requires
1) D has full dimension, i.e. $\operatorname{int} D \neq \emptyset$;
2) for any real number α, the level set $\{\mathbf{x} \in \mathbb{R}^n \colon f(\mathbf{x}) \geq \alpha\}$ is bounded.

Let $\mathbf{x}^0$ be a local minimum and a vertex of D. Denote $\gamma = f(\mathbf{x}^0)$. Since D is full-dimensional, $\mathbf{x}^0$ has at least n adjacent vertices. Let $\mathbf{y}^1, \ldots, \mathbf{y}^p$ denote the vertices adjacent to $\mathbf{x}^0$ $(p \geq n)$. For $i = 1, \ldots, n$, let

$$\theta_i \stackrel{\text{def}}{=} \sup\{t \colon t \geq 0,\ f\left(\mathbf{x}^0 + t(\mathbf{y}^i - \mathbf{x}^0)\right) \geq \gamma\}; \tag{6}$$

denote

$$\mathbf{z}^i \stackrel{\text{def}}{=} \mathbf{x}^0 + \theta_i(\mathbf{y}^i - \mathbf{x}^0).$$

Then the cone originated at $\mathbf{x}^0$ generated by the halflines in the directions $\mathbf{y}^i - \mathbf{x}^0$ covers the feasible region. Because f is concave, any point in the simplex $\text{Spx} \stackrel{\text{def}}{=} \text{conv}\{\mathbf{x}^0, \mathbf{z}^1, \ldots, \mathbf{z}^n\}$ has objective value no less than γ. Therefore, one can cut off Spx from further search for a global minimum. Since $\mathbf{x}^0$ is a vertex of D which has full dimension, one can always find n binding constraints at $\mathbf{x}^0$, and $\mathbf{x}^0$ has n linearly independent edges. Without loss of generality, assume that $\mathbf{z}^1 - \mathbf{x}^0, \ldots, \mathbf{z}^n - \mathbf{x}^0$ are linearly independent. Define

$$\pi = \mathbf{e}^T \text{Diag}(\frac{1}{\theta_1}, \ldots, \frac{1}{\theta_n}) U^{-1},\ U = [\mathbf{y}^1 - \mathbf{x}^0, \ldots, \mathbf{y}^n - \mathbf{x}^0]. \tag{7}$$

Then the inequality

$$\pi(\mathbf{x} - \mathbf{x}^0) > 1 \tag{8}$$

provides a γ-valid cut for (f, D), i.e., any $\mathbf{x}$ having objective value $f(\mathbf{x}) < \gamma$ must satisfy (8). In other words, if (8) has no solution in D, $\mathbf{x}^0$ must be a global minimum. Note that 1) $\theta_i \geq 1$; so Spx contains $\mathbf{x}^0$ and all its neighbor vertices; 2)the larger the θ_i, the deeper the cuts, i.e., the more portion of the feasible region is cut off. Following is the original pure convexity cutting algorithm based on the above idea.

Cutting Algorithm (Algorithm V.1., Chapter V, [6])

Initialization

Search for a vertex $\mathbf{x}^0$ which is a local minimizer of $f(\mathbf{x})$. Set $\gamma = f(\mathbf{x}^0)$, $D_0 = D$.

Iteration $\mathbf{i} = \mathbf{1}, \mathbf{2}, \ldots$

1. At $\mathbf{x}^i$ construct a γ-valid cut π^i for (f, D_i).
2. Solve the linear program

$$\max\ \pi^i(\mathbf{x} - \mathbf{x}^i) \qquad \text{s.t. } \mathbf{x} \in D_i. \tag{9}$$

   Let ω^i be a basic optimum of this LP. If $\pi^i(\omega^i - \mathbf{x}^i) \leq 1$, then stop: $\mathbf{x}^0$ is a global minimum. Otherwise, go to step 3.
3. Let $D_{i+1} = D_i \cap \{\mathbf{x}\colon \pi^i(\mathbf{x} - \mathbf{x}^i) \geq 1\}$. Starting from ω^i find a vertex $\mathbf{x}^{i+1}$ of D_{i+1} which is a local minimum of $f(\mathbf{x})$ over D_{i+1}. If $f(\mathbf{x}^{i+1}) \geq \gamma$, then go to iteration $i + 1$. Otherwise, set $\gamma \leftarrow f(\mathbf{x}^{i+1})$, $\mathbf{x}^0 \leftarrow \mathbf{x}^{i+1}, D_0 \leftarrow D_{i+1}$, and go to iteration 1.

Theorem 1. *(Theorem V.2, [6]) If the sequence $\{\pi^i\}$ is bounded, the above cutting algorithm is finite.*

4.2 The Adapted Tuy's Cutting Algorithm

In this section, we will describe how we construct the concavity cuts. And we will prove the finite convergence of our algorithm and compare it with the k-means algorithm.

Based on the argument at the end of §3, to reduce the number of variables, we consider the following equivalent form of (3).

Let $L_i\,(i = 1, \ldots, N)$ represent the must-link closures, i.e., $\cap_{i=1}^{N} L_i = \emptyset$, $\cup_{i=1}^{N} L_i = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$, and all the patterns in L_i are must-linked together. Let r_i be the number of patterns in L_i. If $r_i = 1$, L_i has only a singleton.

$$\begin{array}{ll} \min_{y_{ij}} & \sum_{j=1}^{k} \sum_{i=1}^{N} y_{ij} \sum_{l \in L_i} \left\| \mathbf{a}_l - \frac{\sum_{i=1}^{N} y_{ij} \sum_{l \in L_i} \mathbf{a}_l}{\sum_{i=1}^{N} y_{ij} r_i} \right\|_{M_j}^2 \\ \text{s.t.} & \sum_{j=1}^{k} y_{ij} = 1 \quad (i = 1, \ldots, N) \\ & y_{pj} + y_{qj} \le 1 \quad (p - q \text{ cannot-linked}; \; j = 1, \ldots, k) \\ & y_{ij} \ge 0 \quad (i = 1, \ldots, N; \; j = 1, \ldots, k)\ . \end{array} \tag{10}$$

Construction of the Cutting Plane. Once we find a local minimum, we need to add a cut to reduce the feasible region. At the mth iteration, let $Y^0 \in \mathbb{R}^{N \times k}$ be a local optimal solution to (10) in D_m and γ be the smallest SSE obtained from the previous iterations. Next, we will give details of how we form the cutting plane (7), including the construction of U and θ_i for (7), although the feasible region of (10) doesn't have full dimension — each vertex is adjacent to at most $N \times (k-1)$ other vertices.

1) Adjacent vertices.
We give $N \times k$ adjacent vertices to Y^0 below.

Let $E_{i,j}$ denote the matrix whose (i, j) entry is 1, the other entries are 0; and let $E_{(i,\cdot)}$ denote the matrix whose ith row are 1's, the remaining entries are 0. The orders of $E_{i,j}$ and $E_{(i,\cdot)}$ will be clear from the context. For $l = 1, \ldots, N$, assume that $L_l = \{\mathbf{a}_{l_i}\}_{i=1}^{r_l}$ is assigned to cluster l_j. Let $Z^{l,i}\,(i = 1, \ldots, k;\, i \ne l_j)$ denote the matrix different from Y^0 only by the assignment of L_l to cluster i. Choose $1 \le l_p \le k$, $l_p \ne l_j$. And let Z^{l,l_j} denote the matrix different from Y^0 only in its (l, l_p) entry being 1 as well, i.e.,

$$Z^{l,i} = \begin{cases} Y^0 - E_{l,l_j} + E_{l,i} & i \ne l_j \\ Y^0 + E_{l,l_p} & i = l_j \end{cases}.$$

Then $Z^{l,i}\,(l = 1, \ldots, N;\, i = 1, \ldots, k)$ are $N \times k$ adjacent vertices of Y^0, although some of them may not be feasible due to cannot-links. We form the vector $\mathbf{y}^0$ by stacking all the columns of Y^0 together. Similarly, we form the vectors $\mathbf{z}^{l,i}$. It is not hard to see that $U = [\mathbf{z}^{1,1} - \mathbf{y}^0, \ldots, \mathbf{y}^{1,k} - \mathbf{y}^0, \ldots, \mathbf{z}^{N,k} - \mathbf{y}^0]$ has full rank. Let I represent the identity matrix. It is straightforward to verify that the corresponding U^{-1} of (8) is a block diagonal matrix with lth block being $I + E_{(l_j,\cdot)} - E_{(l_p,\cdot)} - E_{l_j,l_j}$.

Because Z^{l,l_j} and some Z^{l,l_i} are not feasible to (3), part of the simplex $\text{conv}\{\mathbf{y}^0, \mathbf{z}^{1,1}, \ldots, \mathbf{z}^{1,k}, \ldots, \mathbf{z}^{N,k}\}$ lies outside the feasible region of (10); nevertheless, the concavity cut can exclude some part of the feasible region of (3).

2) The cutting plane.
Next, we will determine the θ_i's of (7), and subsequently the cutting plane π.

It is easy to see that $\theta^{l,l_j} = \infty$, and $\theta^{l,i} = 1$ if $\mathbf{a}_l$ cannot be assigned to cluster i ; otherwise, $\theta^{l,i}$ is a solution to the problem below.

$$\begin{array}{l} \max\ s \\ \text{s.t. } 0 \le s \le \frac{n_{l_j}}{r_l}, \\ \mathrm{SSE}(Y^0) + \mathbf{u} \ge \gamma \end{array} \tag{11}$$

with

$$\mathbf{u} = \begin{cases} -\frac{\left\| sr_l \mathbf{c}_i^{\mathrm{old}} - s\sum_{p=1}^{r_l} \mathbf{a}_{l_p} \right\|_{M_i}^2}{n_i^{\mathrm{old}} + sr_l} - \frac{\left\| sr_l \mathbf{c}_j^{\mathrm{old}} - s\sum_{p=1}^{r_l} \mathbf{a}_{l_p} \right\|_{M_j}^2}{n_j^{\mathrm{old}} - sr_l} \\ \quad + s\sum_{p=1}^{r_l} \left[\left\| \mathbf{a}_{l_p} - \mathbf{c}_i^{\mathrm{old}} \right\|_{M_i}^2 - \left\| \mathbf{a}_{l_p} - \mathbf{c}_j^{\mathrm{old}} \right\|_{M_j}^2 \right] & (n_j^{\mathrm{new}} > 0,\ n_i^{\mathrm{new}} > 0)\ ; \\ -\frac{\left\| sr_l \mathbf{c}_i^{\mathrm{old}} - s\sum_{p=1}^{r_l} \mathbf{a}_{l_p} \right\|_{M_i}^2}{n_i^{\mathrm{old}} + sr_l} \\ \quad + s\sum_{p=1}^{r_l} \left[\left\| \mathbf{a}_{l_p} - \mathbf{c}_i^{\mathrm{old}} \right\|_{M_i}^2 - \left\| \mathbf{a}_{l_p} - \mathbf{c}_j^{\mathrm{old}} \right\|_{M_j}^2 \right] & (n_j^{\mathrm{new}} = 0,\ n_i^{\mathrm{new}} > 0)\ ; \\ 0 & (n_i^{\mathrm{new}} = 0)\ . \end{cases}$$

For simplification, we use j instead of l_j in the formulation for $\mathbf{u}$. The first constraint in (11) keeps the assignment matrix in the feasible region where SSE is concave by Lemma 1.

It is not difficult to solve (11). When $n_j^{\mathrm{new}} = 0$ and $n_i^{\mathrm{new}} > 0$, we have $sr_l = n_j$, i.e. $s = \frac{n_j}{r_l}$; when $n_i^{\mathrm{new}} = 0$, we have $s = 0$. From Y^0 being a local minimum, we have $\theta^{l,i} \ge 1$. It is also easy to verify that

$$\mathrm{SSE}(Y^0) + \mathbf{u} - \gamma \text{ is continuous on } [0, \frac{n_{l_j}}{r_l}] \text{ and is nonnegative at } s = 1\ . \tag{12}$$

When $n_j^{\mathrm{new}} > 0$ and $n_i^{\mathrm{new}} > 0$, multiplying $(n_i^{\mathrm{old}} + sr_l)(n_j^{\mathrm{old}} - sr_l)$ to both sides of $\mathrm{SSE}(Y^0) + \mathbf{u} - \gamma \ge 0$ will reduce it to a cubic polynomial inequality in s. All the coefficients of $\mathrm{SSE}(Y^0) + \mathbf{u} - \gamma$ are real; so it can only have one or three real roots with the possibility of equal roots if $b_3 \ne 0$. The three roots of the corresponding cubic equation can be obtained by Cardano's formula.

The cutting plane is

$$\pi^{l,i} = \begin{cases} \frac{1}{\theta^{l,i}} - \frac{1}{\theta^{l,l_p}} & i \ne l_j \\ -\frac{1}{\theta^{l,l_p}} & i = l_j \end{cases} \qquad \pi \mathbf{y}^0 = -\sum_{l=1}^{n} \frac{1}{\theta^{l,l_p}}\ .$$

Finite Convergence of the Algorithm. The simplex Spx in our algorithm is centered at a local minimum of (11); so each concavity cut eliminates at least one vertex of (11). In addition, the number of vertices of (11) is finite. Therefore, the number of cuts is finite. From this along with the fact that only finite pivots are needed to reach a local minimum of (11), we conclude that our method can find a global minimum of (11) in finite steps.

The distance from the cut to Y^0 is $\frac{1}{\|\pi\|^2}$. The minimal solution to the convex univariate $\sum_{j=1}^{k} (\frac{1}{\theta^{l,j}} - x)^2$ is achieved at $x^* = \frac{\sum_{j=1}^{k} \frac{1}{\theta^{l,j}}}{k}$. Therefore, for deeper cuts, we choose θ^{l,l_P} with

$$l_p = \arg \min_{j \in \{1,\ldots k\}} \left| x^* - \frac{1}{\theta^{l,j}} \right|\ .$$

5 Numerical Examples

We've implemented the above algorithm in Ansi C with the linear program resulting for cutting plane solved by the CPLEX 91 callable library. Numerical results on traditional clustering show that the above cutting method can get a better solution than the k-means algorithm for traditional clustering; see [14]. In this part, we give some test results on semi-supervised clustering. The computation is done on a Toshiba satellite notebook, with Intel Pentium M processor of 1.70 GHz, 496 MB of RAM, Windows XP home edition operating system. Our algorithm stops if 1) a global solution is obtained; or 2) more than 21 cuts are added; or 3) no improvement in SSE after 8 consecutive cuts. We've tested our algorithm on some datasets from the UCI machine learning repository [11]. Table 1 gives a summary of the datasets we've used.

Table 1. Datasets

dataset name	# of instances	# of attributes	# of classes
Vehicle Silhouettes (xaa set)	94	11	4
Zoo	101	16	7
Teaching Assistant Evaluation	151	5	3
Wine Recognition	178	13	3
Glass Identification	214	9	7
Ecoli	336	7	8
Balance Scale Weight & Distance	625	4	3
Yeast	1484	8	10

Below are our numerical results on different datasets. The column 'mustlk' represents the sum of numbers of patterns of all the must-link closures and the column 'cantlk' is the sum of numbers of patterns of all the cannot-link closures. The exact clusters of the datasets are known. So we assign must-link to points randomly drawn from the same cluster, and cannot-link to points randomly taken from different clusters. For each dataset, we randomly choose around (15%, 10%), (10%, 7%), (5%, 3%) of its total points as (must-links, cannot-links). The column 'metric' is the distance metric used, with I represents the Euclidean metric, i.e., M_j's are the identity matrices, and RCA represents M_j's being the within must-link closure covariance matrix (it is shown in [1] that this metric gives the best results). If RCA is singular, we omit the row RCA from the table. For each different combination of 'mustlk', 'cantlk', and 'metric', we run the k-means algorithm and the concavity cut method both with random-start for 100 times. To avoid bias toward large SSE value, we normalize the values by dividing them with the initial SSE. We then take average over the 100 runs. For example, the value under 'k-means' on the row 'Obj' is

$$\frac{1}{100}\sum_{i=1}^{100}\frac{\text{SSE from k-means algorithm at the } i\text{th run}}{\text{SSE from initial partition at the } i\text{th run}}.$$

Table 2. Vehicle Dataset

mustlk	cantlk	metric		k-means	local min	cut
14	10	I	Obj	0.276439	0.258552	0.257897
			CPU	0.001200	0.002210	0.536770
10	6	I	Obj	0.243712	0.230685	0.229439
			CPU	0.002600	0.003400	0.398770
5	4	I	Obj	0.226040	0.208723	0.208723
			CPU	0.002300	0.002800	0.523050

Table 3. Zoo Dataset

mustlk	cantlk	metric		k-means	local min	cut
15	10	I	Obj	0.248311	0.221239	0.215044
			CPU	0.003400	0.006600	0.732250
10	8	I	Obj	0.236324	0.217124	0.214246
			CPU	0.005100	0.007000	0.236640
5	4	I	Obj	0.207460	0.199193	0.186251
			CPU	0.005410	0.007010	0.227320

Table 4. Teaching Assistant Evaluation Dataset

mustlk	cantlk	metric		k-means	local min	cut
23	16	RCA	Obj	0.671734	0.657069	0.642726
			CPU	0.000700	0.001410	0.302940
		I	Obj	0.553348	0.541301	0.524715
			CPU	0.000500	0.000800	0.843630
15	10	I	Obj	0.507037	0.496884	0.480044
			CPU	0.000700	0.001100	0.136990
8	4	I	Obj	0.485393	0.472041	0.460714
			CPU	0.000600	0.001000	0.149910

Table 5. Wine Dataset

mustlk	cantlk	metric		k-means	local min	cut
27	18	RCA	Obj	0.534966	0.532235	0.532197
			CPU	0.002800	0.004510	0.134390
		I	Obj	0.197416	0.187645	0.187278
			CPU	0.003200	0.005500	0.447840
18	12	I	Obj	0.183942	0.177888	0.172741
			CPU	0.005510	0.008310	0.477780
9	6	I	Obj	0.167763	0.150645	0.150578
			CPU	0.004110	0.006720	0.227650

Table 6. Glass Dataset

mustlk	cantlk	metric		k-means	local min	cut
32	22	RCA	Obj	0.108001	0.093677	0.059824
			CPU	0.011120	0.017430	0.486090
		I	Obj	0.298343	0.270086	0.257363
			CPU	0.007310	0.014110	1.825220
21	14	RCA	Obj	0.085489	0.074158	0.033044
			CPU	0.015040	0.019840	1.693230
		I	Obj	0.301583	0.267343	0.257927
			CPU	0.007310	0.013810	1.769340
11	6	I	Obj	0.292819	0.266833	0.249677
			CPU	0.007410	0.013010	1.683520

Table 7. Ecoli Dataset

mustlk	cantlk	metric		k-means	local min	cut
51	34	I	Obj	0.278162	0.270426	0.257343
			CPU	0.011120	0.016140	1.706950
34	24	I	Obj	0.266271	0.260423	0.248139
			CPU	0.011720	0.015620	0.755080
16	10	I	Obj	0.267482	0.259465	0.248567
			CPU	0.011610	0.015520	2.735530

Table 8. Balance Dataset

mustlk	cantlk	metric		k-means	local min	cut
94	62	I	Obj	0.722311	0.715907	0.711963
			CPU	0.005010	0.007620	3.233440
62	44	I	Obj	0.715767	0.711341	0.708332
			CPU	0.005300	0.007200	3.393570
31	18	I	Obj	0.708112	0.705248	0.701182
			CPU	0.003400	0.005310	3.662360

Table 9. Yeast Dataset

mustlk	cantlk	metric		k-means	local min	cut
222	148	I	Obj	0.462682	0.455150	0.438204
			CPU	0.121190	0.167350	2.363490
150	104	I	Obj	0.450784	0.446078	0.430492
			CPU	0.122560	0.157200	2.164910
75	44	I	Obj	0.440515	0.435947	0.424551
			CPU	0.129430	0.171610	0.772810

Similarly, the value under 'local min' on the row 'Obj' is the average normalized SSE of local minima near the solutions of k-means; the value under 'cut' on that row is the average normalized SSE of the solutions obtained from the concavity cutting algorithm. The row 'CPU' is the average CPU time in seconds for the three methods.

For all the test problems, our algorithm terminates within 4 seconds. We observe that the run time of our algorithm is not an inreasing function of the dataset size.

References

1. Aharon Bar-Hillel, Tomer Hertz, Noam Shental, and Daphna Weinshall. Learning a mahalanobis metric from equivalence constraints. *Journal of Machine Learning Research*, 6:937–965, June 2005.
2. Sugato Basu and Ian Davidson. Clustering with constraints: Theory and practice. Online Proceedings of a KDD tutorial, 2006. http://www.ai.sri.com/~basu/kdd-tutorial-2006/.
3. Mikhail Bilenko, Sugato Basu, and Raymond J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *ICML '04: Proceedings of the twenty-first international conference on Machine learning*, page 11, New York, NY, USA, 2004. ACM Press.
4. P. S. Bradley, K. P. Bennett, and A. Demiriz. Constrained k-means clustering. Technical Report MSR-TR-2000-65, Microsoft Research, 2000.
5. A. D. Gordon. A survey of constrained classification. *Comput. Statist. Data Anal.*, 21(1):17–29, 1996.
6. Reiner Horst and Hoang Tuy. *Global optimization*. Springer-Verlag, Berlin, 1993.
7. Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice Hall Advanced Reference Series. Prentice Hall Inc., Englewood Cliffs, NJ, 1988.
8. Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. A local search approximation algorithm for k-means clustering. *Comput. Geom. Theory Appl.*, 28(2-3):89–112, 2004.
9. Dan Klein, Sepandar D. Kamvar, and Christopher D. Manning. From instance-level constraints to space-level constraints: Making the most of prior knowledge in data clustering. In *ICML*, pages 307–314, 2002.
10. Tilman Lange, Martin H. C. Law, Anil K. Jain, and Joachim M. Buhmann. Learning with constrained and unlabelled data. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1*, pages 731–738, Washington, DC, USA, 2005. IEEE Computer Society.
11. P. M. Murphy and D. W. Aha. UCI repository of machine learning databases. Technical report, University of California, Department of Information and Computer Science, Irvine, CA, 1994. http://www.ics.uci.edu/~mlearn/MLRepository.html.
12. H. Tuy. Concave programming under linear constraints. *Soviet Mathematics*, 5:1437–1440, 1964.
13. Kiri Wagstaff, Claire Cardie, Seth Rogers, and Stefan Schroedl. Constrained k-means clustering with background knowledge. In *ICML '01: Proceedings of the Eighteenth International Conference on Machine Learning*, pages 577–584, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

14. Yu Xia and Jiming Peng. A cutting algorithm for the minimum sum-of-squared error clustering. In *Proceedings of the Fifth SIAM International Conference on Data Mining*, pages 150–160, 2005.
15. Eric P. Xing, Andrew Y. Ng, Michael I. Jordan, and Stuart Russell. Distance metric learning with application to clustering with side-information. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 505–512. MIT Press, Cambridge, MA, 2002.

Bender's Cuts Guided Large Neighborhood Search for the Traveling Umpire Problem

Michael A. Trick and Hakan Yildiz

Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA USA, 15213

Abstract. This paper introduces the use of Bender's Cuts to guide a Large Neighborhood Search to solve the Traveling Umpire Problem, a sports scheduling problem inspired by the real-life needs of the officials of a sports league. At each time slot, a Greedy Matching heuristic is used to construct a schedule. When an infeasibility is recognized Bender's cuts are generated, which guides a Large Neighborhood Search to ensure feasibility and to improve the solution.

1 Introduction

In this paper we consider a hybrid use of Bender's Cuts, Large Neighborhood Search and a Greedy Matching Based Heuristic to find good, feasible solutions to the Traveling Umpire Problem (TUP), a sports scheduling problem briefly described in the next section. As shown in the subsequent sections, even finding a feasible solution to the TUP is challenging, making it difficult to find good solutions in a reasonable amount of time.

The rest of the paper is organized as follows. In Section 1 we formally define the TUP. Section 3 discusses exact solution approaches. We present the algorithms we used to solve the TUP in Section 4 and 5. Computational results are given in in Section 6. The conclusion is given in Section 7.

2 Problem Description

Traveling Umpire Problem is a multi-objective sports scheduling problem introduced in [12]. Like the Traveling Tournament Problem for league scheduling, which was introduced by Easton et al. [3], the TUP is based on the most important features of a real sports scheduling problem, the umpires for Major League Baseball. Although the TUP is originally defined as a multi-objective problem, in this paper we consider the single objective version of the problem where we consider is the minimization of the total umpire travel.

TUP extracts the most critical aspects of scheduling the umpires associated with Major League Baseball. There are 2430 games in Major League Baseball's schedule. Each game requires an umpire crew (other sports refer to these as referees). The "real" umpire scheduling problem consists of dozens of pages of constraints, including such ideosyncratic constraints as an umpire's preferred

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 332–345, 2007.

vacation dates. TUP limits the constraints to the key issues: no umpire should be assigned to a team too often in short succession, and every umpire should be assigned to every team some time in a season. Given these constraints, the primary objective is to minimize the travel of the umpires.

Formally, given a double round robin tournament, where every team plays against all other teams twice, on $2n$ teams ($4n-2$ slots), we want to assign one of n umpires to each game and satisfy the following set of constraints.

Constraints
1) Every game gets an umpire
2) Every umpire works exactly one game per slot
3) Every umpire sees every team at least once at the team's home
4) No umpire is in a home site more than once in any n-d_1 consecutive slots
5) No umpire sees a team more than once in any $\lfloor \frac{n}{2} \rfloor$ - d_2 consecutive slots

The parameters for the constraints are not chosen arbitrarily. The structural results regarding the parameters in the constraints are presented in [12].

We present an example of a round robin tournament for 4 teams and a feasible umpire schedule for $d_1 = d_2 = 0$ in Table 2. A game is represented as a pair (i, j) where i is the home team and j is the away team. Rows correspond to umpire schedules and columns correspond to games that are played in the corresponding time slots.

Table 1. Round robin tournament for 4 teams and a feasible umpire schedule for 2 umpires

Slots	**1**	**2**	**3**	**4**	**5**	**6**
Umpire1	(1,3)	(3,4)	(1,4)	(3,1)	(4,3)	(2,3)
Umpire2	(2,4)	(1,2)	(3,2)	(4,2)	(2,1)	(4,1)

3 Exact Solution Approaches

TUP has many characteristics in common with the Traveling Salesman Problem (TSP), due to the emphasis on minimizing travel. But the TSP is solvable exactly for hundreds or thousands of cities. Does TUP have the same characteristic? If so, that would limit interest in the problem since real sports leagues rarely go beyond 30 or so teams.

We formulated the TUP as an Integer Program (IP) and also as a Constraint Program (CP). The IP formulation is presented in the Appendix. Both of the approaches become ineffective as the size of the instances grows, particularly when Constraints 4&5 are more restricting as d_1 and d_2 become closer to 0.

The results for the IP and CP are given in Table 2 for the smaller instances at $d_1 = d_2 = 0$. The IP can solve the 10 team instance to optimality in more than 13 hours, whereas CP was unable to solve that instance. We present these results to demonstrate that the problem becomes very difficult to solve as we

Table 2. IP and CP results for $d_1 = d_2 = 0$

		Time(sec)	
no of Teams	**Total Distance**	**IP**	**CP**
4	5176	0.07	0.02
6	14077	0.27	1.35
8	34311	1.6	869.39
10	48942	47333.7	-

increase the number of teams. Computational results for larger instances are given in Sect. 5.

4 Greedy Matching Heuristic and a Bender's Based Modification

Given the difficulty of the problem, we explore heuristic approaches to find good solutions. Our methods begin with a simple greedy heuristic which we will then extend.

Greedy Matching Heuristic (GMH) is a constructive heuristic, which builds the umpire schedules starting from Slot 1 and ending at Slot $4n-2$. For every slot t, the heuristic assigns umpires to games such that all constraints, except Constraint 3, are satisfied and the best possible assignment is made to minimize the total umpire travel at Slot t. To do that, GMH solves a Perfect Matching Problem on a Bipartite Graph in every slot t by solving an integer program. The partitions in this Bipartite Matching Problem are the Umpires and the Games in slot t. An edge is placed between an umpire u and a game (i, j) if Constraints 4&5 are not violated by assigning u to game (i, j) in slot t, given the assignments from slot 1 to $t-1$. Cost of an edge $(u, (i, j)) = Distance(k, i) - Incentive(u, i)$. In this cost function, k is the venue that u is assigned in slot $t-1$ and $Distance(k, i)$ is the distance between cities k and i. $Incentive(u, i)$ takes a positive value if umpire u has never visited venue i in the previous slots. This reduction in distance guides the GMH towards assigning umpires to cities that they have not visited yet, thus reducing the possibility of having a solution that violates Constraint 3 at the end of the execution of GMH. If the resulting solution violates Constraint 3, then the infeasibility is penalized in the objective function during the course of the Large Neighborhood Search Algorithm, which we explain in Sect. 5.2.

In practice, the GMH often gets stuck at some time slot because there may be no feasible perfect matching. In this case, we can identify a set of previous assignments that are causing this lack of perfect matching. At least one of those previous assignments must be changed in order to create a perfect matching. This set of assignments leads to a *Logic-Based Bender's Cut* in the terminology of [6].

While we could add the Bender's cuts to an integer programming formulation of this problem, in a standard master/subproblem approach to this problem,

instead we use these cuts to guide a large neighborhood search heuristic. Violation of Bender's Cuts is penalized in the objective function with a large cost. Thus, any changes in the schedule that reduces the the number of violated Bender's Cuts or reduces total umpire travel is accepted. When a solution that satisfies all the Bender's Cuts is found, we stop the neighborhood search and solve the Perfect Matching Problem for slot t again. If there is no feasible matching, we repeat the process. A high level pseudo code is presented in Algorithm 1. We explain the generation of Bender's Cuts and the Large Neighborhood Search Algorithm in the next section.

Algorithm 1. Greedy Matching Heuristic with Bender's Cuts Guided Neighborhood Search (GBNS)

```
1: Arbitrarily assign umpires to games in slot 1
2: for all 1 < t ≤ 4n-2 do
3:    Construct a Bipartite Graph G = (V, E) for the Perfect Matching Problem
4:    Find a Minimum Cost Perfect Matching on G
5:    if there is no feasible perfect matching then
6:       Find all Bender's Cuts
7:       Set Objective = (no of violated cuts)*(violation cost)+(total umpire travel)
8:       Set improvement = 1
9:       while at least one of the cuts is violated & improvement > 0 do
10:         Do neighborhood search using 3-Ump and 3-Slot Neighborhoods
11:         if Objective is not improved then
12:            improvement = 0
13:         end if
14:      end while
15:   end if
16: end for
```

5 Bender's Cuts and Large Neighborhood Search

5.1 Generating Bender's Cuts

Bender's Cuts are generated when there is no feasible matching at a slot t during the course of GMH. Such an infeasibility implies that the partial schedule built until slot t can not be completed to obtain a feasible solution. We, then, examine the reasons for this infeasibility and this examination leads to constraints that exclude a number of partial solutions. Clearly, the Bender's Cuts generated in this method are not classical Bender's Cuts, which are obtained from a linear subproblem as in traditional Bender's Decomposition. Instead these cuts are *Logic-Based Bender's Cuts* as defined by Hooker and Ottosson[6], where the Bender's Cuts are obtained from *inference duals*. The difference between the Logic-Based Bender's Cuts and classical Bender's Cuts is that no standard form exists for the Logic Based Bender's Cuts and such cuts are generated by logical inference on the solution of the subproblem. When the subproblem is a feasibility

problem, the inference dual is a condition which, when satisfied, implies that the master problem is infeasible. This condition can be used to obtain a Bender's Cut for cutting off infeasible solutions. Logic Based Bender's Cuts are a special case of "nogoods," a well-known known idea in constraint programming literature, but they exploit problem structure in a way that nogoods generally do not [2]. Logic-Based Bender's Cuts have been successfully used in several studies [5,4,8,9]. In this section we describe the way we generate Logic Based Bender's Cuts and how we use these cuts.

The Bender's Cuts are generated for any set of Umpires (or Games) whose adjacency neighborhood has a cardinality less than the cardinality of the set itself. The *adjacency neighborhood* $N(A)$ of a set A is the set of nodes that are adjacent to a node in A. This condition is known as Hall's Theorem:

Hall's Theorem: *Let $G = (V, E)$ be a bipartite graph with bipartitions X and Y. Then G has a perfect matching if and only if $|N(A)| \geq |A|$ for all $A \subseteq X$*

Because of Hall's theorem, if there is no perfect matching in a time slot, there is a subset of umpires A whose neighborhood $N(A)$ has cardinality smaller than $|A|$. We will generate a constraint that creates at least one edge between A and $Y \setminus N(A)$ as follows. For each pair x, y such that $x \in A$ and $y \in Y \setminus N(A)$ and $(x, y) \notin E$, there exists an already made game-umpire assignment in previous slots that prevents the edge to be in the matching problem. We find all such game-umpire assignments for all the missing edges between $x \in A$ and $y \in Y \setminus N(A)$. Then the corresponding Bender's Cut requires that at least one of these game-umpire assignments should be changed. To obtain all possible Bender's Cuts, we identify Hall sets by checking all subsets of Umpires and their neighborhoods. For small instances, the number of cuts identified when an infeasibility occurs is not too many, though for larger problems it would be necessary to generate only a limited number of cuts.

We present a partial schedule for an example instance for 8 teams and 4 umpires in Table 3. For this instance we assume that $d_1 = d_2 = 0$. Thus, the fourth constraint imposes that an umpire can not visit the same home venue more than once in any 4 consecutive games. The fifth constraint, on the other hand, imposes that an umpire can not see a team more than once in any two consecutive games. For this example the games for the first three slots are scheduled and we are considering the games in the fourth slot.

The matching problem that corresponds to slot 4 is given in Fig. 1. In this figure, set A and $N(A)$, the adjacency neighborhood of A, are circled with dashed lines. The cardinality of A is 4, whereas the cardinality of $N(A)$ is 3. Thus, there is no feasible perfect matching for this graph.

The way we obtain the Bender's Cut from set A is best demonstrated with the help of Fig. 2 and Table 4. The Bender's Cut states that at least one of the edges between the nodes in A, the umpires, and the complement of $N(A)$, Game $(2, 1)$, has to be in the graph. To write this cut in terms of the game-umpire assignments, we identify the game-umpire assignments in slots $1, 2$ and 3, which

Table 3. Partial schedule for 8 teams and 4 umpires. The first three slots are scheduled and the games for the fourth slot are in consideration for assignment.

Slots	1	2	3	4
Umpire1	(7,5)	(2,4)	(5,7)	(2,1)
Umpire2	(1,8)	(3,6)	(4,1)	(4,5)
Umpire3	(2,6)	(1,7)	(6,8)	(6,3)
Umpire4	(4,3)	(5,8)	(3,2)	(8,7)

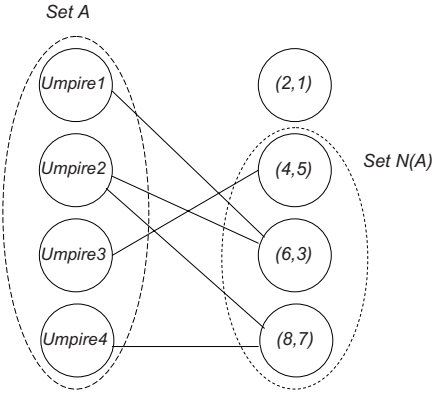


Fig. 1. Bipartite matching problem for slot 4. The partitions are Umpires and the Games in slot 4.

prevents the edges being in the graph. For Umpire1, Game $(2,4)$ in slot 2 is preventing the edge between Umpire1 and Game $(2,1)$, because Umpire1 can not visit venue 2 twice in four consecutive games. Similarly, for Umpire2, Game $(4,1)$ in slot 3 is preventing the edge between Umpire2 and Game $(2,1)$; for Umpire 3, Game $(2,6)$ in slot 1 is preventing the edge between Umpire3 and Game $(2,1)$; for Umpire4, Game $(3,2)$ in slot 3 is preventing the edge between Umpire4 and Game $(2,1)$. Thus the cut generated is

$$assigned[1,2,2] + assigned[2,4,3] + assigned[3,2,1] + assigned[4,3,3] \leq 3$$

$$\text{where, } assigned[u,i,t] = \begin{cases} 1, \text{ if umpire u is at venue i in slot t} \\ 0, \text{ otherwise.} \end{cases}$$

5.2 Very Large Neighborhood Search

A *neighborhood* of a solution S is a set of solutions that are in some sense close to S, i.e., they can be easily computed from S or they share a significant amount of structure with S. An algorithm that starts at some initial solution and iteratively moves to solutions in the neighborhood of the current solution is called a *Neighborhood Search Algorithm* or a *Local Search Algorithm.*

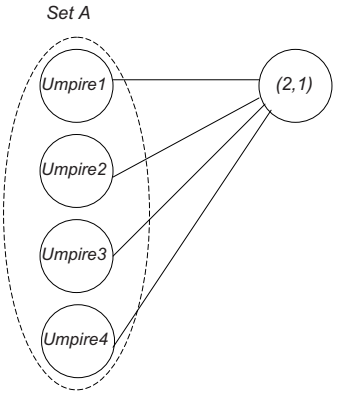


Fig. 2. The missing edges between set A and the complement of N(A). Bender's Cut requires at least one of these to be present in the bipartite perfect matching problem.

Table 4. Games that are in conflict with game (2,1) in slot 4 are highlighted

Slots	1	2	3
Umpire1	(7,5)	**(2,4)**	(5,7)
Umpire2	(1,8)	(3,6)	**(4,1)**
Umpire3	**(2,6)**	(1,7)	(6,8)
Umpire4	(4,3)	(5,8)	**(3,2)**

The Very Large Neighborhood Search(VLNS) for the TUP tries to improve the solution quality at each iteration. It is clear that the larger the neighborhood, the better is the quality of the solutions that can be reached in one single move. At the same time, the larger the neighborhood, the longer it takes to search the neighborhood at each iteration.

To make the notion of *very large neighborhood* clear, we'll first introduce the well known *2-exchange neighborhood*, which is a small polynomially sized neighborhood. Given an umpire schedule, a *2-exchange move* swaps the umpires assigned to two games played in the same slot t. The neighborhood for this move is the set of schedules that can be obtained by performing a single move. We introduce two very large neighborhoods in the following sections. We use both neighborhoods in our search for a feasible solution with the use of Bender's Cuts, when the Greedy Matching Heuristic is unable to assign the games to umpires at a slot. Once an initial solution is found, we further use the K-Umpire Neighborhood to improve that solution. We stop this search when we reach a local optimum.

K-Umpire Neighborhood: We take the schedules of $K \leq n$ umpires and allow exchanges of game assignments within these schedules. The exchanges of game assignments are allowed only within the same slot, not across slots, as we need to make sure that Constraints 1&2 are always satisfied. In each

slot, there are $K!$ different ways of assigning games to umpires. Since there are $4n-2$ slots, the possible solutions that can be reached by one K-Umpire move is $(K!)^{4n-2}$.

The problem of finding the best move is done using the Restricted IP for K-Umpires(RIP-U) for the TUP with only the games for the K umpires in consideration. Since this RIP-U is solved many times during the execution of the algorithm, the solution time should be very low to have the algorithm terminate in a reasonable amount of time. Since the RIP-U becomes a very hard problem for $K \geq 4$ for even small instances, we take $K = 3$ and we look at all possible 3 combinations of the n umpires.

K-Slot Neighborhood: We take the games at $K \leq 4n-2$ slots and allow exchanges of umpire assignments within these slots. The exchanges of umpire assignments are allowed only within the same slot, not across slots, as we need to make sure that Constraints 1&2 are always satisfied. In each slot, there are $n!$ different ways of assigning games to umpires. Since only K slots are considered at a time, the possible solutions that can be reached by one K-Slot move is $(n!)^K$.

The problem of finding the best move is done using the Restricted IP for K-Slots(RIP-S) for the TUP with only the games for the K slots in consideration. Since this RIP-S is solved many times during the execution of the algorithm, the solution time should be very low to have the algorithm terminate in a reasonable amount of time. Since the RIP-S becomes a hard problem as K grows we take $K = 3$ and we look at all possible 3 combinations of the $4n-2$ slots.

6 Computational Results

We have tested the solution approaches presented in this paper on a set of TUP instances. We report these computational results in this section.

6.1 Instance Description

An instance of the TUP has two matrices: The Distance Matrix, which stores the pairwise distances between cities and the Opponents Matrix, which stores the tournament information. We have used instances with 4 teams to 32 teams. The instances with 14 teams or less use the TTP Tournaments as given at [1]. The instances with more than 14 teams use the distance matrix for the National Football League given at [1], and the game schedule is generated using a Constraint Program [11] that creates a round robin tournament. All of the instances used in this study and additional ones are available at [10].

Depending on the choice of d_1 and d_2, the difficulty of the problem changes. Decreasing these parameters makes the problem harder to solve. Choosing a $d_1 = n-1$, which makes $n - d_1 = 1$, or a $d_2 = \lfloor \frac{n}{2} \rfloor - 1$, which makes $\lfloor \frac{n}{2} \rfloor$ - $d_2 = 1$, simply means that Constraint 4 or Constraint 5 is not in effect.

6.2 Summary of Results

All the algorithms are implemented using the script language in ILOG OPL Studio 3.7 [7]. We run the algorithms on a Linux Server with Intel(R) Xeon(TM) 3.2 GHz processor.

6.3 Finding a Feasible Solution

As the problem size increases and as the constraints 4 and 5 become more restricting, even finding a feasible solution to the TUP becomes difficult. To find a feasible solution we used the IP and the GMH with the usage of Bender's Cut's Guided Large Neighborhood Search, which we will refer as GBNS.

Table 5 summarizes the results for IP and GBNS for the instances with 12 and 14 teams. The GBNS approach finds much better solutions much faster in both instances and all the combinations of the parameters for Constraints 4&5. For 12 teams instance the GBNS obtains improvements ranging from 3.1% to 23.6% and for 14 teams instance ranging from 4.5% to 38.6% over the IP.

For 14 teams and for the combination of parameters $(n-d_1, \lfloor \frac{n}{2} \rfloor - d_2) = (6,3)$ and $(n-d_1, \lfloor \frac{n}{2} \rfloor - d_2) = (7,3)$ the IP could not find a feasible solution even after more than 30 hours of execution, whereas GBNS found a solution for each case. Notice that it takes GBNS only 0.7 seconds to solve this instance for $(n-d_1, \lfloor \frac{n}{2} \rfloor - d_2) = (6,3)$, whereas for $(n-d_1, \lfloor \frac{n}{2} \rfloor - d_2) = (7,3)$, it takes GBNS 7322.9 seconds. This drastic difference in time is due to the fact that the problem becomes extremely difficult to solve when $d_1 = d_2 = 0$.

For 12 teams and for the combination of parameters $(n-d_1, \lfloor \frac{n}{2} \rfloor - d_2) = (5,3)$ and $(n-d_1, \lfloor \frac{n}{2} \rfloor - d_2) = (6,3)$ neither the IP nor the GBNS could find a feasible solution even after several hours of running. Since the IP was terminated before concluding to the infeasibility of these cases, we do not currently know if these two instances are actually feasible or not.

6.4 Improving the Solution with VLNS

We run 3-Umpire Neighborhood Search on the initial solutions obtained by the IP and the GBNS. While GBNS used this neighborhood in creating the solution, this was done with costs associated with the Bender's Cuts. Once a feasible solution is found, those costs are no longer required. Furthermore, the local search aspect of GBNS is only applied when there is an infeasibility. So it is entirely possible that further improvements are possible once GBNS terminates, and our computational tests bear that out.

Table 6 summarizes the results for IP and GBNS for the instances with 12 and 14 teams. The quality of the initial solution does not make any significant difference in terms of the quality of the final solution obtained by the neighborhood search. On the other hand, the total times spent on finding the initial solution and then running the neighborhood search is significantly less for GBNS, which is due to the very fast execution of GBNS. In short, the real value of GBNS is the quick generation of a feasible solution.

Table 5. Comparison of the IP and the Greedy Matching Heuristic with Bender's Cuts Guided Neighborhood Search (GBNS) in finding feasible solutions for instances with 12 teams and 14 teams. The columns consist of the parameter for Constraint 4 ($n - d_1$), the parameter for Constraint 5 ($\lfloor \frac{n}{2} \rfloor$ - d_2), IP solution cost (IP), time in seconds, GBNS solution cost (GBNS) and time in seconds, and % cost improvement of GBNS over IP (% Impr.).

12 Teams

$\mathbf{n - d_1}$	$\lfloor \frac{\mathbf{n}}{\mathbf{2}} \rfloor$ - d_2	IP	time(sec)	GBNS	time(sec)	% Impr.
2	1	95024	7.1	72557	0.1	23.6
3	1	97276	10.4	76407	0.1	21.5
4	1	93762	7.4	76756	0.1	18.1
5	1	93030	19.1	76781	0.0	17.5
6	1	99632	67.3	77818	0.1	21.9
2	2	101055	20.8	88277	0.1	12.6
3	2	102399	46.7	88637	0.1	13.4
4	2	101978	36.8	90231	0.1	11.5
5	2	100641	93.4	91951	0.1	8.6
6	2	100372	134.1	91131	0.1	9.2
2	3	100089	7136.3	95072	359.3	5.0
3	3	100797	1025.3	95072	359.3	5.7
4	3	101063	2194.4	97945	28.6	3.1
5	3	–		–		
6	3	–		–		

14 Teams

$\mathbf{n - d_1}$	$\lfloor \frac{\mathbf{n}}{\mathbf{2}} \rfloor$ - d_2	IP	time(sec)	GBNS	time(sec)	% Impr.
2	1	182520	56.5	112142	0.1	38.6
3	1	184069	70.0	117618	0.1	36.1
4	1	180170	42.8	118647	0.1	34.1
5	1	184496	46.6	120781	0.1	34.5
6	1	187357	182.9	121998	0.1	34.9
7	1	182435	196.7	126360	0.0	30.7
2	2	196977	88.4	152658	0.1	22.5
3	2	202383	133.9	153536	0.1	24.1
4	2	199001	194.2	155073	0.1	22.1
5	2	201533	198.7	155525	0.1	22.8
6	2	194975	370.6	157761	0.1	19.1
7	2	206335	621.7	161428	0.1	21.8
2	3	196003	401.4	175884	0.1	10.3
3	3	196394	10085.8	175884	0.1	10.4
4	3	190640	1989.1	182054	0.1	4.5
5	3	205230	2442.9	182746	0.1	11.0
6	3	–		183331	0.7	
7	3	–		186979	7322.9	

Table 6. Comparison of the 3-Umpire Neighborhood search when started from the initial solutions found by IP and GBNS for instances with 12 teams and 14 teams. The columns consist of the parameter for Constraint 4 ($n - d_1$), the parameter for Constraint 5 ($\lfloor \frac{n}{2} \rfloor$ - d_2), solution cost when initial solution is found by IP (IP & 3-Ump), time in seconds, solution cost when initial solution is found by GBNS (GBNS & 3-Ump) and time in seconds, and % cost improvement obtained when started with GBNS over when started with IP (% Impr.).

12 Teams

$\mathbf{n - d_1}$	$\lfloor \frac{\mathbf{n}}{\mathbf{2}} \rfloor$ - d_2	**IP & 3-Ump**	**time(sec)**	**GBNS & 3-Ump**	**time(sec)**	**% Impr.**
2	1	62515	242.2	62374	192.2	0.2
3	1	66159	408.2	66090	325.4	0.1
4	1	66608	352.5	66897	378.5	-0.4
5	1	68312	466.2	69103	351.5	-1.2
6	1	69445	722.2	69775	759.7	-0.5
2	2	82288	256.9	82188	349.0	0.1
3	2	83095	459.3	83504	245.9	-0.5
4	2	83529	407.5	83974	444.5	-0.5
5	2	85192	541.1	85804	495.4	-0.7
6	2	91865	464.2	91018	279.6	0.9
2	3	95924	7367.3	94080	666.7	1.9
3	3	92672	1578.3	94080	666.7	-1.5
4	3	101047	2573.2	97945	288.5	3.1
5	3	–		–		
6	3	–		–		

14 Teams

$\mathbf{n - d_1}$	$\lfloor \frac{\mathbf{n}}{\mathbf{2}} \rfloor$ - d_2	**IP & 3-Ump**	**time(sec)**	**GBNS & 3-Ump**	**time(sec)**	**% Impr.**
2	1	95075	894.6	94748	1053.1	0.3
3	1	103854	1451.3	102021	1427.9	1.8
4	1	106243	1874.4	106751	1508.8	-0.5
5	1	109901	1645.8	110896	1307.5	-0.9
6	1	112385	2920.7	111184	1792.6	1.1
7	1	114077	2283.0	117332	1671.9	-2.9
2	2	140855	1789.8	140570	1221.0	0.2
3	2	142334	1157.1	141936	813.9	0.3
4	2	143746	1435.9	144301	910.9	-0.4
5	2	146129	850.9	146964	1400.2	-0.6
6	2	148860	2122.6	148389	2723.9	0.3
7	2	157402	1969.5	161413	950.8	-2.5
2	3	157444	5678.1	157111	3235.5	0.2
3	3	155776	5793.6	157111	3235.5	-0.9
4	3	166345	4754.3	164263	3625.9	1.3
5	3	177480	5762.4	168995	2469.4	4.8
6	3	–		181052	714.0	
7	3	–		186979	8005.6	

7 Conclusion

In this paper, we introduce a new approach to finding good solutions to the Traveling Umpire Problem. The Traveling Umpire Problem is a highly constrained scheduling problem and we show that conventional methods are ineffective in solving large instances to optimality and even find it difficult to find feasible solutions. Our main contribution is to show how to generate Benders cuts during the execution of a simple Greedy Heuristic. These cuts enforce feasibility requirements that allow the Greedy Heuristic to avoid infeasibilities. This method enables the simple Greedy Matching Heuristic to produce quality feasible solutions very fast.

We introduce two Very Large Neighborhoods, 3-Umpire and 3-Slot Neighborhoods, to search the solution space for the Traveling Umpire Problem. We show that by searching these neighborhoods, guided by the Bender's cuts, we can obtain an initial feasible solution. Moreover, after finding an initial feasible solution, we can improve the quality of the solution by searching the 3-Umpire Neighborhood. Whether the initial solution comes from integer programming or our GBNS approach, the solution after large-neighborhood search seems to have the same quality.

References

1. Challenge Traveling Tournament Instances. http://mat.gsia.cmu.edu/TOURN/, January 2007
2. Dawande, M.W., Hooker, J.N.: Inference-Based Sensitivity Analysis for Mixed Integer/Linear Programming. Operations Research **48 (4)** (2000) 623–634
3. Easton, K., Nemhauser, G.L., Trick, M.A.: The Traveling Tournament Problem: Description and Benchmarks. Principal and Practises of Constraint Programming - CP 2001, Springer Lecture Notes in Computer Science **2239**, 580–585
4. Harjunkoski, I., Grossmann, I. E.: Decomposition Techniques for Multistage Scheduling Problems Using Mixed-integer and Constraint Programming Methods. Computers and Chemical Engineering **26** (2002) 1533-1552
5. Hooker, J.N.: Planning and Scheduling by Logic-based Benders Decomposition. Operations Research (to appear)
6. Hooker, J.N., Ottosson, G.: Logic-based Benders decomposition, Mathematical Programming **96** (2003) 33–60
7. ILOG Inc., ILOG OPL Studio 3.7 Language Manual (2003)
8. Jain, V., Grossmann, I.E.: Algorithms for hybrid MILP-CP models for a class of optimization problems, INFORMS Journal on Computing **13 (4)** (2001) 258–276
9. Rasmussen, R.V., Trick, M.A.: A Benders approach for the constrained minimum break problem, European Journal of Operational Research **177** (2007) 198–213
10. Traveling Umpire Problem. http://www.andrew.cmu.edu/user/hakanyil/TUP/, January (2007)
11. Trick, M.A.: Integer and Constraint Programming Approaches for Round Robin Tournament Scheduling, in PATAT2002, E. Burke and P. Causmaecker (eds), Springer Lecture Notes in Computer Science **2740** (2003) 63-77.
12. Yildiz, H., Trick, M.: The Traveling Umpire Problem. Invited Talk, Informs Annual Conference, Pittsburgh, PA, November (2006)

Appendix: IP Formulation

Model Parameters

T = {1,..,2n} is the set of teams
S = {1,..,4n-2} is the set of slots
U = {1,..,n} is the set of umpires

$$\text{opponents}[t,i] = \begin{cases} j, & \text{if team i plays against team j at venue i in slot t} \\ -j, & \text{if team i plays against team j at venue j in slot t} \end{cases}$$

dist[i,j] = distance between venues i and j

Decision Variables

$$assigned[u,i,t] = \begin{cases} 1, \text{ if umpire u is at venue i in slot t} \\ 0, \text{ otherwise.} \end{cases}$$

$$moves[u,i,j,t] = \begin{cases} 1, \text{ if umpire u is at venue i in slot t and moves to j in slot t+1} \\ 0, \text{ otherwise.} \end{cases}$$

The formulation in the OPL language[7] is as follows:

```
minimize sum (u in U, i in T, j in T, t in S: t < 4*n-2)
dist[i,j]*moves[u,i,j,t]

subject to

//(1): Every game gets an umpire
forall (i in T, t in S: opponents[t,i] > 0)
    sum (u in U) assigned[u,i,t] = 1;

//(2): Every umpire is assigned to exactly one game per slot
forall (u in U, t in S)
    sum (i in T: opponents[t,i] > 0)assigned[u,i,t] = 1;

//(3): Every umpire sees every team at least once at the
team's home forall (u in U, i in T)
    sum (t in S: opponents[t,i] > 0) assigned[u,i,t] >= 1;

//(4): No umpire is in a home site more than once in any n-d1
       consecutive slots
forall (u in U, i in T, t in S: t <= (4*n-2)-(n-d1-1)
      sum (t1 in [0..(n-d1-1)]) assigned[u,i,t+t1] <= 1;

//(5): No umpire sees a team twice in any floor(n/2)-d2
       consecutive slots
forall (u in U, i in T, t in S: t <= (4*n-2)-(n/2-d2-1)
     sum (t1 in [0..(n/2-d2-1)]) (assigned[u,i,t+t1]
     + sum(k in T: opponents[t+t1,k] = i) assigned[u,k,t+t1])
      <= 1;
```

```
//(6): Linkage constraints: If umpire u is assigned to i at t
//             and to j at t+1, then it should move from i to j
               at t.
forall(u in U, i,j in T, t in S: t < 4*n-2)
    assigned[u,i,t] + assigned[u,j,t+1] - 1 <= moves[u,i,j,t];
```

//Additional Valid Inequalities

```
//(7): If team i plays away in slot t, no umpire can be assigned
       to venue i
forall (u in U, i in T, t in S: opponents[t,i] < 0)
    assigned[u,i,t] = 0;

//(8): If umpire u moves from i to j in t, it must be assigned to
       i in t
//(9): If umpire u moves from i to j in t, it must be assigned to
       j in t+1
forall (u in U, i in T, j in T, t in S: t < 4*n-2){
      moves[u,i,j,t] <= assigned[u,i,t];
      moves[u,i,j,t] <= assigned[u,j,t+1]; };

//(10): # of umpires moving to j at t = # of umpires moving from j
        at t+1
forall (u in U, j in T, t in S: t < 4*n-3) {
      sum(i in T) moves[u,i,j,t] = sum(i in T) moves[u,j,i,t+1];};

//(11): Every umpire must move in every slot
forall (u in U, t in S: t < 4*n-2)
      sum(i in T, j in T) moves[u,i,j,t] = 1;}
```

A Large Neighborhood Search Heuristic for Graph Coloring

Michael A. Trick and Hakan Yildiz

Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA USA, 15213

Abstract. We propose a new local search heuristic for graph coloring that searches very large neighborhoods. The heuristic is based on solving a MAX-CUT problem at each step. While the MAX-CUT problem is formally hard, fast heuristics that give "good" cuts are available to solve this. We provide computational results on benchmark instances. The proposed approach is based on similar heuristics used in computer vision.

1 Introduction

Graph Coloring Problem (GCP) is one of the central problems in graph theory, has direct applications in practice, and is related to many other problems such as computer register allocation, bandwidth allocation, and timetabling. Given an undirected graph $G=(V, E)$, a *coloring* f of G is an assignment of a color to each vertex. A *proper (or feasible) coloring* is a coloring such that for each edge $(i, j) \in E$, vertices i and j have different colors. A *conflict* is the situation when two adjacent vertices have the same color assigned to them. We say that a coloring is *improper (or infeasible)* if there exists at least one conflict. The *conflict graph* of a graph G is the graph induced by the vertices that are incident to the conflicts in G.

A *minimum coloring* of G is a feasible coloring with the fewest different colors. It is well known that graph coloring is a hard combinatorial optimization problem [12], and exact solutions can be obtained for only small instances [14]. Therefore, heuristic algorithms are used to solve large instances. In this paper we introduce a new local search algorithm that searches large neighborhoods, based on ideas introduced by Boykov et al. [2].

The rest of the paper is organized as follows. In Section 2 we shortly review the known neighborhood search methods and introduce our very large neighborhood approach. We describe our algorithm in Section 3 and present the experimental results in Section 4. The conclusion is given in Section 5.

2 Local Search for Graph Coloring

Local search is based on the concept of a neighborhood. A *neighborhood* of a solution S is a set of solutions that are in some sense close to S, i.e., they can be easily computed from S or they share a significant amount of structure with S.

Local search for the GCP starts at some initial, improper coloring and iteratively moves to neighboring solutions, trying to reduce the number of conflicts.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 346–360, 2007.

It is clear that the larger the neighborhood, the better is the quality of the solutions that can be reached in one single move. At the same time, the larger the neighborhood, the longer it takes to search the neighborhood at each iteration.

In this paper, we investigate a new local search method that uses very large scale neighborhoods. This is one of the first attempts to solve the GCP using local search in large neighborhoods. The only other large neighborhood searches we are aware of are due to Chiarandini et al.[5] and Avanthay et al.[1].

To make the notion of *large neighborhood* clear, we'll first introduce the well known *1-exchange* and *2-exchange neighborhoods*, which are small polynomially sized neighborhoods. Given a coloring, a *1-exchange move* changes the color of exactly one node and a *2-exchange move* swaps the colors of two vertices. The corresponding neighborhoods for these moves are the set of colorings that can be obtained by performing a single move.

We consider the neighborhoods proposed by Boykov et al. [2] for energy minimization problems in computer vision, and use these neighborhoods to solve the GCP. In the following subsections, we formally describe these neighborhoods and the corresponding moves, which are explained best in terms of partitions. Then we describe how to find the optimal moves by using graph cuts. The structures of the graphs, the cuts on these graphs, and the properties of the cuts are also explained in detail.

2.1 Moves and Neighborhoods

The first neighborhood is the α-β-swap: for a pair of colors $\{\alpha, \beta\}$, this move exchanges the colors between an arbitrary set of vertices colored α and another arbitrary set colored β. The second neighborhood we consider is α-expansion: for a color α, this move assigns the color α to an arbitrary set of vertices.

The GCP can be represented as a partitioning problem, in which a feasible coloring f corresponds to a partition of the set of vertices into K sets such that no edge exists between two vertices from the same color class. Let $\mathbf{V} = \{V_l | l \in L\}$ be such a partition, where L is the set of colors and $V_l = \{v \in V | f(v) = l\}$ is the subset of vertices assigned color $l \in L$.

Given a pair of colors (α, β), a move from a partition $\mathbf{V}$ (coloring f) to a new partition $\mathbf{V}'$ (coloring f') is called an α-β-swap if $V_l = V_l'$ for any color $l \neq \alpha, \beta$. This means that the only difference between $\mathbf{V}$ and $\mathbf{V}'$ is that some vertices that were colored α in $\mathbf{V}$ are now colored β in $\mathbf{V}'$, and some vertices that were colored β in $\mathbf{V}$ are now colored α in $\mathbf{V}'$.

Given a color α, a move from a partition $\mathbf{V}$ (coloring f) to a new partition $\mathbf{V}'$ (coloring f') is called an α-expansion if $V_\alpha \subset V_\alpha'$ and $V_l' \subseteq V_l$ for any label $l \neq \alpha$. In other words, an α-expansion move allows any set of vertices to change their colors to α.

2.2 Size of the Neighborhoods

For an α-β-swap, each vertex either keeps its current color or switches to the other one. Since each partition has $\Omega(n)$ vertices, the possible solutions that can be reached by one swap move is $2^{\Omega(n)}$.

For an α-expansion, each vertex that is not colored α either keeps its old color or acquires the new color α. Since there are $\Omega(n)$ such vertices, the possible solutions that can be reached by one expansion move is $2^{\Omega(n)}$. These imply:

Lemma 1. *The size of both neighborhoods is* $2^{\Omega(n)}$.

2.3 Graph Cuts

The important part of the local search algorithms, which are presented in the following sections, is efficiently finding the best neighboring solution to the current solution by using graph cuts. Let $G = (V, E)$ be a connected and undirected edge weighted graph. A *cut* C of G is a minimal subset of E, which increases the number of connected components by exactly one. The *weight* (or *cost*) of a cut C is the sum of the weights of edges in the cut and is represented by $w(C)$. A *maximum cut* (or a *maxcut*) is then defined as a cut of maximum weight.

2.4 Finding the Optimal Swap Move

Given a coloring f and a pair of colors $\{\alpha, \beta\}$, we want to find a coloring $\hat{f}$ that minimizes the number of conflicts over all colorings within one α-β-swap of f. Our technique is based on computing a coloring that corresponds to a maximum cut on the subgraph $G^{\alpha\beta} = (V^{\alpha\beta}, E^{\alpha\beta})$, which is a clique over the vertices colored with α or β in f. For all $(i, j) \in E^{\alpha\beta}$, we assign a weight equal to one if $(i, j) \in E$ and a weight equal to zero if $(i, j) \notin E$. The latter ensures that $G^{\alpha\beta}$ is connected. The structure of the graph $G^{\alpha\beta}$ is illustrated in Fig. 1.

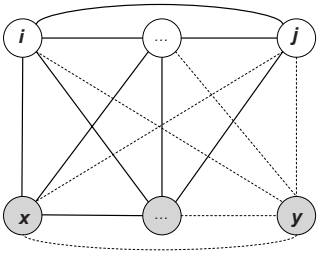


Fig. 1. An example of $G^{\alpha\beta}$. The set of vertices are $V^{\alpha\beta} = V_\alpha \cup V_\beta$ where $V_\alpha = \{i, ..., j\}$ and $V_\beta = \{x, ..., y\}$. Solid edges are induced edges and have weight 1. Dashed edges are artificial edges and have weight 0, which ensure that $G^{\alpha\beta}$ is connected.

Every edge, with a weight 1, between the vertices of the same color is a conflict. Every swap move defines a new bipartition of the vertex set $V^{\alpha\beta}$, possibly with a different number of conflicts. Notice that every cut in this graph defines a swap move that results a bipartition of the vertex set, thus a new coloring. In order to obtain the optimal swap move that results with minimum number of conflicts, we need to minimize the total weight of edges within the partitions. Notice that this is equivalent to maximizing the total weight of edges between

the two partitions, which is equivalent to solving a maxcut problem on $G^{\alpha\beta}$. After finding a maxcut, the vertices in one partition are going to be colored α, and the vertices in the other partition will be colored β. The selection of which partition will be colored α is arbitrary. This implies:

Theorem 1. *Let $G^{\alpha\beta}$ be constructed as described above for a given f and $\{\alpha, \beta\}$ and let T be the total weight of edges in $G^{\alpha\beta}$. A coloring f^C corresponding to a cut C on $G^{\alpha\beta}$ is one α-β-swap away from the initial coloring f. Moreover the optimal α-β-swap move is equivalent to a maxcut C^* in $G^{\alpha\beta}$ and the number of conflicts within $G^{\alpha\beta}$ for the new coloring f^{C^*} is x if $w(C^*) = T - x$.*

2.5 Finding the Optimal Expansion Move

Given an input coloring f and a color α, we want to find a coloring $\hat{f}$ that minimizes the number of conflicts over all colorings within one α-expansion of f. Our technique is based on computing a coloring that corresponds to a maximum cut on the graph $G^{\alpha} = (V^{\alpha}, E^{\alpha})$. The structure of this graph is determined by the current partition $\mathbf{V}$ and by the color α, so the graph dynamically changes after each iteration.

The structure of the graph is illustrated in Fig. 2. The set of vertices include all vertices $v \in V$. Moreover it includes two terminals α and $\overline{\alpha}$, which are auxiliary vertices representing the color α in consideration and the rest of the colors, respectively. In addition, we have six types of auxiliary vertices. For each edge that is incident to two vertices with color α, we create an auxiliary vertex of type A_1. For each edge that is incident to exactly one vertex with color α, we create an auxiliary vertex of type A_2. For each adjacent vertex pair such that neither vertex in the pair is colored with α, we create two auxiliary vertices of types B_1 and B_2 if the pair has different colors. If they are colored with the same color, say γ, we create two vertices of types D_1 and D_2.

We now explain the way we connect the vertices by edges with different wights. The weights assigned to these edges are summarized in Table 1. The two terminals are connected by an edge with a very high weight M to ensure that the maxcut that is found separates the two terminals α and $\overline{\alpha}$. Each vertex $v \in V$ is connected by an edge to the terminals α and $\overline{\alpha}$. Each pair of adjacent vertices $\{i, j\} \in V$ is connected by edges to the auxiliary vertices corresponding to that pair. Each pair of adjacent vertices such that neither of them are colored with α are connected by an edge. In addition to these, type $A1$, $A2$, B_1, and D_1 vertices are connected by an edge to the terminal α, and type B_2 and D_2 vertices are connected by an edge to the terminal $\overline{\alpha}$. As a result, each adjacent vertex pair and the auxiliary vertices corresponding to the pair and the edges incident to these vertices form four different structures, which we call as *gadgets*. The four different gadgets that correspond to four edge types of the original graph G are illustrated in Fig. 2. Formally, for an edge (i, j), there are four possible situations depending on the colors of the vertices incident to those edges:

1. $f(i) = f(j) = \alpha$
2. $f(i) = \alpha$, $f(j) \neq \alpha$, or $f(i) \neq \alpha$, $f(j) = \alpha$

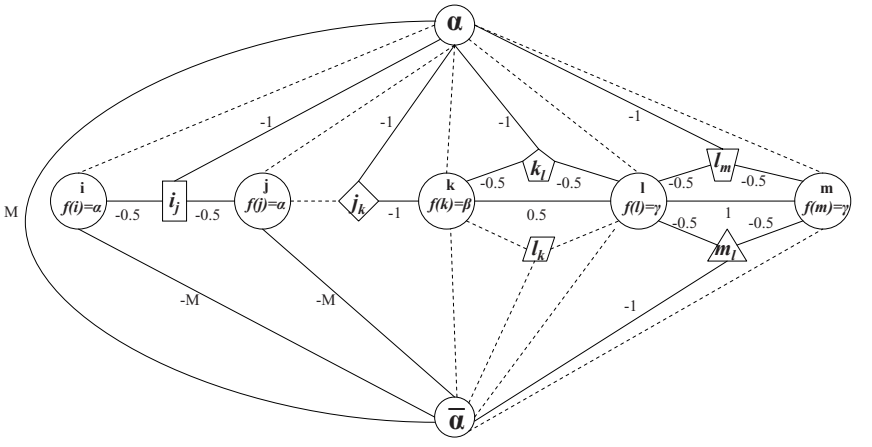


Fig. 2. An example of G^α. Dashed edges have weight 0. The type sets for auxiliary vertices: T=$\{\alpha, \overline{\alpha}\}$, A_1=$\{i_j\}$, A_2=$\{j_k\}$, B_1=$\{k_l\}$, B_2=$\{l_k\}$, D_1=$\{l_m\}$, D_2=$\{m_l\}$.

3. $f(i) \neq \alpha$, $f(j) \neq \alpha$, $f(i) = f(j)$
4. $f(i) \neq \alpha$, $f(j) \neq \alpha$, $f(i) \neq f(j)$

Any cut C on G^α, which separates the two terminals α and $\overline{\alpha}$, must include exactly one of the edges that connect $v \in V$ to the terminals. This defines a natural coloring f^C corresponding to a cut C on G^α. Formally,

$$f^C(v) = \begin{cases} \alpha, & \text{if } (v, \alpha) \in C \\ f(v), & \text{if } (v, \overline{\alpha}) \in C \end{cases}$$

In other words, a vertex v is assigned color α if the cut C separates v from the terminal α and, v is assigned its old color $f(v)$ if C separates v from $\overline{\alpha}$. Clearly this implies:

Lemma 2. *A coloring f^C corresponding to a cut C on G^α, which separates the two terminals α and $\overline{\alpha}$, is one α-expansion away from the initial coloring f. Also, an α-expansion move is equivalent to a cut C on G^α, which separates the two terminals α and $\overline{\alpha}$.*

For each of the four gadgets, a maxcut C on G^α severs some of the edges of the gadgets, and the sum of the weights of the edges severed is consistent with whether the corresponding edge (i, j) is a conflict or not in f^C.

Property 1. For any maxcut C and for any edge $(i, j) \in E$ such that at least one of i and j is colored with α in f

a) *If* (α, i), $(\alpha, j) \in C$ *then either* $(\alpha, i_j) \in C$ *or* (i, i_j), $(i_j, j) \in C$
b) *If* $(\overline{\alpha}, i)$, $(\overline{\alpha}, j) \in C$ *then no other edge from the gadget is in* C
c) *If* (α, i), $(\overline{\alpha}, j) \in C$ *then either* $(i, i_j) \in C$ *or* (α, i_j), $(i_j, j) \in C$
d) *If* (α, j), $(\overline{\alpha}, i) \in C$ *then either* $(i_j, j) \in C$ *or* (α, i_j), $(i, i_j) \in C$

Property 1 follows from the maximality of $w(C)$ and it is illustrated in Fig. 3.

Table 1. The weights assigned to the edges in the graph presented in Fig. 2

edge	weight	for	example(Fig. 2)
(v,α)	0	$v \in V$	(i,α),(j,α),(k,α),(l,α),(m,α)
$(v,\overline{\alpha})$	$-M$	$v \in V_\alpha$	$(i,\overline{\alpha})$,$(j,\overline{\alpha})$
$(v,\overline{\alpha})$	0	$v \in V \backslash V_\alpha$	$(k,\overline{\alpha})$,$(l,\overline{\alpha})$,$(m,\overline{\alpha})$
(a,α)	-1	$a \in A_1, A_2, B_1, D_1$	(i_j,α),(j_k,α),(k_l,α), (l_m,α)
$(b,\overline{\alpha})$	0	$b \in B_2$	$(l_k,\overline{\alpha})$
$(c,\overline{\alpha})$	-1	$c \in D_2$	$(m_l,\overline{\alpha})$
(v,a)	-0.5	$v \in V_\alpha$, $a \in A_1$	(i,i_j), (i_j,j)
(v,a)	-1	$v \in V \backslash V_\alpha$, $a \in A_2$	(j_k,k)
(v,a)	0	$v \in V_\alpha$, $a \in A_2$	(j,j_k)
(v,b)	-0.5	$v \in V \backslash V_\alpha$, $b \in B_1, D_1, D_2$	(k,k_l),(l,k_l),(l,l_m), (l,m_l),(m,l_m),(m,m_l)
(v,b)	0	$v \in V \backslash V_\alpha$, $b \in B_2$	(l_k,k), (l_k,l)
(v,w)	0.5	$v,w \in V \backslash V_\alpha$, $f(v) \neq f(w)$	(k,l)
(v,w)	1	$v,w \in V \backslash V_\alpha$, $f(v) = f(w)$	(l,m)
$(\alpha,\overline{\alpha})$	M	$(\alpha,\overline{\alpha})$	$(\alpha,\overline{\alpha})$

In the case that $f(i) = f(j) = \alpha$, since the weight of the edges that connect i and j to $\overline{\alpha}$ has weight $-M$, the only maxcut possible is one of the cuts described in Property 1(a). Since both of the cuts separate i and j from α, the colors of i and j stay unchanged: $f^C(i) = f^C(j) = \alpha$. If $(\alpha, i_j) \in C$, or if (i,i_j), $(i_j,j) \in C$ then the cost is -1. In both cases, the cost incurred is truly consistent with the fact that (i,j) is a conflict in f^C.

In the case that one of i and j is colored with α in f, assume w.l.o.g. $f(i) = \alpha$, $f(j) = \beta$, the possible cuts are the ones described in Property 1 (a) or (c). The cuts that sever $(i,\overline{\alpha})$ are not possible since the weight of $(i,\overline{\alpha})$ is $-M$. The cost of the cuts having Property 1(a) is -1, which is consistent with the fact that (i,j) is a conflict in f^C. The cost of the cuts having Property 1(c) is 0, which is consistent with the fact that (i,j) is not a conflict in f^C.

Property 2. For any maxcut C and for any edge $(k,l) \in E$ such that $f(k) \neq \alpha$, $f(l) \neq \alpha$:

a) *If* (α,k), $(\alpha,l) \in C$ *then either* $(\alpha,k_l) \in C$ *or* (k,k_l), $(k_l,l) \in C$
b) *If* $(\overline{\alpha},k)$, $(\overline{\alpha},l) \in C$ *then either* $(\overline{\alpha},l_k) \in C$ *or* (k,l_k), $(l_k,l) \in C$
c) *If* (α,k), $(\overline{\alpha},l) \in C$ *then* (k,k_l), $(l_k,l) \in C$
d) *If* (α,l), $(\overline{\alpha},k) \in C$ *then* (k,l_k), $(k_l,l) \in C$

Property 2 follows from the maximality of $w(C)$ and from the fact that no subset of C is a cut. Property 2 is illustrated in Fig. 4.

In the case that $f(k) = f(l) = \beta$, all the cuts described in Property 2 are possible. The cost of the cuts that have Property 2(a) or (b) is -1, which is consistent with the fact that (k,l) is a conflict in f^C in both cases. The cost of the cuts that have Property 2(c) or (d) is 0, which is consistent with the fact that (k,l) is not a conflict in f^C in either case, since the color of exactly one of k, l is changed to α.

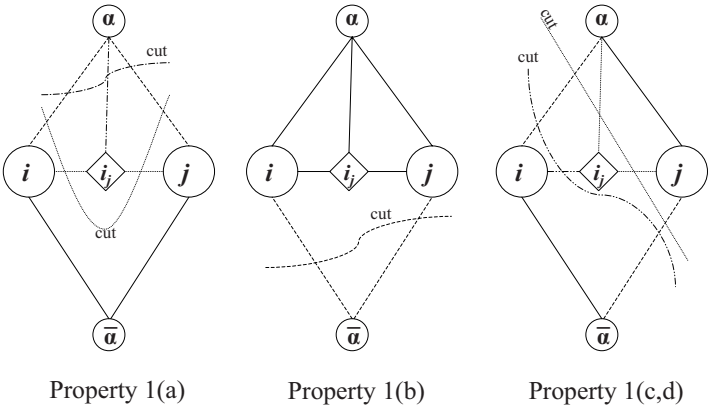


Fig. 3. Properties of a maxcut C on G^α for two vertices $i,j \in V$ such that at least one of them is colored with α

In the case that $f(k) = \beta$, $f(l) = \gamma$, all the cuts described in Property 2 are possible. The cost of the cuts that have Property 2(a) is -1, which is consistent with the fact that (k, l) is a conflict in f^C. The cost of the cuts that have Property 2(b), (c) or (d) is 0, which is consistent with the fact that (k, l) is not a conflict in f^C in those cases, since the color of at least one of k, l is not changed to α. Lemma 2, Property 1 and Property 2 implies:

Theorem 2. *A coloring f^{C^*} corresponding to a maxcut C^* on G^α is one α-expansion away from the initial coloring f. Moreover the optimal α-expansion move is equivalent to a maxcut C^* in G^α and the number of conflicts for the new coloring f^{C^*} is x if $w(C^*) = M - x$.*

3 Algorithms

In this section we first introduce swap-move, check-bipartite and expansion-move algorithms. These three algorithms are used as subroutines in the expansion-swap algorithm, which is the main algorithm.

3.1 Swap-Move Algorithm

Input: A coloring f of G with K colors.

1. Set Success:=0, Any_Imp(swap):=0
2. Set Improvement(swap):= 0
3. For each pair of colors $\{\alpha, \beta\} \subset L$
   3.1 Find $\hat{f}$ that minimizes the number of conflicts among all possible new colorings within one α-β-swap of f
   3.2 IF the number of conflicts is reduced, f:= $\hat{f}$, Improvement(swap):=1, Any_Imp(swap):=1

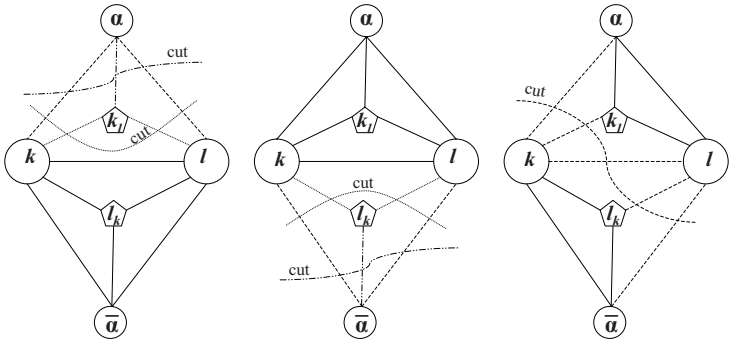


Fig. 4. Properties of a maxcut C on G^{α} for two vertices k,$l \in V$ such that none of them is colored with α

4. IF f has no conflicts, return f with Success := K
5. IF Improvement(swap) = 1, goto 2
6. Return f with Success:= 0 & Any_Imp(swap).

3.2 Expansion-Move Algorithm

Input: A coloring f of G with K colors.

1. Success:=0, Any_Imp(expansion):=0
2. Set Improvement(expansion):= 0
3. For each color $\alpha \in L$
   3.1 Find $\hat{f}$ that minimizes the number of conflicts among all possible new colorings within one α-expansion of f
   3.2 IF the number of conflicts is reduced, f:= $\hat{f}$, Improvement(expansion):=1, Any_Imp(expansion):=1
4. IF f has no conflicts, return f with Success:= K & Any_Imp(expansion)
5. IF Improvement(expansion)= 1, goto 2
6. Return f with Success:= 0 & Any_Imp(expansion).

3.3 Check-Bipartite Algorithm

If the conflict graph of an infeasible coloring f of G with K colors is bipartite, one can remove all conflicts without creating new conflicts and end up with a feasible $K+1$ coloring by coloring the vertices of one partition with a new color $K+1$. A short algorithm based on this observation is given below.

Input: A coloring f of G with K colors.

1. If f has conflicts, let $G^{'} = (V^{'},E^{'})$ be the subgraph induced by the conflicting vertices
   1.1 Starting from an arbitrary source vertex, color the vertices and their neighbors in alternation with colors α and β.
   1.2 If the resulting coloring is proper, then $G^{'}$ is bipartite. Introduce a new color $K+1$ to color the vertices in one partition, return f with Success:= $K+1$.

3.4 Expansion-Swap Algorithm

Expansion-swap is the main algorithm that takes advantage of both neighborhoods introduced and uses the swap-move, check-bipartite and expansion-move algorithms as subroutines. Namely, when the swap-move no longer improves the current solution, expansion-move tries to improve the current solution without starting from scratch. When expansion-move is unable to improve the solution, swap-move starts running again. When neither algorithm is able to improve the solution, first we check if the conflict graph is bipartite with the Check-bipartite algorithm. If it is not bipartite, then at Step 8, a new color $K+1$ is introduced by finding $\hat{f}$ that minimizes the number of conflicts among $f^{'}$ within one $(K+1)$-expansion of f.

Introducing a new color by an expansion move is a better approach than introducing it by finding an independent set on the conflict graph, since expansion allows for the creation of new conflicts in exchange for removing more current conflicts. However in the independent set case, coloring adjacent vertices with the new color is not permitted. This is easy to see when we assume that the conflict graph is a dense graph, such as a clique, where the maximum independent set is only one vertex.

Introducing the new color $K+1$ will definitely improve the solution if not actually remove all the conflicts. After this improvement, the swap-move phase will search for a better solution, and the whole cycle repeats until a feasible coloring is found.

Input: A graph G and an initial number of colors K.

1. Randomly color G with K colors, resulting in coloring f
2. Swap-move(G, f, K)
3. IF Success = 0, Expansion-move(G, f, K)
4. IF Success $>$ 0 return f
5. IF Any_Imp(expansion) = 1, Swap-move(G, f, K); ELSE goto 7
6. IF Success $>$ 0 return f;
   ELSE IF Success = 0 & Any_Imp(swap) = 1, goto 3
7. Check-bipartite(G, f, K)
8. IF Success = 0, find $\hat{f}$ that minimizes the number of conflicts among all possible new colorings within one $(K+1)$-expansion of f, set $f := \hat{f}$
9. IF f has no conflicts, return f with Success := $K+1$; ELSE set $K := K+1$, goto 2.

3.5 Finding a Maximum Cut

Given an undirected graph G with edge weights, the *MAX-CUT problem* consists of finding a maxcut of G. MAX-CUT is a well-known NP-Hard problem [12]. Since all our algorithms rely on solving MAX-CUT problems several times, the solution times for our algorithms can be expected to be out of our limits for a local search heuristic. So rather than determining the maxcut at each move, we can find a "good" cut, which has a weight that is close to the weight of a maxcut. This allows us to search the introduced neighborhood approximately and fast. However, since Theorem 2 is dependent on the cut found being a maxcut, we can not use the total weight of the cut found to calculate the number of conflicts. But notice that we can still use any cut to find a new coloring as Lemma 2 holds for any cut. For this reason, we use the cut obtained to define the new coloring and we calculate the actual number of conflicts by checking the adjacency matrix and the new coloring.

There are many heuristic and approximation algorithms that have been computationally tested and/or with theoretical performance guarantee. Goemans and Williamson [13] proposed a randomized algorithm that uses semidefinite programming to achieve a performance guarantee of 0.87856. More recent algorithms for solving the semidefinite programming relaxation are particularly efficient, because they exploit the structure of the MAX-CUT problem. Burer, Monteiro, and Zhang [4] proposed a rank-2 relaxation heuristic for MAX-CUT and described a computer code, called *Circut* [7], that produces better solutions in practice than the randomized algorithm of Goemans and Williamson. Circut does not assume the edge weights are positive. This property is necessary for our algorithm as the graphs created by our algorithm have edges with negative weights. Moreover, since the performance of Circut on many different problems has been shown to be very good, and the code is available for outside use, we decided to use Circut to solve MAX-CUT problems in our algorithms.

4 Experimental Results

4.1 Implementation Details

Our algorithm is implemented in C. Since the MAX-CUT solver code, Circut, is implemented in Fortran90, the input/output transaction between the main code and Circut is made through text files. For large size problems, writing into and reading from files takes very long times. This becomes a serious issue especially for the expansion graphs created for the expansion move since the size of the expansion graphs are much larger than the original graph. To overcome this disadvantage, we used the following strategy for large graphs: Expansion moves were only used for introducing a new color when swap-move is stuck with the current solution. We introduce the new color by finding the best expansion move on the conflict graph rather than the original graph, since the color of non-conflicting vertices do not change during an expansion move. The conflict graph

is smaller than the original graph in almost all cases, and becomes smaller as the number of conflicts is reduced during the execution of the algorithm. This observation made it possible for us, not fully but at least partially, to use the expansion move idea for large instances.

In addition to the modification described above, we have made two more changes in the original expansion-swap algorithm in order to decrease the execution time: First, we use the simple 1-exchange moves after Step 5 and Step 9 of the expansion-swap algorithm. Second, after introducing a new color at Step 9, we have looked at the best swap moves only between the new color and the old ones, but not between all the old colors.

4.2 Summary of Results

We run the expansion-swap algorithm on a 450 MHz Sun UltraSPARC-II workstation with 1024MB of RAM. We tested our algorithms on some of the benchmark instances proposed for COLOR02/03/04 [8].

Table 2 and Table 2 compare the results of the Expansion-Swap algorithm(ES) to the results of the heuristics proposed by Croitoru et al.(CL)[9], Galinier et al.(GH)[11], Bui and Patel(BP)[3], Phan and Skiena(PS)[15], Chiarandini and Stuetzle(CS)[6]. These results are summarized in [8]. Not all heuristics reported their results for all instances. Thus many cells in the table are empty.

Of the 68 test instances solved with the Expansion-Swap Algorithm, there are 18 instances with reported chromatic numbers [8]. Of these 18 instances our algorithm found the optimal solution for 10 of them.

The results of the ES algorithm for 32 instances are either equal to the optimal solution, or as good as the best result found by other heuristics listed. The results of ES for these instances are highlighted in bold in Tables 2 and 2. For 15 instances, ES either could not find the optimal solution or at least one of the other heuristics obtained a better solution. For 14 instances, ES obtained the worst results. And for the remaining 7 instances, we cannot make a comparison as we only have the results of ES but we see that these results are only one more than the clique number of 4 of these 7 instances.

In terms of instance types, we can say that our algorithm performed very well on myciel and FullIns instances, and on school1-nsh and mugg100-25. It also has a good performance on all other graphs except DSJ instances and latin-square-10. For DSJ and latin-square, the quality of the solutions are not as good, especially for the large and dense instances.

In terms of solution times, 16 instances are solved in less than 1 second, 39 instances are solved in more than 1 second but in less than 1 minute, 5 instances are solved in more than 1 minute but less than 4 minutes, 4 instances are solved in more than 4 minutes but in less than 8 minutes, and the remaining 4 instances are solved in more than 8 minutes but in less than 16 minutes.

Figure 5 presents the relationship of the solution time with the density of the graphs. As one would expect, the hardest instances are those with high density, though this does not fully explain the heuristic's running time since some high density instances can be solved quickly.

Table 2. Comparison of the results of the expansion-swap(ES) algorithm to the results of other heuristics. The columns in the table consist of the name of the graph, number of vertices (n), number of edges (m), density of the graph (d), clique number (cl), optimum solution(OPT), lower bound(LB), results due to (CL), (GH), (BP), (PS), (CS), results for expansion-swap (ES) algorithm, and time in seconds for ES (time).

Graph	n	m	d	cl.	OPT	LB	CL	GH	BP	PS	CS	ES	time
le450_5a.col	450	5714	6%	5	5	5			5	14		**5**	*3.6*
le450_5b.col	450	5734	6%	5	5	5			5	13		**5**	*8.6*
le450_5c.col	450	9803	10%	5								**5**	*4.2*
le450_5d.col	450	9757	10%	5	5	5				16		**5**	*4.1*
le450_15a.col	450	8168	8%	15		15	18		15	23	15	*18*	*51.2*
le450_15b.col	450	8169	8%	15	15	15	18		15	23	15	*18*	*46.4*
le450_15c.col	450	16680	17%	15		15	27	15		32	16	*25*	*90.8*
le450_15d.col	450	16750	17%	15		9		15		31	16	*26*	*36.5*
le450_25a.col	450	8260	8%	25								*26*	*21.3*
le450_25b.col	450	8263	8%	25								*26*	*19.8*
le450_25c.col	450	17343	17%	25		25		26		36	26	*32*	*29.3*
le450_25d.col	450	17425	17%	25		13		26		37	26	*31*	*100.9*
queen8_8.col	64	728	36%	9	9	9						*10*	*7.2*
queen8_12.col	96	1368	30%	12								*13*	*4.5*
queen9_9.col	81	2112	65%	10					10			*11*	*12.2*
queen10_10.col	100	2940	59%									*13*	*3.5*
queen11_11.col	121	3960	55%	11					12			*14*	*4.5*
queen12_12.col	144	5192	50%									*15*	*12.8*
queen13_13.col	169	6656	47%	13					14		14	*16*	*107.9*
queen14_14.col	196	8372	44%									*17*	*75.9*
queen15_15.col	225	10360	41%						17			*19*	*9.9*
queen16_16.col	256	12640	39%							21	18	*19*	*30.0*
myciel5.col	47	236	22%	6								**6**	*0.5*
myciel6.col	95	755	17%	7					7			**7**	*0.9*
myciel7.col	191	2360	13%	8					8			**8**	*1.4*
1-Insertions_4.col	67	232	10%	4		4	4		4			*5*	*0.4*
1-Insertions_5.col	202	1227	6%			4	6			6		**6**	*0.9*
1-Insertions_6.col	607	6337	3%				7			15		**7**	*5.1*
2-Insertions_3.col	37	72	11%	4								**4**	*0.2*
2-Insertions_4.col	149	541	5%	4	4	4	5		4	5	5	*5*	*0.4*
2-Insertions_5.col	597	3936	2%			4	6			11		**6**	*4.2*
3-Insertions_3.col	56	110	7%	4								**4**	*0.2*
3-Insertions_4.col	281	1046	3%			3	5			5	5	**5**	*1.2*
3-Insertions_5.col	1406	9695	1%				6			29	6	**6**	*35.4*
4-Insertions_3.col	79	156	5%	3		3			4			**4**	*0.2*
4-Insertions_4.col	475	1795	2%			3				7		**5**	*2.3*
1-FullIns_3.col	30	100	23%	4	4	4	4					**4**	*0.2*
1-FullIns_4.col	93	593	14%	5	5	5	5					**5**	*0.4*
1-FullIns_5.col	282	3247	8%	6	6	6	6			7		**6**	*1.1*
2-FullIns_3.col	52	201	15%	5		5	5					**5**	*0.3*

Table 2. *(continued)*

Graph	n	m	d	cl.	OPT	LB	CL	GH	BP	PS	CS	ES	time
2-FullIns_4.col	212	1621	7%			5	6			7		**6**	*0.7*
2-FullIns_5.col	852	12201	3%			6	7			23		**7**	*9.1*
3-FullIns_3.col	80	346	11%		5	5	6					**6**	*0.4*
3-FullIns_4.col	405	3524	4%			6	7			11	7	**7**	*4.6*
3-FullIns_5.col	2030	33751	2%			6	8			59	8	**8**	*53.7*
4-FullIns_3.col	114	541	8%	7	7	7	7					**7**	*1.0*
4-FullIns_4.col	690	6650	3%			7	8			19		**8**	*7.6*
4-FullIns_5.col	4146	77305	1%				9				9	*11*	*325.0*
5-FullIns_3.col	154	792	7%	8	8	8				8		**8**	*2.2*
5-FullIns_4.col	1085	11395	2%							27		**10**	*11.8*
DSJC125.1.col	125	736	9%	5	5	5			5	7		*6*	*1.0*
DSJC125.5.col	125	3891	50%	12		12	20		18	21		*21*	*11.8*
DSJC125.9.col	125	6961	90%	27		30			42	46		*48*	*50.2*
DSJC250.1.col	250	3218	10%		8	8			9			*10*	*3.3*
DSJC250.5.col	250	15668	50%			13	37		22		28	*36*	*46.9*
DSJC250.9.col	250	27897	90%			35			72	79		*82*	*230.4*
DSJC500.1.col	500	12458	10%			6	16	12		20	12	*15*	*42.0*
DSJC500.5.col	500	62624	50%			16	66	48	51		50	*61*	*256.7*
DSJC500.9.col	500	112437	90%	35		42		126			127	*156*	*838.3*
DSJR500.1.col	500	3555	3%	12	12	12	12					**12**	*5.3*
DSJR500.1c.col	500	121275	97%	63	63	63	56			105		*94*	*418.1*
DSJR500.5.col	500	58862	47%	26	26	26			129	155	124	*143*	*474.2*
DSJC1000.1.col	1000	49629	10%			6		20		41		*26*	*50.5*
DSJC1000.5.col	1000	249826	50%			17		84				*111*	*793.4*
DSJC1000.9.col	1000	449449	90%	37		54		224				*289*	*796.6*
latin_sq_10.col	900	307350	76%						101		99	*123*	*901.5*
school1_nsh.col	352	14612	24%	14	14	14			14	33		**14**	*29.3*
mugg100_25.col	100	166	3%	4								**4**	*0.3*

5 Conclusion

We studied a new local search algorithm using two very large-scale neighborhoods for the GCP. The first type of move allows us to swap the colors of sets of vertices. The second type of move allows any set of vertices to change their colors to a particular color. The algorithm proposed combines these two types of moves.

The key part of the algorithm is efficiently finding the best neighboring solution to the current solution by solving a MAX-CUT problem. Since MAX-CUT is a hard problem, we considered approximate algorithms that are able to find "good" solutions very fast. It is important to note that the success of the algorithms presented in this paper hinges on fast algorithms that can solve MAX-CUT problems optimally or approximately.

This study is one of the first attempts to solve the GCP using local search in very large neighborhoods. Although we could not fully take advantage of the

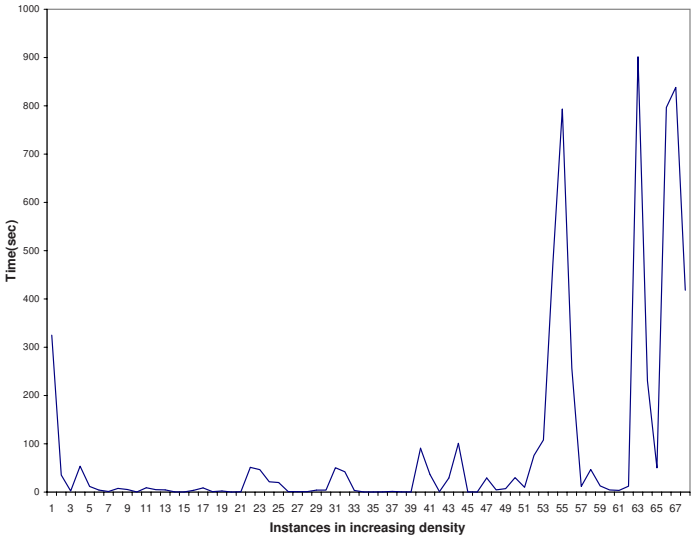


Fig. 5. The x-axis is the ordered list of 68 instances solved. The instances are ordered in ascending order of density. Instance number 1 is the least dense and instance number 68 is the most. The y-axis is the solution time.

neighborhoods by solving the MAX-CUT problems optimally, the results we present here are promising. However, further research efforts are still required to make large scale neighborhood techniques fully competitive.

One possible extension is to use exact or better approximate algorithms and to fully integrate them with the main code to solve the MAX-CUT problems. Another one is to investigate if a best-improvement variant of the Expansion-Swap algorithm would perform better than the current first-improvement search approach we use. That is instead of accepting the first improving move, using the move that gives the best improvement in conflicts.

References

1. Avanthay, C., Hertz, A., Zufferey, N.: A variable neighborhood search for graph coloring. European Journal of Operational Research, **151** (2003) 379–388.
2. Boykov, Y., Veksler, O., Zabih, R.: Fast approximate energy minimization via graph cuts. IEEE Transactions on Pattern Analysis and Machine Intelligence, **23(11)** (2001) 1222–1239
3. Bui, T.N., Patel, C.M.: An Ant System Algorithm for Coloring Graphs. In D.S. Johnson, A. Mehrotra, and M. Trick, editors, Pro- ceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
4. Burer, S., Monteiro, R.D.C., Zhang T.: Rank-two relaxation heuristics for MAX-CUT and other binary quadratic programs. SIAM Journal on Optimization, **12** (2001) 503-521

5. Chiarandini, M., Dumitrescu, I., Stuetzle, T.: Local search for the colouring graph problem. A computational study. Technical Report AIDA-03-01, FG Intellektik, TU Darmstadt (2003)
6. Chiarandini, M., Stuetzle, T.: An application of Iterated Local Search to Graph Coloring Problem. In D. S. Johnson, A. Mehrotra, M. Trick, editors, Proceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
7. CirCut: A Fortran 90 Code for Max-Cut, Max-Bisection and More. http://www.caam.rice.edu/ zhang/circut/
8. COLOR02/03/04: Graph Coloring and its Generalizations. http://mat.gsia.cmu.edu/COLOR04
9. Croitoru, C., Luchian, H., Gheorghies, O., Apetrei, A.: A New Genetic Graph Coloring Heuristic. In D. S. Johnson, A. Mehrotra, M. Trick, editors, Proceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
10. Galinier, P., Hertz, A.: A Survey of Local Search Methods for Graph Coloring. Computers & Operations Research **33** (2006) 2547–2562.
11. Galinier, P., Hertz, A., Zufferey, N.: Adaptive Memory Algorithms for Graph Coloring. In D.S. Johnson, A. Mehrotra, M. Trick, editors, Proceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
12. Garey, M.R., Johnson, D.S.:Computers and Interactibility: A Guide to the Theory of NP-Completeness. Freeman, San Francisco, CA, USA, (1979).
13. Goemans, M.X., Williamson, D.P.: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. Journal of ACM **42** (1995) 1115-1145.
14. Mehrotra, A., Trick, M.: A column generation approach for graph coloring. INFORMS Journal On Computing **8(4)** (1996) 344-354
15. Phan, V., Skiena, S.: Coloring Graphs With a General Heuristic Search Engine. In D.S. Johnson, A. Mehrotra, M. Trick, editors, Proceedings of the Computational Symposium on Graph Coloring and its Generalizations, Ithaca, NY (2002)
16. Thompson, P.M, Psaraftis, H.N.: Cyclic transfer algorithms for multivehicle routing and scheduling problems. Operations Research **41** (1993) 70-79

Generalizations of the Global Cardinality Constraint for Hierarchical Resources

Alessandro Zanarini and Gilles Pesant

Département de génie informatique
École Polytechnique de Montréal
C.P. 6079, succ. Centre-ville
Montreal, Canada H3C 3A7
{azanarini,pesant}@crt.umontreal.ca

Abstract. We propose generalizations of the Global Cardinality Constraint (gcc) in which a partition of the variables is given. In the context of resource allocation problems, such constraints allow the expression of requirements, in terms of lower and upper bounds, for resources with different capabilities. Alternate models using gcc's are shown to be weaker. We present filtering algorithms based on flow theory that achieve domain consistency and give experimental evidence of the usefulness of such constraints. We consider an optimization version of the constraints and discuss its relationship with the cost_gcc.

1 Introduction

Resource allocation problems occur in many real-life problems whenever it is necessary to assign resources to tasks that need to be accomplished. It can be thought of as a one to one assignment or, more generally, a many to one relation in which tasks can be assigned one or more resources. Typically, for each task a minimum and maximum number of required resources is defined. Resources may be *homogeneous* in the sense that they have identical capabilities or skills. In Constraint Programming, problems with homogeneous resources can be easily modeled by a Global Cardinality Constraint [6] (gcc) in which each resource is represented by a variable whose domain is the set of tasks and each task defines its resource requirements through the bounds on the number of occurrences in the definition of the constraint. However for some real-world problems this scenario is too simplistic: resources are heterogeneous and tasks require resources with different capabilities or skill levels. We further distinguish three cases: in the first, referred to as *disjoint heterogeneous resources*, the different skill levels are considered independently i.e. a resource with a given skill level can only satisfy requirements defined on this level; in the second, referred to as *nested heterogeneous resources*, resources are organized in a total order, that is, a resource with skill level ℓ is able to satisfy requirements of level ℓ or below; in the third, which we call *hierarchical heterogeneous resources*, the relationship between resources generalizes beyond the linear order of the nested case to a tree-like hierarchy.

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 361–375, 2007.

Problems with disjoint heterogeneous resources are also easily modeled, this time using a set of gcc's, each of them representing a single skill level as before, and domain consistency can still be achieved. Unfortunately a similar model for the nested and hierarchical cases does not guarantee domain consistency. This paper focuses on the important cases of nested and hierarchical heterogeneous resources for which we propose generalizations of the global cardinality constraint that achieve domain consistency.

Example 1. We need to accomplish two tasks $T1$ and $T2$ that have different requirements of resources of level 1 and 2. Three resources R_1^1, R_2^1, R_3^1 of level 1 and three resources R_1^2, R_2^2, R_3^2 of level 2 are available. Each resource can be assigned to any task. Both tasks $T1$ and $T2$ need between 1 and 2 resources of level 2, and between 2 and 3 resources of level 1. In a disjoint heterogeneous resources setting, resources can only satisfy requirements of their level. Since the tasks need at least 4 resources of level 1 the problem is unsatisfiable. In a nested heterogeneous resources setting, resources can satisfy requirements of their level or below. The minimum requirements of resources of level 2 is equal to 2 (one for each task). Then, one resource of level 2 can be assigned to a task for satisfying the requirements of level 1. Thus, the problem is satisfiable.

Note that in the case of nested heterogeneous resources the problem can be restated as follows: both tasks $T1$ and $T2$ need respectively between 3 and 5 resources of level 1 or higher, and among them 1 or 2 must be of level 2.

The initial motivation for this work came from a real-life manpower planning and scheduling problem proposed in [8] by France Telecom for the 2007 ROADEF Challenge. A subproblem consists of forming teams of technicians that have to accomplish a set of tasks. The technicians have different skill levels and a technician can satisfy task requirements of his level or below. Each task defines the minimum number of technicians required for each skill level. This corresponds exactly to nested heterogeneous resources. Another important application area is nurse rostering. Here a minimum number of nurses on duty is specified for each work shift and sometimes a minimum is also given for senior nurses acting in a supervisory role but who can perform the duties of regular nurses as well. Applications of hierarchical heterogeneous resources are found in the computer software industry or generally in large projects with multiskilled resources.

The paper is organized as follows: Section 2 gives a brief background of Constraint Programming and Network Flows that will be used in the following sections. In Section 3, we formally introduce the `nested_gcc`, its graph representation as well as the theoretical basis for achieving domain consistency. Section 4 is dedicated to a generalization of the `nested_gcc` called `hierarchical_gcc`. In Section 5 we show experimental evidence of the usefulness of the presented constraints. Section 6 considers an optimization version of `nested_gcc` that allows the expression of preferences. Finally, conclusions are drawn in Section 7.

2 Preliminaries

2.1 Constraint Satisfaction Problem

A *Constraint Satisfaction Problem* (CSP) consists of a finite set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ with finite domains $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ such that $x_i \in D_i$ for all i, together with a finite set of constraints $\mathcal{C}$, each on a subset of $\mathcal{X}$. A constraint $C \in \mathcal{C}$ is a subset $T(C)$ of the Cartesian product of the domains of the variables that are in C. We write $X(C)$ to denote the set of variables involved in C and we call tuple $\tau \in T(C)$ an allowed combination of values of $X(C)$. Given a set of variables $X' \subseteq X(C)$, $\tau \downarrow_{X'}$ is the projection of the tuple τ over the set X'. The number of occurrences of a value d in a tuple τ is denoted by $\#(d, \tau)$; analogously $\#(d, \tau \downarrow_{X'})$ is the number of occurrences of d in the projection of the tuple τ over the set X'. An assignment $(d_1, \ldots, d_k) \in X(C)$ satisfies a constraint C if it belongs to $T(C)$. A *solution* to a CSP is an assignment of all the variables such that it satisfies all the constraints.

In Constraint Programming (see [2]), the solution process consists of iteratively interleaving search phases and propagation phases. During the search phase all the combinations of values are evaluated. It is generally performed on a tree-like structure and each step consists of instantiating a variable to a value of its domain. In order to avoid the systematic generation of all the combinations and reduce the search space, the propagation phase shrinks the search space: each constraint propagation algorithm removes values that a priori cannot be part of a solution w.r.t. the partial assignment built so far. The propagation can be eventually performed an exponential number of times thus it needs to be efficient and effective. In order to be effective, each constraint filtering algorithm should remove as many variable domain values as possible and possibly achieve *domain consistency* (also referred to as hyper-arc consistency or generalized-arc consistency).

Definition 1 (Domain Consistency). *Given a constraint C defined on the variable set $x_1, \ldots, x_n$ with respective domains $D_1, \ldots, D_n$, the constraint is* domain consistent *iff for each variable x_i and each value $d_i \in D_i$ there exists a value $d_j \in D_j$ for all $j \neq i$ such that $(d_1, \ldots, d_n) \in T(C)$.*

2.2 Network Flows

In this section we recall the main results and definitions that will be used in the following sections (see [1] for further explanations). An oriented graph is defined as $G = (V, A)$ where V is a set of vertices and A is a set of ordered pairs (arcs) from V. We write $\delta^{out}(v)$ to refer to the set of outgoing arcs of v: $\delta^{out}(v) = \{(v, u) \,|\, (v, u) \in A\}$. Similarly, the set of ingoing arcs of v is denoted by $\delta^{in}(v) = \{(u, v) \,|\, (u, v) \in A\}$. An oriented path in a oriented graph $G = (V, A)$ is a sequence of vertices $v_0, v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in A$ with $i = 0, \ldots, k-1$. An oriented graph is called *strongly connected* iff for each ordered pair (u, v) of vertices there exists an oriented path from u and v. A *strongly connected*

component of an oriented graph $G = (V, A)$ is a strongly connected subgraph G' of G such that no other strongly connected subgraph of G contains G'.

Let $G = (V, A)$ be an oriented graph, $l(a)$ and $c(a)$ the demand and the capacity of each arc $a \in A$ $(0 \leq l(a) \leq c(a))$. We define an *s-t flow* as a function $f : A \rightarrow \mathbb{R}$ such that:

$$\forall v \in V \setminus \{s, t\} : \sum_{a \in \delta^{out}(v)} f(a) = \sum_{a \in \delta^{in}(v)} f(a)$$

where s and t are respectively *source* and *sink* of the flow. The flow is feasible if $\forall a \in A : l(a) \leq f(a) \leq c(a)$. The value of a flow f is defined as $value(f) = \sum_{\{a \in \delta^{out}(s)\}} f(a) - \sum_{\{a \in \delta^{in}(s)\}} f(a)$. A feasible flow f is maximum if there is no other feasible flow f' such that $value(f') > value(f)$.

Theorem 1. *If all arc demands and capacities are integer and there exists a feasible flow then the maximum flow problem has an integer maximum flow.*

Given a flow f on a graph $G = (V, A)$, the residual graph is defined as $G_f = (V, A_f)$ where $A_f = \{(u, v) \in A : f((u, v)) < c((u, v))\} \bigcup \{(v, u) : (u, v) \in A, l((u, v)) < f((v, u))\}$.

3 Nested Global Cardinality Constraint

Let $\{X^k\}_{1 \leq k \leq \ell}$ represent a family of ℓ disjoint sets of variables. Define further $\mathbb{X}^k = \bigcup_{k \leq j \leq \ell} X^j$, with $\mathbb{X} = \mathbb{X}^1$ for short. Observe that this new family of sets is nested: $\mathbb{X}^\ell \subseteq \mathbb{X}^{\ell-1} \subseteq \cdots \subseteq \mathbb{X}^1$. The variables $X^k = \{x_1^k, \ldots, x_{n_k}^k\}$ are defined over the domains $D_1^k, \ldots, D_{n_k}^k$. We write D_{X^k} for the union of the domains of the variables in X^k; analogously $D_{\mathbb{X}}$ stands for the union of all the domains of the variables in $\mathbb{X}$.

We denote by l_d^k and u_d^k the lower and upper bounds on the number of occurrences of value $d \in D_{\mathbb{X}}$ among $\mathbb{X}^k$. It follows that we should have $l_d^{k+1} \leq l_d^k$ and $u_d^{k+1} \leq u_d^k$ for $k = 1, \ldots, \ell - 1$. For example $l_d^1 = 5$, $u_d^1 = 7$, $l_d^2 = 3$, $u_d^2 = 4$ means that value d occurs between 5 and 7 times in $\mathbb{X}^1$, including between 3 and 4 times in $\mathbb{X}^2$. The related vectors of lower and upper bounds are denoted by l^k and u^k for $k = 1, \ldots, \ell$.

Definition 2 (nested gcc). *The nested global cardinality constraint is formally defined as*

$$\texttt{nested\_gcc}(\mathtt{X}^1, \ldots, \mathtt{X}^\ell, (\mathtt{l}^1, \mathtt{u}^1), \ldots, (\mathtt{l}^\ell, \mathtt{u}^\ell)) =$$

$$\{\tau = (d_1^1, \ldots, d_{n_1}^1, d_1^2, \ldots, d_{n_\ell}^\ell) \,|\, d_i^k \in D_i^k, \forall 1 \leq k \leq \ell, \forall d \in D_{\mathbb{X}} : l_d^k \leq \#(d, \tau \downarrow_{\mathbb{X}^k}) \leq u_d^k\}$$

Note that it is possible to model the `nested_gcc` as a set of traditional `gcc`'s: for each set $\mathbb{X}^k$, we define a `gcc` in which we set the corresponding upper and lower bounds. But such a formulation is strictly weaker, as we shall see in Proposition 1.

Going back to our resource allocation problem, the tasks would correspond to the values and the resources to the variables, arranged in disjoint sets according to their level. As defined, the constraint requires that each resource be assigned to a task. In some problems, like rostering, it might be useful to find an assignment that, while satisfying the lower bound constraints, keeps some resources unassigned. This can be easily modeled by adding an extra value (without requirements) representing a fake activity; resources that are assigned to it are in fact unused.

Example 2. A company needs to form some teams in order to accomplish 5 tasks. Only 6 technicians with different skills are available. Three of them have skill level equal to 2 (capable of accomplishing a job requiring skill level 1 or 2) and the remaining three have a skill level equal to 1. Moreover the three technicians with basic skills are not allowed to be assigned to the task 5. We model the problem with 6 variables representing the resources divided in two sets: $X^1 = \{x_1^1, x_2^1, x_3^1\}$ and $X^2 = \{x_1^2, x_2^2, x_3^2\}$. The variable domains are respectively: $D_1^1 = D_2^1 = D_3^1 = \{d_1, d_2, d_3, d_4\}$ and $D_1^2 = D_2^2 = D_3^2 = \{d_1, d_2, d_3, d_4, d_5\}$. The tasks require respectively a minimum of 1, 1, 1, 1 and 2 technicians of skill level at least 1. Tasks 3 and 4 each need at least one technician of level 2. None of the tasks can accommodate more than 3 technicians (independently from the level). We would model this situation as $\texttt{nested\_gcc}(\{\texttt{x}_1^1, \texttt{x}_2^1, \texttt{x}_3^1\}, \{\texttt{x}_1^2, \texttt{x}_2^2, \texttt{x}_3^2\}, ([\texttt{1}, \texttt{1}, \texttt{1}, \texttt{1}, \texttt{2}], [\texttt{3}, \texttt{3}, \texttt{3}, \texttt{3}, \texttt{3}]), ([\texttt{0}, \texttt{0}, \texttt{1}, \texttt{1}, \texttt{0}], [\texttt{3}, \texttt{3}, \texttt{3}, \texttt{3}, \texttt{3}]))$. The alternate model using two gcc's is illustrated at Figure 1.

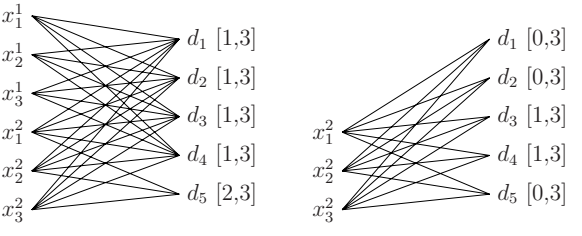


Fig. 1. Traditional GCC modelling for the Nested-GCC

Proposition 1. *Modelling the constraint* nested_gcc *as a set of traditional* gcc *does not achieve domain consistency.*

Proof. Consider Example 2. Both gcc constraints are domain consistent however the instance is unsatisfiable. Two variables in X^2 must take the value d_3 and d_4 (from the level 2 gcc), hence there is only one variable left to assign to d_5 that requires a minimum of two variables (from the level 1 gcc).

Even though it has been proven ([3]) that finding a consistent solution to a set of overlapping gcc's is an NP-Complete problem, the particular nested structure is such that it is possible to find a consistent assignment in polynomial time as we shall see in the next section.

3.1 Graph Representation

We propose a new graph representation for the `nested_gcc`. Informally, it contains vertices representing the variables from the X^k sets and vertices that denote the values; differently from the traditional `gcc`, the value vertices are duplicated for each set X^k while variable vertices remain singletons; arcs connect successive replications of value vertices. In order to identify duplicate value vertices, for each value $d_i \in D_{\mathbb{X}}$ we add a superscript that refers to its corresponding set X^k. We write $D^k_{\mathbb{X}}$ to denote the set of values in $D_{\mathbb{X}}$ with superscript k; hence $d^k_i \in D^k_{\mathbb{X}}$ and $d^{k'}_i \in D^{k'}_{\mathbb{X}}$ represent the value d_i but two different value vertices for sets X^k and $X^{k'}$. The directed graph $G = (V, A)$ is defined as follows:

$$V = \mathbb{X} \cup \Big(\bigcup_{k=1}^{\ell} D^k_{\mathbb{X}}\Big) \cup \{s, t\}$$

$$A = A_s \cup \Big(\bigcup_{k=1}^{\ell} A_{X^k}\Big) \cup \Big(\bigcup_{k=1}^{\ell} A^k_{req}\Big)$$

where

$$\begin{aligned} A_s &= \{(s, x^k_i) \mid k = 1, \dots, \ell\,,\ i = 1, \dots, n_k\} \\ A_{X^k} &= \{(x^k_i, d^k_j) \mid i = 1, \dots, n_k, d^k_j \in D^k_i\} \\ A^k_{req} &= \begin{cases} \{(d^1_i, t) \mid i = 1, \dots, |D_{\mathbb{X}}|\} & \text{if } k = 1 \\ \{(d^k_i, d^{k-1}_i) \mid i = 1, \dots, |D_{\mathbb{X}}|\} & \text{if } 2 \le k \le \ell \end{cases} \end{aligned}$$

The lower bounds and upper bounds of the arcs $a \in A_s$ are unitary; they are respectively null and unitary for the arcs $a \in A_{X^k}$. For each arc (d^k_i, v) the lower bound is equal to l^k_i and the upper bound is u^k_i.

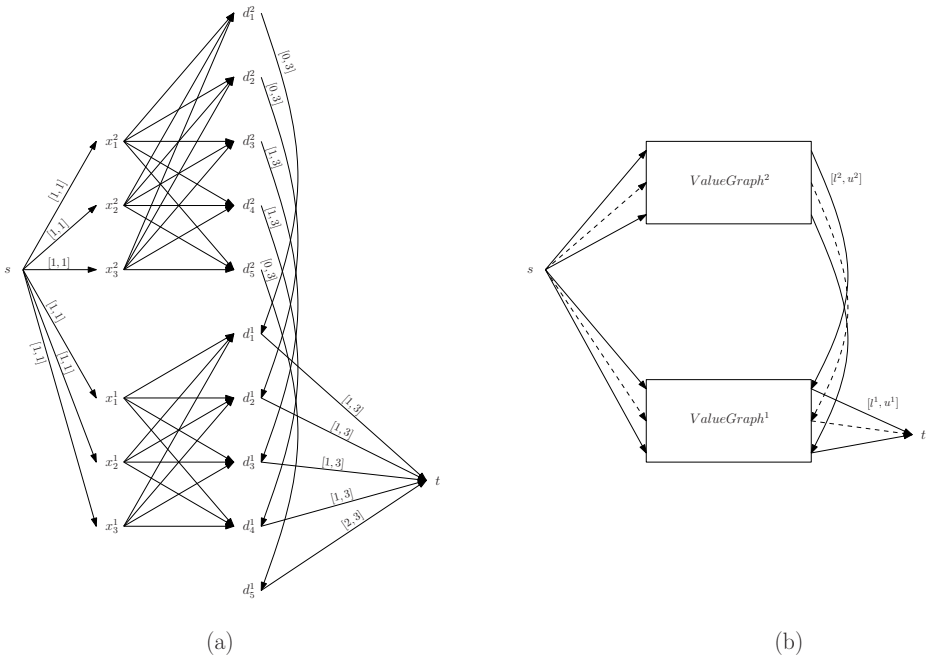
The graphical representation of Example 2 is given at Figure 2.

3.2 Domain Consistency and Propagation Algorithm

A feasible flow on the introduced graph representation reflects a feasible assignment of the `nested_gcc` constraint. A flow going from a variable vertex x^k_i to a value vertex d^k corresponds to the assignment $x^k_i = d$. In addition a value vertex d^k collects all the flow coming from duplicate value vertices $d^j, j \ge k$, which means it receives a flow equal to the number of assignments of d to variables in $\mathbb{X}^k$. For a given value vertex, the bounds on the single outgoing arc constraint the number of occurrences according to the definition of the constraint, by construction.

Theorem 2. *There is a bijection between solutions to the* `nested_gcc` *and feasible flows in the related graph representation G.*

Proof. $\Rightarrow$ *Given a solution, we can build a feasible flow setting a unitary flow in the arc (x^k_i, d^k_j) for each assignment $x^k_i = d_j$. The arcs in A_s are all saturated. An arc $(d^k_j, v) \in A^k_{req}$ has a flow equal to $\#(d_j, \tau \downarrow_{\mathbb{X}^k})$. Note that for any given k and vertex d^k_j, demands and capacities of the arc $(d^k_j, v) \in A^k_{req}$ are satisfied*



(a) (b)

Fig. 2. (a) Nested-GCC Graph Representation for Example 2: if not shown the lower and upper bounds are respectively null and unitary. (b) Schematic graph representation for Example 2.

since the related flow is equivalent to the sum of the flow coming from level k and higher.
⇐ *Given a feasible (integral) flow, we build an assignment setting* $x_i^k = d_j$ *whenever* $f(x_i^k, d_j^k) = 1$.

Consider again Example 2: the constraint is infeasible and there is no feasible flow in the related graph G of Figure 2.

Corollary 1. *Let G be the graph representation of a* `nested_gcc` *and f a feasible flow on G. The constraint is domain consistent iff for each arc $a \in A_{X^k}$ there exists a feasible flow such that $f(a) = 1$.*

Proof. From Theorem 2, if there exists a feasible flow that has $f(a) = 1$ with $a = (x_i^k, d^k)$ then there exists a solution with $x_i^k = d$. Analogously, if there exists a solution with $x_i^k = d$ then there exists a flow with $f(a) = 1$ where $a = (x_i^k, d^k)$.

Following Régin in [6], we can design a filtering algorithm in which we find a feasible flow in the graph representation in order to check the feasibility of the constraint. If it does not exist then the constraint is infeasible. Otherwise, we compute the strongly connected component [7] of the residual graph and then every arc that does not belong to any strongly connected component can be removed.

3.3 Complexity

In the following, we use N to indicate the total number of variables and d for $|D_{\mathbb{X}}|$. The propagation of the `nested_gcc` requires $O(nm)$ time to find a feasible flow (Ford-Fulkerson) and $O(n+m)$ to find infeasible values where n is the number of vertices and m is the number of arcs of the `nested_gcc` graph representation. Here, n is in $O(N+d\ell)$ and m is in $O(Nd+d\ell)$. Note that the equivalent set of `gcc`'s representing the `nested_gcc` requires the propagation of ℓ different `gcc`'s.A single `gcc` propagates in $O(\sqrt{n'}m')$ [4] where n' and m' are respectively the number of vertices and the number of arcs of the `gcc` graph representation and $n' \in O(N+d)$ and $m' \in O(Nd)$.

4 Further Generalization

So far, skill levels have been considered linearly ordered: a resource of level k is able to satisfy requirements of levels $k, k-1, \ldots, 1$. The main challenge is now how far we can generalize relations between skill levels in order to address more complex problems while still using the flow algorithm.

Different skill level relations are shown in Figure 3: in (a) the skill levels are linearly ordered while in (b) levels are organized in a tree-like fashion. The semantic of 3b is that both resources of type β and γ can accomplish a task with requirements of type α, whereas resources of type β cannot satisfy requirements of type γ (and the other way around). Equivalently, we write $\delta \succ \beta$, $\beta \succ \alpha$, $\epsilon \succ \gamma$, $\zeta \succ \gamma$, $\eta \succ \gamma$ and $\gamma \succ \alpha$, where we consider $\succ$ a reflexive, antisymmetric and non-transitive relation. The transitive closure of $\succ$ is denoted by $\succ^*$ (hence, for instance $\delta \succ^* \alpha$). Note that the relation between resource classes is not a partial order relation: we cannot have $\lambda \succ \mu$ and $\lambda \succ \nu$ or, in other words, lower classes cannot rejoin in a single higher class. The reason of this limitation will be clarified in the next paragraph. Furthermore the relation set is such that there exists only a single root: a definition of multiple roots (a forest) simply gives rise to different constraints.

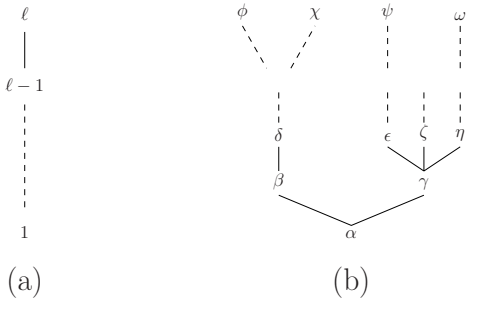


Fig. 3. Skill level relations: (a) linearly ordered skill levels and (b) tree-like ordered skill levels

Example 3. A company is planning to develop two software components c_1 and c_2 for an application. The component c_1 requires between 7 and 10 programmers while c_2 between 8 and 10. Particularly, both components need 1 or 2 expert developers and 3 or 4 testers. A programmer is either a basic developer or an expert developer or a tester, however both expert developers and testers can accomplish duties as a basic developer ("expert developer" $\succ$ "basic developer", "tester" $\succ$ "basic developer"). The company has 4 novices, 8 testers and 3 expert developers. The different relations and component requirements are depicted in Figure 4. A possible solution is to assign 4 testers for each component; one tester for each component should work as a basic developer. Novices are evenly divided between the two tasks, one expert developer will be assigned to the development of component c_1 and finally the remaining two expert developers will work for the component c_2 (one as a basic developer).

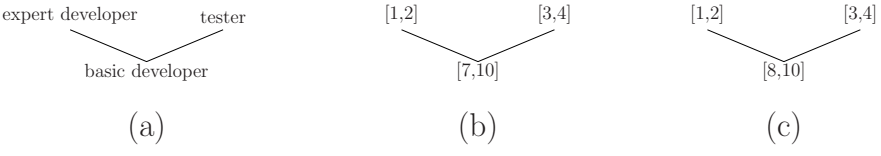


Fig. 4. (a) Programmers skill relations. (b) Requirements for component c_1. (c) Requirements for component c_2.

Note that, more generally, whenever we have a taxonomy or hierarchy of resources, we can easily derive the resource relations. This scenario fits perfectly applications in which resources are represented as classes in a UML class diagram and they are organized in a hierarchy (with single inheritance); a subclass by definition is a specialization of the superclass, it is able to act as the superclass (the subclass "is" a kind of superclass) but it has additional capabilities.

We now formally introduce the constraint that models the described problem substructure. In the following, Σ represents the set of different resource classes where, arbitrarily, α is considered the lower level (i.e. the root class). The variables representing resources of class $\gamma \in \Sigma$ are denoted by X^γ. We write $\mathbb{X}^\lambda = \bigcup\{X^\gamma \,|\gamma \in \Sigma, \gamma \succ^* \lambda\}$ to represents the union of the variables of level λ and higher w.r.t. the relation $\succ$[1]. For short, we write $\mathbb{X} = \mathbb{X}^\alpha$.

Definition 3 (hierarchical gcc). *The hierarchical global cardinality constraint is formally defined as*

$$\texttt{hierarchical\_gcc}(\texttt{X}^\alpha, \ldots, \texttt{X}^\omega, (\texttt{l}^\alpha, \texttt{u}^\alpha), \ldots, (\texttt{l}^\omega, \texttt{u}^\omega), \succ) =$$

$$\{\tau = (d_1^\alpha, \ldots, d_{n_\alpha}^\alpha, d_1^\beta, \ldots, d_{n_\omega}^\omega) \,|\, d_i^\gamma \in D_i^\gamma, \forall \gamma \in \Sigma, \forall d \in D_\mathbb{X} : l_d^\gamma \leq \#(d, \tau \downarrow_{\mathbb{X}^\gamma}) \leq u_d^\gamma\}$$

[1] Of which the linear order relation used for the `nested_gcc` is a special case.

4.1 Graph Representation

The graph representation is similar to the one introduced for the `nested_gcc`; it differs mainly in how the `gcc` subgraphs are connected. We have a `gcc` substructure for each resource class; value vertices are still duplicated and they are connected to the equivalent value vertices following the resource relations. Note again that resources can be only of a given class, hence a resource is represented by exactly one vertex. The total amount of flow coming out from a variable vertex is still unitary.

More formally, the graph $G = (V, A)$ is defined as follows:

$$V = \mathbb{X} \cup \Big(\bigcup_{\gamma \in \Sigma} D^{\gamma}_{\mathbb{X}} \Big) \cup \{s, t\}$$

$$A = A_s \cup \Big(\bigcup_{\gamma \in \Sigma} A_{X^{\gamma}} \Big) \cup \Big(\bigcup_{\gamma \in \Sigma} A^{\gamma}_{req} \Big)$$

where

$$\begin{aligned} A_s &= \{(s, x^{\gamma}_i) \mid \gamma \in \Sigma\,,\ i = 1, \ldots, n_{\gamma}\} \\ A_{X^{\gamma}} &= \{(x^{\gamma}_i, d^{\gamma}_j) \mid \gamma \in \Sigma, d^{\gamma}_j \in D^{\gamma}_i\} \\ A^{\gamma}_{req} &= \begin{cases} \{(d^{\alpha}_i, t) \mid i = 1, \ldots, |D_{\mathbb{X}}|\} & \text{if } \gamma = \alpha \\ \{(d^{\gamma}_i, d^{\lambda}_i) \mid i = 1, \ldots, |D_{\mathbb{X}}| : \gamma \succ \lambda\} & \text{if } \gamma \neq \alpha \end{cases} \end{aligned}$$

Arcs in A_s have unitary lower and upper bounds, whereas arcs in $A_{X^{\gamma}}$ have null lower bounds and unitary upper bounds. Each arc $(d^{\gamma}_i, v) \in A^{\gamma}_{req}$ has lower and upper bound respectively equal to $l^{\gamma}_{d_i}$ and $u^{\gamma}_{d_i}$.

An example is given in Figure 5; four classes of resources are defined with the following relations: $\delta \succ \beta$, $\beta \succ \alpha$ and $\gamma \succ \alpha$. The equivalent constraint graph representation is shown in Figure 5b.

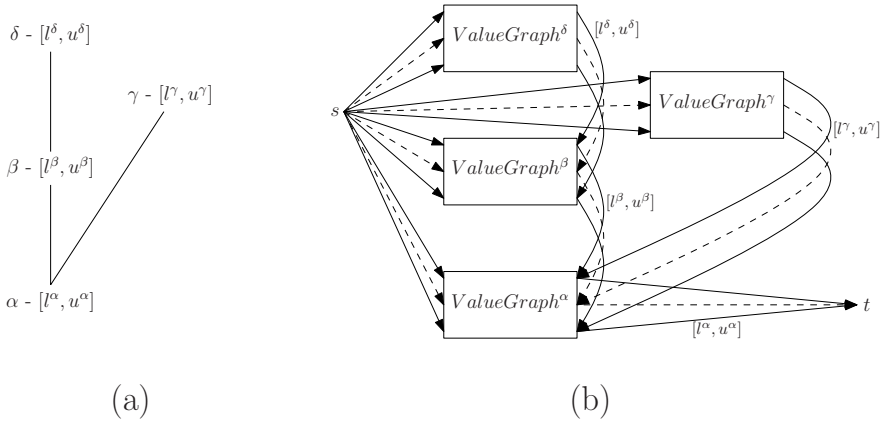


Fig. 5. (a) Resource relation. (b) Constraint graph representation.

One of the reasons why it is not possible to express resource relations as a partially ordered set (poset) is that we might have a situation like: $\lambda \succ \mu$ and $\lambda \succ \nu$. Thus, the outcome of a gcc substructure may have to flow in two different gcc substructures: $\texttt{gcc}^{\lambda}$ should output both in $\texttt{gcc}^{\mu}$ and $\texttt{gcc}^{\nu}$. Doubling the input flow (and consequently the output flow) of the $\texttt{gcc}^{\lambda}$ will clearly lead to inconsistencies inside the gcc substructure.

4.2 Domain Consistency and Propagation Algorithm

Theorem 3. *There is a bijection between solutions to the* hierarchical_gcc *and feasible flows in the related graph representation G.*

Proof. $\Rightarrow$ *Given a solution, we can build a feasible flow setting a unitary flow in the arc (x_i^γ, d_j^γ) for each assignment $x_i^\gamma = d_j$. The arcs in A_s are all saturated. An arc $(d_j^\gamma, v) \in A_{req}^\gamma$ has a flow equal to $\#(d_j, \tau \downarrow_{\mathbb{X}^\gamma})$.*
$\Leftarrow$ *Given a feasible (integral) flow, we build an assignment setting $x_i^\gamma = d_j$ whenever $f(x_i^\gamma, d_j^\gamma) = 1$.*

Corollary 2. *Let G be the graph representation of a* hierarchical_gcc *and f a feasible flow on G. The constraint is domain consistent iff for each arc $a \in A_{X^\gamma}$ there exists a feasible flow such that $f(a) = 1$.*

Proof. From Theorem 3, if there exists a feasible flow that has $f(a) = 1$ with $a = (x_i^\gamma, d_j^\gamma)$ then there exists a solution with $x_i^\gamma = d_j$. Analogously, if there exists a solution with $x_i^\gamma = d_j$ then there exists a flow with $f(a) = 1$ where $a = (x_i^\gamma, d_j^\gamma)$.

The propagation algorithm works exactly as in the nested_gcc with the only difference given by the graph. The resulting complexity is then equivalent, that is, $O(nm)$ where n and m are respectively the number of vertices and edges of the graph representation. Here, n is in $O(N + d\ell)$ and m is in $O(Nd + d\ell)$.

5 Experimental Results

We implemented the nested_gcc and hierarchical_gcc and we compared them with the equivalent set of gcc's. Due to time constraints, we were able to generate significant instances only for the nested_gcc setting. We chose as benchmark some random instances of the ROADEF challenge. We recall briefly that the problem consists of grouping technicians in teams in order to accomplish a set of tasks. A technician has skills in different domains and, particularly, he has associated a skill level for each domain; a technician is able to satisfy requirements of his skill level and lower. A task requires a specified number of technicians for each pair domain-level in order to be accomplished. The goal is to form teams of technicians such that they are able to perform a given set of tasks.

The problem is modeled as a set of nested_gcc's one for each domain where the variables represent the technicians and the values represent the tasks. As variable selection heuristic, we developed an ad-hoc heuristic that chooses the

Table 1. Experimental results

Instance	Perc.	nested_gcc		gcc's		ILOG gcc's	
		Time (secs)	Backtracks	Time (secs)	Backtracks	Time (secs)	Backtracks
data11-a	0.1	0.01	4	0.01	4	0.01	4
data11-a	0.2	0.01	0	0.01	0	0.01	0
data11-a	0.3	13.61	73381	14.12	73381	**11.92**	73381
data11-a	0.4	0.40	2261	0.38	2261	**0.33**	2261
data11-a	0.5	0.02	68	**0.01**	68	**0.01**	68
data11-b	0.1	-	1944273	-	2444379	-	3800974
data11-b	0.2	**35.75**	106324	56.27	202690	51.76	202690
data11-b	0.3	**3.85**	10992	4.65	17454	3.96	17454
data11-b	0.4	3.1	8267	2.75	9219	**2.51**	9219
data11-b	0.5	1.76	4986	**1.36**	5382	1.52	5382
data11-c	0.1	6.43	23531	5.49	23778	**3.52**	23778
data11-c	0.2	3.18	11771	2.71	11979	**1.81**	11979
data11-c	0.3	0.07	247	0.07	247	**0.05**	247
data11-c	0.4	0.01	1	0.01	1	0.01	1
data11-c	0.5	0.01	1	0.01	1	0.01	1
data11-d	0.1	**4.15**	16478	7.04	26766	5.59	26766
data11-d	0.2	**0.01**	11	**0.01**	38	0.02	38
data11-d	0.3	0.01	4	0.01	15	0.01	15
data11-d	0.4	0.01	1	0.01	1	0.01	1
data11-d	0.5	0.01	1	0.01	1	0.01	1
data12-a	0.2	-	2219874	-	2495188	-	2763002
data12-a	0.3	**140.43**	422107	-	2373366	-	2650290
data12-a	0.4	**0.16**	419	135.44	505243	154.01	505243
data12-a	0.5	**0.11**	300	6.47	22099	6.64	22099
data12-a	0.6	**0.01**	1	0.03	75	0.04	75
data12-b	0.2	-	1327519	-	1601952	-	1231394
data12-b	0.3	-	1376981	-	1642615	-	1169144
data12-b	0.4	**20.16**	45634	36.00	109762	53.24	109762
data12-b	0.5	**0.38**	827	0.52	1274	0.74	1274
data12-b	0.6	0.01	1	0.01	1	0.01	1
data12-c	0.2	-	1810092	-	2184929	-	3317824
data12-c	0.3	**13.24**	36336	20.12	75787	14.74	75787
data12-c	0.4	1.12	2835	1.55	5475	**1.02**	5475
data12-c	0.5	**0.12**	266	0.28	1033	0.26	1033
data12-c	0.6	**0.05**	69	0.83	2618	0.74	2618
data12-d	0.2	-	1434098	-	2363013	-	2361900
data12-d	0.3	-	1483539	-	2185816	-	2030271
data12-d	0.4	179.00	353462	**148.62**	445726	172.92	445726
data12-d	0.5	98.02	185798	74.77	203920	**74.04**	203920
data12-d	0.6	0.09	152	0.06	152	**0.04**	152

most skilled technicians first; from preliminary tests this heuristic seemed to narrow the gap between nested_gcc and the set of gcc representations. The testbed has been generated as follows: each task has associated an optimistic approximation of the technicians needed; then from the set of tasks, we chose randomly

a subset such that the sum of the approximations is less than the number of available technicians. Furthermore, we randomly removed values from variable domains according to an input percentage.

The constraint has been implemented with Ilog Solver 6.2 and the tests were performed on a machine with an AMD Dual Opteron 250 (2.4GHz) with 3GB RAM (note however that only one processor has been used). We set a time limit of 600 seconds for each run. We compared two different models: the former exploits the `nested_gcc`, the latter uses traditional `gcc`'s. For a fair comparison, the second model has been solved using both our implementation of the `gcc` and the ILOG's one. The results are shown in Table 1. Instances from data11 have 4 skill domains with 4 skill levels each whereas instances from data12 have 5 domains with 3 skill levels each. Each instance has been tried with different percentages of domain value removals (shown in the second column). The remaining columns show the running times (for finding a solution or proving infeasibility) and the number of backtracks respectively for the `nested_gcc`, `gcc` and the ILOG's one; in all three approaches the same variable and value ordering heuristics have been used. If the running time is not shown, the solver timed out either without finding a solution or without proving the infeasibility of the instance (however in those cases we show the number of backtracks performed within the time limit).

Depending on the instance, the reduction on the number of backtracks using the `nested_gcc` can go from null to two orders of magnitude; whenever the reduction is significant, we obtain better running times. Nonetheless there are instances in which even with our implementation of the `gcc` we get better performances over the `nested_gcc`. Hence, further studies are required in order to better characterize the instances and understand when the use of the `nested_gcc` is likely to lead to better running times. We think that an instance generator with a more fine grained parameterization could help us in this task as well as in generating a broader testbed. In fact, the basic generator produced too many instances either too easy or too hard, the same problem that we encountered also during the generation of a testbed for the `hierarchical_gcc`.

6 Expressing Preferences

For ease of presentation, in this section we take into consideration only linearly ordered resources. However the results can be easily extended to the hierarchical version.

In the constraints presented, there might be cases in which we would like to define some preferences among different consistent assignments. Imagine, for example, that a given task requires one resource of level 1 and one of level 2; however only one resource of level 2 and one of level 3 are available. The constraint does not allow to express a difference between a solution in which the level 3 resource will perform level 1 duties or a solution in which the level 2 resource will carry out level 1 duties and the level 3 resource level 2 duties. Nonetheless, in both solutions we simply have the two resources assigned to the same task without any information about who is going to perform what. We

should then enrich the model in order to express this new information. In this new setting, domains should contain for each task different values to denote different duty levels. So, d_j^k represents the task j with duty level k; assignment $x_i^{k'} = d_j^k$ means that resource i of level k' is going to perform duties of level k of task j. For the sake of clarity, note that in the `nested_gcc` we would have had simply $x_i^{k'} = d_j$; the differentiation of duty levels is only present inside the graph representation but not at the constraint level; for expressing preferences such differentiation needs to be brought up to the constraint level.

In the graph representation of `nested_gcc` with preferences, a variable $x_i^{k'}$ is connected directly to value vertices d_j^k with $k \leq k'$. Lower and upper bounds of value occurrences are expressed directly for each d_j^k. It follows that occurrences of a value d_j^k do not interfere with the ones of value $d_j^{k'}$ with $k \neq k'$. Hence, the graph representation does not contain anymore the arcs A_{req}^k that connect value vertices of different levels but rather value vertices are directly connected to the sink. In order to express preferences, we should also introduce positive costs on the arcs $(x_i^{k'}, d_i^k)$ whenever $k < k'$. Thus, the overall graph representation is similar to a particular case of the `cost_gcc` [5]. Figure 6 shows an example of a `nested_gcc` with preferences in which we have 5 variables and 9 values (5 resources with 3 different skill levels and 3 tasks leading to an overall of 9 different values). With such a representation it is also straightforward to constraint the eventual gap in a given assignment between the resource level and the duty level.

Note that this particular graph representation could not be used for the `nested_gcc`. For instance, take in consideration Figure 6: if this would have been a traditional `nested_gcc`, value vertices represent simply tasks, hence d_2^2 denotes the task 2 as well as d_2^1. Suppose furthermore that task 2 requires at

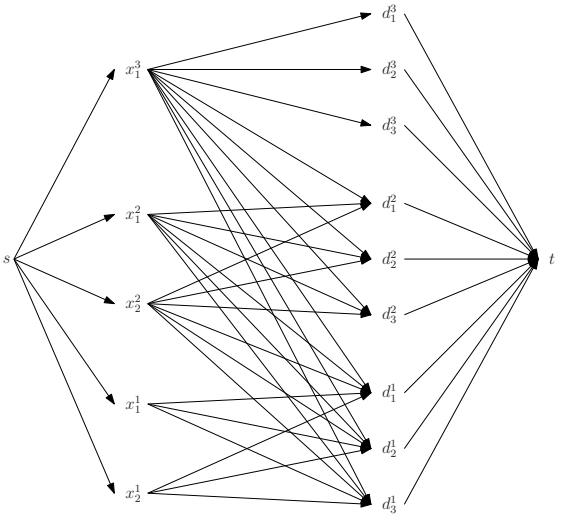


Fig. 6. Graph representation for the nested gcc with preferences

most 1 resource of level 2 or higher. The constraint would be consistent even with assignments $x_1^2 = d_2^1$ and $x_2^2 = d_2^2$ hence leading to contradiction.

7 Conclusions

We proposed generalizations of the `gcc` to address certain resource allocation problems. We showed both theoretically and empirically that such generalizations can outperform an alternate formulation using `gcc`'s. As future work, we plan to do a more extensive empirical analysis in order to better characterize the hardness of the instances. Finally we will consider filtering on occurrence variables.

Acknowledgements

The authors would like to thank the anonymous referees for their helpful comments.

References

1. R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. Network Flows. Prentice Hall, 1993.
2. K.R. Apt. Principles of Constraint Programming. Cambridge Unievristy Press, 2003.
3. K. Elbassioni, I. Katriel, M. Kutz, and M. Mahajan. Simultaneous Matchings. *Proceedings of the Sixteenth International Symposium on Algorithms and Computation (ISAAC 2005)*, Springer LNCS 3827: 106-115.
4. C-G. Quimper, Alejandro López-Ortiz, P. van Beek and Alexander Golynski. Improved Algorithms for the Global Cardinality Constraint. *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP 2004)*, Springer LNCS 3258: 542-556.
5. J-C. Régin. Arc Consistency for Global Cardinality Constraints with Costs. *Proceedings of the Fifth International Conference on Principles and Practice of Constraint Programming (CP 1999)*, Springer LNCS 1713: 390-404.
6. J-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, AAAI Press: 209-215.
7. R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146-160, 1972.
8. Challenge Roadef 2007, http://gilco.inpg.fr/ChallengeROADEF2007/ , 2007-01-30.

A Column Generation Based Destructive Lower Bound for Resource Constrained Project Scheduling Problems*

J.M. van den Akker, G. Diepen, and J.A. Hoogeveen

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80089, 3508 TB Utrecht, The Netherlands
marjan@cs.uu.nl, diepen@cs.uu.nl, slam@cs.uu.nl

Abstract. In this paper we present a destructive lower bound for a number of resource constrained project scheduling (RCPS) problems, which is based on column generation. We first look at the problem with only one resource. We show how to adapt the procedure by Van den Akker et al. [1] for the problem of minimizing maximum lateness on a set of identical, parallel machines such that it can be used to solve these RCPS problems. We then consider a number of variants of the RCPS problem with one or more resources and show how these can be solved by our approach. Because of the close relation between RCPS and the cumulative constraint in constraint programming, our method can be used as an efficient filtering algorithm for the cumulative constraint as well.

1980 Mathematics Subject Classification (Revision 1991): 90B35.

Keywords and Phrases: resource constrained project scheduling, cumulative constraint, linear programming, column generation, generalized precedence constraints.

1 Introduction

In this paper we consider a number of basic problems from project scheduling; we refer to the survey paper by Brucker, Drexl, Möhring, Neumann, and Pesch [6] for an overview of this area. We further refer to Van den Akker, Hoogeveen, and Van de Velde [2], Baptiste, Le Pape, and Nuijten [3], and Bazaraa, Jarvis, and Sherali [4] for an overview of the application of column generation in scheduling, for an overview of the application of constraint programming in scheduling, and for an overview of linear programming in general, respectively.

The basic resource constrained scheduling problem we are looking at is defined as follows. We are given a set of n jobs, which we denote by $J_1, \ldots, J_n$. For each job J_j we are given its processing time p_j, its release date r_j, its deadline $\bar{d}_j$, and its

* Supported by BSIK grant 03018 (BRICKS: Basic Research in Informatics for Creating the Knowledge Society).

P. Van Hentenryck and L. Wolsey (Eds.): CPAIOR 2007, LNCS 4510, pp. 376–390, 2007.

resource consumption pattern, which gives the amount of resource needed during its execution; for the time being, we assume that there is only one kind of resource. For each job J_j we are asked to find a valid starting time S_j and completion time $C_j = S_j + p_j$ such that job J_j does not start before its release date ($S_j \geq r_j$), it is completed by its deadline ($C_j \leq \bar{d}_j$), and such that the total resource consumption of the jobs at any time t does not exceed the amount of resources available at that time. Moreover, between each pair of jobs J_i and J_j, there can be generalized precedence constraints, which define a lower bound and/or upper bound on $S_i - S_j$. In case the upper and lower bound are equal, we say that there is a *no-wait* constraint between J_i and J_j. The goal is to minimize either the makespan $C_{\max}$ or the maximum lateness $L_{\max} = \max_j L_j$, where the lateness L_j of job J_j is defined as the difference between the completion time C_j and the due date d_j, which denotes a target completion time. In fact, our approach can easily be generalized further to deal with any regular minimax function.

The resource constrained project scheduling problem has received attention from both operations research and constraint programming. We only discuss a few contributions. Brucker and Knust ([7], [8]) have applied column generation to a number of resource constrained project scheduling problems in which the goal is to minimize the makespan. Here they first formulate the problem as a decision problem and then use linear programming to check whether it is possible to execute all jobs in a feasible preemptive schedule; here the decision variables refer to the length of a time slice during which a given set of jobs is executed simultaneously. Cesta, Oddi, and Smith [9] have applied constraint programming to the makespan problem. The key here is to determine a schedule that is feasible for all constraints except for the resource consumption. Then resource conflicts are determined and resolved.

Van den Akker, Hoogeveen, and van Kempen [1] have looked at the special case of the above model in which the available amount of resources is constant over time (say m) and each job has a constant resource consumption pattern of one, that is, at any time during its execution, it consumes one unit of resource. This problem is then equivalent to the parallel machine scheduling with m parallel, identical machines. Van den Akker et al. [1] have presented a column generation based method to solve it, which yields a lower bound that turned out to be tight in all their computational experiments. They further gave a method to find a feasible solution with value equal to the lower bound. We will briefly discuss their method in Section 2. In Section 3 we will describe how we can extend their method to solve a number of basic RCPS problems. In Section 4 we consider two other extensions, concerning change-over times and machine maintenance. Finally, we draw some conclusions and present directions for future research in Section 5.

2 Reviewing the Basic Method

Here, we briefly review the column generation approach presented in [1] for the problem of minimizing $L_{\max}$ on a set of m parallel, identical machines. Each

job needs exactly one machine during its processing. Furthermore, there are release dates, deadlines, and generalized precedence constraints. The minimization problem is turned into a feasibility problem by putting an upper bound L on $L_{\max}$; this is equivalent to adding deadlines $\bar{d}_j \leftarrow d_j + L$ $(j = 1, \ldots, n)$. Since a feasible schedule corresponds to a collection of at most m feasible, single-machine schedules containing all n jobs, the decision problem can be reformulated as: *is it possible to partition the jobs in at most m subsets such that for each subset we can find a feasible single-machine schedule*? Finally, the latter decision problem is solved by answering the question: what is the minimum number of feasible single-machine schedules that are needed to accommodate all jobs?

This problem is formulated as an integer linear programming problem as follows. We call a subset of jobs that allow a feasible single-machine schedule with respect to the release dates and deadlines a *machine schedule*. Let S be the set containing all machine schedules. We introduce binary variables x_s $(s = 1, \ldots, |S|)$ that take value 1 if machine schedule s is selected and 0 otherwise. For each machine schedule s we encode whether job J_j is included (then $a_{js} = 1$) or not $(a_{js} = 0)$, and we encode the starting times S_{js} of the jobs with $a_{js} = 1$ $(j = 1, \ldots, n)$. Since two jobs that are connected through a precedence constraint do not have to be executed by the same machine, the generalized precedence constraints are not included in the feasibility of the machine schedules, and we include a constraint in the integer linear programming formulation for each of the generalized precedence constraints. We define A^1 as the arc set containing all pairs (i, j) such there exists a precedence constraint of the form $S_j - S_i \geq q_{ij}$; similarly, we define A^2 and A^3 as the arc sets that contain an arc for each pair (i, j), for which $S_j - S_i \leq q_{ij}$ and $S_j - S_i = q_{ij}$, respectively. Note that the intersection of A^1 and A^2 does not have to be empty. We denote the union of A^1, A^2, and A^3 by the multiset A. This leads to the following integer linear programming formulation

$$\min \quad \sum_{s \in S} x_s$$

subject to

$$\sum_{s \in S} a_{js} x_s = 1, \text{ for each } j = 1, \ldots, n, \tag{1}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{s \in S} S_{is} x_s \geq q_{ij} \text{ for each } (i, j) \in A^1; \tag{2}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{s \in S} S_{is} x_s \leq q_{ij} \text{ for each } (i, j) \in A^2; \tag{3}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{s \in S} S_{is} x_s = q_{ij} \text{ for each } (i, j) \in A^3; \tag{4}$$

$$x_s \in \{0, 1\}, \text{ for each } s \in S.$$

We relax the integrality constraints to $x_s \geq 0$; the upper bound $x_s \leq 1$ follows from the other constraints. The LP-relaxation is solved by applying column generation. To start, we give each job its own machine. Given the outcome of the current LP, we find dual multipliers λ_j for Constraints 1 and δ_{ij} for the

Constraints 2-4 in which jobs J_i and J_j are involved. The reduced cost of machine schedule s is then equal to

$$c'_s = 1 - \sum_{j=1}^{n} a_{js}\lambda_j - \sum_{j=1}^{n} \left[\sum_{h \in Prec_j} \delta_{hj} S_{js} - \sum_{k \in Suc_j} \delta_{jk} S_{js} \right],$$

where $Prec_j$ and Suc_j are defined as the sets containing all predecessors and successors of job J_j in A, respectively. The pricing problem is then to find a machine schedule with minimum reduced cost. Since solving the LP-relaxation by column generation only renders us a lower bound when the column generation procedure has finished, we compute an intermediate lower bound as

$$\sum_{s \in S} x_s \geq \left[\sum_{j=1}^{n} \lambda_j + \sum_{(j,k) \in A} \delta_{jk} q_{jk} \right] / (1 - c^*),$$

where c^* denotes the outcome value of the pricing problem.

Since the pricing problem is $\mathcal{NP}$-hard to solve, we do not solve it to optimality in each iteration. We apply a two-phase Simulated Annealing procedure to find a good solution. In the first phase, we decide which jobs are included in the machine schedule and in which order. In the second step, we find the optimal starting times of the included jobs. After that, we change the choices made in phase 1, etc. We mostly use the local search procedure to find good solutions to the pricing problem, but after 50 iterations, or when we cannot find any improving column, we turn to a time-indexed linear programming formulation of the pricing problem. Here we use binary variables x_{jt} to indicate whether job J_j starts at time t or not; the corresponding cost coefficients c_{jt} are easily determined. The ILP-formulation (ignoring the constant) then becomes

$$\min \sum_{j=1}^{n} \sum_{t=r_j}^{\bar{d}_j - p_j} c_{jt} x_{jt}$$

subject to

$$\sum_{t=r_j}^{\bar{d}_j - p_j} x_{jt} \leq 1 \quad \forall j = 1, \ldots, n; \tag{5}$$

$$\sum_{j=1}^{n} \sum_{s=t-p_j+1}^{t} x_{js} \leq 1 \quad \forall t = 0, \ldots, T-1; \tag{6}$$

$$x_{jt} \in \{0,1\} \quad \forall j = 1, \ldots, n; \forall t = r_j, \ldots, \bar{d}_j - p_j.$$

Here T denotes the latest point in time at which at least two jobs can be executed. Constraint 5 decrees that each job can be chosen at most once, and Constraint 6 states that at most one job should be executed at any time.

Since we need to find out whether there exists a solution using at most m machines, we stop as soon as the outcome of the LP-relaxation has hit m. If the

outcome of the current LP is bigger than m, and we cannot find an improving column, then we can compute the outcome value of the pricing problem that we need such that the intermediate lower bound equals m. We can then ask the ILP-solver whether there exists a solution to the pricing problem with that value or less. If it does not exist, then we have proven that m is not achievable, and we are done; if we can find it, then this is an improving column that we add, etc. In this way, we do not have to solve the time-indexed formulation to optimality.

Finally, when we have found the smallest upper bound L on $L_{\max}$ that cannot be proven impossible, then we try to construct a feasible schedule with $L_{\max}$ equal to L by formulating the problem as a time-indexed ILP. Solving this ILP from scratch only works for small instances. But when we insert our knowledge of the lower bound by adding the constraint $LMAX = L$, then our ILP-solver CPLEX finds a feasible solution rather quickly, if it exists. So far (and we have run a lot of experiments), we have not found an instance in which the optimum solution is not equal to the lower bound.

In the remainder of this section, we discuss the computational experiments by Van den Akker et al.[1]. Note that in these experiments, we did not include any no-wait precedence constraints. In our experiments we compared our hybrid algorithm, i.e. column generation and then for the identified lower bound L solving the time-indexed ILP with $LMAX = L$, to the approach of letting CPLEX solve the time-indexed ILP formulation without knowing the value of the lower bound; from now on, we will refer to this as the *ignorant ILP*. We have applied both algorithms on 6 scenarios; for each scenario we ran ten test instances. The scenarios are described in Table 1. The algorithms were encoded

Table 1. Test scenarios

Number	p_j	r_j	d_j	n	m	# prec
0	U[1,20]	U[0,60]	U[50,80]	40	4	20
1	U[1,20]	U[0,40]	U[30,60]	70	5	35
2	U[1,20]	U[0,40]	U[60,80]	100	9	40
3	U[1,20]	U[0,60]	U[80,110]	180	10	60
4	U[1,20]	U[0,60]	U[40,80]	60	5	30
5	U[1,20]	U[0,60]	U[50,80]	30	3	15

in Java and the experiments were run on a Pentium 4, 3 Ghz PC with 1 GB memory. For each instance we let each algorithm run for at most 30 minutes.

Our results clearly showed that our hybrid algorithm outperformed the method of letting CPLEX solve the ignorant ILP by far. For all instances we managed to solve, the derived lower bound was equal to the optimal value. There are some instances for which we could not check whether optimum and lower bound coincided, for we could not solve them within 30 minutes. This may be due to a gap between the lower bound and the optimum. However, we were never able to show that the lower bound differed from the optimum for any instance. Altogether we may draw the conclusion that our lower bound is extremely strong.

If we compare solving the ignorant ILP with the second part of the hybrid algorithm, then we see that specifying the optimum makes a lot of difference. Most likely the preprocessing steps performed by CPLEX play an important role in this. Therefore, we may expect the technique of constraint satisfaction to work very well to find a solution of value L if such a solution exists.

3 The RCPS Problem

3.1 One Resource

Unit resource consumption
We first look at the case that the resource consumption pattern is constantly equal to 1 for each job J_j, but the available amount of nonrenewable resources is not constant over time. We capture this situation in the general framework of [1] by issuing dummy jobs, which 'eat up' the missing resources. This is achieved in the following way. We define the number of machines m to be equal to the maximum amount of resource available at any time. Now we determine the amount of resource that is missing over time with respect to m; this will yield a figure with a number of piles of blocks on top of each other, where the higher you come, the smaller the block is. For each block, we introduce a dummy job with the following characteristics: it has processing time equal to the length of the block; release date equal to the left time point of the block; and deadline equal to the right time point of the block. In the example of Figure 1, we have a pile with three blocks, which lead to the three jobs called D_1, D_2, D_3. The release dates and deadlines of these jobs are r_i and $\bar{d}_i$; the processing times are equal to $\bar{d}_i - r_i$ $(i = 1, \ldots, 3)$. We further assume that each dummy job has a due date that is unrestrictively large to prevent any interference with the $L_{\max}$ value. Note that the choice of dummy jobs is not unique. We can, for example, mingle the dummy jobs D_1 and D_2 to obtain D_1' and D_2' by swapping the deadlines and adjusting the processing times: a solution with these two dummy jobs can be translated into a solution with the two original dummy jobs by applying a 'cross-over' operation of the two involved machine schedules at time $\bar{d}_2$. Similarly, dummy jobs can be split. Anyway, it is easily seen that each feasible solution for this instance that uses no more than m machines corresponds to a feasible solution for the RCPS with equal objective value.

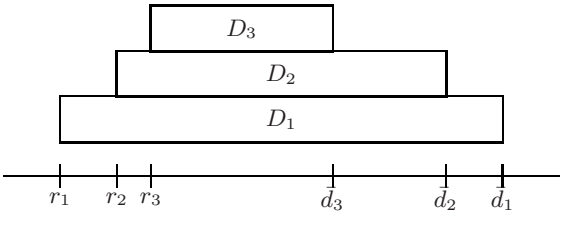


Fig. 1. Example of the dummy jobs

Arbitrary integral resource consumption
We now assume that the resource consumption pattern is constant for each job, but it can be any arbitrary integral value greater than or equal to 1. Suppose that J_j is some job that needs a constant amount of $k \geq 2$ units of resource during its execution. We capture this situation in the general framework by replacing job J_j by job J'_j and $k-1$ additional dummy jobs. Here J'_j is identical to J_j, except for its resource consumption, which we put equal to one. Furthermore, each dummy job has processing time equal to p_j, but it has no release date and deadline, and it is independent of all other jobs, except for J'_j: we force that all these dummy jobs and J'_j are started at the same time by means of a no-wait constraint. It is easily seen that solving the resulting instance with unit resource consumption is equivalent to solving the original instance.

If the resource consumption of job J_j is not constant over time, but can attains arbitrary integral values, then we replace J_j by a set of new jobs with a constant resource consumption pattern equal to 1, and we glue these together by no-wait constraints, such that their joint resource consumption pattern is equivalent to that of the original job J_j.

We can now use the approach of [1] to find the lower bound. Furthermore, we can use the time-indexed formulation of [1] to look for a schedule with value equal to the lower bound. Since the dummy jobs that replace an original job J_j are glued together by no-wait constraints, and since the time-indexed formulation uses variables x_{jt} indicating whether job J_j starts at time t, we can restrict ourselves to the original jobs (with their varying resource consumption patterns) in the time-indexed formulation. Obviously, we must then adjust Constraints 6 to deal with the consumption patterns.

Note the close connection between the above RCPS problem and the cumulative constraint (see the on line Global Constraint Catalog by Beldiceanu and Demassey [5]. The cumulative constraint decrees that we should find for a given set of jobs starting times, which obey the release dates and deadlines, such that the total resource consumption should never exceed the available amount of resource. To filter this constraint, we must check whether a feasible schedule exists for the above resource constraint project scheduling problem without initial precedence constraints. Note that fixing the start time of some job can be easily included in the model by adjusting the available amount resource.

3.2 Multiple Resources

We assume in this subsection that there are only two resources involved, but each model can be easily generalized to deal with any number of resources. We first transform it to an instance in which each job consumes only one resource during its execution: this is easy to achieve by replacing a job that needs both resources with two copies that only need one of the individual resources but are identical otherwise. These copies are then tied together by no-wait constraints such that they start at the same time. Next, we use the transformations described above to achieve that each job uses exactly one amount of resource (either resource 1 or 2) at any time during its execution. We now have transformed the problem into

a parallel machine scheduling problem in which there are two different sets of identical machines; we assume that there are m_1 (m_2) machines corresponding to resource 1 (2).

We apply the same solution strategy as Van den Akker et al. [1] We divide the jobs and the machines into two groups, where jobs are only assigned to machines of the right group, which is easily incorporated in the pricing problem. We again minimize the total number of machines that is used, but we add the constraint that we use at least m_1 (m_2) machines of group 1 (2): if we then find a solution using no more than $m_1 + m_2$ machines, then we know that we do not use too much of resources 1 and 2 separately. Note that we could have added constraints decreeing that we use no more than m_1 (m_2) machines of group 1 (2) instead, but then we run into problems when we look for a feasible solution of the LP-relaxation to start with. Finally, we add some 'empty' columns, such that these two constraints can always be met.

As an illustration, we work things out for the case in which there are two resources, and each job J_j has a constant resource consumption pattern, requiring either 0 or 1 unit of resource 1 and 2. We assume for the ease of exposition that initially there are no precedence constraints. We denote the set of jobs requiring resource 1 only by R_1; similarly, we use R_2 to denote the jobs requiring resource 2 only. We denote the set of jobs that need both resources by $R_{1,2}$; these jobs will be split into two operations. These two operations need only one resource and are connected by a no-wait constraint. We use S and V to denote the set of machine schedules for resources 1 and 2, and we use x_s and y_v as binary decision variables. Moreover, we use a_{js} and b_{jv} to indicate whether job J_j is included in machine schedules s and v for resource 1 and 2, respectively. This leads to the following ILP-formulation, where with a little abuse of notation, a job $J_j \in R_{1,2}$ in fact consists of its two operations.

$$\min \quad \sum_{s\in S} x_s + \sum_{v\in V} y_v$$

subject to

$$\sum_{s\in S} a_{js}x_s = 1, \text{ for each } j \in R_1 \cup R_{12} \tag{7}$$

$$\sum_{v\in V} b_{jv}y_v = 1, \text{ for each } j \in R_2 \cup R_{12} \tag{8}$$

$$\sum_{s\in S} S_{js}x_s - \sum_{v\in V} S_{jv}y_v = 0, \text{ for each } j \in R_{1,2} \tag{9}$$

$$\sum_{s\in S} x_s \geq m_1 \tag{10}$$

$$\sum_{v\in V} y_v \geq m_2 \tag{11}$$

$$x_s, y_v \in \{0,1\}, \text{ for each } s \in S \text{ and } v \in V.$$

When we solve the LP-relaxation by column generation, we find that the reduced cost of a schedule $s \in S$ is equal to

$$c'_s = 1 - \lambda_0 - \sum_{j\in R_1\cup R_{12}} a_{js}\lambda_j - \sum_{j\in R_{1,2}} \delta_j S_{js};$$

here λ_0 is the dual multiplier corresponding to Constraint 10, λ_j $(j \in R_1)$ are the dual multipliers corresponding to the Constraints 7, and δ_j $(j \in R_{1,2})$ are the dual multipliers corresponding to the Constraints 10. The reduced cost of machine schedule $v \in V$ is computed in an equivalent way. It is readily verified that the pricing problem is similar to the one of [1], which implies that the local search procedure and time-indexed formulation to solve it can still be applied. Furthermore, we can compute an intermediate lower bound as follows. Let c_1^* denote the optimal value of the pricing problem for resource 1. If we fill in $c'_s \geq c_1^*$ in the formula of the reduced cost, then we find that

$$1 \geq c_1^* + \lambda_0 + \sum_{j \in R_1 \cup R_{12}} a_{js}\lambda_j + \sum_{j \in R_{1,2}} \delta_j S_{js}.$$

Hence,

$$\sum_{s \in S} x_s \geq \sum_{s \in S} \left[c_1^* + \lambda_0 + \sum_{j \in R_1 \cup R_{12}} a_{js}\lambda_j + \sum_{j \in R_{1,2}} \delta_j S_{js} \right] x_s =$$
$$(c_1^* + \lambda_0) \sum_{s \in S} x_s + \sum_{j \in R_1 \cup R_{12}} \lambda_j \sum_{s \in S} [a_{js} x_s] + \sum_{j \in R_{1,2}} \delta_j \sum_{s \in S} S_{js} x_s =$$
$$(c_1^* + \lambda_0) \sum_{s \in S} x_s + \sum_{j \in R_1 \cup R_{12}} \lambda_j + \sum_{j \in R_{1,2}} \delta_j \sum_{s \in S} S_{js} x_s.$$

Similarly, we find that

$$\sum_{v \in V} y_v \geq (c_2^* + \mu_0) \sum_{v \in V} y_v + \sum_{j \in R_2 \cup R_{12}} \mu_j - \sum_{j \in R_{1,2}} \delta_j \sum_{v \in V} S_{js} y_v;$$

here μ_0 is the dual multiplier corresponding to Constraint 11, μ_j $(j \in R_1 \cup R_{12})$ is the dual multiplier corresponding to Constraint 8, and c_2^* is the outcome value of the pricing problem for resource 2. If we add these two inequalities up, then the terms containing S_{js} cancel out, because of Constraints 9. Rearranging the terms, we find that

$$(1 - c_1^* - \lambda_0) \sum_{s \in S} x_s + (1 - c_2^* - \mu_0) \sum_{v \in V} y_v \geq \sum_{j \in R_1 \cup R_{12}} \lambda_j + \sum_{j \in R_2 \cup R_{12}} \mu_j.$$

If $1 - c_1^* - \lambda_0 = 1 - c_2^* - \mu_0$, then we can divide by this term and find a lower, provided that $1 - c_1^* - \lambda_0 > 0$, which issue we discuss later. Suppose that $1 - c_1^* - \lambda_0 > 1 - c_2^* - \mu_0$; the other case can be dealt with in the same way. Then we add to this inequality $(c_2^* - c_1^* + \mu_0 - \lambda_0)$ times inequality 11, and we find the intermediate lower bound

$$\sum_{s \in S} x_s + \sum_{v \in V} y_v \geq \frac{((c_2^* - c_1^* + \mu_0 - \lambda_0)m_2 + \sum_{j \in R_1 \cup R_{12}} \lambda_j + \sum_{j \in R_2 \cup R_{12}} \mu_j}{(1 - c_1^* - \lambda_0)}.$$

What is left to show is that $(1 - c_1^* - \lambda_0) > 0$. We know that $c_1^* \leq 0$, since any column that is used in the current LP solution has zero reduced cost. Moreover,

if both λ_0 and μ_0 are positive, then both constraints are binding, which implies that we have found a solution with value $m_1 + m_2$, which means that we can stop. Hence, at least one of λ_0 and μ_0 is zero, which implies that the maximum of $1 - c_1^* - \lambda_0$ and $1 - c_2^* - \mu_0$ is positive.

Finally, we look at the problem of finding a feasible solution with this value. It is easily verified that the time-indexed formulation of [1] to find a feasible solution can be used, but we must split the m machines into two sets representing the m_1 and m_2 units of resources 1 and 2, respectively.

Machine scheduling with operators

A special case of the above is the situation in which each job needs an operator to start it up, which takes 1 time unit per job. Hence, we should not start more jobs at any moment than there are operators available. We can model the operators as a second resource, but alternatively we can add the starting times to the machine schedules and force the restriction on the number of operators by adding constraints. Here, we work out the second option, which has the additional advantage that we can model a varying number of available operators. We again assume without loss of generality that there are no additional precedence constraints. We use o_{st} to indicate whether a job starts at time t in machine schedule s; we use Op_t to denote the number of operators available at time t. We then arrive at the ILP-formulation:

$$\min \quad \sum_{s \in S} x_s$$

subject to

$$\begin{aligned} \sum_{s \in S} a_{js} x_s &= 1, \text{ for each } j = 1, \ldots, n \\ \sum_{s \in S} o_{st} x_s &\leq Op_t, \text{ for all } t = 0, \ldots, T-1 \\ x_s &\in \{0, 1\}, \text{ for each } s \in S, \end{aligned} \tag{12}$$

where T denotes a given time horizon. The reduced cost of a machine schedule s is then equal to

$$c'_s = 1 - \sum_{j=1}^{n} a_{js} \lambda_j - \sum_{t=0}^{T} o_{st} \pi_t,$$

where π_t denotes the dual variable corresponding to Constraints 12. The corresponding pricing problem can be minimized using the local search procedure and the time-indexed formulation of [1]. Furthermore, since $\pi_t \leq 0$ $(t = 0, \ldots, T)$, it is readily determined that

$$\left\lceil \sum_{j=1}^{n} \lambda_j + \sum_{t=0}^{T} \pi_t Op_t \right\rceil / (1 - c^*)$$

is an intermediate lower bound on the outcome of the LP-relaxation.

Finally, we can use the time-indexed formulation of [1] to find a solution with value equal to the lower bound, but we have to add constraints to ensure that the required number of operators is no more than the available number at any time

$$\sum_{j=1}^{n} x_{jt} \leq Op_t, \text{ for all } t = 0, \ldots, T-1.$$

3.3 Computational Experiments

We tested our hybrid algorithm for the case with one type of resource, unit resource consumption, and variable resource availability over time. We consider the instances from Table 1. Besides the basic scenario with full resource availability, we consider two scenarios for each instance. In the first scenario, there is one pile of dummy jobs (reflecting the resource unavailability) where the pile is located around half of the estimated makespan of the schedule. In the second scenario there are two shorter piles around one third and two third of the estimated makespan, respectively. In both scenarios the maximum amount of unavailable resources is about $\lceil \frac{m}{2} \rceil$. The first scenario is denoted by Hi-T1 and the second by Hi-T2. We report the number of times out of 10 that an optimum was found ('# success'), and we report the average and maximum amount of time in seconds needed for the successful runs ('Avg t' and 'Max t'). For the hybrid algorithm, we denote by ('#LB=OPT') the number of times that we could prove that the lower bound equalled the optimum. Next, we report the average and maximum time needed to find the lower bound for the successful runs ('Avg t LB' and 'Max t LB'). By ('Avg #ILP' and 'Max #ILP'), we denote the number of times that we solved the ILP formulation of the pricing problem; this was conducted after each series of 50 runs of the local search algorithm, since we wanted to find out whether the intermediate lower bound could decide the problem already, and whenever the local search algorithm could not find an improving column. Finally, we report on the increase of the lateness because of resource unavailability (Avg incL and Max incL, both in percentages). Again, the maximal running time is 30 minutes. The results are given in Table 2. Our computational results indicate that the resource unavailability increases the running time of the algorithm but that in most cases the algorithm is still able to solve the problem within 30 minutes. For the largest instances (of type 3), we were able to compute the lower bound but could not complete the ILP within 30 minutes. In most cases the scenario with one pile is more difficult than the one with two piles. Finally, most cases were solved and moreover, for all these cases the lower bound equals the optimum, which emphasizes the strength of our lower bound.

We further have tested the suitability of using the destructive lower bounding technique for filtering the cumulative constraint. Hereto, we conducted some experiments to find out the time it take to test whether a schedule with $L_{\max} \leq L$ can exist for a specific value of L. Given the optimum L^* of the instance, we checked for the first two instances of Table 2 whether a schedule can exist with

Table 2. Results of the hybrid algorithm with resource unavailability

	# success	Avg t	Max t	#LB =OPT	Avg t LB	Max t LB	Avg #ILP	Max #ILP	Avg incL	Max incL
H0	10	30	62	10	27	60	8	43		
H0-T1	9	41	81	9	35	72	20	94	42	69
H0-T2	10	39	71	10	32	62	19	79	27	54
H1	10	191	336	10	108	156	16	45		
H1-T1	9	166	207	9	83	119	13	38	37	45
H1-T2	10	437	1238	10	190	926	15	30	42	52
H2	9	183	302	9	117	217	16	68		
H2-T1	9	297	497	9	137	383	14	20	87	107
H2-T2	10	340	582	10	112	158	11	16	125	155
H3	9	1033	1579	9	534	640	45	78		
H3-T1	6	1393	1736	6	578	730	55	75	29	33
H3-T2	9	1288	1473	9	642	927	52	88	35	40
H4	10	54	173	10	42	153	16	92		
H4-T1	9	76	121	9	56	103	29	91	13	28
H4-T2	9	84	165	9	60	139	34	97	18	37
H5	9	26	77	9	24	76	13	76		
H5-T1	9	61	139	9	47	135	17	46	112	179
H5-T2	10	77	214	10	59	205	10	31	144	258

$L_{\max} \leq L$, where $L = L^* - 8, L^* - 4, L^* - 2, L^* - 1, L^*, L^* + 1, L^* + 2, L^* + 4, L^* + 8$. Note that we computed each test from scratch. We further have added the time needed to establish the lower bound of L^* in the column with header $LB - time$. The average running times (in seconds) are displayed in Table 3. The running times do not show a clear picture. In many cases, showing infeasibility becomes more difficult when approaching L^*, but for the instances of type 2 the toughest is showing infeasibility of $L^* - 2$. It clearly becomes easier to conduct the test for values L that are greater than or equal to L^* for bigger values of L.

4 Other Extensions

4.1 Set-Up Times and Change-Over Times

So far, we have assumed that as soon as a machine has finished a job, it can start the next one. In many applications, however, there can be a mandatory delay, which is called a *set-up time* or a *change-over time*. A set-up time just depends on the job that is to be started; the change-over time depends on both the job that is to be started and the job that has just been completed. Here we assume that the change-over times obey the triangle inequality.

We first deal with the set-up times, since this is fundamentally easier than the case with change-over times. The basic idea is to add the set-up time to the processing time; we then consider the first part of processing the job as setting it up. We must then update the release date by subtracting the set-up time

Table 3. Running times for testing feasibility

#	type	L*-8	L*-4	L*-2	L*-1	L*	L*+1	L*+2	L*+4	L*+8	LB-time
0	T1	5.7	7.6	7.3	10.5	3.7	2.5	2	1.6	1.3	27.4
0	T2	5.8	4.7	6.7	7.4	5.7	2.7	2.4	1.6	1.3	28.5
1	T1	15.5	15.2	17.1	19.8	8.5	5.8	4.8	3.7	3.1	78.3
1	T2	13.8	13	14	17.8	14	7.8	5.6	3.9	3.2	127.8
2	T1	22.6	20.4	30.8	25.9	13.4	8.9	7.4	6.3	3.4	106.3
2	T2	25.8	25	43.6	26.9	10.9	9.2	8.7	5.8	4.7	114.2
3	T1	123	407.1	171.1	229.3	60.2	35.8	24.9	19.9	11.6	558.1
3	T2	85.4	125.9	144.6	137.6	161	36.9	27.5	17.6	11.8	456.6
4	T1	.1	0	0	0	5.2	3.5	2.5	1.9	1.5	11.4
4	T2	0.6	0	0	12.9	15.5	3.4	2.8	2.4	1.7	45.1
5	T1	6	6.3	6.7	15.2	7.6	2.8	2.1	1.8	1.3	42.5
5	T2	6.2	6.7	8	8.4	2.9	2.7	2.2	1.4	1.2	36.5

from it, which might lead to a negative release date. We may further have to update the right-hand-sides of the generalized precedence constraints, but this is simply a matter of administration. An optimal solution for the problem with set-up times is then readily obtained from the optimal solution for the adjusted instance without set-up times.

Sequence-dependent change-over times are much harder. We incorporate this type of constraint in the column generation: we look for single machine schedules that obey the release dates, deadlines, and the change-over times. This implies that the ILP formulation remains the same; we only must add another constraint to the pricing problem. It is easily dealt with in the local search procedure that Van den Akker et al. use to solve the pricing problem approximately, but it cannot be incorporated in the time-indexed formulation to solve the pricing problem. If we want to solve the pricing problem then, we might use branch-and-bound. Moreover, we cannot use the time-indexed formulation of [1] to find an optimal solution. Very recently, Pereira Lopes and Valério de Carvalho [10] have presented a branch-and-price algorithm for this problem, but with an additive objective function.

4.2 Machine Unavailability and Planned Maintenance

Machine unavailabilities are similar to varying resource availabilities, but they are more restrictive, since we put a label on a machine with its unavailability pattern instead of aggregating the capacities of all machines. One way to tackle this problem is to label the machines and determine for each one a separate set of machine schedules, from which we must select one. An alternative and quicker way is to add dummy jobs to the instance which correspond to unavailabilities. In a correct solution, we will have for each unavailability pattern that a feasible machine schedule will be selected that contains the dummy jobs corresponding to this unavailability pattern, which gives us a schedule for the corresponding machine. In case of a planned maintenance, we know that the machine is being

repaired for a given time, but we do not know when this time period starts: we then give the dummy job a release date and deadline corresponding to the earliest start time and the latest completion time of the repair. The only difficulty left is to ensure that a given set of dummy jobs corresponding to the unavailabilities and repairs of a given machine all end up in the same, selected machine schedule. Just like in the previous subsection, we put these constraints in the pricing problem. These additional constraints to a machine schedule are easily being dealt with in the local search procedure. When we want to solve the pricing problem to optimality, we can use the time-indexed formulation, but we must add a constraint for each pair of jobs that must be executed on the same machine or on different machines: if J_i and J_j are to be executed on the same machine, then we add the constraint

$$\sum_{t=r_i}^{\bar{d}_i - p_i} x_{it} = \sum_{t=r_j}^{\bar{d}_j - p_j} x_{jt};$$

if J_i and J_j must go on different machines, then we require

$$\sum_{t=r_i}^{\bar{d}_i - p_i} x_{it} + \sum_{t=r_j}^{\bar{d}_j - p_j} x_{jt} \leq 1.$$

Note that we do not have to solve a pricing problem for each machine separately. Since each job has to be executed, there will be one machine 'executing' the set of dummy jobs that we introduced to mimic the unavailability pattern of this machine. Unfortunately, after having determined the lower bound, we cannot straightaway use the time-indexed formulation of [1] to look for a solution with equal value, since we must force the set of dummy jobs representing the machine unavailability pattern on one machine that does not execute any other dummy job. We can use a similar formulation in which we distinguish between the machines by using variables x_{ijt} indicating that job J_j starts at time t on machine i, but this will blow up the model tremendously, since we cannot aggregate the machines and require that at most m are used then anymore.

5 Conclusions and Future Research

We have described how the framework by Van den Akker et al. [1] can be used to solve a number of basic resource project scheduling problems. We further have shown how to incorporate change-over times and machine maintenance. Except for the case with change-over times, we can use the same tool kit as in [1] to compute the lower bound. This lower bound always coincided with the optimum in the computational experiments conducted in [1], and we found the same phenomenon in our experiments for the case of the strongly related problem with a varying amount of resources available. We are working on more elaborate computational experiments including other cases. When it comes to finding a solution with value equal to the lower bound, we can in many cases

use the time-indexed formulation of [1] in which we specify the wanted optimum beforehand. Van den Akker et al. conjectured that this is presumably due to the preprocessing step within CPLEX, which suggest that the technique of constraint programming should be able to find such a solution more quickly, or show that it does not exist. Constraint programming seems to be the most eminent candidate to look for a solution with value equal to the lower bound for the problems with machine unavailabilities and change-over times. This is one of the directions that we work on.

References

1. J.M. van den Akker, J.A. Hoogeveen, and J.W. van Kempen (2006). Parallel machine scheduling through column generation: minimax objective functions (extended abstract). Y. Azar and T. Erlebach (Eds.) *ESA 2006*. LNCS 4168, Springer, 648–659.
2. J.M. van den Akker, J.A. Hoogeveen, and S.L. Van de Velde (2005). Applying column generation to machine scheduling. G. Desaulniers, J. Desrosiers, and M.M. Solomon (eds.). *Column Generation*, Springer, 303–330.
3. P. Baptiste, C. Le Pape, and W. Nuijten (2001). *Constraint-based scheduling*: Applying constraint programming to scheduling problems. Kluwer Academic Publishers, Dordrecht, The Netherlands.
4. M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali (1990). *Linear Programming and Network Flows*, Wiley, New York.
5. N. Beldiceanu and S. Demassey (2007). *Global Constraint Catalog* www.emn.fr/x-info/sdemasse/gccat/index.html
6. P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch (1999). resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research 112*, 3–41.
7. P. Brucker and S. Knust (2000). A linear programming and constraint propagation-based lower bound for the RCPSP. *European Journal of Operational Research 127*, 355–362.
8. P. Brucker and S. Knust (2003). Lower bounds for resource-constrained project scheduling problems. *European Journal of Operational Research 149*, 302–313.
9. A. Cesta, A. Oddi, and S.F. Smith (2002). A constraint-based method for project scheduling with time windows. *Journal of Heuristics 8*, 109–136.
10. M.J. Pereira Lopes and J.M. Valério de Carvalho (2007). A branch-and-price algorithm for scheduling parallel machines with sequence dependent setup times. *European Journal of Operational Research 176*, 1508–1527.

Author Index