The 19th International Conference on Principles and Practice of Constraint Programming

Proceedings

ModRef 2013: The Twelfth International Workshop on Constraint Modelling and Reformulation

Monday September 16th, 2013

Uppsala, Sweden

George Katsirelos and Claude-Guy Quimper

Preface

The Twelfth International Workshop on Constraint Modelling and Reformulation was held in Uppsala (Sweden) on September 16th, 2013 jointly with the 19th International Conference on Principles and Practice of Constraint Programming. The objective of this workshop is to promote discussion of novel ideas related to modeling with constraint programming. The program committee selected 7 papers from the submissions to appear in these proceedings and to be presented at the workshop. We would like to thank the program committee for reviewing the papers and sharing their comments with the authors. We would also like Laurent Michel for giving an invited talk entitled "What to Expect from Modeling Facilities".

September 2013

George Katsirelos
Claude-Guy Quimper

Organizers

George Katsirelos
Claude-Guy Quimper

Program committee

Jessica Davies, University of Toronto
Simon Degivry, INRA, Toulouse
Alan Frisch, University of York
Emmanuel Hebrard LAAS, CNRS
Christopher Jefferson, University of St. Andrews, UK
George Katsirelos, INRA, Toulouse, France
Zeynep Kiziltan, University of Bologna
Jimmy Lee, The Chinese University of Hong Kong
Michele Lombardi, DEIS, University of Bologna
Christopher Mears, Monash University, Australia
Karen Petrie, University of Dundee
Gilles Pesant, Ecole Polytechnique de Montréal
Claude-Guy Quimper, Université Laval, Québec, Canada
Jean-Charles Régin, University of Nice-Sophia Antipolis / I3S / CNRS
Ashish Sabharwal, IBM Research
Horst Samulowitz, IBM Research
Meinolf Sellmann, IBM Research
Thomas Schiex , INRA, Toulouse

Table of Content

On SAT-Encodings of the At-Most-One Constraint

Steffen Hölldobler and Van-Hau Nguyen

International Center for Computational Logic
Technische Universität Dresden, 01062 Dresden, Germany
{sh,hau}@iccl.tu-dresden.de

Abstract. One of the most widely used constraint during the process of translating a practical problem into a propositional satisfiability instance is the at-most-one constraint. This paper proposes a new encoding for the at-most-one constraint, the so-called *bimander* encoding which can be easily extended to encode cardinality constraints. Experimental results reveal that the new encoding is competitive. We prove that the *bimander* encoding allows unit propagation to achieve arc consistency. Furthermore, we show that a special case of the *bimander* encoding outperforms the widely used *binary* encoding in all our experiments.

1 Introduction

Solving propositional satisfiability (SAT) problems is one of the most successful automated reasoning methods in the last decade in Computer Science by solving a wide range of both industrial and academic problems [27,13]. SAT solving comprises two essential phases: encoding a certain problem into a SAT instance and, thereafter, finding solutions by advanced SAT solvers. Notwithstanding the steadily increasing diffusion and availability of SAT solvers, understanding of SAT encodings is still limited and challenging [25].

An increasing number of real-world applications in computer science can be expressed as constraint satisfaction problems (CSPs) [36]. While CSPs can be solved directly using appropriate solvers, the generality and success of SAT solvers in recent years has let to a fruitful competition between the CSP and the SAT community. To utilize state-of-the-art SAT solvers, CSPs need to be encoded as SAT instances (see e.g. [41,5,32,40,39,35,3]). Such encodings should not only be efficiently generated, but should also be efficiently solved by SAT solvers. Currently, mapping a CSP into a SAT instance is regarded more of an art than a science [41,21,31,30,25,35], and detailed studies of different encodings are needed in order to better understand these mappings.

Generally, different encodings of CSPs into SAT instances yield different sizes and different run time behaviour of the used SAT solver. There does not seem to be general knowledge why a particular encoding performs better than others. In this study, we will compare different encodings with respect to the following features:

- the number of auxiliary variables required,
- the number of clauses required,
- the number of variables per clause,
- the strength of the encoding in terms of performance of unit propagation,
- the runtime of a SAT solver on benchmark problems.

Although many encodings have been proposed [41,21,5,32,40,39,35], the most straightforward way of mapping a CSP into a SAT instance is the *direct encoding* (see [41]). The *direct encoding* requires the translation of global constraints like the at-least-one (ALO) and at-most-one (AMO) constraints requiring that a CSP variable has at least one and at most one value assigned to it, respectively. Whereas the ALO constraint can be easily encoded by a single clause, the encoding of the AMO constraint is more complicated and has been intensively studied [26,17,14,34,33]. This is due to the fact that many different applications such as the discrete tomography problem [9], partial Max-SAT [6,7], or cardinality constraints [17] contain the AMO constraint. Furthermore, better encodings of the AMO constraint in conjunctive normal form (CNF) could also help Max-SAT solving, because some state-of-the-art SAT-based Max-SAT solvers encode AMO constraints in CNF [4].

In the direct encoding, if a propositional variable is used to represent the binding of a CSP variable to a particular value, then the AMO constraint requires that at most one of n propositional variables is bound to *TRUE*. Herein, this will be denoted by $\leq_1(X_1, ..., X_n)$, where X_i, $1 \leq i \leq n$, are propositional variables.

Inspired by many interesting and recent results [26,17,14], especially when Prestwich used the *binary* encoding [18,19] to successfully solve many large instances with a standard SAT solver [34,33], we will survey several widely used encodings of the AMO constraint. In particular, we will point out the similarity of the *relaxed ladder* [33] encoding, the *sequential* [38] encoding, and several others. Finally, we will introduce a new encoding, the *bimander* encoding. The new encoding requires $\lceil log_2 m \rceil$[1] auxiliary variables and $\frac{n^2}{2m} + n\lceil log_2 m \rceil - \frac{n}{2}$ binary clauses, where m is the number of disjoint subsets used by dividing the given set $\{X_1, ..., X_n\}$ of propositional variables.

Additionally, the *bimander* encoding can be easily extended to cardinality constraints, denoted by $\leq_k(X_1, ..., X_n)$, which expresses that less than k of n propositional variables $X_i, 1 \leq i \leq n$, can be simultaneously assigned to *TRUE*. To the best of our knowledge, our encoding is the one that requires the least number of auxiliary variables among known encodings except for the *pairwise* encoding, which needs no auxiliary variables at all. With respect to scalability, the *bimander* encoding can be adjusted by changing the parameter m. For example, by setting the parameter m to specific values, the *binary* and *pairwise* encodings can be expressed as special cases of the *bimander* encoding. Interestingly, the special case of the *bimander* encoding where $m = \lceil \frac{n}{2} \rceil$, outperforms the *binary* encoding in all our experiments. It is important to note that our encoding allows unit propagation (UP) to preserve arc consistency, one of the most important technique in Constraint Programming (see [12]).

The structure of the paper is as follows. In Section 2, we briefly represent many known encodings of the AMO constraint. In Section 3, we describe the new *bimander* encoding and prove several important properties. In Section 4, we compare the *bimander* encoding with other encodings through experiments. Finally, we conclude and outline future research in Section 5.

[1] $\lceil x \rceil$ is the smallest integer not less than x.

2 Existing Encodings

Before giving a brief survey of some of the most widely used AMO encodings, we first define notions and notations, mainly following Frisch and Giannoros [17].

Let $X = \{X_i \mid 1 \leq i \leq n, n \in \mathbb{N}\}$ be a finite set of propositional variables, let A be a finite, possibly empty set of auxiliary propositional variables, and let $\phi(X, A)$ be a propositional formula in conjunctive normal form (CNF) encoding the constraint $\leq_1(X_1, ..., X_n)$. The encoding $\phi(X, A)$ is *correct* if and only if

- any (partial) assignment $\widehat{x}$ that satisfies $\leq_1(X_1, ..., X_n)$ can be extended to a complete assignment that satisfies $\phi(X, A)$, and
- for any (partial) assignment $\widehat{x}$ for X which assigns more than one variable of X to *TRUE*, unit propagation (UP) detects a conflict, i.e., repeated applications of UP yield the empty clause.

UP plays a crucial role in SAT solving as modern SAT solvers spent more than 80% of their runtime on it [15,29], whereas arc consistency is one of the most important techniques in CSP solvers because it is a very good trade-off between the amount and the cost of pruning. Therefore, when translating a CSP to a SAT instance one should pay much attention to determine whether UP on the resulting SAT instance enforces arc consistency. UP of a SAT encoding of the constraint $\leq_1(X_1, ..., X_n)$ achieves the same pruning as arc consistency on the original CSP if the following holds [17]:

- at most one propositional variable in X is assigned to *TRUE*, and
- if any variable $X_i \in X$ is assigned to *TRUE*, then all the other variables occurring in X must be assigned to *FALSE* using UP.

In the following sections, generally $AMO(X)$, $ALO(X)$, and $EO(X)$ denote the at-most-one, at-least-one, and exactly-one clauses, respectively, for the set of propositional variables X. For the sake of convenience, we will illustrate those encoding on a running example through the set consisting of 8 Boolean variables, $X = \{X_1, ..., X_8\}$.

2.1 The Pairwise Encoding

This encoding has several different names: the *naive* encoding [38,26], the *pairwise* encoding [37,33], or the *binomial* encoding [17]. In this paper, we refer to it as the *pairwise* encoding. The idea of this encoding is to express that no pair of two variables are simultaneously assigned to *TRUE*, therefore as soon as one literal is assigned to *TRUE*, then all others must be assigned to *FALSE*:

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} (\overline{X}_i \vee \overline{X}_j)$$

In the running example, the *pairwise* encoding produces the following clauses:

$$\begin{aligned}\overline{X}_1 \vee \overline{X}_2,\ \overline{X}_1 \vee \overline{X}_3,\ \overline{X}_1 \vee \overline{X}_4,\ \ldots,\ \overline{X}_1 \vee \overline{X}_8,\\ \overline{X}_2 \vee \overline{X}_3,\ \overline{X}_2 \vee \overline{X}_4,\ \ldots,\ \overline{X}_2 \vee \overline{X}_8,\\ \overline{X}_3 \vee \overline{X}_4,\ \ldots,\ \overline{X}_3 \vee \overline{X}_8,\\ \vdots\\ \overline{X}_7 \vee \overline{X}_8.\end{aligned}$$

The *pairwise* encoding is a traditional way of encoding the AMO constraint into SAT. Although this encoding does not need any auxiliary variables, it requires a quadratic number of clauses (see Table 1). As a result, this encoding produces impractically large formulas on problems with large domains. Nevertheless, the *pairwise* encoding is not only widely used in practice, but also easily combined with other encodings [26,40,14]. The encoding allows UP to maintain arc consistency.

2.2 The Binary Encoding

Frisch et al. [18,19] proposed the *binary* encoding. Independently, Prestwich called it *bitwise* encoding [33,35]) and used it to successfully solve a number of large instances of CSPs with a standard SAT solver [34,33].

The binary encoding requires new Boolean variables $B_1, ..., B_{\lceil log_2 n \rceil}$ with the understanding that B_j (or $\overline{B}_j$) is the bit j of $i-1$ represented by a binary string is. The desired clauses are

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil log_2 n \rceil} (\overline{X}_i \vee \phi(i,j)),$$

where $\phi(i,j)$ denotes B_j (or $\overline{B}_j$) if bit j of $i-1$ represented by a binary string is 1 (or 0). The running example is represented by the *binary* encoding as follows:

$$\begin{array}{llll}\overline{X}_1 \vee \overline{B}_1, & \overline{X}_2 \vee B_1, & \overline{X}_3 \vee \overline{B}_1, & \ldots, \quad \overline{X}_8 \vee B_1,\\ \overline{X}_1 \vee \overline{B}_2, & \overline{X}_2 \vee \overline{B}_2, & \overline{X}_3 \vee B_2, & \ldots, \quad \overline{X}_8 \vee B_2,\\ \overline{X}_1 \vee \overline{B}_3, & \overline{X}_2 \vee \overline{B}_3, & \overline{X}_3 \vee \overline{B}_3, & \ldots, \quad \overline{X}_8 \vee B_3.\end{array}$$

The idea is to create the different sequences of $\lceil log_2 n \rceil$-tuples $B_j, 1 \leq j \leq \lceil log_2 n \rceil$, such that whenever any X_i is assigned to *TRUE*, $1 \leq i \leq n$, then we immediately infer that the other variables $X_{i'}$ must be assigned to *FALSE*, for any $1 \leq i' \neq i \leq n$. One should observe that with the binary encoding UP maintains arc consistency.

2.3 The Commander Encoding

Klieber and Kwon [26] described the *commander* encoding by dividing the set $X = \{X_1, ..., X_n\}$ of propositional variables into m, $1 \leq m \leq n$, disjoint subsets denoted by $\{G_1, ..., G_m\}$, and introducing a *commander* variable c_i for each subset G_i, $1 \leq i \leq m$. The *commander* encoding is defined as follows:

1. Exactly one variable in each set $G_i \cup \{\overline{c}_i\}$ is assigned to *TRUE*.

$$\bigwedge_{i=1}^{m} EO(\{\overline{c}_i\} \cup G_i) = \bigwedge_{i=1}^{m} AMO(\{\overline{c}_i\} \cup G_i) \wedge \bigwedge_{i=1}^{m} ALO(\{\overline{c}_i\} \cup G_i).$$

   Whereas the ALO constraint is easily translated into a single clause, AMO can be encoded either by the pairwise on any other encoding. In the running example, we select $m = 4$ and divide the set $X = \{X_1, ..., X_8\}$ into the disjoint subsets $G_1 = \{X_1, X_2\}$, $G_2 = \{X_3, X_4\}$, $G_3 = \{X_5, X_6\}$, and $G_4 = \{X_7, X_8\}$. Then, four commander variables c_1, c_2, c_3, and c_4 are added. We obtain:

$$AMO(\overline{c}_1, X_1, X_2) \wedge (\overline{c}_1 \vee\ X_1 \vee X_2) \wedge \ldots \wedge AMO(\overline{c}_4, X_7, X_8) \wedge (\overline{c}_4 \vee X_7 \vee X_8).$$

2. At most one commander variable is assigned to *TRUE*. This constraint can be encoded either by the *pairwise* encoding or by another encoding (even by a recursive application of the commander encoding):

$$\bigwedge_{i=1}^{m} AMO(c_i).$$

   In the running example, we obtain:

$$AMO(c_1, c_2, c_3, c_4).$$

Compared with the *pairwise* encoding, the *commander* encoding requires fewer clauses but introduces (an acceptable number of) new variables (see Table 1). The *commander* encoding also allows UP to preserve arc consistency.

2.4 The Product Encoding

Chen [14] proposed an AMO encoding which is called *product* encoding. Instead of encoding the AMO constraint $\leq_1(X_1, ..., X_n)$ he encoded a constraint consisting of n corresponding points, denoted by $\leq_1\{(u_i, v_j) \mid 1 \leq i \leq p, 1 \leq j \leq q, p * q \geq n\}$. The idea can be explained as follows:

1. Each variable $X_k, 1 \leq k \leq n$, is mapped onto a corresponding point (u_i, v_j), where $u_i \in U = \{u_1, ..., u_p\}$ and $v_i \in V = \{v_1, ..., v_q\}$.
2. Then, the *product* encoding is obtained as:

$$AMO(X) = AMO(U) \wedge AMO(V) \bigwedge_{1 \leq i \leq p, 1 \leq j \leq q}^{1 \leq k \leq n, k=(i-1)q+j} ((\overline{X}_k \vee u_i) \wedge (\overline{X}_k \vee v_j)),$$

where $AMO(U)$ and $AMO(V)$ can be encoded by either another encoding or a recursive application of the *product* encoding.

With regard to our running example, we choose $p = 3$, $q = 3$, and use the pairwise encoding for $AMO(U)$ and $AMO(V)$. The derived clauses are:

$$\begin{aligned}
AMO(U) &= (\overline{u}_1 \vee \overline{u}_2) \ \wedge (\overline{u}_1 \vee \overline{u}_3) \wedge (\overline{u}_2 \vee \overline{u}_3),\\
AMO(V) &= (\overline{v}_1 \vee \overline{v}_2) \ \wedge (\overline{v}_1 \vee \overline{v}_3) \ \wedge (\overline{v}_2 \vee \overline{v}_3),\\
AMO(X) &= AMO(U) \wedge AMO(V) \wedge\\
&\quad (\overline{X}_1 \vee u_1) \wedge (\overline{X}_1 \vee v_1) \wedge (\overline{X}_2 \vee u_2) \wedge (\overline{X}_2 \vee v_1) \wedge\\
&\quad (\overline{X}_3 \vee u_3) \wedge (\overline{X}_3 \vee v_1) \wedge (\overline{X}_4 \vee u_1) \wedge (\overline{X}_4 \vee v_2) \wedge\\
&\quad (\overline{X}_5 \vee u_2) \wedge (\overline{X}_5 \vee v_2) \wedge (\overline{X}_6 \vee u_3) \wedge (\overline{X}_6 \vee v_2) \wedge\\
&\quad (\overline{X}_7 \vee u_1) \wedge (\overline{X}_7 \vee v_3) \wedge (\overline{X}_8 \vee u_2) \wedge (\overline{X}_8 \vee v_3).
\end{aligned}$$

One should observe that UP on the *product* encoding achieves arc consistency.

2.5 The Sequential Encoding

By building a count-and-compare hardware circuit and translating this circuit to an equivalent CNF formula, Sinz [38] introduced an encoding of $\leq_k(X_1, ..., X_n)$ which he called the *sequential* encoding. Here, we only consider the case $k = 1$ and obtain the following clauses:

$$(\overline{X}_1 \vee s_1) \wedge (\overline{X}_n \vee \overline{s}_{n-1}) \bigwedge_{1<i<n} ((\overline{X}_i \vee s_i) \wedge (\overline{s}_{i-1} \vee s_i) \wedge (\overline{X}_i \vee \overline{s}_{i-1})), \quad (1)$$

where $s_i, 1 \leq i \leq n-1$, are auxiliary variables.

The running example is represented by the following clauses:

$$\begin{array}{lll}
\overline{X}_1 \vee s_1, & & \\
\overline{X}_2 \vee s_2, & \overline{s}_1 \vee s_2, & \overline{X}_2 \vee \overline{s}_1,\\
\overline{X}_3 \vee s_3, & \overline{s}_2 \vee s_3, & \overline{X}_3 \vee \overline{s}_2,\\
\overline{X}_4 \vee s_4, & \overline{s}_3 \vee s_4, & \overline{X}_4 \vee \overline{s}_3,\\
\overline{X}_5 \vee s_5, & \overline{s}_4 \vee s_5, & \overline{X}_5 \vee \overline{s}_4,\\
\overline{X}_6 \vee s_6, & \overline{s}_5 \vee s_6, & \overline{X}_6 \vee \overline{s}_5,\\
\overline{X}_7 \vee s_7, & \overline{s}_6 \vee s_7, & \overline{X}_7 \vee \overline{s}_6,\\
\overline{X}_8 \vee \overline{s}_7. & &
\end{array}$$

As Marques-Silva and Lynce [37] pointed out, the sequence $s_1, ..., s_{n-1}$ is of the form *"0...01...1"* and whenever any Boolean variable X_i is assigned to ***TRUE*** (or *1*), $1 \leq i \leq n$, consequently, under UP all the other variables X_j must be forced to ***FALSE*** (or *0*), $1 \leq j \neq i \leq n$. In other words, this encoding allows UP to guarantee the arc consistency property.

2.6 The Ladder Encoding

Gent and Nightingale [22] used the ladder structure, originally proposed by Gent et al. [23], to describe a new encoding of the all-different constraint into SAT. It was named the *ladder* encoding.

Without loosing the correctness property, we consider the sequence $s_1, ..., s_{n-1}$ to play the role of the sequence $y_1, ..., y_{n-1}$ in [22] with the the condition reversed. Now the ladder validity clauses are represented as follows:

$$\bigwedge_{i=1}^{n} (\overline{s}_{i-1} \vee s_i), \tag{2}$$

where we set

$$s_0 = 0 \wedge s_n = 1. \tag{3}$$

We add the channelling clauses [22]

$$\bigwedge_{i=1}^{n} ((s_i \wedge \overline{s}_{i-1}) \leftrightarrow X_i), \tag{4}$$

and adding the conjunction of (2) and (4) we obtain the *ladder* encoding [22]:

$$\bigwedge_{i=1}^{n} ((\overline{s}_{i-1} \vee s_i) \wedge (\overline{s}_i \vee s_{i-1} \vee X_i) \wedge (\overline{X}_i \vee s_i) \wedge (\overline{X}_i \vee \overline{s}_{i-1})). \tag{5}$$

It is easy to check that the clauses $(\overline{s}_i \vee s_{i-1} \vee X_i)$ occurring in (5) are redundant because they do not affect the correctness of the AMO constraint. After removing these redundant clauses the *ladder* encoding is simplified to:

$$\bigwedge_{i=1}^{n} ((\overline{s}_{i-1} \vee s_i) \wedge (\overline{X}_i \vee s_i) \wedge (\overline{X}_i \vee \overline{s}_{i-1})). \tag{6}$$

Replacing (6) by (3) yields the *sequential* encoding:

$$(\overline{X}_1 \vee s_1) \wedge (\overline{X}_n \vee \overline{s}_{n-1}) \bigwedge_{1<i<n} ((\overline{X}_i \vee s_i) \wedge (\overline{s}_{i-1} \vee s_i) \wedge (\overline{X}_i \vee \overline{s}_{i-1})). \tag{7}$$

Furthermore, Prestwich supposed the *relaxed ladder* in [33]. Particularly, instead of adding the condition (4), he added the following condition:

$$\bigwedge_{i=1}^{n} ((s_i \wedge \overline{s}_{i-1}) \leftarrow X_i). \tag{8}$$

Hence, the *relaxed ladder* encoding is exactly the *ladder* encoding without the redundant clauses. Consequently, the *relaxed ladder* encoding and the *sequential* encoding are identical. Argelich et al. [7] also noticed that the *sequential* encoding is a reformulation of a *regular* encoding [5]. In fact, it is a simple matter to prove that the *regular* encoding and the *ladder* encodings are identical.

Summarizing, we showed that the *relaxed ladder* encoding and the *sequential* encoding are identical. These encodings are obtained from the *ladder* or the *regular* encoding by removing redundant clauses. One should observe that Tamura et al. used the *ladder* structure in the *order* encoding to translate CSPs to SAT instances in their SAT-based solving system [39]. Bailleux et al. [9] also used this structure, called *unary representation*, during their translation of cardinality constraints and pseudo-Boolean constraints to SAT formulas [9,16,10].

Recently, Martins et. al [28] compared both encodings, the *sequential* encoding and the *ladder* encoding. As the result, the experiment shown in their paper indicated very small difference between the two encodings.

In conclusion, here we claim the similarity of the *ladder*, *sequential*, *relaxed ladder*, *regular*, *order* encodings, and the *unary representation*. We hope that this work could help the SAT community to recognize the similarities of these encodings.

3 The Bimander Encoding

The general idea of the new encoding is based on both the ideas of the *binary* encoding and the *commander* encoding. We refer to it as the *bimander* encoding.

Similarly to the *commander* encoding, with a given positive number $m, 1 \leq m \leq n$, we partition a set of propositional variables $X = \{X_1, ..., X_n\}$ into m disjoint subsets $\{G_1, ..., G_m\}$ such that each subset G_i consists of $g = \lceil \frac{n}{m} \rceil$ variables. However, instead of introducing commander variables like in the *commander* encoding, we introduce a set of auxiliary propositional variables $B_1, ..., B_{\lceil log_2 m \rceil}$ like in the *binary* encoding. The variables $B_1, ..., B_{\lceil log_2 m \rceil}$ play the role of the commander variables in the *commander* encoding.

The *bimander* encoding is the conjunction of the following clauses:

1. At most *one* variable in each subset can be *TRUE*. We encode this constraint for each subset $G_i, 1 \leq i \leq m$, by using the *pairwise* encoding:

$$\bigwedge_{i=1}^{m} AMO(G_i). \tag{9}$$

   In our running example we choose $m = \lceil \sqrt{n} \rceil = 3$ to obtain:

$$AMO(X_1, X_2, X_3) \wedge AMO(X_4, X_5, X_6) \wedge AMO(X_7, X_8).$$

2. The following clauses are generated by the constraints between each variable and commander variables in a subset:

$$\bigwedge_{i=1}^{m} \bigwedge_{h=1}^{g} \bigwedge_{j=1}^{\lceil log_2 m \rceil} (\overline{X}_{i,h} \vee \phi(i,j)), \tag{10}$$

   where $\phi(i,j)$ denotes B_j (or $\overline{B}_j$) if bit j of $i-1$ represented by a unique binary string is 1 (or 0).

The following set of clauses is generated for the running example:

$$\begin{array}{lll} \overline{X}_1 \vee \overline{B}_1, & \overline{X}_4 \vee B_1, & \overline{X}_7 \vee \overline{B}_1, \\ \overline{X}_1 \vee \overline{B}_2, & \overline{X}_4 \vee \overline{B}_2, & \overline{X}_7 \vee B_2, \\ \overline{X}_2 \vee \overline{B}_1, & \overline{X}_5 \vee B_1, & \overline{X}_8 \vee \overline{B}_1, \\ \overline{X}_2 \vee \overline{B}_2, & \overline{X}_5 \vee \overline{B}_2, & \overline{X}_8 \vee B_2, \\ \overline{X}_3 \vee \overline{B}_1, & \overline{X}_6 \vee B_1, & \\ \overline{X}_3 \vee \overline{B}_2, & \overline{X}_6 \vee \overline{B}_2. & \end{array}$$

Compared with the *commander* encoding, the *bimander* encoding does not require any constraints among the sequences of auxiliary variables because any combination of such variables $B_1, ..., B_{\lceil log_2 m \rceil}$ of a corresponding subset is different from any combinations of all the other groups. Let us prove some important properties of the *bimander* encoding, following the definitions mentioned at the beginning of Section 2.

Correctness Let $\widehat{x} = (X_1, ..., X_l), 1 \leq l \leq n$, be a partial assignment with *at most one* variable assigned to *TRUE*. In case all variables are mapped to *FALSE*, then condition (9) is trivially satisfied. The same holds for condition (10). In case that only one variable, say $X_i, 1 \leq i \leq n$, is mapped to *TRUE*, then there is a corresponding sequence of *TRUE* values assigned to the corresponding sequence of $\{B_1, ...B_{\lceil log_2 m \rceil}\}$. Hence, condition (10) is satisfied as well. Therefore, the partial assignment $\widehat{x}$ can possibly be extended to a complete assignment that satisfies the two conditions.

Now suppose that we have a partial assignment $\widehat{x} = (X_1, ..., X_l), 1 \leq l \leq n$, with more than one variable assigned to *TRUE*. Suppose that $X_i = TRUE$ and $X_j = TRUE$, for $1 \leq i \neq j \leq l$. In order to satisfy the condition (9), the variables X_i and X_j must belong to different subsets. That leads to two differently corresponding patterns of the sequence $\{B_1, ...B_{\lceil log_2 m \rceil}\}$ which are assigned to *TRUE*. As a result, the sequence contains one propositional variable $B_k, 1 \leq k \leq \lceil log_2 m \rceil$, that is assigned to both *TRUE* and *FALSE* at the same time, which is impossible. Hence, if any partial assignment has more than one variable assigned to *TRUE*, then UP produces an empty clause. It means that this partial assignment can not be extended to a complete assignment.

In conclusion, the *bimander* encoding correctly encodes AMO.

Propagation Strength Let $\widehat{x} = (X_1, ..., X_l)$, $1 \leq l \leq n$, be a partial assignment where *TRUE* is assigned to exactly one variable. Now we will show that UP will assign all other variables to *FALSE*. Suppose that variable $X_{i,j} = TRUE$, which is the j^{th} variable in the subset $G_i, 1 \leq i \leq m$, then this assignment forces a corresponding pattern of the sequence $\{B_1, ...B_{\lceil log_2 m \rceil}\}$ to *TRUE*. Because $X_{i,j} = TRUE$, all other variables in the subset G_i are set to *FALSE* due to condition (9). Due to condition (10), all the other variables in the subsets $G_{i'}, 1 \leq i' \neq i \leq m$, are set to *FALSE* because they have different patterns of the sequence $\{B_1, ...B_{\lceil log_2 m \rceil}\}$ corresponding to $X_{i,j} = TRUE$. In conclusion, UP on the *bimander* encoding maintains arc consistency.

Complexity As we mentioned, we need a set of $\lceil log_2 m \rceil$ auxiliary variables. Condition (9) using the *pairwise* encoding requires $m * [\frac{g(g-1)}{2}] = \frac{n(\frac{n}{m}-1)}{2}$ new clauses. Condition (10) requires $m * [g * log_2 m] = n * \lceil log_2 m \rceil$ clauses. Hence, the encoding uses $\frac{n(\frac{n}{m}-1)}{2} + n\lceil log_2 m \rceil = \frac{n^2}{2m} + n\lceil log_2 m \rceil - \frac{n}{2}$ clauses.

Generalization It is worth pointing out that the *bimander* encoding can be easily generalized to encode the at-most-k constraint. Again, the set of variables is partitioned into several subsets.

1. For each subset, the at-most-k constraint is encoded by a modified pairwise (or another) encoding.
2. The constraints between each variable and the commander variables in a subset are encoded by the following clauses:

$$\bigwedge_{i=1}^{m} \bigwedge_{h=1}^{g} \bigvee_{l=1}^{k} \bigwedge_{j=1}^{\lceil log_2 m \rceil} (\overline{X}_{i,h} \vee \phi(i,h,l,j)),$$

   where $\phi(i,h,l,j)$ denotes $B_{l,j}$ (or $\overline{B}_{l,j}$) if bit j of $i-1$ represented by a binary string is 1 (or 0).

Special Cases One should observe that the *bimander* encoding is a general case of several encodings. For example,

- the *pairwise* encoding is a special case of the *bimander* encoding by setting $m = 1$;
- the *commander* encoding is a special case of the *bimander* encoding by setting $m = 2$ (when both encodings divide into *2* subsets); and
- the *binary* encoding is a special case of the *bimander* encoding by setting $m = n$.

4 Comparison and Experimental Evaluation

In this section, we first summarize key features of SAT encodings of the AMO constraint. Thereafter, we experimentally evaluate the encodings presented in Section 3 using different domains.

4.1 Comparison

Table 1 presents the key features of many approaches for encoding the AMO constraint (column *enc*). The columns *clauses* and *aux vars* depict the number of clauses required and auxiliary variables, respectively. The column *AC* indicates whether UP achieves arc consistency. The column *origin* refers to the original publications where the encoding had been introduced. *m* denotes the disjointed subsets by dividing the set of propositional variables $\{X_1, ..., X_n\}$ in the *bimander* encoding. In addition to the encodings of the AMO constraint presented in Section 3, we also mention several more encodings that are mainly used for cardinality constraints. As we can see in Table 1, the *bimander* encoding requires the least auxiliary variables among known encodings – with the exception of the *pairwise* encoding. The *totalizer* encoding proposed by Bailleux al et. [9] requires clauses of size at most *3*, and the *commander* encoding proposed by Klieber and Kwon [26] needs m (number of disjointed subsets) clauses of size $\lceil \frac{n}{m} + 1 \rceil$, whereas the *product*, *sequential*, *binary* and *bimander* encodings require only binary clauses. This is a considerable advantage of the new encoding.

Table 1. A summary of almost all known encodings of the AMO encoding.

enc	*clauses*	*aux vars*	*AC*	*origin*
pairwise	$\binom{n}{2}$	0	yes	none
linear	$8n$	$2n$	no	[42]
totalizer	$O(n^2)$	$O(nlog(n))$	yes	[9]
binary	$nlog_2n$	$\lceil log_2n \rceil$	yes	[19]
sequential	$3n-4$	$n-1$	yes	[38]
sorting networks	$O(nlog_2^2n)$	$O(nlog_2^2n)$	yes	[16]
commander	$\sim 3n$	$\sim \frac{n}{2}$	yes	[26]
product	$2n+4\sqrt{n}+O(\sqrt[4]{n})$	$2\sqrt{n}+O(\sqrt[4]{n})$	yes	[14]
card. networks	$6n-9$	$4n-6$	yes	[8]
PHFs-based	$nlog_2n$	$\lceil log_2n \rceil$	yes	[11]
bimander	$\frac{n^2}{2m}+nlog_2m-\frac{n}{2}$	$log_2m, 1 \leq m \leq n$	yes	this paper
bimander ($m=\frac{n}{2}$)	$nlog_2n-\frac{n}{2}$	$\lceil log_2n \rceil - 1$	yes	this paper

4.2 Experimental Evaluation

For the experimental evaluation we have selected some well-known problems which have been used in recent CSP and SAT competitions. In case of the *bimander* encoding, we have considered two different values for the parameter m, viz. $m=\sqrt{n}$ and $m=\frac{n}{2}$.

Our experiments were conducted using CLASP 2 [20] with default configuration on a 2.66-GHz Intel Core 2 Quad processor with 3.8 GB of memory. Bold font indicates the minimum time for each benchmark. We abbreviate *pairwise*, *sequential*, *commander*, *binary*, *product*, and *bimander* encodings as *pw*, *seq*, *cmd*, *bin*, *pro* and *bim*, respectively. For the *commander* encoding, the set of variables is recursively divided into 2 disjoint subsets since the encoding in that case conducted on our problems gives a best result in term of the average time.

Pigeon-Hole Problems The goal of the problem is to prove that *p* pigeons can not fit in $h=p-1$ holes. Table 2 shows the results from different encodings on unsatisfiable Pigeon-Holes instances.

Table 2. A comparison of the run times for Pigeon-Hole problems. Run times are in seconds.

enc	*pw*	*seq*	*cmd*	*bin*	*pro*	*bim*($\sqrt{n}$)	*bim*($n/2$)
10	2.16	0.73	0.56	0.80	0.22	0.33	**0.22**
11	22.15	5.79	4.46	6.59	6.13	5.10	**2.10**
12	244.59	117.83	43.27	29.52	43.21	38.19	**26.06**
13	>3600.00	1604.14	352.53	142.60	736.25	546.91	**64.91**
14	>3600.00	>3600.00	>3600.00	1271.24	>3600.00	>3600.00	**560.03**
average	>1493.78	>1065.69	> 800.16	290.15	> 877.16	> 838.10	**130.66**

It can be seen that the *bimander* encoding (with $m=\frac{n}{2}$) performs best in all cases, followed by the *binary* encoding, whereas the *bimander* encoding (with $m=\sqrt{n}$) outperforms the rest.

Hamiltonian Cycle Problems Given a directed or undirected graph with a set of vertices and a set of edges between vertices. A Hamiltonian cycle is a path that visits each vertex exactly once, except for the vertex that is both first and the last, which is visited twice. The different instances were taken from [1].

The run times for the different encodings of the Hamiltonian cycle problem are presented in Table 3. Both *bimander* encodings achieve the best average run time, followed by the *binary* encoding. The *commander* and *product* encodings are faster than the *sequential* encoding, whereas *pairwise* encoding performs worse.

Table 3. A comparison of run times for satisfiable Hamiltonian cycle instances. Run times are in seconds.

enc	*pw*	*seq*	*cmd*	*bin*	*pro*	*bim*($\sqrt{n}$)	*bim*($n/2$)
miles750	135.48	25.42	**13.67**	38.19	14.18	32.55	22.92
miles1000	67.77	10.93	7.65	**7.38**	12.45	9.52	8.19
miles1500	30.01	3.30	2.60	2.95	**2.46**	3.74	3.16
queen10_10	13.87	4.16	**3.54**	3.77	3.68	4.00	3.75
queen11_11	32.34	9.75	8.32	8.43	8.41	9.23	**8.16**
queen12_12	**1.73**	22.46	20.13	21.49	18.43	20.44	21.13
queen13_13	3.10	40.99	38.43	1.58	36.30	1.45	**1.39**
queen14_14	5.17	2.47	2.53	2.42	**2.09**	2.27	2.20
queen15_15	7.75	3.64	3.42	3.76	**3.17**	3.47	3.37
queen16_16	11.26	4.80	**5.21**	5.44	5.14	5.37	5.25
average	30.84	12.79	10.55	9.54	10.63	9.20	**7.95**

All-Interval Series Problems The goal of the problem is to arrange a permutation of the n integers ranging from 1 to n in such a way that the differences between adjacent numbers are also a permutation, of the numbers from 1 to $n-1$. As a result, the performance of this benchmarks is heavily influenced by the encoding of the AMO constraint. All-interval series problems are one of the classical CSPs and are usually regarded as a difficult benchmark to find all solutions (see prob007 in [24]).

Table 4. A comparison of run times for All-Interval Series problems. Run times are in seconds. *sol* shows the number of all solutions of the corresponding instance.

enc	*pw*	*seq*	*cmd*	*bin*	*pro*	*bim*($\sqrt{n}$)	*bim*($n/2$)	*sol*
7	0.05	0.03	0.02	0.02	0.05	**0.01**	0.02	32
8	0.56	1.07	0.63	**0.20**	0.49	0.62	0.62	40
9	5.33	8.92	0.37	0.27	5.61	0.33	**0.24**	120
10	61.72	104.02	1.72	1.58	60.71	1.95	**1.46**	296
11	972.54	1387.67	11.96	8.94	269.43	11.34	**6.72**	648
12	>3600.00	>3600.00	78.91	49.24	>3600.00	69.52	**43.81**	1328
13	>3600.00	>3600.00	517.72	356.64	>3600.00	504.61	**276.34**	3200
14	>3600.00	>3600.00	3200.21	2748.69	>3600.00	3537.74	**2005.18**	9912
average	>1480.02	>1537.71	476.44	395.69	>1392.03	515.76	**291.79**	

Table 4 summaries the run times for different encodings on AIS instances. Except for the cases $n = 7$ and $n = 8$, the table shows that the *bimander* encoding with $m = \frac{n}{2}$ significantly surpasses all the others. Moreover, for the final three instances this one performs in a reasonable time, whereas the *pairwise*, *sequential*, and *product* encodings carry out more than 3600 seconds. The *binary* encoding gives rather good results, while the *bimander* encoding in case $m = \sqrt{n}$ and the *commander* encoding perform similarly. The *pairwise*, *sequential*, and *product* encodings perform worse.

Quasigroup With Holes Problems A quasigroup is a square of values x_{ij}, $1 \leq i, j \leq n$, where each number $[1..n]$ occurs exactly oncee in each row and column. Achlioptas et al. [2] introduced an encoding for generating satisfiable quasigroups with holes instances in which some cells are filled. Quasigroup with holes instances can be consider as a multiple permutation problem in which the variables may occur in more than one permutation problem. Moreover, the encoding can tune the generator to output hard instances. We experimented with instances with different levels of hardness.

Table 5 shows the results from different encodings on satisfiable quasigroup with holes problems. The *bimander* encoding with parameter $m = \sqrt{n}$ is clearly the fastest with the exception of the instance qwh.order40.holes544. Surprisingly, the *pairwise* encoding performs very well followed by the *commander* encoding. The *bimander* encoding with parameter $m = \frac{n}{2}$, the *binary*, and the *product* encoding are quite similar. Although the *sequential* encoding was the fastest on the instance qwh.order40.holes544, its overall performance is poor.

Table 5. A comparison of run times for satisfiable quasigroup with holes problems. Run times are in seconds.

enc	*pw*	*seq*	*cmd*	*bin*	*pro*	*bim*($\sqrt{n}$)	*bim*($n/2$)
qwh.order30.holes320	0.46	0.28	0.23	0.25	0.23	**0.20**	0.22
qwh.order35.holes405	3.62	3.51	10.35	6.51	5.73	**1.60**	2.14
qwh.order40.holes528	134.71	115.62	124.26	120.47	241.20	**58.90**	159.21
qwh.order40.holes544	39.26	**14.57**	47.82	123.72	46.7	70.81	154.03
qwh.order40.holes560	121.74	65.36	55.68	119.66	33.16	**21.22**	53.27
qwh.order33.holes381	58.73	435.90	174.29	94.22	108.03	**12.74**	92.30
average	358.52	635.24	412.63	464.83	435.05	**165.47**	461.17

5 Conclusions and Future Works

Inspired by being remarkably successful at solving hard and practical problems of SAT solving, many problems that were solved previously by other encodings can now be solved more effectively by translating them into SAT instances and applying advanced SAT solvers to find solutions. During the encoding phase, one of the most important constraints occurring naturally in a wide range of real world applications, is the AMO constraint. Hence, many problems may benefit from effective encodings of this constraint.

The paper has four contributions. Firstly, we pointed out that the *ladder* encoding [22] is nothing but the *sequential* encoding [38] and a set of redundant clauses.

Moreover, the *relaxed ladder* encoding [33] and the *sequential* encoding are identical. Another two encodings, viz. the *ladder* and the *regular* [5] encoding, are also identical. Moreover, the *relaxed ladder* and the *sequential* encodings can be obtained from the *ladder* and the *regular* encoding after removing redundant clauses. Interestingly, these ideas were exploited in the *unary representation* [9] and the *order* encoding [39].

Secondly, we proposed a new encoding for AMO, the so-called *bimander* encoding. Compared to many other well-known AMO encodings, the *bimander* encoding requires the least auxiliary variables (with the exception of the *pairwise* encoding which does not erequire any auxiliary variables at all). Although the *commander* encoding and the *bimander* encoding use the same approach by dividing the original set of propositional variables, the *commander* encoding requires clauses of size $\lceil \frac{n}{m} + 1 \rceil$ (where m is the number of disjoint subsets), whereas the *bimander* encoding requires only binary clauses. We believe that this helps the *bimander* encoding to perform better than the *commander* encoding in our experimental evaluation. Moreover, the *bimander* encoding has the advantage of high scalability, and it can easily be adjusted in terms of the number of additional propositional variables to obtain particular encodings. For example, the *pairwise* or *binary* encodings are special cases of the *bimander* encoding.

Thirdly, this paper also proposes a special case, when dividing the propositional variables into $m = \lceil \frac{n}{2} \rceil$ disjoint subsets. From a theoretical point of view, this case is better than the *binary* encoding due to fewer auxiliary variables and clauses. From a practical point of view, we show that this special case of the *bimander* encoding ($m = \lceil \frac{n}{2} \rceil$) performs better than the *binary* encoding in all experiments in term of run time.

Fourthly, in practice, the *bimander* encoding is practical and easy to implement. Our results reveal that two particular cases of the *bimander* encoding are very competitive in a comparison with other well-known encodings.

A future research is to study how the number of disjoint subsets could affect the *bimander* encoding in realistic problems. It would be particularly useful to extend our findings to the at-most-k constraint. Finally, the ultimate goal should carry out a profound study of not only analytical, but also theoretical knowledge of variants of well-known encodings. We expect that this will help us to deepen our understanding on what encoding one should select given a particular problem.

Acknowledgments We would like to thank Christoph Wernhard, Norbert Manthey for many fruitful suggestions, and Martin Gebser for his helpful discussions. We also wish to thank Carla Gomes for her kindly providing us the QWH's generator.

References

1. A Computational Symposium at Cornell University, Ithaca, NY, USA, 2002. `http://mat.gsia.cmu.edu/COLOR03/`
2. Achlioptas, D., Gomes, C.P., Kautz, H.A., Selman, B.: Generating Satisfiable Problem Instances. In: Kautz, H.A., Porter, B.W. (eds.) Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA. pp. 256–261. AAAI Press / The MIT Press (2000)

3. Anbulagan, Grastien, A.: Importance of variables semantic in CNF encoding of cardinality constraints. In: Bulitko, V., Beck, J.C. (eds.) Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009. AAAI (2009)
4. Ansótegui, C., Bonet, M.L., Levy, J.: Solving (weighted) partial maxsat through satisfiability testing. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. pp. 427–440. Lecture Notes in Computer Science, Springer (2009)
5. Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables into problems with boolean variables. In: SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings. pp. 1–15. Springer LNCS (2004)
6. Argelich, J., Cabiscol, A., Lynce, I., Manyà, F.: Sequential Encodings from Max-CSP into Partial Max-SAT. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5584, pp. 161–166. Springer (2009)
7. Argelich, J., Cabiscol, A., Lynce, I., Manyà, F.: New Insights into Encodings from MaxCSP into Partial MaxSAT. In: 40th IEEE International Symposium on Multiple-Valued Logic, ISMVL 2010, Barcelona, Spain, 26-28 May 2010. pp. 46–52. IEEE Computer Society (2010)
8. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality Networks: a Theoretical and Empirical Study. Constraints 16(2), 195–221 (2011)
9. Bailleux, O., Boufkhad, Y.: Efficient CNF Encoding of Boolean Cardinality Constraints. Principles and Practice of Constraint Programming 9th International Conference CP-2003 pp. 108–122 (2003)
10. Bailleux, O., Boufkhad, Y., Roussel, O.: New Encodings of Pseudo-Boolean Constraints into CNF. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5584, pp. 181–194. Springer (2009)
11. Ben-Haim, Y., Ivrii, A., Margalit, O., Matsliah, A.: Perfect Hashing and CNF Encodings of Cardinality Constraints. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7317, pp. 397–409. Springer (2012)
12. Bessiere, C.: Chapter 3 Constraint Propagation, vol. 2, pp. 27 – 81. Elsevier (2006)
13. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
14. Chen, J.C.: A new SAT Encoding of the At-Most-One Constraint. In: Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation (2010)
15. Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem-proving. Commun. ACM 5(7), 394–397 (Jul 1962)
16. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2, 1–26 (2006)
17. Frisch, A.M., Giannoros, P.A.: SAT Encodings of the At-Most-k Constraint. Some Old, Some New, Some Fast, Some Slow. In: Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation (2010)
18. Frisch, A.M., Peugniez, T.J., Doggett, A.J., Nightingale, P.W.: Solving Non-Boolean Satisfiability Problems with Stochastic Local Search. In: in Proc. IJCAI-01. pp. 282–288 (2001)
19. Frisch, A.M., Peugniez, T.J., Doggett, A.J., Nightingale, P.W.: Solving Non-Boolean Satisfiability Problems with Stochastic Local Search: A Comparison of Encodings. J. Autom. Reason. 35, 143–179 (October 2005)

20. Gebser, M., Kaufmann, B., Schaub, T.: The Conflict-Driven Answer Set Solver clasp: Progress Report. In: Erdem, E., Lin, F., Schaub, T. (eds.) Logic Programming and Nonmonotonic Reasoning, 10th International Conference, LPNMR 2009, Potsdam, Germany, September 14-18, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5753, pp. 509–514. Springer (2009)
21. Gent, I.: Arc Consistency in SAT. In fifteenth European Conference on Artificial Intelligence pp. 121–125 (2002)
22. Gent, I., Nightingale, P.: A New Encoding of AllDifferent into SAT. In: Frisch, A.M., Miguel, I. (eds.) Proceedings 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems. pp. 95–110. Springer (2004)
23. Gent, I., Prosser, P., Smith, B.: A 0/1 encoding of the gaclex constraint for pairs of vectors. In: ECAI 2002 workshop W9: Modelling and Solving Problems with Constraints. University of Glasgow (2002)
24. Hnich, B., Miguel, I., Gent, I.P., , Walsh, T.: CSPLib is a Library of Test Problems for Constraint Solvers. `http://www.csplib.org/`, [Online; accessed 24-April-2013]
25. Kautz, H., Selman, B.: The State of SAT. Discrete Appl. Math. 155, 1514–1524 (June 2007)
26. Klieber, W., Kwon, G.: Efficient CNF Encoding for Selecting 1 from N Objects . In: the Fourth Workshop on Constraint in Formal Verification (CFV) (2007)
27. Marques-silva, J.: Practical Applications of Boolean Satisfiability. In: In Workshop on Discrete Event Systems (WODES. IEEE Press (2008)
28. Martins, R., Manquinho, V.M., Lynce, I.: Exploiting Cardinality Encodings in Parallel Maximum Satisfiability. In: IEEE 23rd International Conference on Tools with Artificial Intelligence, ICTAI 2011, Boca Raton, FL, USA, November 7-9, 2011. pp. 313–320 (2011)
29. 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 2001, Las Vegas, NV, USA, June 18-22, 2001. pp. 530–535. ACM (2001)
30. Prestwich, S.D.: Local Search on SAT-Encoded Colouring Problems. In: Lecture Notes in Computer Science. vol. 2919. Springer Verlag (2003)
31. Prestwich, S.D.: SAT Problems with Chains of Dependent Variables. Discrete Applied Mathematics 130(2), 329–350 (2003)
32. Prestwich, S.D.: Full Dynamic Substitutability by SAT Encoding. In: Wallace, M. (ed.) Principles and Practice of Constraint Programming - CP 2004, 10th International Conference,September 27 - October 1, 2004, Toronto, Canada. Lecture Notes in Computer Science, vol. 3258, pp. 512–526. Springer (2004)
33. Prestwich, S.D.: Finding Large Cliques using SAT Local Search. vol. Trends in Constraint Programming, pp. 273–278. ISTE (2007)
34. Prestwich, S.D.: Variable Dependency in Local Search: Prevention Is Better Than Cure. In: Silva, J.M., Sakallah, K.A. (eds.) Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4501, pp. 107–120. Springer (2007)
35. Prestwich, S.D.: CNF Encodings. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 75–97. IOS Press (2009)
36. Rossi, F., Beek, P.v., Walsh, T.: Handbook of Constraint Programming (Foundations of Artificial Intelligence). Elsevier Science Inc., New York, NY, USA (2006)
37. Silva, J.M., Lynce, I.: Towards Robust CNF Encodings of Cardinality Constraints . In: Proc. 13th International Conference on Principles and Practice of Constraint Programming CP-2007. Lecture Notes in Computer Science, vol. 4741, pp. 483–497. Springer (2007)
38. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: Principles and Practice of Constraint Programming, 11th International Conference, CP 2005,

Spain, October 2005, Proceedings. Lecture Notes in Computer Science, vol. 3709, pp. 827–831. Springer (2005)
39. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling Finite Linear CSP into SAT. Constraints 14(2), 254–272 (2009)
40. Velev, M.N.: Exploiting Hierarchy and Structure to Efficiently Solve Graph Coloring as SAT. In: International Conference on Computer-Aided Design (ICCAD'07), November 5-8, 2007, San Jose, CA, USA. pp. 135–142. IEEE (2007)
41. Walsh, T.: SAT v CSP. In: Principles and Practice of Constraint Programming - CP2000. Lecture Notes in Computer Science, vol. 1894, pp. 441–456. Springer (2000)
42. Warners, J.P.: A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form. Information Processing Letters 68(2), 63–69 (1998)

Table Constraints in Clause Learning CSP Solvers

Ozan Erdem[1], George Katsirelos[2], and Fahiem Bacchus[1]

[1] Department of Computer Science, University of Toronto,
Toronto, Ontario, Canada, M5S 3H5,
{ozan, fbacchus}@cs.toronto.edu
[2] MIAT, INRA, Toulouse, France,
george.katsirelos@toulouse.inra.fr

Abstract. We investigate alternative methods for implementing table constraints in clause learning CSP solvers (CL solvers). CL solvers have been an important development in CP solving as they can provide important performance improvements on some problems. Furthermore, table constraints remain an important and useful modeling tool in CP. Hence, it is important to be able to utilize table constraints in CL solvers effectively. Here we compare different ways of achieving GAC propagation over table constraints in a CL solver. These methods require different representations of the constraint. First we utilize a CNF encoding of the table constraint which has the property that unit propagation achieves GAC. We compare this with the use of a traditional GAC propagation algorithm for tables, Simple Tabular Reduction (STR). To utilize STR in a CL solver we also develop a method for extracting clausal explanations for pruned values. We also develop and test a negative version of STR which more compactly represents tables that have fewer falsifying than satisfying tuples. This version also generates clausal explanations. We implement these different methods in the CL solver minicsp, and compare them to the CNF encodings. Even though our proposed methods do not perform as well as the encodings in general, we believe that they will stimulate further research on this area.

1 Introduction

A number of robust and powerful CSP solvers have been developed and are publicly available. A recent development in CSP solver technology has been the importation of ideas from SAT, specifically clause learning [9, 10, 12, 13]. Clause learning improves the theoretical power of a CSP solver, and recently well engineered clause learning CSP solvers (CL solvers) have demonstrated very good empirical performance on a range of problems (see, e.g., the results of recent MiniZinc Challenges, http://www.minizinc.org/).

One of the main technologies exploited by CL solvers is the ability to generate clausal explanations from global constraints. Specifically, when a global constraint prunes a domain value, a clause justifying that pruning needs to be

generated (also called an explanation). Once the solver can obtain a clause for each pruned value, it can perform clause learning much like a regular SAT solver. In particular, when a contradiction is detected the CL solver must be able to obtain the clausal reasons for the various domain prunings that lead up to the contradiction, and resolve these reasons against the base contradiction.

A considerable amount of work has been done showing how to generate clausal reasons (typically on-demand or lazily) from propagators for various global constraints, e.g., [9, 14, 5, 4]. Methods for generating explanations from table constraints were examined in [9], but from a practical point of view these methods have not been previously examined in the light of the performance tradeoffs of modern CL solvers.

Table constraints are of course very important in constraint programming. They are easy for inexperienced users to use, problem domains often contain special ad-hoc constraints that are most easily encoded extensionally as table constraints, and information stored in databases is often accessed most conveniently as a table constraint. It is not surprising then that much research has been done on efficiently achieving GAC on table constraints.

In this paper we examine some different options for implementing table constraints in a CL solver so as to efficiently achieve GAC. We look at a clausal decomposition, i.e., representing the table constraint as a set of clauses (CNF). The main inference method available on clauses in a CL solver is unit propagation, hence we examine a clausal encoding on which unit propagation is sufficient to achieve GAC. A CL solver can also treat a table constraint as a black box—just as if it was any other propagator for achieving GAC over a specific type of constraint. As mentioned above, all that the CL solver needs is the ability to generate (or obtain on the fly) clausal reasons for the values pruned by the constraint. We examine one of the most efficient and simplest "propagators" for table constraints, Simple Tabular Reduction [11]. Although not typically viewed as being a propagator, an STR represented table constraint interfaces with a CL solver in the same way that any other propagator would. Finally, we examine a new STR-like algorithm for propagating table constraints represented by their falsifying rather than satisfying tuples. Negative STR can be more effective for table constraints that have many more satisfying tuples than falsifying tuples. We give an algorithm for achieving GAC on this negative STR table. We compare these different ways of handling table constraints in CL solvers and draw some conclusions about their relative effectiveness in practice.

2 Background

A constraint problem $\mathcal{P}$ consists of a finite set $\mathcal{V} = \{V_1, ..., V_n\}$ of variables, each with a finite domain of values $Dom[V_i]$ and a finite set of constraints $\mathcal{C} = \{c_1, \ldots, c_m\}$. Each constraint c_i is defined over some subset of variables, $scope(c_i) \subseteq \mathcal{V}$. An assignment $\mathcal{A}$ is a set of variable value assignments $\{V^1 = d^1, \ldots, V^k = d^k\}$ in which any variable is assigned at most a single value. Let

$vars(\mathcal{A}) \subseteq \mathcal{V}$ denote the set of variables assigned in $\mathcal{A}$. $\mathcal{A}$ is said to **cover** a constraint c if $scope(c) \subseteq vars(\mathcal{A})$.

A constraint c can be viewed as a Boolean function from assignments $\mathcal{A}$ that cover it to *true* (in which case $\mathcal{A}$ is said to be satisfying) or *false* ($\mathcal{A}$ is said to be falsifying). c is actually a function only of the variables in $scope(c)$, so if $\mathcal{A}$ covers c and $V \notin scope(c)$, then $\mathcal{A}$ with any assignments to V removed will be mapped to the same value as $\mathcal{A}$.

Often we consider $scope(c)$ to be ordered, and then for assignments $\mathcal{A}$ such that $vars(\mathcal{A}) = scope(c)$ we can order its variable assignments in the same way. Then without loss of information we can remove the variables leaving only a sequence of values. This ordered sequence of values is called a **tuple** for c. If $\mathcal{A}$ is satisfying then its corresponding tuple is said to be a positive tuple (p-tuple) for c, otherwise it is a negative tuple for c (n-tuple). If t is a tuple we let $vars(t)$ denote the variables t specifies values for. If $V \in vars(t)$ we let $t[V]$ denote the value assigned to V in t, and we say that t is **valid** iff for all $V \in vars(t)$ we have that $t[V] \in Dom[V]$. A valid p-tuple t for c is said to be a **support** for the value $d \in Dom[V]$ (where $V \in scope(c)$) if $t[V] = d$.

A positive table constraint (p-table) is a constraint that is specified by a set of p-tuples, a negative table constraint is specified by a set of n-tuples. In particular these sets of tuples are complete: t satisfies a p-table constraint T if and only if $t \in T$ (similarly for n-tables).

Finally, a constraint c is said to be generalized arc-consistent (GAC) if $\forall V \in scope(c), d \in Dom[V]$ there exists a valid p-tuple t for c such that $t[V] = d$.

A *propositional variable* is a variable with domain $\{0, 1\}$. By convention, if x is a propositional variable, we write x for $x = 1$ and $\neg x$ for $x = 0$. x and $\neg x$ are *literals* of the variable x and they are called complementary to each other. If l is a literal, we denote its complement by $\bar{l}$. A *clause* is a constraint which is a set of literals, interpreted as their logical disjunction, e.g., $(x, y, \overline{z})$, which is interpreted as $x \vee y \vee \neg z$.

In a CL solver, for each variable we maintain (implicitly or explicitly) a *clausal encoding* of its domain. There are several options for this; we use the order encoding in [12, 3, 1]. For every multi-valued variable V with $Dom[V] = \{d_1, \ldots, d_k\}$ we have k Boolean assignment variables $A_{V=d_j}$ and $k + 1$ *order* variables[3] $A_{V \leq d_j}$, $0 \leq j \leq k$. The variable $A_{V=d_j}$ is true when $V = d_j$, and is false when d_j has been pruned from V's domain. The variable $A_{V \leq d_j}$ is true when all values greater than d_j have been pruned from the domain of V and false when all values less than or equal to d_j have been pruned. We also have $O(k)$ clauses that encode $A_{V=d} \iff A_{V \leq d} \wedge \neg A_{V \leq d-1}$ and $A_{V \leq d} \rightarrow A_{V \leq d+1}$ and the unit clauses $(A_{V \leq d_k})$ and $(\neg A_{V \leq d_0})$. These ensure that each variable is assigned exactly one value.

[3] d_0 is a sentinel value, which we use here only to simplify the exposition. In practice it is optimized away.

3 Clausal Decompositions of Table Constraints

We aim to examine methods for implementing table constraints in CL solvers. Such solvers have efficient mechanisms for handling clauses, as they can learn many clauses during solving, and it is very easy to use the same mechanisms to deal with an initial set of input clauses. Hence, one way of implementing table constraints is to convert them into a set of input clauses. The only restriction is that the solver only has access to unit propagation (UP) to reason about these clauses. Thus to achieve GAC we must use an encoding on which UP achieves GAC. One significant advantage of clausal encodings is that the clausal reasons are already available—no extra computation is needed to obtain them.

3.1 Support Tuple Encoding

In [2] a CNF encoding for a table constraint c was given on which UP achieves GAC. This encoding was an adaptation of encodings presented in [8].

To encode the constraint c we utilize additional propositional variables t_1, ..., t_m, each one representing one of the m different p-tuples of c. Let τ_i be the p-tuple represented by the propositional variable t_i. Using the t_i variables we can write the clauses capturing C as follows. For the variable $V \in \mathit{scope}(c)$ and value $d \in \mathit{Dom}[V]$, let $\{s_1, \ldots, s_i\}$ be the subset of $\{t_1, \ldots, t_m\}$ such that the satisfying tuples represented by the s_i are precisely the set of tuples τ_i such that $\tau_i[V] = d$ (these are the supports of $V = d$). For each variable and value $V = d$ we have the clause $(s_1, \ldots, s_i, \neg A_{V=d})$ ($V = d$ must be false if it has no support). Finally, we have for each p-tuple of C, τ_i, and assignment $V = d \in \tau_i$ the clause $(A_{V=d}, \neg t_i)$, which captures the condition that the tuple of assignments τ_i cannot hold if $V = d$ cannot be true and also the condition that if τ_i holds then so do all of its variable assignments.

For example consider the table constraint C, where $\mathit{scope}(C) = \{V_1, V_2\}$ and $\mathit{Dom}[V_1] = \mathit{Dom}[V_2] = \{1, 2\}$. Let the table be represented by a set of tuples as $\{(1, 1), (2, 1), (2, 2)\}$. To encode this constraint we associate a new variable with each row, namely t_1, t_2 and t_3. We have the following clauses: $(t_1, \neg A_{V_1=1})$, $(t_2, \neg A_{V_1=2})$, $(t_1, t_2, \neg A_{V_2=1})$, $(t_3, \neg A_{V_2=2})$ for capturing the loss of supports for each variable value pair. In addition, we have the following clauses to capture the relation between assignments and the new variables: $(A_{V_1=1}, \neg t_1)$, $(A_{V_2=1}, \neg t_1)$, $(A_{V_1=2}, \neg t_2)$, $(A_{V_2=1}, \neg t_2)$, $(A_{V_1=2}, \neg t_3)$,$(A_{V_2=2}, \neg t_3)$.

It has been shown [2] that this encoding enforces GAC on a table constraint. It can also be observed that the size of this encoding (i.e., the sum of the lengths of the clauses) is linear in the size of the constraint's p-table representation: $O(mr)$ where m is the number of satisfying tuples, and r is the constraint's arity. Hence UP on this encoding will operate in time linear in the size of the constraint's p-table representation.

4 Clausal Reasons from Table Propagators

In this section we examine an algorithmic representation for achieving GAC on a table constraint. There are many such algorithms, but in this paper we focus on simple tabular reduction, a GAC algorithm that is particularly efficient under backtracking. When we implement table constraints using non-clausal representations the CL solvers must be able to derive clausal reasons for pruned values. First we discuss the standard STR algorithm that works on p-table constraints. Then we turn our attention to a negative version of STR that works on n-table constraints. Our negative STR algorithm is new and might be of interest beyond the context of CL solvers.

4.1 Positive STR

Simple Tabular Reduction (STR) is an efficient GAC algorithm which dynamically maintains tables in order to keep track of supports. It was first introduced by [15], and used in the context of a backtracking search algorithm by [11] along with a number of optimizations under the name STR2+. In this paper we will refer to STR2+ as *positive STR*, and as *STR* when the context is clear. In positive STR, the table t of a constraint is divided into its upper and lower parts called $top(t)$ and $bottom(t)$ with all the tuples in *bottom* being invalid.

As described in [11] achieving GAC with STR involves processing the tuples in *top* to determine if they are valid (these tuples become invalid as the values they assign to variables are pruned). If a tuple t is found to be invalid it is moved to *bottom*, while if t is found to be valid all of the values it assigns are marked as having a support (i.e., these values are GAC). After all tuples in *top* are processed, the unsupported variable values are pruned. The key contribution of STR is that the invalid tuples need never be examined, the valid tuples need be examined only once, and backtracking can be achieved by simply restoring the variable domains and moving the marker that divides *top* from *bottom*.

Our addition to STR is to compute clausal reasons for the values pruned. These reasons can also be computed on demand (lazily). This is achieved as follows. When processing the tuples in *top*, for any invalid tuple t we detect, we remember the variable value pair that made t invalid. This is some $V = d$ such that $t[v] = d$ and $d \notin Dom[V]$. There must be at least one such pair in t for it to be classified as invalid. If there are multiple pairs we simply choose the first one found. Hence, each tuple in *bottom* is marked with a proposition $A_{V_i=d_j}$. Then if we prune a value, e.g., $V = d$ to compute a reason we scan all tuples in *bottom*, locate those tuples t with $t[V] = d$ and accumulate their propositional reasons into a set. The conjunction of these propositions then implies the loss of all supporting tuples for $V = d$, and thus implies $\neg A_{V=d}$. This implication is a clause and supplies the reason we wanted to compute. In [7] an alternative method is presented for computing reasons from table constraints, where they use a trie containing tuples of the table. This trie is built using a static ordering of the variables and hence some extra processing is required to extract a reason

Algorithm 1: Initialize n-STR

```
1  init-nSTR (c);
2  dprod = ∏_{V∈scope(c)} |Dom[V]|;
3  foreach X ∈ scope(c) do
4  |  c.dprod[X] = dprod / |Dom[X]|;
5  |  foreach d ∈ Dom[X] do
   |  |  // Initialize Forbidden Tuple Counts
6  |  |  c.ftc[X][d] ← # of n-tuples t for c with t[X] = d;
7  |  |  if c.ftc[X][d] == c.dprod[X] then
8  |  |  |  prune (X,d);
9  |  |  |  c.update = true;
10 if c.update then
11 |  prop-nSTR (c);
```

compatible with the order of the variables along the current search path. Our approach more naturally yields a reason compatible with the current search path.

Since we must scan all tuples in *bottom* computing reasons can be fairly expensive, and it can be done lazily by simply restoring the marker between *top* and *bottom* to its state when the value was pruned and then examine all tuples in *bottom*. Alternatively, by using more memory these reasons could be accumulated as the tuples are detected to be invalid. We did not experiment with this alternative, as the STR approach in general is not that promising for CL solvers (see Sec. 5).

4.2 Negative STR

The new STR-based algorithm that we describe for negative table constraints c reuses the table data structures of the positive STR. In addition, it maintains the following data structures:

- **dprod**: is an array of size $r = |scope(c)|$, such that **dprod**$[i]$ is the number of valid (satisfying or falsifying) tuples that contain each value of variable i.
- **ftc**: is a two dimensional array giving the number of valid falsifying tuples for every variable/value pair.

The initialization of the n-STR propagator is shown in Algorithm 1 and the propagator itself in Algorithm 2. They both compute **dprod** and **ftc** in a straightforward manner. The only complication is that when we prune a value, the counts have to be recomputed from scratch, hence we set the variable *update* to *true* in order to repeat the loop.

To see the correctness of n-STR, observe that it computes exactly for each $X = d$, the size of the set $\mathcal{V}(X = d)$ of valid tuples t with $t[X] = d$ and the size of the set $\mathcal{F}(X = d)$ of valid falsifying tuples t with $t[X] = d$. We see that $\mathcal{F}(X = d) \subseteq \mathcal{V}(X = d)$ and moreover the set of supporting tuples $\mathcal{S}(X = d)$ satisfies $\mathcal{S}(X = d) \subseteq \mathcal{V}(X = d)$, $\mathcal{F}(X = d) \cup \mathcal{S}(X = d) = \mathcal{V}(X = d)$ and $\mathcal{S}(X = d) \cap \mathcal{F}(X = d) = \emptyset$. Therefore $|\mathcal{S}(X = d)| = |\mathcal{V}(X = d)| - |\mathcal{F}(X = d)|$.

Algorithm 2: Achieve GAC on an n-STR

```
1  prop-nSTR (c);
   // Propagate The Pruned Values
2  while c.update do
3  |  c.update = false;
4  |  dprod = ∏_{V∈scope(c)} |Dom[V]|;
5  |  foreach X ∈ scope(c) do
6  |  |  c.dprod[X] = dprod / |Dom[X]|;
7  |  foreach n-tuple nT in c.top do
8  |  |  if ¬valid(nT) then
9  |  |  |  foreach X ∈ scope(c) do
10 |  |  |  |  c.ftc[X][nT[X]] -= 1;
   |  |  |  // Move nT to inactive part of c
11 |  |  foreach X ∈ scope(c), d ∈ Dom[X] do
12 |  |  |  if c.ftc[X][d] == c.dprod[X] then
13 |  |  |  |  prune (X,nT[X]);
14 |  |  |  |  c.update = true;
```

We must prune $X = d$ when $|\mathcal{S}(X = d)|$ reaches 0, or $|\mathcal{V}(X = d)| = |\mathcal{F}(X = d)|$. But $|\mathcal{V}(X = d)| = \mathbf{dprod}[X]$ and $|\mathcal{F}(X = d)| = \mathbf{ftc}[X][d]$, so since these counts are maintained correctly, n-STR is sound and enforces GAC.

Let r be the size of the constraint's scope, d the size of the maximum variable domain and T the number of n-tuples in the constraint. The complexity of computing **dprod** is $O(r)$ per iteration (lines 4–6) and the complexity of updating **ftc** is $O(rT)$. The loop can be executed at most $O(rd)$ times, but this is cumulative over an entire branch. This gives a total cumulative complexity over a branch of $O(r^2 + r^2dT)$. Crucially, this complexity is linear in T, although it is worse than that of STR and other algorithms for positive table constraints.

To compute clausal reasons from n-STR is more complex to implement. We chose to implement a simple but non-minimal way of computing reasons. If n-STR c prunes $V = d$ then it is easy to see that the conjunction of all of the other pruned values for the other variables in c's scope is a sufficient reason. For example, if $scope(c) = \{V_1, V_2, V_3\}$, and the value $V_2 = a$, $V_2 = d$, and $V_3 = a$ have been pruned from V_2 and V_3 at the time c prunes $V_1 = d$ (note these values could be, and probably are, pruned by other constraints), then we know that $\neg V_2 = a \wedge \neg V_2 = d \wedge \neg V_3 = a \rightarrow \neg V_1 = d$. This can be encoded in a clause using the $A_{V=d}$ variables. These reasons are not as compact at those produced by positive STR.

5 Empirical Results

To evaluate these three methods, we have implemented them in the CL solver `minicsp` (`http://www7.inra.fr/mia/T/katsirelos/minicsp.html`). The evaluation was performed using XCSP benchmarks (`http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html`). All experiments were performed on a Intel Core 2 Duo, 2.00 GHz.

We experimented with a set of 163 real instances of consisting of mostly crossword puzzles (`lexPuzzle`, `ukPuzzle`, `ogdPuzzle`, `ogdHerald`, `lexHerald`, `driver`, `renault`), a set of 757 patterned instances consisting of mostly graph isomorphism problems (`si2-m4D`, `si4-m4D`, `si4-bvg`, `si6-m4D`, `si6-bvg`, , `BH-4-4`, `BH-4-7`) and a set of 50 randomly generated instances to evaluate the different methods for implementing p-table constraints in `minicsp`.

Figure 1 reports the number of solved real instances and Figure 2 reports the number of solved patterned instances using the CNF support tuple encoding (STE) and the positive STR method (p-STR).[4] We see that although on the real instances the performance is similar, on the patterned instances the clausal STE encoding performs much better (solving more problems in less time).

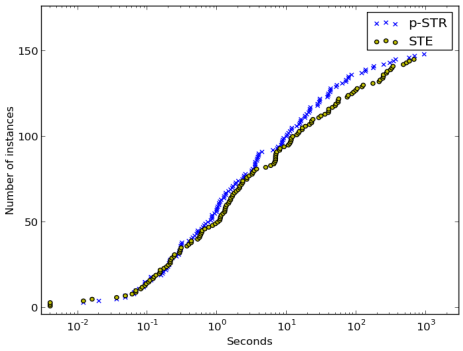


Fig. 1. Real instances

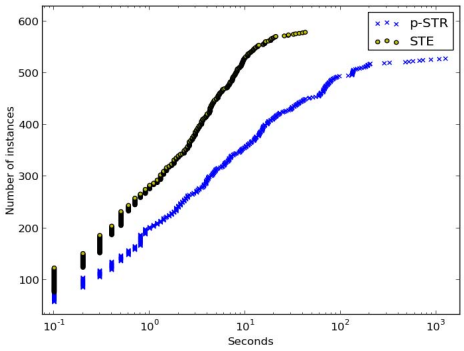


Fig. 2. Patterned instances

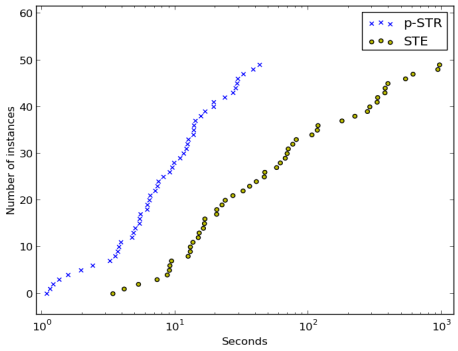


Fig. 3. Random instances

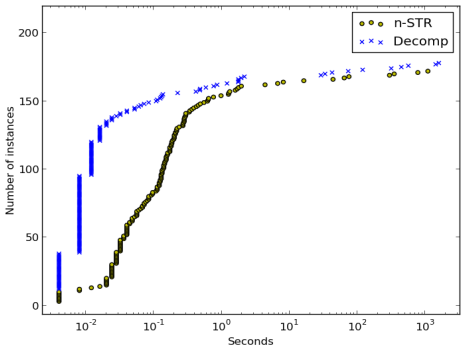


Fig. 4. Instances with negative tables

[4] In addition to the support tuple encoding we also experimented with a MDD clausal encoding. However, we do not report results for this since it was not competitive with the other methods.

The 50 random instances all consisted of 20–25 positive table constraints, each with arity between 5–10 variables, each with domain size 8–9. Each tuple of values was inserted as a p-tuple into the table with probability 0.5, so that these constraints had a tightness of approximately 1/2. Figure 3 show the comparison between our p-STR implementation and the support tuple encoding (STE).

The results reveal that the support tuple encoding shows the best performance for real and patterned instances. A special purpose propagator such as STR yields better results when the table constraints tend to be tight as seen on the random problems. However, none of the XCSP benchmarks we used contain tables with a tightness close to 50%.

Although it is known that the crossword instances lack an underlying structure which can be exploited by conflict analysis [6] they remain an apt choice as benchmark, since a clause learning solver must generate reasons from all of the problems' constraints. Profiling of STR shows that in the crossword instances 60% of the runtime is spent extracting clausal reasons while pruning values. Time spent on this extraction can be as high as 70% in the patterned instances. However, this does not account for the performance difference in patterned instances, as seen in the logarithmic gap between STR and STE is shown in Figure 2. This gap demonstrates that STE would still be faster even if the reasons from STR could be generated for free. We could further confirm this by verifying that the solver generates fewer conflicts when using STE, on average. We conjecture that the additional propositions in the STE encoding allows the solver to generate smaller proofs.

Finally, to evaluate our n-STR approach we experimented on 182 patterned instances (`bqwh-15-106`, `travellingSalesman-20`, `QCP-10`, `cril`, `QWH-10`, `coloring`) specified with n-tables. Since the support tuple encoding works with p-tuples, we tested the n-STR algorithm against a naive encoding where each conflicting tuple is directly represented as a clause. Figure 4 shows the number of solved instances using these two methods, where *n-STR* stands for negative STR and *Decomp* stands for the naive encoding. For n-STR we see that it is generally inferior to the naive encoding on this problem set.

In general, it would seem that once the simplicity of using the clausal encoding is considered for CL solvers a CNF encoding will probably be the best choice for representing table constraints. The results could change, however, depending on future developments: (1) potentially a more memory expensive way of computing better reasons from p-STR tables might benefit p-STRs; (2) better reasons could potentially be computed from n-STRs. These are useful questions for future research.

References

1. Ansótegui, C., Manya, F.: Mapping problems with finite-domain variables to problems with boolean variables. Theory and Applications of Satisfiability Testing pp. 899–899 (2005)
2. Bacchus, F.: Gac via unit propagation. In: Principles and Practice of Constraint Programming (CP). pp. 133–147. Springer-Verlag, New York (2007)

3. Crawford, J.M., Baker, A.B.: Experimental results on the application of satisfiability algorithms to scheduling problems. In: In Proceedings of the Twelfth National Conference on Artificial Intelligence. pp. 1092–1097 (1994)
4. Downing, N., Feydy, T., Stuckey, P.J.: Explaining flow-based propagation. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems (CPAIOR). pp. 146–162 (2012)
5. Gange, G., Stuckey, P.J.: Explaining propagators for s-dnnf circuits. In: Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems (CPAIOR). pp. 195–210 (2012)
6. Gange, G., Stuckey, P.J., Szymanek, R.: Mdd propagators with explanation. Constraints 16(4), 407–429 (Oct 2011), http://dx.doi.org/10.1007/s10601-011-9111-x
7. Gent, I.P., Miguel, I., Moore, N.C.A.: Lazy explanations for constraint propagators. In: Proceedings of the 12th international conference on Practical Aspects of Declarative Languages. pp. 217–233. PADL'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-11503-5_19
8. Hebrard, E., Bessière, C., Walsh, T.: Local consistencies in sat. In: Proceedings of Theory and Applications of Satisfiability Testing (SAT). pp. 400–407 (2003)
9. Katsirelos, G.: Nogood Processing in CSPs. Ph.D. thesis, University of Toronto (2008)
10. Katsirelos, G., Bacchus, F.: Generalized nogoods in csps. In: Proceedings of the AAAI National Conference (AAAI). pp. 390–396 (2005)
11. Lecoutre, C.: Str2: optimized simple tabular reduction for table constraints. Constraints 16(4), 341–371 (2011)
12. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = lazy clause generation. In: Principles and Practice of Constraint Programming (CP). pp. 544–558 (2007)
13. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints 14(3), 357–391 (2009)
14. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. Constraints 16(3), 250–282 (2011)
15. Ullmann, J.R.: Partition search for non-binary constraint satisfaction. Inf. Sci. 177(18), 3639–3678 (Sep 2007), http://dx.doi.org/10.1016/j.ins.2007.03.030

Finite Type Extensions in Constraint Programming

Rafael Caballero[1], Peter J. Stuckey[2] and Antonio Tenorio-Fornés[1]

[1] University Complutense of Madrid
[2] NICTA and the University of Melbourne

Abstract. Many problems are naturally modelled by extending an existing type with additional values. For example for modelling database problems with nulls natural models use booleans and integers with an additional null value. Similarly models involving integers may naturally be extended to handle $-\infty$ and $+\infty$. We extend MiniZinc to MiniZinc$^+$ to allow modelling with extended types. The user can specify both the extension of a predefined type with new values, and the behavior of the operations with relation to the new types. The resulting model MiniZinc$^+$ model is transformed to a MiniZinc model which is equivalent to the original model. We illustrate the usage of MiniZinc$^+$ to model Boolean circuits allowing undefined inputs and scheduling problems considering special time values.

1 Introduction

Constraint programming languages aim at providing mechanisms that allow the user to represent complex problems in a natural way. With that purpose, this paper presents a technique for expressing constraints over extended types in the constraint modelling language MiniZinc [9].

For example, within our framework, it is possible to extend the int predefined MiniZinc domain to support the representation of the value positive infinity. The new type intE is introduced by the reserved word extended:

```
extended intE = int ++ [posInf];
```

where posInf is a new *extended constant*. Once a new extended type has been declared, the user can also define new operations as extensions of the predefined operations allowed by the language. For instance, in this example one could define result of the addition of two intE variables x, y as x+y if both x and y are in the subtype int, or posInf if at least one of the two values is posInf (as in IEEE standard 754 [5]).

Apart from extended arithmetic, the extension of standard domains is an approach used in a multitude of disciplines, such as the design and test of digital circuits [1], the representation of null values to represent the unknown data in database query languages such as SQL [3], or the many-valued logics [8]. All these problems can be successfully modeled in the language proposed in this paper, which we call MiniZinc$^+$.

In order to solve the constraints over the extended types we present a transformation from MiniZinc$^+$ into MiniZinc. The transformation represents each extended decision variable as a pair of variables in MiniZinc. The first variable contains the possible value in the source, standard type. The second variable contains the value in the extended type and also works as a switch that selects one of the two variables during the search. The transformation applies not only for constraint satisfaction problems, but also for optimization problems.

The next section introduces both the syntax of MiniZinc with functions [10] and the syntax of MiniZinc$^+$. Section 3 explains that the transformation is the composition of two phases. The first phase, the elimination of local declarations and functions is decribed in other papers and is not discussed here. The second part is itself split into two section: first, Section 4 introduces the transformation over expressions, and then Section 5 generalizes the transformation to top-level constructions such as constraints and declarations. The soundness of the approach is discussed in Section 6. Finally, Section 7 presents the conclusions and discusses possible future work.

2 Extending MiniZinc

2.1 Syntax

In this section we introduce the syntax of MiniZinc$^+$, our proposed extension of MiniZinc (with functions) [10]:

$$
\begin{aligned}
\text{typeE} &\longrightarrow \textit{extended}\ \textbf{tId} = [c_{-n}, \ldots, c_{-1}] \mathbin{++} \text{type} \mathbin{++} [c_1, \ldots, c_m] \\
\text{exp} &\longrightarrow \textbf{vId} \mid \textbf{constant} \mid \textbf{vId}[\text{exp}] \mid \text{arrayexp}[\text{exp}] \mid \text{setexp} \mid \text{arrayexp} \\
&\quad \mid \textit{if}\ \text{exp}\ \textit{then}\ \text{exp}\ \textit{else}\ \text{exp}\ \textit{endif} \\
&\quad \mid \textbf{vId}(\text{exp}^{*[,]}) \mid \textit{let}\ \{\text{decl}^{*[,]}\ \text{const}^{*[,]}\}\ \textit{in}\ \text{exp} \\
\text{arrayexp} &\longrightarrow [\text{exp}^{*[,]}] \mid [\text{exp} \mid \text{genvar}^{+[,]}\ \textit{where}\ \text{exp}] \\
\text{setexp} &\longrightarrow \{\ \text{exp}^{*[,]}\ \} \mid \text{range} \mid \{\text{exp} \mid \text{genvar}^{+}\textit{where}\ \text{exp}\} \\
\text{genvar} &\longrightarrow \textbf{vId}^{+[,]}\ \textit{in}\ \text{setexp} \mid \textbf{vId}^{+[,]}\ \textit{in}\ \text{arrayexp} \\
\text{range} &\longrightarrow \text{exp} \mathbin{..} \text{exp} \\
\text{decl} &\longrightarrow \text{vtype} : \textbf{vId} \mid \textit{array}[\text{range}]\ \textit{of}\ \text{vtype} : \textbf{vId} \\
&\quad \mid \textit{set of}\ \text{type}: \textbf{vId} \mid \textit{var set of}\ \text{setexp}: \textbf{vId} \\
\text{assig} &\longrightarrow \textbf{vId} = \text{exp} \\
\text{const} &\longrightarrow \textit{constraint}\ \text{exp} \\
\text{funct} &\longrightarrow \textit{function}\ \text{decl}\ (\text{decl}^{*[,]}) = \text{exp} \\
\text{pred} &\longrightarrow \textit{predicate}\ \textbf{vId}(\text{decl}^{*[,]}) = \text{exp} \\
\text{solv} &\longrightarrow \textit{solve satisfy} \mid \textit{solve minimize}\ \text{vId} \mid \textit{solve maximize}\ \text{vId} \\
\text{out} &\longrightarrow \textit{output}\ (\textit{[}\ \text{show}^{*[,]}\ \textit{]}) \\
\text{show} &\longrightarrow \textit{show(}\text{exp}\textit{)} \mid \text{“}\textbf{string}\text{“} \\
\text{type} &\longrightarrow \textit{int} \mid \textit{bool} \mid \textit{float} \mid \textbf{tId} \mid \text{range} \\
\text{vtype} &\longrightarrow \text{type} \mid \textit{var}\ \text{type} \\
\text{model} &\longrightarrow \text{typeE}^{*[;]}; \text{decl}^{*[;]};\ \text{assig}^{*[;]};\ \text{pred}^{*[;]};\ \text{funct}^{*[;]};\ \text{const}^{*[;]};\ \text{solv};\ \text{out}
\end{aligned}
$$

where *model* is the start symbol of the grammar, **vId** and **tId** are identifiers for: parameters, variables, functions, or predicates and new types, respectively.

string represents an arbitrary string constant. The values c_i represent new constant identifiers. The notation $n^{*[s]}$ / $n^{+[s]}$ indicates zero or more / one or more repetitions of the nonterminal n such that these repetitions are separated by string s. *Italic* words are reserved words of the language. The only difference of this grammar with respect to the standard MiniZinc with functions presented in [10] is the new nonterminal *typeE* and the inclusion of type identifiers (**tId**) as possible types.

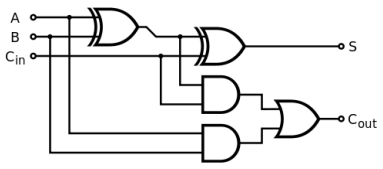
2.2 Example: Extending the Boolean type for a full adder combinational circuit

Suppose that we wish to extend the Boolean type with a new constant *undef* in order to model combinational circuits with undefined (i.e. neither true or false) signals [1]. The definition in MiniZinc$^+$ of the new type can be found in the first line of the model in Figure 1. Note that replacing `boolEx` with `bool` in lines (3-6) and omitting lines (8-24) would give a standard MiniZinc model for this problem.

The model redefines the behavior of the Boolean connectives $\wedge$, $\vee$ and *xor* taking into account the new constant as indicated in the truth tables of Figure 2 (where 0 stands for *false*, 1 for *true* and $\perp$ stands for *undef*). For instance, the standard MiniZinc operator *xor* is redefined in MiniZinc$^+$ as shown in lines (8-11) of Figure 1. The function first defines a local decision variable *c1*, which uses the predefined function *sv* in order to check if both parameters *a* and *b* contain standard values, that is, values different from *undef*. If this is the case, then the function returns the result of using the standard MiniZinc operator *xor*, represented by *predef(xor)*. Otherwise, if either *a* or *b* is *undef*, then the result is *undef* according to the table for extended *xor* of Figure 2. The schema of this function will be usual in all the *conservative* redefinition of standard operators. The code for functions redefining $\wedge$ and $\vee$ is analogous.

Note that although the functions xor, $\wedge$ and $\vee$ have been redefined, they are used as the original functions inside function declarations by wrapping them with `predef`.

Using these definitions we model the behavior of a n-bit adder digital circuit in lines (25-28). The basic piece of the circuit is the *full adder*:



which adds binary numbers and accounts for values carried in as well as out. The code of lines (25-28) employs n full adders to obtain a n-bit adder. In particular, line (26) defines the output s using two *xor* gates, while lines (27-28) model the carries employing two *and* and one *or* gates.

```
1 extended boolEx = bool ++ [undef];
2 int n;
3 array[1..n]   of var boolEx: x;
4 array[1..n]   of var boolEx: y;
5 array[1..n+1] of var boolEx: s;
6 array[1..n+1] of var boolEx: c;
7
8 function var boolEx:xor(var boolEx:a, var boolEx:b) =
9   let{var boolEx:r, var bool:c1=sv([a,b]),
10       constraint (c1 /\ r = (a predef(xor) b)) \/
11                  (not c1 /\ r=undef)} in r;
12 function var boolEx:/\(var boolEx:a, var boolEx:b) =
13  let{var boolEx:r, var bool:c1=sv([a,b]),
14      var bool:c2= (a=false \/ b=false),
15      constraint (c1 /\ r = (a predef(/\) b)) \/
16                 (not c1 /\ c2 /\ r=false) \/
17                 (not c1 /\ not c2 /\ r= undef)} in r;
18 function var boolEx:\/(var boolEx:a, var boolEx:b) =
19  let{var boolEx:r, var bool:c1=sv([a,b]),
20      var bool:c2= (a=true \/ b=true),
21      constraint (c1 /\ r = (a predef(\/) b)) \/
22                 (not c1 /\ c2 /\ r=true) \/
23                 (not c1 /\ not c2 /\ r=undef)} in r;
24
25 constraint c[1]=false /\ s[n+1]=c[n+1]
26 constraint forall([s[i]=x[i] xor y[i] xor c[i]|i in 1..n])
27 constraint forall([c[i+1]=(x[i] /\ y[i]) \/
28                    ((x[i] xor y[i]) /\ c[i])|i in 1..n]);
29 solve satisfy;
```

Fig. 1: A n bit full adder in MiniZinc$^+$: $x + y = s$

After transforming this model into an standard MiniZinc model, we can use MiniZinc to obtain solutions such as the following:[3]

$$\begin{array}{l} \mathrm{x} = 1\ \bot\ 0\ 1 \\ \mathrm{y} = 1\ \bot\ 0\ 0 \\ \underline{\mathrm{c} = 0\ 1\ \bot\ 0\ 0} \\ \mathrm{s} = 0\ \bot\ \bot\ 1\ 0 \end{array}$$

The least significant digit (and thus the first position of each array) is displayed on the left. Observe that in the second position from the left the addition $\bot + \bot + 1$ (1 is the carry from the previous position) yields $\bot$ in the result. In

[3] The *output* sentence is omitted in Figure 1 for simplicity.

	1	**0**	$\bot$
1	1	1	1
0	1	0	$\bot$
$\bot$	1	$\bot$	$\bot$

(a) $\vee$

	1	**0**	$\bot$
1	1	0	$\bot$
0	0	0	0
$\bot$	$\bot$	0	$\bot$

(b) $\wedge$

	1	**0**	$\bot$
1	0	1	$\bot$
0	1	0	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

(c) xor

Fig. 2: Truth tables including the undefined value

particular this means that the carry is undefined as well, and thus in the third position $0 + 0 + \bot$ produces the output $\bot$. However, in this case we can ensure that the carry is 0, and thus in the fourth position we have $1 + 0 + 0 = 0$ as output with 0 carry and as last bit.

3 From MiniZinc$^+$ to MiniZinc

The main goal of this paper is to present an automatic translation from MiniZinc$^+$ to MiniZinc. Thanks to this translation, the models written in the extended setting can be solved using all the features (optimizations, different types of solvers, etc.) included in MiniZinc. The translation can be presented as a process in two phases:

1. First, functions, predicates and local declarations of variables are removed from the model.
2. Finally, the resulting MiniZinc$^+$ model, now containing neither functions nor local declarations, is translated into MiniZinc.

Observe that the first phase can be applied to both MiniZinc and MiniZinc$^+$ indistinctly. In particular, the function elimination is done by unrolling the function calls following ideas similar to those described in [10] (we assume in our setting the use of *total* functions), which simplifies the task. The elimination of constraints included in local declarations is managed using the relational semantics [4] of MiniZinc where these constraints "float" to the nearest enclosing Boolean context where they are added as a conjunct. Analogously, the local variable declarations are converted to global variable declarations, see [6] for a more detailed discussion.

In the rest of the paper we describe the second phase, which converts a MiniZinc$^+$ model without functions and local declarations into a semantically equivalent MiniZinc model.

4 Transforming MiniZinc$^+$ expressions

In the case of MiniZinc$^+$ expressions, the transformation is defined in terms of two auxiliary transformations, the first one representing the standard MiniZinc part of the expression (transformation $\tau_s(c)$), and the second one keeping a representation of the extended part (transformation $\tau_e(c)$).

4.1 Notation

First we introduce some auxiliary notation:

We use t for type identifiers (either standard as bool, int and float or extended such as bEx). The functions $st(t)$ and $et(t)$ return whether t is either a standard (st) or an extended (et) type.

The notation $\mathsf{ord}_t(\mathsf{k})$ maps constants k of type t to an integer that represents the *distance* to k from the base type following the textual order in its definition (the sub-index t in ord is omitted when it is clear from the context). For instance, given the definition

```
extended int3 = [negInf]++int++[undef,posInf];
```

then: $\mathsf{ord}_{int3}(\mathsf{negInf}) = \mathsf{-1}$, $\mathsf{ord}_{int3}(\mathsf{undef}) = \mathsf{1}$, and $\mathsf{ord}_{int3}(\mathsf{posInf}) = \mathsf{2}$.

For every constant k, $\mathsf{ord}_t(k) \neq \mathsf{0}$ iff k is extended. We define $\mathsf{ord}_t(k) = 0$ if k is a standard constant. The function $\mathsf{eRan}(t)$ (extended Range) is defined for an extended type t as follows: define a set S as $S = \{\mathtt{ord}_t(k) | k \in t\} \cup \{0\}$, then $\mathtt{eRan}(t) = \min(S) \,..\, \max(S)$. In the example of int3 above: $-1 \,..\, 2$. We choose for each type t a *default value* $k_{o(t)}$ which will be used in the representation of extended constants. The notation $o(t)$ refers to the base type of t if it is extended, or to t itself otherwise. Additionally, for each type t we define a value z_t, which is 0 if t is an atomic type, the array of n zeros ($[0, \ldots, 0]$) if t is an array of size n, the empty set ($\{\}$) if t is a set, and the minimum value in the base type in the case of an integer subrange. In the rest of the paper we assume that MiniZinc$^+$ models are well-typed following the type inference rules for MiniZinc which can be found in [2], and use the notation *type(e)* to refer to the type of e.

Next we explain the transformation of MiniZinc$^+$ expressions, distinguishing between the different possibilities enunciated in the grammar (Section 2.1).

4.2 Identifiers, constants, array and set expressions

Base identifiers and constants The transformations τ_s and τ_e for identifiers and constants of base types are defined as follows:

	τ_s	τ_e
Identifiers : x, $t = type(x)$		
$st(t)$	x	z_t
$et(t)$	$s(x)$	$e(x)$
Constants : k, $t = type(k)$		
$st(t)$	k	z_t
$et(t)$	$k_{o(t)}$	$\mathtt{ord}_t(k)$

Observe that here identifiers represent both decision variables and parameters. Identifiers of standard type are mapped to the original form, with the second component fixed to zero, representing a standard value. Extended type identifiers x are mapped to the associated new identifiers $s(x)$ of type t and $e(x)$ of type $\mathsf{eRan}(t)$. Constants are mapped to themselves paired with z_t if standard, or to the default constant from the underlying type and their order number if they are extended, new values.

Array expressions and identifiers Array expressions of the form: $e = [e_1, \ldots, e_n]$ are transformed simply mapping the transformations τ_s, τ_e:

$$\tau_s(e) = [\tau_s(e_1), \ldots, \tau_s(e_n)] \quad \tau_e(e) = [\tau_e(e_1), \ldots, \tau_e(e_n)]$$

For instance, if we consider the array expression e defined as [*true,false,undef*], then $\tau_s(e)$ = [*true,false,false*], and $\tau_e(e)$ = [*0,0,1*]. Observe that the underlined *false* corresponds to the arbitrary constant k_{bool} chosen to replace *undef* and it is only used to keep the array with the same length and with the standard constants in the same positions. Array identifiers a always have known length after unfolding so they are treated analogously to array expressions: e.g. $a = [a[1], \ldots, a[n]]$ means that $\tau_s(a) = [\tau_s(a[1]), \ldots, \tau_s(a[n])]$ and $\tau_e(a) = [\tau_e(a[1]), \ldots, \tau_e(a[n])]$.

Array access An array access of the form $a[exp]$ with $type(a) = <array\ of\ t>$ is transformed as:

$$\tau_s(a[exp]) = \tau_s(a)[\tau_s(exp)] \quad \tau_e(a[exp]) = \tau_e(a)[\tau_s(exp)]$$

We make use of the fact that MiniZinc arrays are always indexed by integers. Consider the subexpression `c[1]` in line 25 of Figure 1. We have $type(c) = <array\ of\ bEx>$, and thus *st(bEx)* is false and *et(bEx)* holds. Therefore, $\tau_s(c[1]) = cs[1]$, $\tau_e(c[1]) = ce[1]$, assuming $s(c[1])$ is defined as the new identifier cs[1] and $e(c[1])$ as ce[1].[4]

Set expressions [5] Set expressions of the form $e = \{ e_1, \ldots, e_n \}$ with $type(e_1) = \cdots = type(e_n) = t$ are transformed depending on the type t:

- if $st(t)$, then $\tau_s(e) = \{ \tau_s(e_1), \ldots, \tau_s(e_n) \}$, and $\tau_e(e) = \{\}$
- if $et(t)$, then $\tau_s(e) = \{[\tau_s(e_1), \ldots, \tau_s(e_n)][\text{i}] \mid \text{i in 1..n}$ where $[\tau_e(e_1), \ldots, \tau_e(e_n)][\text{i}] = 0 \}$
  $\tau_e(e) = \{[\tau_e(e_1), \ldots, \tau_e(e_n)][\text{i}] \mid \text{i in 1..n}$ where $[\tau_e(e_1), \ldots, \tau_e(e_n)][\text{i}] \mathrel{!}= 0 \}$

The overall idea is that the elements in the set are split into standard and extended parts.

4.3 Array and set comprehensions

Let $\langle$ exp | genvars where cond $\rangle$ be an array or set comprehension (with $\langle$,$\rangle$ representing [,] or {,}). The translation of this expression consists of two phases. The first phase processes each generator g in genvars. We use the notation $e[x \mapsto e']$ to indicate that all the occurrences of x in e must be replaced by e'.

- If $g \equiv$ gId in genExp with genExp a set or array of standard type, then apply the replacement genvars[g $\mapsto$ gId in τ_s(genExp)].

[4] For simplicity we use the suffixes s and e to generate new identifiers for the standard and extension parts of a construction in the rest of the paper.

[5] For set variables see Section 5.2.

– If g is of the form gId in arrayexp and arrayexp is an array of extended type then:
  - Apply the replacement
    genvars[g $\mapsto$ f in index_set(τ_s(arrayexp))], where f is a fresh variable.
  - Apply the replacements
    exp[gId $\mapsto$ arrayexp[f]] and cond[gId $\mapsto$ arrayexp[f]]
– If $g \equiv$ gId in setexp and setexp is a set of extended type then: Let fresh array expression a be [ord_t^{-1}(x) | x in τ_e(setexp) where x<0]++ [x | x in τ_s(setexp)]++ [ord_t^{-1}(x) | x in τ_e(setexp) where x>0].
  Then:
  - Apply the replacement genvars[g $\mapsto$ f in index_set(τ_s(a))], where f is a fresh variable.
  - Apply the replacements exp[gId $\mapsto$ a[f]] and cond[gId $\mapsto$ a[f]]

Let $\langle$(exp') | genvars' where cond' $\rangle$ be the result of applying this transformation to all the generators in the array/set comprehension. Then, the second phase of the translation is defined as:
- Array comprehensions:
τ_s = [τ_s(exp') | genvars' where τ_s(cond')]
τ_e = [τ_e(exp') | genvars' where τ_s(cond')]
- Set comprehensions:
τ_s = { τ_s(exp') | genvars' where τ_s(cond') $\wedge$ τ_e(exp)=0 }
τ_e = { τ_e(exp') | genvars' where τ_s(cond') $\wedge$ τ_e(exp)!=0 }

For example, let *intE* be the integer type extended with constant *posInf*, and consider the following expression:

```
e = [ y | x in [posInf, 4, 9, -1], y in {8, -1, 8, posInf}
          where x=y]
```

In order to simplify the presentation we use L to represent the list [posInf, 4, 9, -1], and S to represent the set {8, -1, 8, posInf}. Therefore, the array comprehension is represented as [y | x in L, y in S where x=y].

First we select the first generator x in L, choosing i as new variable and taking into account that τ_s(L) = [0, 4, 9, -1]. Applying the replacements we obtain [y | i in index_set([0,4,9,-1]), y in S where L[i]=y].

The second generator is y in S. Attending to the translation of set expressions we have
$\tau_s(S)$ =[[8,-1,8,0][i] | i in 1..4 where [0,0,0,1]=0]
$\tau_e(S)$ =[[0,0,0,1][i]| i in 1..4 where [0,0,0,1]!=0]

Then the array expression a is defined as:
a = [ord$_t^{-1}$(x) | x in τ_e(S) where x < 0] ++
[x | x in τ_s(S)] ++
[ord$_t^{-1}$(x) | x in τ_e(S) where x >0]

Observe that during the evaluation of the model a will be evaluated to []++[-1,8]++[posInf] = [-1,8,posInf]. The idea behind a is to obtain the list of elements in S without repetitions and respecting the order among elements. This mimics in MiniZinc$^+$ the behaviour of MiniZinc where [x | x in {3,4,5,3,4}] is evaluated to [3,4,5].

The translation proceeds by replacing the second generator by a new variable j, obtaining
[a[j] | i in index_set([0,4,9,-1]), j in index_set(τ_s(a)) where L[i]=a[j]]

Finally:
$\tau_s(e)$= [τ_s(a[j])|i in index_set([0,4,9,-1]), j in index_set(τ_s(a)) where τ_s(L[i]=a[j])]

and
$\tau_e(e)$= [τ_e(a[j])|i in index_set([0,4,9,-1]), j in index_set(τ_s(a)) where τ_s (L[i]=a[j])]

During the evaluation the system will obtain: τ_s(e) = [0,-1], and τ_e(e) = [1,0], which corresponds to the MiniZinc representation of the MiniZinc$^+$ list [posInf,-1].

4.4 Conditional and logical expressions

Expressions $e \equiv$ if c then e1 else e2 endif are transformed as:
τ_s(e) = if τ_s(c) then τ_s(e1) else τ_s(e2) endif
τ_e(e) = if τ_s(c) then τ_e(e1) else τ_e(e2) endif

Note: the exists and forall constructions are simply expanded to disjunctions and conjunctions respectively and then transformed.

4.5 Predefined function and predicate calls

We consider the following predefined function and predicate calls:
- $c \equiv$ sv([exp$_1$,...,exp$_n$]). The purpose of this Boolean function is to ensure that all the expressions correspond to standard values. Therefore: $\tau_s(c) = (\tau_e(\text{exp}_1)=z_{t_1}) \wedge \cdots \wedge (\tau_e(\text{exp}_n)=z_{o(t_n)})$, with $z_{o(t_i)}$ the zero value associated to the type t_i of expression exp$_i$.

- $c \equiv$ predef(f)(exp$_1$, ..., exp$_n$), or alternatively $c \equiv$ exp$_1$ predef(f) exp$_2$, with f a predefined function or an infix operator. predef indicates that this call corresponds to the predefined MiniZinc function/operator f even if it has been redefined by the user. Thus, τ_s(c)= f(τ_s(exp$_1$), ..., τ_s(exp$_n$)), or τ_s(c)=τ_s(exp$_1$) f τ_s(exp$_2$) if f is an infix operator, and τ_e(c) = z_t where t is the output type of f. Thus, the user should ensure, usually by adding some constraints using sv that exp$_1$, ..., exp$_n$ can only correspond to standard values, otherwise the result of evaluating this function can be unsound.

- $c \equiv$ (exp $_1$ = exp$_2$), assuming that = has not been redefined. Then: $\tau_s(c)$ is defined as

$$(\tau_s(\text{exp}_1) = \tau_s(\text{exp}_2) \wedge \tau_e(\text{exp}_1) = \tau_e(\text{exp}_2))$$

and $\tau_e(c)$ is defined as z_{bool}. The result of the comparison depends both on the standard and on the extended value. It is not enough to check only the standard part, because in case of two different extended constants a, b with base type t we have $\tau_s(b) = \tau_s(a) = k_t$, but the result should be false. Analogously, the extended part is not enough because for instance considering the standard constants 3, 4, we have $\tau_e(3) = \tau_e(4) = z_{bool}$. The translation of exp$_1$!= exp$_2$ is simply not(exp$_1$ = exp$_2$), applying then the translation of =.

- $c \equiv (e\ in\ S)$, assuming that *in* has not been redefined. Then: $\tau_s(c) = (\tau_e(e) = 0 \wedge \tau_s(e)\ in\ \tau_s(S)) \vee (\tau_e(e) \neq 0 \wedge \tau_e(e)\ in\ \tau_e(S))$ and $\tau_e(c) = 0$. Other set operations such as card, union or intersect can be defined analogously.

This ends the transformation part for expressions. It only remains to define the transformation applied to top-level constructions.

5 Transforming MiniZinc$^+$ models

The transformation of a MiniZinc$^+$ model $\mathcal{M}$, denoted by $\tau(\mathcal{M})$ is obtained transforming each of these top-level constructions as described in this section.

5.1 Declarations of extended types

The declarations of extended types are useful for obtaining the names of the new types, their base standard types, the names of the extended constants, and for generating the ord function described above. However, these declarations do not generate directly any code in the transformed MiniZinc model.

5.2 Declarations of variables and parameters

If $c \equiv decl$ is a declaration of a variable or a parameter, then it is translated to MiniZinc as $c^{\mathcal{T}} \equiv \tau(decl)$ as defined by the following table:

	τ
Var. or param. declarations: [var] $t : x$, with $o(t) \in$ {int, float, bool }	
$st(t)$	[var] $t : x$
$et(t)$	[var] o(t): $s(x)$; [var] eRan(t): $e(x)$; C_1
array [S] of [var] t: a	
$st(t)$	array [S] of [var] t: a;
$et(t)$	array [S] of [var] $o(t)$: $s(a)$; array [S] of [var] eRan(t): $e(a)$; C_2
set of t: x	
$st(t)$	set of t: x;
$et(t)$	set of o(t): $s(x)$; set of eRan(t) : $e(x)$
var set of $setexp : x$, $type(setexp) = <$set of $t >$	
t=int	var set of $setexp : x$
$et(t)$	var set of $\tau_s(setexp) : s(x)$; var set of $\tau_e(setexp) : e(x)$;

with the constraints C_1 and C_2 defined as

$$C_1 \equiv \text{constraint } e(x) = z_{o(t)} \text{ -> } s(x) = k_{o(t)};$$

and

$$C_2 \equiv \text{constraint forall([}e(a[i]) \neq z_{o(t)} \text{ ->} s(a[i]) = k_{o(t)} \mid \text{i in S]);}$$

The first column of the table distinguishes the different possible cases. The constraints C_1 and C_2 are introduced to avoid the repetition of equivalent solutions that is produced if the standard variables are not constrained. This is done, by ensuring that if the variable takes an extended value (extended part $\neq z_{o(t)}$), then the standard part of the variable takes some arbitrary value $k_{o(t)}$.

In our circuit example, the array x is transformed into:

```
array[1..n] of var bool: xs;
array[1..n] of var 0..1: xe;
constraint forall([xe[i]!=0 -> xs[i]=false | i in 1..n]);
```

assuming that false is the arbitrary constant k_{bool}.

5.3 Assignments and Constraints

Assignments of the form $c \equiv vId = exp$, with $type(vId) = t$ are transformed as follows:

	τ
$st(t)$	$vId = \tau_s(exp)$
$et(t)$	$\tau_s(vId) = \tau_s(exp)$; $\tau_e(vId) = \tau_e(exp)$

Thus, the idea is to constrain the standard (respectively extended) part of the identifier to the standard (respectively extended) part of the expression.

Constraints have the form $c \equiv$ constraint *exp*;, where *exp* is a Boolean expression. In this case the transformation simply takes into account that the type of *exp* is standard, and therefore $c^{\mathcal{T}} \equiv$ constraint $\tau_s(exp)$.

5.4 Output Item

The translation of an output item adds a new requirement, being able to print extended types. An expression of the form show(exp) must return a string representing the possibly extended expression exp. An extended type definition of the form

extended tId [c_{-n}, ..., c_{-1}]++type++ [c_1, ..., c_m];

creates an array of string tnames

array[eRan(tId)] of string: tnames = [c_{-n}, ..., c_{-1}, "dummy", c_1, ..., c_m];

and replaces each show(e) by

if(fix($\tau_e(e)$)==0) then show($\tau_s(e)$) else show(tnames[τ_e(e)]) endif

For example output [show(x)]; where x is of type int3 creates

```
array[-1..2] of string: int3names =
        ["neginf","dummy","undef","posInf"];
output [ if (fix(xe) == 0) then show(xs)
         else show(int3names[xe]) endif ];
```

5.5 Satisfaction and Optimization

A *satisfaction problem* is encoded in MiniZinc$^+$ using the solve item solve satisfy. In the translation to MiniZinc this is unchanged.

MiniZinc also allows defining *optimization problems*, using solve minimize e or solve maximize e. In MiniZinc$^+$ we also allow the optimization of expressions with extended range, extending implicitly the order $<$ to the new elements accordingly to their position with respect to the standard type in the definition of the type extension (see Section 4).

In standard MiniZinc, the optimization of an arithmetic expression is treated as the optimization of a variable constrained to be equal to the expression. Thus we consider goals either of the form *solve minimize y;* or *solve maximize y;* with y a variable of some extended type t.

In order to compare values k of extended types in the transformation we consider the lexicographical ordering over pairs of the form $(\tau_e(k), \tau_s(k))$. Let a be the minimum base type value in t if this exists, and b be the maximum base type value in t if this exists. If a and/or b dont exist then we may be able to determine $a = \min(\tau_s(y))$ and $b = \max(\tau_s(y))$. As a last resort, if we are to use a solver which artificially represents unbounded objects of the base type in a finite range $a..b$ we can use these values. Note that most finite domain solvers have this restriction. If we cannot determine either a or b then the optimization cannot be translated.[6] Given a and b can be determined we transform minimize/maximize y to minimize/maximize $\tau_e(y) * (b - a + 1) + \tau_s(y)$.

For instance, the example in Figure 3 models the time required to perform some task. The time is measured in hours, from 0 to 23, plus an especial value *oneDayOrMore*. The addition operator $+$ is redefined accordingly, ensuring that if the sum of the two values exceeds 23 then the value *oneDayOrMore* is returned. For this type $a = 0$ and $b = 23$.

In the example, the sum of the values of the parameters exceed 23 hours, and therefore even assuming the minimum possible value for c (which is 0), the expression takes the value *oneDayOrMore*. After transforming the model MiniZinc yields the expected values for variables *total* and c:

```
Total=oneDayOrMore t2=0
```

6 Theoretical results

In this section we present the theoretical result that supports our proposal. The idea is to prove that both the MiniZinc$^+$ and its transformation represent the same set of solutions. The solutions are represented by well-typed substitutions:

Definition 1. *Let $\mathcal{M}$ be a MiniZinc$^+$ model, Γ its associated type context, and θ a substitution. We say that θ is a* well-typed *substitution for $\mathcal{M}$ iff*

[6] We are aiming to extend MiniZinc to directly handle lexicographic objectives, in which case this problem would disappear.

```
extended time = (0..23) ++ [oneDayOrMore];

function var time:+(var time:x, var time:y) =
 let {var time:r, var bool:c=sv([x,y]),
     constraint (c /\ x + y>23 /\ r=oneDayOrMore)
                \/ (c /\ x + y<=23 /\ r=x+y ) \/
                (not c /\ r=oneDayOrMore) } in r;
time: t1 = 5;
var time:t2;
var time:total = t1 + t2 + 21;
solve minimize total;
output(["Total=",show(total)," t2=",show(t2),"\n"]);
```

Fig. 3: Modelling time with an extended value..

- *The domain of θ is the set containing the decision variables declared in $\mathcal{M}$.*[7]
- *For all $x \in dom(\theta)$, $type(x) =< t >$ iff $type(x\theta) =< t >$*

The key idea for defining the concept of solution is the evaluation of an expression in a model with respect to a given well-typed substitution.

Definition 2. *Let $\mathcal{M}$ be a MiniZinc$^+$ model, e an expression occurring in $\mathcal{M}$, and θ be a well-typed substitution for $\mathcal{M}$. The* evaluation of e with respect to θ, *denoted by $\| e \|_\theta$, is defined distinguishing cases according to the definition of MiniZinc$^+$ expressions (refer to non-terminal* exp *in the grammar)*

1. $\| id \|_\theta = id\theta$, id *any identifier.*
2. $\| k \|_\theta = k$, k *any constant.*
3. *Set Expressions:*
   (a) $\| \{e_1, \ldots, e_n\} \|_\theta = ord(\{\| e_1 \|_\theta, \ldots, \| e_n \|_\theta\})$.
   ord is defined as the function that given a set of values, eliminate the repetitions and sort the values according to order $\preceq$ that extends ord_t defined in Section 4.1 where:

$$a \preceq b = \begin{cases} a \leq b & a, b \text{ standard constants} \\ ord_t(a) < 0 & a \text{ extended}, b \text{ standard} \\ ord_t(b) > 0 & a \text{ standard}, b \text{ extended} \\ ord_t(a) \leq ord_t(b) & \text{otherwise} \end{cases}$$

   (b) $\| e_i..e_f \|_\theta = \{\| e_i \|_\theta, \| e_i \|_\theta + 1, \ldots, \| e_f \|_\theta\}$
4. *Array Expressions:* $\| [e_1, \ldots, e_n] \|_\theta = [\| e_1 \|_\theta, \ldots, \| e_n \|_\theta]$
5. *Array Access:*

[7] The decision variables are the variables declared either at top level, or in local *let* statements. The parameter names in the declarations of user functions and predicates are *not* considered decision variables in our setting.

   (a) $\| a[e] \|_\theta = t_i$, *with* a *an array identifier with index range* $m \dots n$, $i = \| e \|_\theta - m + 1$, $1 \leq i \leq n - m + 1$, *and* $\| a \|_\theta = [t_1, \dots, t_{n-m+1}]$.
   (b) $\| e_1[e_2] \|_\theta = t_i$, *with* e_1 *not an array identifier,* $\| e_1 \|_\theta = [t_1, \dots, t_n]$, *and* $i = \| e_2 \|_\theta$, $1 \leq i \leq n$.
6. *Set/list comprehensions of the form* $lc = \langle e \mid g_1, \dots, g_m$ *where* $c \rangle$, *where:*
   (a) $\langle , \rangle$ *represents either {,} or [,].*
   (b) g_j *is of the form* id_j *in arrayexp or* id_j *in setexp.*
   (c) *In particular we suppose that* $g_1 \equiv id$ *in* e'. *Let* $\| e' \|_\sigma$ *be* $\langle\ e_1, \dots, e_n\ \rangle$ *and define* $\sigma_1 = \sigma \uplus \{id \mapsto e_1\}$, $\dots$, $\sigma_n = \sigma \uplus \{id \mapsto e_n\}$.

   Moreover, in the definition we use the following notation:
   - $\diamond$ *represents the array concatenation or set union depending on what* $\langle , \rangle$ *is representing.*
   - $\mathcal{C}(e, c)$ *being* $\langle e \rangle$ *if* c *holds and* $\langle\rangle$ *in other case.*

   Then, $\| lc \|_\theta$ *is defined recursively as:*
   (a) *If* $m = 1$, *then* lc *contains only one generator* g, *which must be of the form* id *in* e'. *Then:*
   $\| \langle e \mid g$ *where* $c \rangle \|_\sigma = \mathcal{C}(\| e \|_{\sigma_1}, \| c \|_{\sigma_1})\ \diamond \dots \diamond\ \mathcal{C}(\| e \|_{\sigma_n}, \| c \|_{\sigma_n})$
   (b) *If* $m > 1$ *then* lc *contains more than one generator. Analogously to the previous item, suppose that the first generator is* g_1. *Then:*
   $\| \langle e \mid g_1, \dots, g_m$ *where* $c \rangle \|_\sigma =$
   $\| \langle e \mid g_2 \dots, g_m$ *where* $c \rangle \|_{\sigma_1}\ \diamond \dots \diamond\ \| \langle e \mid g_2 \dots, g_m$ *where* $c \rangle \|_{\sigma_n}\}$
7. $\| sv([e_1, \dots, e_n]) \|_\theta = st(t_1) \wedge \dots \wedge st(t_n)$ *with* $\Gamma \vdash \| e_1 \|_\theta :: t_1, \Gamma \vdash \| e_n \|_\theta :: t_n$
8. $\| e_1 = e_2 \|_\theta = true$ *if* $\| e_1 \|_\theta$ *and* $\| e_2 \|_\theta$ *are the same constant, false otherwise.*
9. $\| p(e_1, \dots, e_n) \|_\theta = p(\| e_1 \|_\theta, \dots, \| e_n \|_\theta)$, *with* p MiniZinc *predefined (that* p *is a relational operator or predefined arithmetic function such as* $>$,$<$,$+$ $\dots$ *)* .
10. Forall, exists *constructions:*
    Let $\| a \|_\theta$ *be* $[v_1, \dots, v_n]$, *then:*
    - $\| forall(a) \|_\theta = v_1 \wedge \dots \wedge v_n$
    - $\| exists(a) \|_\theta = v_1 \vee \dots \vee v_n$

Thus, the overall idea is simply to evaluate the expressions after replacing the variables by their values. Now we can define the concept of solution.

Definition 3. *Let* $\mathcal{M}$ *be a MiniZinc*$^+$*model,* $\mathcal{M} = T; D; A; P; F; C; S$, *with* T *the sequence of type extensions declarations,* D *a sequence of declarations,* A *a sequence of assignments,* C *a sequence of constraints, and* S *the* solve *statement. Let* θ *be a well-typed substitution for* $\mathcal{M}$. *Then, we say that* θ *is a solution of* $\mathcal{M}$ *if:*

1. *For every assignment* a *in* A, $\| a \|_\theta = true$.
2. *For every constraint* constraint c *in* C, $\| c \|_\theta = true$.
3. *If* S *is of the form* maximize f *(respectively* minimize f*) then there is no well-typed substitution* σ *for* $\mathcal{M}$ *verifying 1) and 2) and such that* $f\sigma > f\theta$ *(respectively* $f\sigma < f\theta$*)*

Definition 4. *Let $\mathcal{M}$ be a MiniZinc$^+$ model and σ be a well-typed substitution of $\mathcal{M}$, then,*

$$\sigma^{\mathcal{T}} = \{\tau_s(x) \mapsto \tau_s(v) \mid (x \mapsto v) \in \sigma\} \cup \\ \{\tau_e(x) \mapsto \tau_e(v) \mid (x \mapsto v) \in \sigma,\ \Gamma \vdash x :: t,\ \tau_e(x) \neq z_t\}$$

Finally, we can establish the theoretical result.

Theorem 1. *A well-typed substitution θ is solution of a* MiniZinc$^+$ *model $\mathcal{M}$ iff $\theta^{\mathcal{T}}$ is well-typed solution of $\mathcal{M}^{\mathcal{T}}$.*

Proof Idea

We must check that both θ verifies the three items of Definition 4 with respect to $\mathcal{M}$ iff $\theta^{\mathcal{T}}$ verifies the same Definition with respect to $\mathcal{M}^{\mathcal{T}}$.

For items 1 and 2, the result is a consequence of a similar auxiliary lemma applied to expressions:

For every expression e and well-typed substitution θ:

- $\| \tau_s(\| e \|_\theta) \|_{id} = \| \tau_s(e) \|_{\theta^{\mathcal{T}}}$
- $\| \tau_e(\| e \|_\theta) \|_{id} = \| \tau_e(e) \|_{\theta^{\mathcal{T}}}$

where id represents the identity substitution. These results can be proven using structural induction on the form of e.

Analogously, item 3 requires a generalization of the following result: *For every pair of constants k, k' of some type t in $\mathcal{M}$ $k \leq k'$ (with the order $<$ extended to the new types) iff*

$$\tau_e(k) * (b - a + 1) + \tau_s(k) \leq \tau_e(k') * (b - a + 1) + \tau_s(k')$$

where a and b are respectively the minimum and the maximum constants in the base type for t. A detailed proof can be found in [2].

7 Conclusions and Future Work

The possibility of extending predefined types with new constants allows the representation of many constraint satisfaction problems in a more natural way. Some examples are models representing circuits including undefined entries (representing for instance failing connections), database problems including *null* values, problems that can be modelled using many-valued logics, or scheduling problems with optional tasks.[8]

The system MiniZinc$^+$ presented in this paper extends the constraint system MiniZinc to include this feature. The modeller can define new types by adding new constants to already existing types, and redefine accordingly the behaviour of the predefined operations. We present a model transformation that converts the models in the new system into a standard MiniZinc model. Thus, all the

[8] Although for these scheduling problems there are approaches [7] which support stronger propagation.

facilities included in MiniZinc such as intensional lists, local definitions, sets, or predicates are available in the new setting.

As future work we plan to allow the possibility of extending already extended types. The framework will give rise to lattices of extensions and will allow modelling more complex problems.

References

1. F. Azevedo. Thesis: Constraint solving over multi-valued logics - application to digital circuits. *AI Commun.*, 16(2):125–127, 2003.
2. R. Caballero, P. J. Stuckey, and A. Tenorio-Fornés. Finite Type Extensions in Constraint Programming (extended version). Technical Report SIC-05/13, Facultad de Informática, Universidad Complutense de Madrid, 2013. `http://gpd.sip.ucm.es/rafa/minizinc/cptr.pdf`.
3. E. F. Codd. Missing information (applicable and inapplicable) in relational databases. *SIGMOD Record*, 15(4):53–78, 1986.
4. A. Frisch and P. Stuckey. The proper treatment of undefinedness in constraint languages. In I. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *LNCS*, pages 367–382. Springer-Verlag, 2009.
5. IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, Aug. 1985.
6. L. D. Koninck, S. Brand, and P. J. Stuckey. Constraints in non-boolean contexts. In *ICLP (Technical Communications)*, pages 117–127, 2011.
7. P. Laborie and J. Rogerie. Reasoning with conditional time-intervals. In D. C. Wilson and H. C. Lane, editors, *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference*, pages 555–560. AAAI Press, 2008.
8. G. Malinowski. *Many-Valued Logics*. Oxford University Press, 1993.
9. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In *In: Proc. of 13th International Conference on Principles and Practice of Constraint Programming*, pages 529–543. Springer, 2007.
10. P. J. Stuckey and G. Tack. Minizinc with functions. In *Proceedings of the 10th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*, LNCS, page to appear. Springer, 2013.

Constraint Models for the Container Pre-Marshaling Problem

Andrea Rendl and Matthias Prandtstetter

AIT Austrian Institute of Technology GmbH
Mobility Department, Dynamic Transportation Systems
Giefinggasse 2, 1210 Vienna, Austria
{matthias.prandtstetter|andrea.rendl}@ait.ac.at

Abstract. The container pre-marshaling problem (PMP) is an important optimization problem that arises in terminals where containers are stacked in bays before they are collected by vessels. However, if a due container is stacked underneath other containers, it is blocked, and additional relocations are necessary to retrieve the container from the bay, which can cause significantly delays. To avoid this effect, the PMP is concerned with finding a minimal number of container movements that result in a container bay without blocked containers.
In this paper, we introduce a Constraint Programming (CP) formulation of the PMP and propose a specialized search heuristic that attempts to try out the most promising moves first. Furthermore, we present a robust variant of the problem that considers the uncertainty of the arrival time of collecting vessels. We show how the robust PMP can be very naturally formulated using CP and give some preliminary results in an experimental evaluation.

1 Introduction

Containers are an essential means for transporting large quantities of goods. The main reason for this is the standardized format of containers through which containers are transportable by all major means of transportation. This is important since transporting goods often requires a chain of different transport modes, such as ship, train or truck. For example, a container that transports goods from a Chinese factory to a European seller, will most likely first travel by train from the factory to a harbor, then by ship to Europe, and then by truck to its final destination.

An important aspect in the transportation chain of containers is the hub between different modes of transport: the *container terminal*. Container terminals handle the exchange of containers between different vessels, as well as the storage of containers until their respective vessel arrives. For illustration, Fig. 1 shows a container terminal for trains and trucks. Since container terminals have to handle an increasingly large amount of incoming and outgoing vessels, it is increasingly difficult to manage all the traffic as well as the container storage within the terminal.

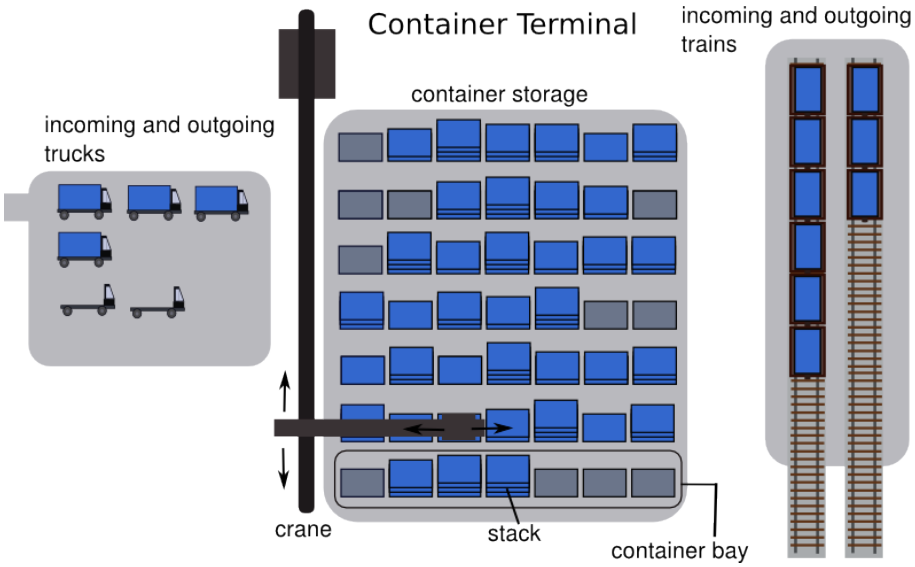


Fig. 1: Container terminal that operates as a hub for trucks (left) and trains (right). Containers are stored in several container bays in the center. A gantry crane moves the containers.

One of the many problems in container terminal optimization is the management of *container bays*, in which incoming containers are stored until they are due for another vessel. A container bay consists of a row of adjacent *stacks* in which containers are stacked upon arrival. When a container is due, it is removed from the bay by the gantry crane. However, due containers are often blocked by other containers in the stack, which results in additional re-locations to access the container. This can significantly increase the loading time of a vessel and cause dramatic (chains of) delays, resulting in displeased terminal clients. Therefore, terminals *pre-marshal* their container bays by re-locating containers to free all blocked containers before incoming vessels arrive. Thus, all containers in the pre-marshaled bay are directly accessible at their due date.

1.1 Related Work

Pre-marshaling is known to be NP-hard [6] and different solution approaches have been proposed for tackling it. Among them are dynamic programming [7, 12], corridor method based approaches [7], neighborhood search based methods [10], greedy heuristics [9], an A*-search [9], a tree-search based approach [5], a multi-commodity flow formulation [11] as well as an integer linear programming approach [11]. However, to the best of our knowledge, no Constraint Programming (CP) approach has been proposed so far for the PMP. In this paper, we show how the PMP can be formulated and solved using Constraint Programming.

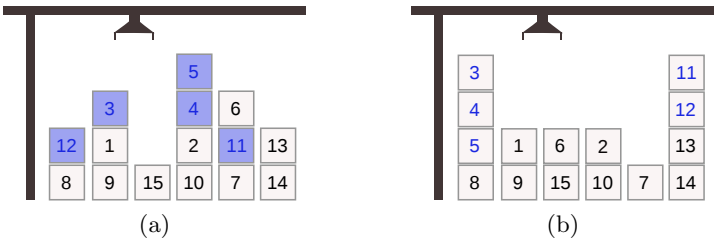


Fig. 2: Container bay before (a) and after (b) pre-marshaling. Shaded containers are blocking other containers; lower numbers indicate earlier due dates.

The PMP is only one of several optimization problems arising in container terminals; we refer to [15–17] for an overview. Furthermore, the PMP is closely related to the famous Blocks World Problem (BWP) [14] where the objective is to relocate an initial configuration of blocks into a specific goal configuration. We want to highlight, however, the differences between the PMP and the BWP [13]: unlike the BWP, in the PMP the number of stacks, as well as their height, is limited. Furthermore, in the BWP, the objective is to arrange the blocks in a pre-specified configuration, while in the PMP, the final configuration is unknown, but simply characterized by some features.

2 The Pre-marshaling Problem (PMP)

In container terminals, containers are stored in *bays* that consist of a number of stacks of maximal height. Every container has a due date at which a vessel will arrive to collect it. The due time of a container c is typically represented by *priority* p_c. The smaller the value of p_c, the sooner c is due for delivery. For instance, container with priority $p_c = 3$ is the third container that will be removed from the bay. Figure 2(a) shows a container bay with 6 stacks of maximal height 4 that holds 15 containers that are labeled with their priority.

The priority of a container is often not known at its arrival in the bay, therefore containers are often stacked randomly. As a consequence, due containers are often blocked by other containers that are due at a later time. In this case, the gantry crane has to perform additional relocations, so-called idle strokes, to 'free' the blocked container. This can result in a severe overhead in processing time which can lead to significant (chains of) delays within the terminal. Figure 2 (a) illustrates a bay with five blocking containers.

The pre-marshaling problem (PMP) is concerned with re-locating containers in a bay such that every container can be removed at its due date without any further relocations. More specifically, the PMP deals with finding a minimal number of container movements that result in a container bay without blocking containers. Figure 2(b) shows the result of pre-marshaling the container bay from Figure 2(a).

2.1 Formal problem formulation

We consider a container bay with S stacks of maximal height H where a set of containers $\mathcal{C}$ are stored. Each container $c \in \mathcal{C}$ is assigned a priority $p_c \in \mathbb{N}$ that indicates when the container will be transported from the bay: the smaller p_c, the earlier the container will be stowed into a vessel. If a container c is stored on top of a container that has a lower priority than c, then c is called a *blocking container*. The number of containers with priority c (multiplicity of c) is given by $\mu(c)$.

The initial bay setup is denoted $\mathcal{B}$ where $\mathcal{B}_{s,t}$ is the t-th container in the s-th stack, with $(s,t) \in \mathcal{S} \times \mathcal{T}$ and write, $\mathcal{B}_{s,t} = 0$ if slot $\mathcal{B}_{s,t}$ is empty. A stack $\mathcal{B}_s, s \in \mathcal{S}$ is said to be *valid* iff

$$\mu_s(c) \leq \mu(c), \qquad \text{for } c \in \mathcal{C}, \tag{1}$$

$$\mathcal{B}_{s,t} = 0 \Rightarrow \mathcal{B}_{s,t+1} = 0, \qquad \text{for } t \in \mathcal{T} \setminus \{H\} \tag{2}$$

Thus, a stack is valid if it does not contain more containers of priority c than are available (Eq. 1) and if a slot at position t is empty, all subsequent slots must also be. A stack $\mathcal{B}_s, s \in \mathcal{S}$ is called *perfect*, if it is valid and

$$\mathcal{B}_{s,t} \geq \mathcal{B}_{s,t+1}, \qquad \text{for } t \in \mathcal{T} \setminus \{H\} \tag{3}$$

holds, i.e. no container in the stack lies underneath a container with higher priority (no containers are blocked). Furthermore, container bay $\mathcal{B}$ is valid, iff

$$\mathcal{B}_s \text{ is valid}, \qquad \text{for } s \in \mathcal{S} \tag{4}$$

$$\sum_{s \in \mathcal{S}} \mu_s(c) = \mu(c), \qquad \text{for } c \in \mathcal{C} \tag{5}$$

thus, if each stack in $\mathcal{B}$ is valid (Eq. 4) and for every priority $c \in \mathcal{C}$, the bay contains exactly the number of corresponding containers (Eq. 5). A bay is called *perfect* if all stacks are perfect.

The bay can be altered by performing particular moves (actions): the topmost container from one stack can be moved to the top of another stack. Therefore, we define the container relocation $r = (i,j)$ as the movement of the topmost container of stack $\mathcal{B}_i$ to the top of stack $\mathcal{B}_j$. A move is *valid* iff $\mathcal{B}_{i,1} \neq 0$ and $\mathcal{B}_{j,H} = 0$, i.e., at least one container is stored in the i-th stack and the number of containers in the j-th stack is less than the maximum height.

The aim of the PMP is to find a sequence of moves that transforms the initial bay $\mathcal{B}$ into a bay without blocking containers. Thus, $\sigma = (r_1, \ldots, r_K)$ is a solution to the PMP if every move $r \in \sigma$ is valid and σ transforms $\mathcal{B}$ into a perfect bay. A solution $\sigma^* = (r_1, \ldots, r_{K^*})$ is said to be optimal if $K^* \leq K$ for all valid solutions σ. Let us denote by $\mathcal{K} = \{1, \ldots, K\}$ the set of steps in a solution and by $\mathcal{K}_0 = \mathcal{K} \cup \{0\}$.

3 A Constraint Model for the PMP

We can easily solve the PMP via Constraint Programming (CP) by iteratively trying to find a solution with exactly $k \geq 0$ container moves, as outlined in

Algorithm 1: PMP(bay layout $\mathcal{B}$)

Input: initial bay layout $\mathcal{B}$

```
1 k ← getLowerBound(B);
2 while true do
3 |   σ ← solveDecisionProblem(k, B);
4 |   if σ ≠ NIL then
5 |   |   return solution;
6 |   k ← k + 1;
```

Alg. 1. When starting with a lower bound for k and iteratively increasing k if no solution could be found, the first returned solution provides an optimal solution for the original PMP formulation. If the lower bound on k is tight, the number of iterations is kept low. Therefore, we utilize the lower bound computation methods proposed by Bortfeldt and Forster [5] deriving the lower bound according to different features of the container bay, such as the number of blocking containers.

Our constraint model is based on an AI planning perspective [3, 2], where we consider *states* that can be altered by a limited set of *actions* (moves). Starting with an initial state (the initial bay layout), we search for a sequence of moves, until a desired end state (perfect bay) is reached. In our model, the main decision variables concern the container bay states.

3.1 Variables

We use two kinds of variables to represent the PMP. First, variables $bay^k_{s,t}$ represent the bay state after performing move k, with $k \in \mathcal{K}_0$, $1 \leq s \leq W$, and $1 \leq t \leq H$. All variables $bay^k_{s,t}$ range over the domain $\mathcal{C}_0$ (set of containers including the empty slot). Thus, $bay^K_{s,t}$ represents the final bay state and $bay^0_{s,t}$ corresponds to the initial layout.

Second, we introduce 0-1-variables $move^k_{s,t}$ that indicate whether a container is moved to the t-th tier of the s-th stack during move k (with $1 \leq s \leq W$, $1 \leq t \leq H$, and $k \in \mathcal{K}$).

3.2 Constraints

First, we assign the initial state of the bay, $\mathcal{B}$, to the variables bay^0:

$$bay^0_{s,t} = \mathcal{B}[s][t] \qquad \text{for } (s,t) \in \mathcal{S} \times \mathcal{T} \tag{6}$$

Next we state that no container may disappear or appear twice in a bay after each move k. More specifically, we state that each priority c must occur as often as its multiplicity $\mu(c)$:

$$\left|\left\{ bay^k_{s,t} : bay^k_{s,t} = c \right\}\right| = \mu(c) \qquad \text{for } c \in \mathcal{C}_0, k \in \mathcal{K} \tag{7}$$

We implemented Constraint (7) using multiple occurrence constraints. Furthermore, the multiplicity of the empty slot is $\mu(0) = S*H-\hat{c}$ where $\hat{c}$ is the number of containers in the bay. Constraints (8) ensure that unchanged slots stay the same: all slots that are neither '0' at steps $k-1$ and k have obviously not been changed and must stay the same:

$$bay_{s,t}^{k-1} \neq 0 \wedge bay_{s,t}^{k} \neq 0 \Rightarrow bay_{s,t}^{k-1} = bay_{s,t}^{k} \quad \text{for } (s,t) \in \mathcal{S} \times \mathcal{T}, k \in \mathcal{K} \tag{8}$$

Since expressing Constraints (8) with standard available (global) constraints resulted in considerably more, additional variables (and constraints), we decided to implement a user defined constraint which straightforwardly checks whether the Constraints (8) are fulfilled. Furthermore, we state that only one container may be moved in each step k:

$$\sum_{(s,t)\in\mathcal{S}\times\mathcal{T}} move_{s,t}^{k} = 1 \qquad \text{for } k \in \mathcal{K} \tag{9}$$

and we link the *bay* variables with the *move* variables, i.e. we ensure that the corresponding *move* variable is set when a container is moved to a (new) position in step k.

$$move_{s,t}^{k} \leq bay_{s,t}^{k} \qquad \text{for } (s,t) \in \mathcal{S} \times \mathcal{T}, k \in \mathcal{K} \tag{10}$$

$$move_{s,t}^{k} \leq 1 - \min\left\{1, bay_{s,t}^{k-1}\right\} \qquad \text{for } (s,t) \in \mathcal{S} \times \mathcal{T}, k \in \mathcal{K} \tag{11}$$

$$move_{s,t}^{k} \geq \min\left\{1, bay_{s,t}^{k}\right\} - bay_{s,t}^{k-1} \qquad \text{for } (s,t) \in \mathcal{S} \times \mathcal{T}, k \in \mathcal{K} \tag{12}$$

$$\tag{13}$$

Finally, we specify the perfect bay setup by stating that priority of slot t in stack s has to be less or equal to the priority in slot $t-1$.

$$bay_{s,t}^{K} \leq bay_{s,t-1}^{K} \qquad \text{for } (s,t) \in \mathcal{S} \times \mathcal{T} \setminus \{1\} \tag{14}$$

Note that this is feasible since we represent empty slots by '0'.

3.3 A Specialized Search Heuristic

Variable Selection Search is only performed on the *bay* variables; the *move* variables are set by the constraints. We apply a (semi) static variable ordering assuring that all variables bay^k for step k are instantiated before variables for step $k' > k$ are selected. This way we search the bay states step by step. However, since the value selection (that determines which container to move within the bay) is dynamic, the variable order *within* one step k is determined heuristically, as explained below.

Value Selection A smart value selection heuristic requires some knowledge about which container move would be beneficial in the current bay state. Therefore, we employ the heuristic of Bortfeldt and Forster [5] that, given a bay layout, returns a sorted list of promising moves $\mathcal{M}$, where the potentially best moves are first in the list. The list $\mathcal{M}$ is calculated by classifying moves according to how they improve or impair the current bay state. For instance, a move that renders an imperfect stack *perfect*, is called a *bad-good* move, since it transforms a bad into a good state. In the same fashion, *good-bad*, *good-good* and *bad-bad* moves are defined. Note, that moves that worsen the bay state, *good-bad* and *bad-bad*, are often essential to 'dig out' containers and thus reach a solution.

In summary, we compute $\mathcal{M}$ for the current state at step $k \in \mathcal{K}$, and try out every move $m \in \mathcal{M}$, starting with the first (the most promising one). Then, the selected move m determines which state variable to search on next, as well as its value. More specifically, we set the state variable of the target position of the moved container with the container value. This choice sets all other state variables, which is essential since therefore mainly *one* variable-value choice is needed to reach the next step $k + 1$.

4 A robust variant of the PMP

In real-world settings, the exact arrival time of a vessel is quite uncertain. In fact, most vessels are expected to arrive within a time window instead of at an exact time. Since the container priorities in the classical PMP are based on the scheduled arrival *times* instead of *time windows* of vessels (and those time windows often overlap), the *initially expected* priority of a container may differ from the *actual* priority. Thus, perfect bays that are obtained through the classical PMP approach are easily rendered imperfect, if the arrival time windows of vessels are not considered. We therefore aim at finding a sequence of container moves that produces a final bay setup that is *robust* concerning expected vessel delays.

The (expected) arrival time of a vessel can be represented by some probability distribution, for instance a normal distribution with the scheduled arrival time as mean and the expected delay as standard deviation. However, we simplify this notion by considering the arrival time as a simple time window (where the arrival at any time within the time window has the same probability and thus represents a uniform distribution). Based on the arrival time window, we can deduce a *priority range* for the container that will be collected. More specifically, a container c has a *priority range* $p_c = \{a_l, a_u\}$ where a_l is the earliest possible priority, and a_u is the latest. For instance, a priority range of $p_c = \{4, 6\}$ denotes that container c may be the fourth, fifth or sixth container to be removed from the container bay. Figure 3 illustrates a bay where for some containers the priorities are given as ranges.

A stack $\mathcal{B}_s, s \in \mathcal{S}$ is called *robust*, if it is valid and

$$p_l^{\mathcal{B}_{s,t}} \geq p_u^{\mathcal{B}_{s,t+1}}, \qquad \text{for } t \in \mathcal{T} \setminus \{H\} \tag{15}$$

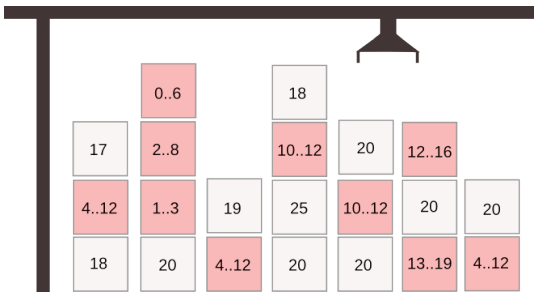


Fig. 3: Container bay where the priorities of some containers are ranges

holds, i.e. the upper bound of the priority of the container at $\mathcal{B}_{s,t+1}$ that is stacked upon container $\mathcal{B}_{s,t}$, has to be smaller or equal to the lower bound of container c_2. We say that the container bay $\mathcal{B}$ is robust, iff it is valid and all stacks are robust.

Please note, that a given container bay with priority ranges may not have a robust configuration. For instance, consider a 4×4 bay with 5 containers that all have the same priority range 0..5. This bay cannot be altered to a robust bay configuration, since in some stack, two containers with range $\{0..5\}$ have to be stacked over another. Therefore, it can be useful to determine beforehand if a robust configuration exists, which can be easily formulated using CP. This is part of our current work.

To the best of knowledge, very little has been investigated into studying robust or stochastic variants of the pre-marshaling problem. In very recent work, Borjan et.al. [4] present a mathematical model for the dynamic version of the Container Relocation Problem (CRP), and also consider uncertainty. The CRP is very similar to the PMP, where the goal is to additionally empty the bay while performing relocations.

5 A Constraint Model for the Robust PMP

The constraint model for the robust PMP is based on the PMP model from Sec. 3. The main difference lies within the notion of container priorities: in the robust PMP, we consider container priorities as range $\{lb..ub\}$ where $lb < ub$ and lb denotes the lower bound of the container's priority, and ub the upper bound.

Furthermore, we assign a unique id $c \in \{1, \ldots, C\}$ to each container. This is necessary in order to track containers to assure that no container occurs multiple times or disappears during a move. More specifically, in the PMP model (Sec. 3), we were able to assure that all containers stay in the bay after each move by checking the multiplicity of each priority (Constraint (7)). However, in the robust PMP, priorities are ranges and considering the multiplicity of ranges is much more expensive. Therefore, we assign a unique id to each container to enhance modeling.

The priority range for container c is denoted by p_i^c with $i \in \{l, u\}$ where p_l^c is the lower bound, and p_u^c is the upper bound. Note, that the empty slot '0' has priority $p_l^0 = p_u^0 = 0$ and containers c with a single priority value v have the same upper and lower bound $p_l^c = p_u^c = v$.

5.1 Variables

We apply the same set of variables as for the classical PMP (see Section 3.1). However, the meaning of the *bay* variables is slightly different: while in the classical PMP model the domain $\mathcal{C}$ of variables *bay* represented the container priorities, in the robust PMP its domain represents the container *id*.

5.2 Constraints

The constraints for the robust PMP are the same as for the classical PMP, with the exception of the final bay configuration. Therefore, after stating constraints (6)-(12), we constrain the final bay setup by:

$$p_u^{bay^K_{s,t}} \leq p_l^{bay^K_{s,t-1}} \qquad \text{for } (s,t) \in \mathcal{S} \times \mathcal{T} \setminus \{1\} \tag{16}$$

More specifically, Constraint (16) states that the upper bound of the priority range of the container in slot t has to be less or equal than the lower bound of the container in slot $t-1$. This is necessary to avoid an overlap within the container priority range of two stacked containers. For instance, container c_1 with priority range $p^{c_1} = \{0..4\}$ may not be stacked upon container c_2 with $p^{c_2} = \{2..9\}$ since the ranges overlap: $\{0..4\} \cap \{2..9\} = \{2..4\}$. Therefore, if for instance, c_1's priority is realized with 3 and c_2 with 2, then c_1 would block c_2.

6 Preliminary Results

We conducted a computational evaluation to assess the performance of the presented CP models. Note that the CP models were implemented in Java 7 using the freely available Choco framework [8] version 2.1.5 as CP solver. We first discuss results for the PMP model and then continue with the robust PMP model.

6.1 Evaluating the CP Model for the PMP

We compare the PMP model from Sec. 3 to one of the fastest current approaches [12]: a dynamic program combined with branch-and-bound, denoted DPBnB. Each approach is given a maximum of 600s computation time to find an optimal solution for the test instances.

As test instances, we decided to rely on the instance generator provided by Expósito-Izquierdo et al. [9] via [1]. Unfortunately, the original instances used in [9] are not publicly available. The instances are grouped into sets according

		DPBnB		CP	
		#solved	time[s]	#solved	time[s]
04 x 04 050	001	100	0.03	100	1.79
	002	100	0.02	100	0.55
	003	100	0.02	100	0.49
	004	100	0.02	100	0.43
04 x 04 075	001	100	0.12	–	–
	002	100	0.10	51	117.41
	003	100	0.06	87	46.96
	004	100	0.06	85	26.16

Table 1: Results for solving the PMP on instances with 4 stacks of maximal height 4 with either 50% or 75% of container coverage and 4 difficulty levels (001 is the most difficult level).

to the *bay size* (4×4 stands for 4 stacks of maximal height 4), the *utilization rate* (either 50% or 75% of the available tiers are filled with containers), and the *difficulty level* (1 to 4) that is a rough estimation on how disordered the bays are (with 1 being the hardest). Each of these sets contains 100 (pairwise different) instances.

The experiments were carried out on a single core of an Intel® Core™2 Quad CPU Q9300 with 2.50GHz and 3GB RAM.

Table 1 documents how often each algorithmic setup reached an optimal solution, as well as the average computation times (over all 100 instances). Columns report the number of instances solved to optimality and the average computation time for each approach; rows represent the set of 100 instances with respect to their size, coverage rate, and difficulty level.

We observe that the CP model is not yet competitive with the current state of the art, in particular on dense instances, where the container bay contains a lot (75%) of containers. While for less dense instances (50% coverage), the CP model can solve all instances, it only manages to solve 85% of the easiest dense instances, and for the most difficult level, cannot solve a single instances within the given time limit.

We believe that one major drawback of the current CP formulation is the lack of a mechanism (in form of a constraint or search heuristic) that stops search on bay states that have already been reached (and thus should not be part of an optimal solution). This is difficult to formulate effectively as a constraint in CP. However, since the DPBnB approach is able to eliminate these bay states naturally, we are currently working on an integration of this approach into our CP model.

6.2 Evaluating the CP Model for the Robust PMP

We conducted some first tests on the robust PMP model from Sec. 5. The instances for the robust PMP are based on the instances for the PMP that we

instance	# moves	time [s]	nodes	backtracks
0	5	0.186	41	66
1	6	0.625	474	933
2	5	0.222	66	116
3	5	0.125	41	51
4	5	0.938	1301	2561

Table 2: Preliminary results for the robust PMP on instances with 50% coverage and medium difficulty level.

modify: we first randomly assign each container to a vessel (train, truck or ship) and assign each vessel an arrival time that is reflected by the priority of the container in the PMP instance. Then we deduce each container's priority range by the vessels' arrival time window. More specifically, we create an arrival time window for each vessel that is based on a distribution of expected delays.

The experiments were carried out on a single core of an Intel® Core™ i7 CPU M 640 with 2.80GHz and 1GB RAM.

The preliminary results are summarized in Tab. 2 that shows the number of moves to render the bay perfect, the solving time, as well as some information on search. We see that we are able to solve these rather small instances quickly, however, we still need to assess our approach on larger and more complex problem instances.

7 Conclusions

In this paper, we introduced a new problem to the Constraint Programming community, the container pre-marshaling problem (PMP). The PMP is concerned with finding a minimal sequence of container relocations such that the resulting container bay hosts no blocked containers. Our contributions are two-fold: first, we present the first constraint model for the PMP and second, we introduce a robust variant of the PMP and show how the CP model for the PMP can be easily and naturally extended to the robust formulation.

In an initial experimental evaluation, we assess both constraint models. We observe that the PMP model is still not competitive and requires enhancement. In particular, we compare the CP model to a dynamic programming based approach that represents the current state of the art. Our current (and future) work is focused on improving the constraint model by integrating features of the mentioned DP-based approach into the CP model. Some initial experiments on the robust PMP (for which there are no alternative approaches yet) show good results on small instances.

For future work, we plan to enhance and extend both our CP models to render them competitive with existing approaches. Furthermore, we want to consider alternative formulations and explore alternative solving approaches for tackling the robust PMP.

Acknowledgments

This work is part of the project TRIUMPH, partially funded by the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) within the strategic programme I2VSplus under grant 831736. The authors thankfully acknowledge the TRIUMPH project partners Logistikum Steyr (FH OÖ Forschungs & Entwicklungs GmbH), Ennshafen OÖ GmbH and via donau – Österreichische Wasserstraßen-Gesellschaft mbH.

References

1. `http://sites.google.com/site/gciports` (last accessed April 2013)
2. Barták, R., Toropila, D.: Reformulating constraint models for classical planning. In: Wilson, D., Lane, H.C. (eds.) FLAIRS Conference. pp. 525–530. AAAI Press (2008)
3. van Beek, P., Chen, X.: Cplan: A constraint programming approach to planning. In: Hendler, J., Subramanian, D. (eds.) AAAI/IAAI. pp. 585–590. AAAI Press / The MIT Press (1999)
4. Borjan, S., Manshadi, V., Barnhart, C., Jaillet, P.: Dynamic stochastic optimization of relocations in container terminals. working paper, submitted, MIT (2013), http://web.mit.edu/jaillet/www/general/container13.pdf
5. Bortfeldt, A., Forster, F.: A tree search procedure for the container pre-marshalling problem. European Journal of Operational Research 217(3), 531–540 (2012)
6. Caserta, M., Schwarze, S., Voß, S.: Container rehandling at maritime container terminals. In: Böse, J.W. (ed.) Handbook of Terminal Planning, Operations Research/Computer Science Interfaces Series, vol. 49, pp. 247–269. Springer New York (2011)
7. Caserta, M., Voß, S.: A corridor method-based algorithm for the pre-marshalling problem. In: Giacobini, M., Brabazon, A., Cagnoni, S., Di Caro, G., Ekárt, A., Esparcia-Alcázar, A., Farooq, M., Fink, A., Machado, P. (eds.) Applications of Evolutionary Computing, Lecture Notes in Computer Science, vol. 5484, pp. 788–797. Springer (2009)
8. choco Team: choco: an Open Source Java Constraint Programming Library. Research report 10-02-INFO, École des Mines de Nantes (2010), http://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf
9. Expósito-Izquierdo, C., Melián-Batista, B., Moreno-Vega, M.: Pre-marshalling problem: Heuristic solution method and instances generator. Expert Systems with Applications 39(9), 8337–8349 (2012)
10. Lee, Y., Chao, S.L.: A neighborhood search heuristic for pre-marshalling export containers. European Journal of Operational Research 196(2), 468–475 (2009)
11. Lee, Y., Hsu, N.Y.: An optimization model for the container pre-marshalling problem. Computers & Operations Research 34(11), 3295–3313 (2007)
12. Prandtstetter, M.: A dynamic programming based branch-and-bound algorithm for the container pre-marshalling problem. Tech. rep., AIT Austrian Institute of Technology (2013), submitted to European Journal of Operational Research
13. Rodríguez-Molins, M., Salido, M.A., Barber, F.: Intelligent planning for allocating containers in maritime terminals. Expert Syst. Appl. 39(1), 978–989 (2012)
14. Slaney, J.K., Thiébaux, S.: Blocks world revisited. Artif. Intell. 125(1-2), 119–153 (2001)

15. Stahlbock, R., Voß, S.: Operations research at container terminals: a literature update. OR Spectrum 30, 1–52 (2008)
16. Steenken, D., Voß, S., Stahlbock, R.: Container terminal operation and operations research - a classification and literature review. OR Spectrum 26, 3–49 (2004)
17. Vis, I.F., de Koster, R.: Transshipment of containers at a container terminal: An overview. European Journal of Operational Research 147(1), 1–16 (2003)

Boosting Weighted CSP Resolution with Shared BDDs*

Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret

Departament d'Informàtica, Matemàtica Aplicada i Estadística
Universitat de Girona, Spain
{mbofill,mpalahi,suy,villaret}@imae.udg.edu

Abstract. We present a new approach for solving Weighted Constraint Satisfaction Problems (WCSP). The method is based on encoding the violation cost of soft constraints as a pseudo-Boolean objective function, and successively calling a decision procedure bounding the maximum allowable cost. The novelty of our approach consists in building a Binary Decision Diagram (BDD) for the objective function, using state-of-the-art generalized arc-consistent SAT encodings for it. Moreover, with our approach we maximize the reuse of the BDDs for the objective function between successive calls to the decision procedure by creating a shared BDD. The method has been incorporated into the WCSP solving system WSimply, based on reformulation into SMT, with preliminary encouraging results.

1 Introduction

A Constraint Satisfaction Problem (CSP) is a decision problem where the goal is to determine whether an assignment of values to a set of variables exists which satisfies a given set of constraints. It is usually to find CSPs where, additionally to determine if there exists a solution for the problem, the possible solution has to minimize or maximize some objective function. These kind of CSP are known as Constraint Optimization Problems (COP).

Occasionally, some real-world CSP instances have no solution. In such situations, we can relax the CSP by allowing the violation of a subset of the constraints, and try to maximize the number of satisfied constraints. This CSP variant is known as Maximum CSP (MaxCSP) [18]. Furthermore, there can exist preferences over which constraints to violate. A convenient way of expressing these preferences is by giving a weight to each constraint, denoting its violation cost. The constraints that can be violated (the ones with a non-infinite weight) are usually called *soft*, while those constraints that must be satisfied are called *hard*. Then, the objective is to find an assignment which satisfies all hard constraints and minimizes the aggregated cost of the violated soft constraints [21]. These problems are known as Weighted CSP (WCSP) [20] or, alternatively, as Cost Function Networks (CFN) [14].

* Partially supported by the Spanish Ministry of Science and Innovation through the projects TIN2012-33042, and by the Universitat de Girona under grant BR2010.

`WSimply` [4,6] is a language and system for solving intensionally represented WCSPs by reformulation into Satisfiability Modulo Theories (SMT) [8], namely, into SAT modulo Linear Integer Arithmetic (LIA). An SMT formula can be seen as a generalization of propositional Boolean formula, where some predicates have predefined interpretations from background theories, and any satisfying assignment has to be compatible with those theories. Leveraging the advances made in SAT solvers in the last decade, SMT solvers have proved to be competitive with classical decision methods in many areas, and in particular in CSP solving [5,10]. Most modern SMT solvers integrate a SAT solver with decision procedures (theory solvers) for sets of literals belonging to each theory. For example, variations of the simplex method are used for dealing with LIA predicates. This way, one can hopefully get the best of both worlds: in particular, the efficiency of the SAT solver for the Boolean reasoning and the efficiency of special-purpose algorithms for the theory reasoning.

`WSimply` benefits from the expressiveness of the SMT language and the performance of current SMT solvers. However, SMT solvers are decision procedures, and they scarcely support optimization. A few solvers support (weighted partial) MaxSMT [16,15,13], and there is a recent attempt to introduce optimization into SMT by means of a theory of costs [12]. In `WSimply`, optimization is implemented by means of successive calls to the decision procedure in several (user choosable) ways: performing sequential or binary search, or using algorithms based on unsatisfiable cores like WPM1 [7].

In this paper we extend the WCSP solving capability of `WSimply` by introducing a new optimization approach, based on representing the objective function (generated from the violation cost of the soft constraints) as a BDD [2]. This allows us to encode the objective function as a pure propositional formula, following the generalized arc-consistent encodings proposed in [1]. This way, we tighten the link between optimization and the logical structure of the problem, with the hope of benefiting from crucial capabilities of the underlying solver, such as conflict driven learning. An interesting aspect of our approach is the reutilization of BDDs in successive calls to the decision procedure. Although changing the bounds of the objective function implies building a new BDD, some parts can be easily reused. We create a Shared BDD [19], also known in the literature as Multi-Rooted BDD, keeping all the generated BDDs. This allows not only to improve the performance of the BDD construction algorithm, but also to keep a number of learned clauses from the solver, as we reuse the clauses representing the previous BDDs.

Since the size of BDDs strongly depends on the number of variables involved, and we use them to represent objective functions encoding the violation cost of soft constraints, our method is especially well suited for WCSP instances involving a *small* number of soft constraints. We provide encouraging preliminary results on the Soft Balanced Academic Curriculum Problem introduced in [4,6] and on a WCSP version of the Maximal Density Still Life Problem.

The paper is structured as follows. In Section 2 we introduce the required background on Weighted Constraint Satisfaction Problems (WCSP). In Sec-

tion 3 we introduce Pseudo-Boolean constraints and (Reduced Ordered) Binary Decision Diagrams (ROBDD). In Section 4 we present our method for solving WCSPs using shared ROBDDs. In Section 5 we study the performance of the new method and we compare it with the previous ones implemented in `WSimply`. Finally, in Section 6 we conclude and propose future work.

2 Solving WCSPs with SMT

The expert reader can skip the first two subsections and directly go to the solving algorithms described in Subsection 2.3.

2.1 CSPs, WCSPs and COPs

A *Constraint Satisfaction Problem (CSP)* instance is defined as a triple $\langle X, D, C\rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, $D = \{d(x_1), \ldots, d(x_n)\}$ is a set of domains containing the values the variables may take, and $C = \{C_1, \ldots, C_m\}$ is a set of constraints. Each constraint $C_i = \langle S_i, R_i\rangle$ is defined as a relation R_i over a subset of variables $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ (called the *constraint scope*) which specifies the allowable combinations of values for that subset. Usually, the relation R_i is represented *intensionally* as a condition that the assignments to the variables must satisfy (e.g., $x_1 < x_2$). An *assignment* v for a CSP instance $\langle X, D, C\rangle$ is a mapping that assigns to every variable $x_i \in X$ an element $v(x_i) \in d(x_i)$. An assignment v *satisfies* a constraint $\langle\{x_{i_1}, \ldots, x_{i_k}\}, R_i\rangle$ in C if and only if $\langle v(x_{i_1}), \ldots, v(x_{i_k})\rangle \in R_i$. A *solution* to a CSP instance is an assignment to its variables that satisfies all the constraints. The Constraint Satisfaction Problem for a CSP instance consists in finding a solution for that instance.

A *weighted CSP (WCSP)* instance is a triple $\langle X, D, C\rangle$, where X and D are variables and domains, respectively, as in a CSP. A constraint C_i is now defined as a pair $\langle S_i, f_i\rangle$, where $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ is the constraint scope and $f_i : d(x_{i_1}) \times \cdots \times d(x_{i_k}) \to \mathbb{N} \cup \{\infty\}$ is a *cost (weight) function* that maps tuples to its associated cost (a natural number or infinity). Forbidden tuples receive infinite cost. The cost of a constraint C_i induced by an assignment v in which the variables of $S_i = \{x_{i_1}, \ldots, x_{i_k}\}$ take values $b_{i_1}, \ldots, b_{i_k}$ is $f_i(b_{i_1}, \ldots, b_{i_k})$. A *solution* to a WCSP instance is an assignment to its variables which makes the sum of the costs of the constraints minimal. The Weighted Constraint Satisfaction Problem for a WCSP instance consists in finding a solution for that instance.

In this paper we assume to deal with weighted constraints $(c, w(c))$, where c is a constraint as defined for a CSP and $w(c)$ is the cost corresponding to its falsification, which can be a natural number or infinity. Note that this corresponds to an intensional formulation of the definition of *cost function* given above. We call those constraints whose associated cost is infinity *hard*, if otherwise *soft*. In this setting, the cost of a variable assignment corresponds to the sum of all weights of the constraints that are violated under the assignment.

A *Constraint Optimization Problem (COP)* instance consists of an optimization variable O matched to an objective function to be minimized (or maximized)

subject to the constraints of a CSP instance $\langle X, D, C\rangle$ where $O \in X$. A solution to a COP instance is a solution to the CSP instance that minimizes (or maximizes) the value of the optimization variable O.

2.2 SMT and Weighted SMT

A *Satisfiability Modulo Theories (SMT)* instance is a generalization of a Boolean formula in which some propositional variables have been replaced by predicates with predefined interpretations from background theories such as, e.g., linear integer arithmetic. For example, a formula can contain clauses like $p \vee q \vee (x + 2 \leq y) \vee (x > y + z)$, where p and q are Boolean variables and x, y and z are integer variables. A *solution* to an SMT instance is an assignment that satisfies the formula. Predicates over non-Boolean variables, such as linear integer inequalities, are evaluated according to the rules of a background theory [8]. As in the CSP case, we can extend SMT to *weighted SMT (WSMT)* as follows.

A *weighted SMT clause* is a pair (C, w), where C is an SMT clause[1] and w is a natural number or infinity (indicating the penalty for violating C). A *weighted SMT formula* is a multiset of weighted SMT clauses

$$\{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\}$$

where the first m clauses are soft and the last m' clauses are hard. The optimal cost of a formula is the minimal cost of all its assignments. An optimal assignment is an assignment with optimal cost. The *WSMT problem*[2] for a WSMT formula is the problem of finding an optimal assignment for that formula.

2.3 Solving WCSPs

Now we describe the reformulations and solving procedures of `WSimply` that are relevant for the present work. We remark that in this paper we assume that costs of soft constraints are constant natural numbers.[3]

The input of `WSimply` is a WCSP instance written in the `WSimply` language (we refer the reader to [9,6] for details). This is reformulated (according to a command line option indicating the solving method) either as a COP or as a WSMT instance:

– When reformulating a WCSP instance into a COP instance, each soft constraint C_i with weight w_i is replaced by the following constraints:

$$C_i \rightarrow (o_i = 0) \tag{1}$$
$$\neg C_i \rightarrow (o_i = 1) \tag{2}$$

[1] In fact these can be general SMT formulas, not necessarily disjunctions of literals.
[2] In the literature the weighted SMT problem is also referred to as weighted MaxSMT, same as in the SAT formalism. We prefer to talk about WSMT because it is closer to WCSP.
[3] In `WSimply` costs can be defined by a linear integer arithmetic expression.

where o_i is a fresh (pseudo-Boolean) variable reifying the violation of C_i. Secondly, the objective function is defined by introducing a fresh integer variable O to be the aggregation of violation costs, with the constraint:

$$O = \sum_{i=1}^{m} w_i * o_i \tag{3}$$

The variable O is the variable to be minimized in the resulting COP. Original hard constraints are kept without modification.

– The reformulation of a WCSP instance into a WSMT instance is trivial: each soft constraint C_i with weight w_i is replaced by the WSMT clause (C'_i, w_i) where C'_i is the translation of C_i into SMT as described in [9] (recall that our method is based on translation of CSPs into SAT modulo linear integer arithmetic). Finally, hard constraints are replaced by their equivalent hard SMT clauses.

Once the WCSP instance has been reformulated, the system can solve it by means of one of three methods, called `yices`, `core` and `dico` respectively. The two first methods allow to solve WSMT instances, while the last one allows to solve COP instances:

– The `yices` method uses an algorithm that performs a sequence of satisfiability checks until the optimum is found. It is the default Yices [16] algorithm for solving WSMT (`WSimply` is built on top of Yices). This algorithm is not exact since Yices defines a maximum number of iterations for the search.[4] We are not aware of any document describing the procedures used there.
– The `core` method is an implementation, introduced in [6], of the core based WPM1 algorithm [7] from the MaxSAT field.
– In the `dico` method, the system first translates the constraints of the COP into SMT formulae, and then incrementally calls an SMT solver, bounding the optimization variable O by adding the unit clause $O \leq K$, where K is an integer constant determined by the system using binary search. Note that bounding the objective function of a COP which encodes a WCSP instance, where the weights of soft constraints are natural numbers, results into a pseudo-Boolean constraint (see Section 3).

3 SAT Encodings of Pseudo-Boolean Constraints using BDDs

Pseudo-Boolean (PB) constraints [11] are constraints of the form $a_1x_1 + \cdots + a_nx_n \ \# \ K$, where the a_i and K are integer coefficients, the x_i are pseudo-Boolean (0/1) variables, and the relation operator $\#$ belongs to $\{<, >, \leq, \geq, =\}$. For our purposes we assume that $\#$ is $\leq$ and that the a_i and K are positive. Under these assumptions, these constraints are monotonic (decreasing) Boolean

[4] `http://yices.csl.sri.com/language.shtml`

functions $C : \{0,1\}^n \rightarrow \{0,1\}$, i.e., any solution for C remains a solution after flipping input values from 1 to 0.

A typical data structure to represent Boolean functions is a *Binary Decision Diagram (BDD)*, which consists of a rooted, directed, acyclic graph, where each non-terminal (decision) node corresponds to a Boolean variable x and has two child nodes with edges representing a *true* and a *false* assignment to x, respectively. We talk about the *true child* (resp. *false child*) to refer to the child node linked by the *true* (resp. *false*) edge. Terminal nodes are called 0-terminal and 1-terminal, representing the truth value of the formula for the assignment leading to them. A BDD is called *ordered* if different variables appear in the same order on all paths from the root. A BDD is said to be *reduced* if the following two rules have been applied to its graph until fixpoint:

- Merge any isomorphic subgraphs.
- Eliminate any node whose two children are isomorphic.

A *Reduced Ordered Binary Decision Diagram* (ROBDD) is canonical (unique) for a particular function and variable order. Figure 1 and Figure 2 illustrate, respectively, a BDD and a ROBDD for the same PB constraint.

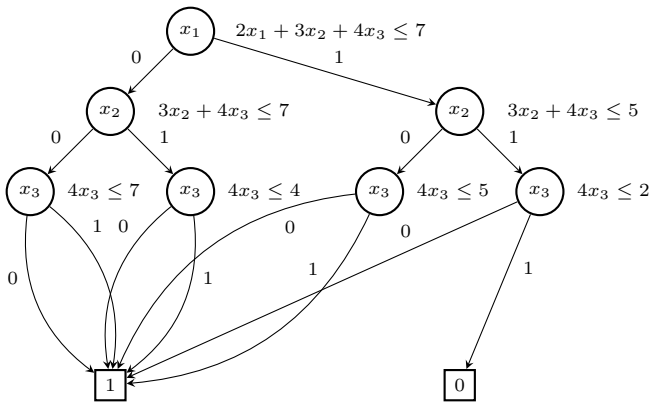


Fig. 1. BDD for $2x_1 + 3x_2 + 4x_3 \leq 7$.

There exist several BDD-based approaches for reformulating PB constraints into SAT clauses [17]. We focus on the recent work of [1], that proposes a simple and efficient algorithm to construct ROBDDs and a corresponding Generalized Arc Consistent (GAC) SAT encoding for monotonic Boolean functions.

A key point of that ROBDD construction algorithm is the reuse of BDDs, which is sustained by the so-called concept of *PB intervals*. Therefore, first of all we define the concept of PB interval. Let C be a constraint of the form $a_1x_1 + \cdots + a_nx_n \leq K$. The *interval of* C is the set of all integers M such that the constraint $a_1x_1 + \cdots + a_nx_n \leq M$, seen as a Boolean function, is equivalent to C (i.e., that the corresponding Boolean functions have the same truth table).

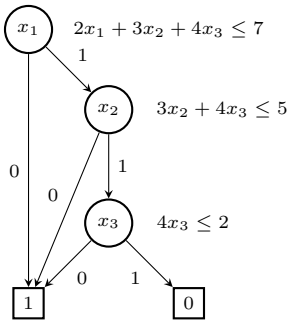


Fig. 2. ROBDD for $2x_1 + 3x_2 + 4x_3 \leq 7$.

For instance, the interval of $2x_1+3x_2+4x_3 \leq 7$ is $[7, 8]$ since, as no combination of coefficients adds to 8, we have that the constraint $2x_1 + 3x_2 + 4x_3 \leq 7$ is equivalent to $2x_1 + 3x_2 + 4x_3 \leq 8$. Since each node in a BDD represents a PB constraint, we can naturally overload the notion of interval and refer also to intervals of nodes.

The algorithm of [1] is a dynamic, bottom up BDD construction algorithm, which keeps the BDD intervals corresponding to the already visited nodes in *layers*: for a given variable ordering of a PB constraint, say $x_1, x_2, \ldots, x_n$, a list of layers $\mathcal{L} = L_1, \ldots, L_{n+1}$ is created. A layer L_i is a set of pairs of the form $([\beta, \gamma], \mathcal{B})$, where $\mathcal{B}$ is the ROBDD of the constraint $a_i x_i + \cdots + a_n x_n \leq K$ for every $K \in \{\beta..\gamma\}$. These intervals are used to detect if some needed ROBDD has already been constructed. That is, if for some node at level i, the ROBDD for the constraint $a_i x_i + \cdots + a_n x_n \leq K$ is needed for a given K, and K belongs to some interval already computed in layer L_i, then the same ROBDD can be used for this node. It is important to recall here that ROBDDs are unique for a given function and variable ordering.

The algorithm first initializes each layer L_i, with i in $1..n+1$, with the pairs $((-\infty, -1], \mathbf{0})$ and $([\sum_{j=i}^{n} a_j, \infty), \mathbf{1})$. The ROBDD construction procedure has the following parameters: a PB constraint $a_i x_i + \cdots + a_n x_n \leq K$, the list of layers $\mathcal{L}$ and an index i denoting the current layer.

For instance, following the example of $2x_1 + 3x_2 + 4x_3 \leq 7$, the algorithm initializes $\mathcal{L}$ with:

$$L_1 = \{((-\infty, -1], \mathbf{0}), ([9, \infty), \mathbf{1})\}$$
$$L_2 = \{((-\infty, -1], \mathbf{0}), ([7, \infty), \mathbf{1})\}$$
$$L_3 = \{((-\infty, -1], \mathbf{0}), ([4, \infty), \mathbf{1})\}$$
$$L_4 = \{((-\infty, -1], \mathbf{0}), ([0, \infty), \mathbf{1})\}$$

Then, the construction procedure is called with $2x_1 + 3x_2 + 4x_3 \leq 7$, the list of layers $\mathcal{L}$, and index 1.

The first step of the construction procedure consists in searching in layer L_i if there exists a ROBDD $\mathcal{B}$ for the current K, i.e., if a pair $([\beta, \gamma], \mathcal{B})$ with $K \in \{\beta..\gamma\}$ exists in L_i. If so, the existing pair is returned, otherwise the procedure is recursively called for the two descendants increasing the index layer to $i+1$ and updating the K of the true child to $K - a_i$. Once the two descendants' ROBDD

are returned a new pair for the layer L_i is created. If the two returned ROBDDs are different, the $\mathcal{B}$ of the new pair will be a new ROBDD created from them, otherwise $\mathcal{B}$ will be the returned ROBDD of the children.

Following our example, the algorithm recursively calls the construction procedure for the two descendants with the parameters:

- True child: $3x_2 + 4x_3 \leq 5$, $\mathcal{L}$ and 2
- False child: $3x_2 + 4x_3 \leq 7$, $\mathcal{L}$ and 2

Since the two returned ROBDDs are different, the algorithm creates a new ROBDD $\mathcal{B}_1$ with pair $[7, 8]$, which will be inserted into L_1, resulting in:

$$\big\{((-\infty, -1], \mathbf{0}), ([7, 8], \mathcal{B}_1), ([9, \infty), \mathbf{1})\big\}$$

With the ROBDD constructed, we only have to encode it to SAT. As usual, the encoding introduces an auxiliary variable for every node. Let v be a node with selector variable x and auxiliary variable n. Let f be the variable of its false child and t be the variable of its true child. We only need to add two clauses per node:

$$\bar{f} \rightarrow \bar{n} \qquad \bar{t} \wedge x \rightarrow \bar{n}$$

and a unit clause with the variable of the 1-terminal and another one with the negation of the 0-terminal. Finally, we only have to add a unit clause forcing the variable of the root node to be *true*. This encoding is GAC.

Additional details of the BDD construction or the SAT encoding can be found in [1].

4 Solving WCSPs using Shared ROBDDs

As we have seen in Subsection 2.3, we can use a PB constraint to encode the objective function of a COP (possibly resulting from a WCSP). Moreover, we can use a ROBDD to represent this PB constraint and then encode it into SAT, as a GAC formula.

Our WCSP solving method consists in reformulating a WCSP into a COP, written in the SMT language, and solving the optimization problem by iteratively calling an SMT solver with the problem instance together with successively tighter bounds for the objective function. This is accomplished by adding the SAT encoding of the (tighter) PB constraints representing the objective function.

The fact of constructing ROBDDs to represent the PB constraints, using the same variable ordering and the same coefficients (we only change the K), may lead to have isomorphic subgraphs between ROBDDs. When multiple BDDs have isomorphic subgraphs they can be joined into a single *Shared BDD (SBDD)*, that is, a BDD with several roots [19].

The key point of our solving method is to construct a SBDD, in our case a Shared ROBDD (SROBDD), using the ROBDD construction method presented in Section 3. Note that when the ROBDD construction procedure finishes, in $\mathcal{L}$ we will have all the ROBDD nodes, with their corresponding intervals. Using

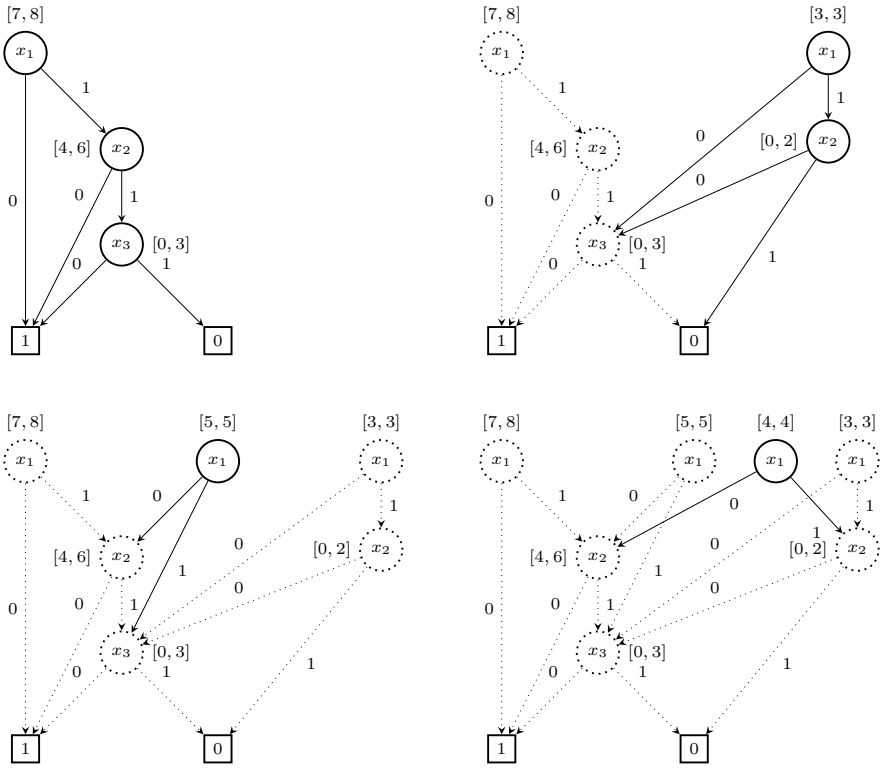


Fig. 3. ROBDDs and intervals for $2x_1 + 3x_2 + 4x_3 \leq 7$ (top left), $2x_1 + 3x_2 + 4x_3 \leq 3$ (top right), $2x_1 + 3x_2 + 4x_3 \leq 5$ (bottom left) and $2x_1 + 3x_2 + 4x_3 \leq 4$ (bottom right), illustrating the reuse of previous ROBDDs.

these pairs during the creation of all new constraints to bound the objective function, it is possible to create a SROBDD.

Figure 3 shows the evolution of the SROBDD representing the bounding constraint $2x_1 + 3x_2 + 4x_3 \leq K$, for $K = 7$ (sat), $K = 3$ (unsat), $K = 5$ (sat) and $K = 4$ (sat).

4.1 Optimization algorithm

Algorithm 1 describes our new WCSP solving method. First of all, we reify the soft constraints, we create a formula φ with the reified soft constraints together with the hard constraints, and check the satisfiability of φ. The function SMT_algorithm returns a tuple with the satisfiability (st) of φ and, if satisfiable, a model (M) of φ. If φ is unsatisfiable then the problem does not have any solution. Otherwise, we use the solution found to compute an upper bound ub of the objective function by aggregating the weights of the violated soft constraints, and set the lower bound lb to -1.

Algorithm 1 Binary Search Algorithm using Shared ROBDD

Input: $\varphi_s = \{(C_1, w_1), \ldots, (C_n, w_m)\}$, $\varphi_h = \{C_{m+1}, \ldots, C_{m+m'})\}$
Output: Cost of $\varphi_s \cup \varphi_h$ or *UNSAT*

```
φ ← φ_h ∪ reif_soft(φ_s)
(st, M) ← SMT_algorithm(φ)
if st = UNSAT then
   return UNSAT
else
   ub ← sum({w_i | (C_i, w_i) ∈ φ_s and C_i is falsified in M})
end if
lb ← −1
L ← init_layers(φ_s)
while ub > lb + 1 do
   K ← ⌊(ub + lb)/2⌋
   ([β, γ], B) ← construct_ROBDD(φ_s, K, L)
   root ← BDD2SAT(B, φ)
   φ ← φ ∪ root
   (st, M) ← SMT_algorithm(φ)
   if st = UNSAT then
      lb ← γ
      φ ← (φ \ root) ∪ ¬root
   else
      ub ← min(β, sum({w_i | (C_i, w_i) ∈ φ_s and C_i is falsified in M}))
   end if
end while
return ub
```

Before starting a binary search procedure, we initialize the list of layers $\mathcal{L}$ using the objective function, i.e., the weights of the soft constraints φ_s, as we have explained Section 3.

In the first step of the `while` statement, we determine a new tentative bound K for the objective function. Then, we call the ROBDD construction method described in Section 3, with the set of soft clauses φ_s, K and $\mathcal{L}$, being this last an input/output parameter. This way, $\mathcal{L}$ will contain the SROBDD with all the computed ROBDDs, and may be used in the following iterations of the search, significantly reducing the construction time and avoiding the addition of repeated clauses. This procedure returns the ROBDD $\mathcal{B}$ representing the objective function for the specific K in the current iteration.

In the next step we call the BDD2SAT procedure, which generates the SAT clauses from $\mathcal{B}$, as explained in Section 3, but only for the new nodes. Then the procedure inserts these clauses into φ and returns the auxiliary variable associated to the root node of $\mathcal{B}$. This variable is inserted into φ as a unit clause to effectively force that the objective function has to be less or equal than K.

At this point we call the SMT solver to check the satisfiability of φ. If φ is satisfiable we can keep all the learned clauses. Otherwise, we only have to remove the unit clause for the root node. This way, we will only remove the

learned clauses related to this unit clause. In addition, we add a unit clause with the negation of the root node variable, stating that the objective function is not less or equal than K, i.e., it has to be greater than K.

Finally, we update the bounds of the search using the interval $[\beta, \gamma]$ of $\mathcal{B}$:

- If φ is unsatisfiable: $lb = \gamma$
- If φ is satisfiable: $ub = min(\beta, sum(\{w_i \mid (C_i, w_i) \in \varphi_s, C_i = false\}))$

Note that, thanks to the intervals, in fact we are checking the satisfiability of the PB constraints for several values at the same time and, hence, we can compute more refined bounds.

Since we have the invariant that the lower bound always corresponds to an unsatisfiable case, while the upper bound corresponds to a satisfiable case, when $ub = lb + 1$ we are done.

5 Benchmarking

In this section we compare the performance of our new solving method to that of the methods already existing in `WSimply`. In particular, we use the following two problems (and some variants of them) for benchmarking:

- A softened version of the well-known Balanced Academic Curriculum Problem (BACP), the so-called Soft BACP (SBACP) [4,6]. In the BACP, a number of courses have to be scheduled in a limited number of periods, balancing students' load, and satisfying some prerequisite constraints between courses. In the SBACP the number of periods is reduced until all instances become unsatisfiable due to the prerequisites chain, and then the prerequisite constraints are considered to be soft. We use the five variants of the SBACP presented in [4,6] for this performance study.
- The Maximal Density Still Life Problem (Still Life) is to place the maximum number of live cells in a given board, so that the board configuration is stable under the Conway's Game of Life. We have translated this COP into a WCSP written in `WSimply`. We have considered the three harder variants of the problem found in the benchmarks folder of the MiniZinc distribution.[5] This COP is well-suited for our purposes because it has a PB optimization function.

Experiments have been run on an Intel® Core™ i5 CPU@2.66GHz, with 3GB of RAM, under 32-bit openSUSE 11.2 (kernel 2.6.31). We use `WSimply` with the API of the Yices 1.0.33 [16] SMT solver, with a cutoff of 600 seconds per run. By calling Yices through its API, we are able to keep learned clauses from previous calls that are still valid.

Table 4 shows the aggregated time per problem variant and solving method. We consider the solving methods `yices`, `core` and `dico`, already described in Subsection 2.3, plus the new method using SROBDDs, with two different variable

[5] `http://www.minizinc.org/`

orderings: `sbdd≤` where variables in the BDD are ordered from small (root) to big (leaves) coefficients, and `sbdd≥` where variables are ordered from big (root) to small (leaves) coefficients.

	#	dico	yices	core	sbdd≤	sbdd≥
sbacp	28	95.95	58.94	394.48 (15 t.o.)	12.79	
sbacp_h1	28	5.68	13.28	544.22 (12 t.o.)	5.99	
sbacp_h2	28	32.01	35.05	1128.93 (13 t.o.)	10.68	
sbacp_h2_ml2	28	344.95	153.40	114.29 (19 t.o.)	126.35	94.30
sbacp_h2_ml3	28	866.63	741.48	59.42 (24 t.o.)	145.76	258.32
still life	10	160.87 (1 t.o.)	382.76 (1 unk.)	0.87 (4 t.o.)	269.64	
still life free	10	553.95 (2 t.o.)	553.33 (3 unk.)	46.86 (3 t.o.)	126.28	
still life no border	10	296.77 (1 t.o.)	407.96 (1 unk.)	0.57 (4 t.o.)	220.74	

Fig. 4. Aggregated times in seconds for the solved instances of the 5 variants of the SBACP and the 3 variants of Still Life. The column # indicates the number of instances per set. The indication (n t.o.) refers to the number of unsolved instances, with a cutoff of 600s per instance. The indication (n unk.) refers to the number of instances for which the solver returned *unknown*.

In the first three variants of the SBACP and in the Still Life sets, the `sbdd≤` and `sbdd≥` methods are the same because the coefficients are 1 for all variables of PB constraints (with an average of 67 variables per PB constraint in the SBACP and of 30 in Still Life).

In the last two variants of the SBACP the coefficients are different. In the *sbacp_h2_ml2* set, PB constraints have an average of 312 variables, with coefficients 1 and 246, while in the *sbacp_h2_ml3* set, PB constraints have an average of 332 variables, with coefficients 1, 21 and 5166. In both cases the coefficients are stratified, for example, in the *sbacp_h2_ml2* set, PB constraints have an average of 245 variables with coefficient 1 and 67 variables with coefficient 246.

In these two latter sets of instances of the SBACP, the two distinct variable orderings for constructing the BDDs result into alternated best performance between `sbdd≤` and `sbdd≥`. In any case, our new solving method with SROBDDs is clearly the best for all sets of instances (except for *sbacp_h1*), the improvement being even clearer in the hardest sets, *sbacp_h2_ml3* and *still life free*. It is worthy to notice that in *still life free*, `dico` gives two time outs while `yices` three unknowns (recall that the Yices MaxSMT solver is non exact and incomplete).

Finally, the `core` method has really bad performance for these kind of problems. This is probably due to the bad quality of the cores found during the solving process.

We also provide particular comparisons between the `sbdd≤` and `dico` methods, and between the `sbdd≤` and `yices` methods. Figure 5 shows the comparison for all the 140 (28×5) instances of the SBACP variants, where `sbdd≤` is able to solve 104 instances in less time than `yices` (left), and 100 instances in less time than `dico` (right).

Figure 6 shows the comparison for all the 30 (10×3) instances of the Still Life variants, where the `sbdd`$\leq$ method is able to 19 instances in less time than `yices`, and `yices` cannot find a solution in 5 instances (left), and the `sbdd`$\leq$ method is able to solve 23 instances in less time than `dico` (right).

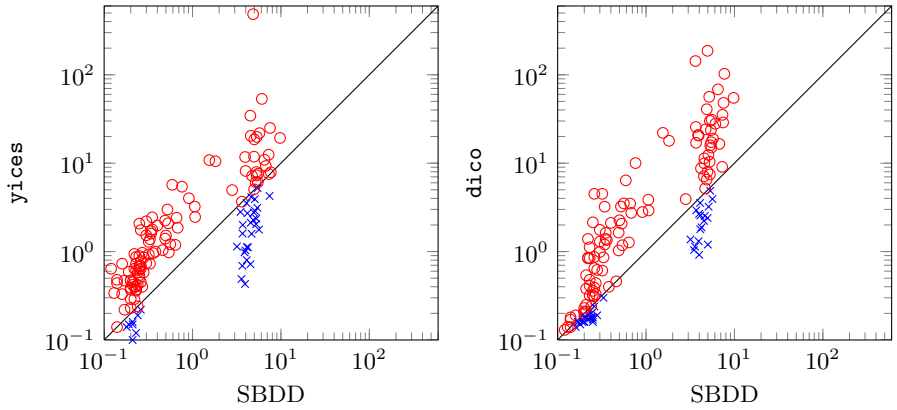


Fig. 5. Scatter plot of the solving times (in seconds) of the `sbdd`$\leq$ and `yices` methods (left), and of the `sbdd`$\leq$ and `dico` methods (right), for the 140 (28×5) instances of the SBACP. The `sbdd`$\leq$ method solves 104 instances in less time than `yices`, and 100 in less time than `dico`.

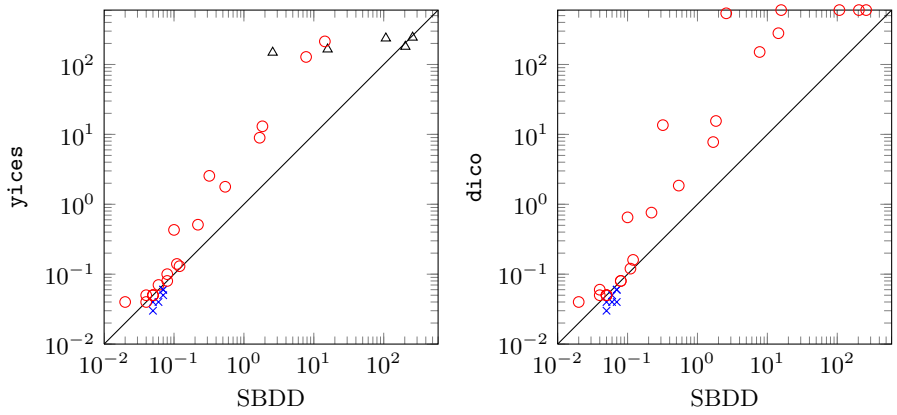


Fig. 6. Scatter plot of the solving times (in seconds) of the `sbdd`$\leq$ and `yices` methods (left), and of the `sbdd`$\leq$ and `dico` methods (right) for the 30 (10×3) instances of Still Life. The `sbdd`$\leq$ method solves 19 instances in less time than `yices`, which moreover cannot find a solution for 5 instances (marked with a triangle). The `sbdd`$\leq$ method solves solves 23 instances in less time than `dico`.

We have tried to figure out where is the gain in efficiency when using BDDs to deal with the objective function. Therefore, we have analyzed the time spent per iteration due to SROBDD construction, clauses assertion and satisfiability checking in the `sbdd`$\leq$ method, which in general is the best solving method.

Since Still Life instances have in average 30 variables in the PB constraint representing the bound of the objective function, all of them with coefficient 1, and we do binary search when seeking for the optimum, the number of iterations of the solving algorithm is at most 6 for these sets. A similar behavior appears for the three first sets of instances of the SBACP. For this reason, we only illustrate the time analysis for the *sbacp_h2_ml2* and *sbacp_h2_ml3* sets, in which the variables of the PB constraint have bigger coefficients, resulting in an average of 13 and 16 iteration steps respectively. Moreover, to see how useful is the usage of shared BDDs we have also computed the average percentage of node reuse in the construction of the new SROBDD per iteration.

Figure 7 has two plots showing this analysis for the *sbacp_h2_ml2* set (top) and the *sbacp_h2_ml3* set (bottom). As we can see, the majority of the time is spent in clauses assertion (this is the difference between the total time and the others). However, between iterations 5 and 9, in both sets, node reuse raises from 15% to almost 90% and therefore, the clauses assertion time strongly decreases. Naturally, at the same iterations we can appreciate a decrease on the total solving time, which in the last five iterations turns to be (nearly) just the time to check the satisfiability of the instance.

The node reuse behavior is similar for the rest of the instances of the SBACP and of Still Life, where the coefficients of the variables are always 1, making the BDDs very reusable. The only exception with respect to reuse is in the `sbdd`$\geq$ method, which has a high node reuse percentage in the *sbacp_h2_ml2* set (being fairly better than the `sbdd`$\leq$ method on the same set) and a low node reuse percentage in the *sbacp_h2_ml3* set. This can be appreciated in Table 4, where `sbdd`$\leq$ and `sbdd`$\geq$ swap their performance. We want to remark that, for the *sbacp_h2_ml3* set, the size of the first constructed ROBDD is very similar for both the `sbdd`$\leq$ and `sbdd`$\geq$ methods, with an average of 10395 nodes and an average of 11411 nodes, respectively. But the final SROBDD has in average 66128 nodes for `sbdd`$\leq$ and of 146791 nodes for `sbdd`$\geq$. Clearly, an important aspect to study in the future is how to find a good variable ordering for the objective function in order to get a highly reusable SROBDD.

6 Conclusions and future work

We have presented a new WCSP solving method, implemented in the `WSimply` system, based on using SROBDDs to generate SAT clauses representing the objective function. Although this is a preliminary work, we have shown that the new method clearly outperforms the previous `WSimply` solving methods on the two tested problems. In addition, we have shown how to boost the ROBDD generation for objective functions taking advantage of previously generated ROBDDs, more precisely constructing a SROBDD.

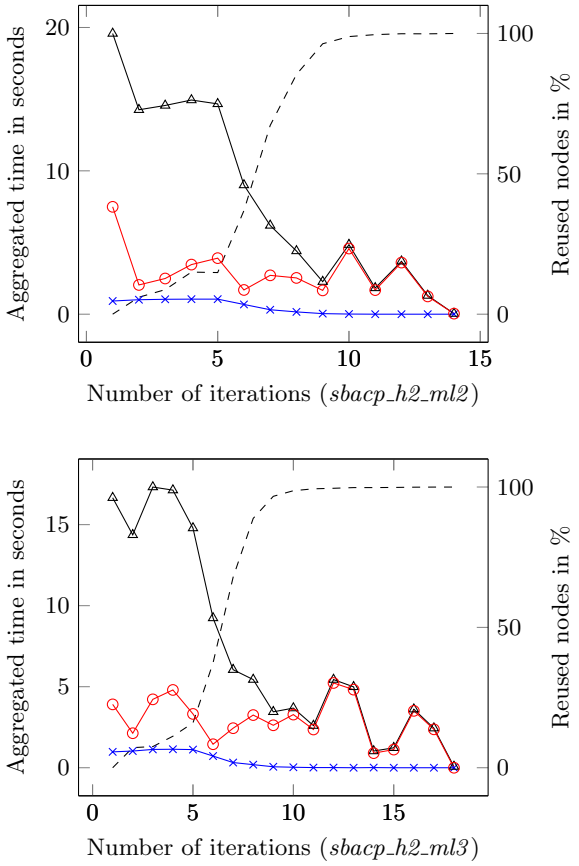


Fig. 7. Average of the percentage of SROBDD reused nodes per iteration (dashed line), aggregated SROBDD construction time (blue cross line), aggregated SROBDD construction time plus aggregated check time (red circle line) and aggregated total time (black triangles line) for the *sbacp_h2_ml2* set (top) and the *sbacp_h2_ml3* set (bottom).

As future work we want to study more deeply the efficiency of our method on other weighted constraint satisfaction problems and compare it with state-of-the-art WCSP solvers like *toulbar2* [3]. Also, as pointed out, an important aspect to study is how to find a good variable ordering for the objective function. Although the problem of finding the optimal variable ordering in order to generate a minimal BDD is known to be NP-hard, we are interested in finding a variable ordering that maximizes the node reuse through iterations. Another aspect that could be interesting to explore is to extend the new method to deal with objective functions with (finite domain) integer variables, using Multi-valued Decision Diagrams (MDDs) to represent them. Finally we also would like to check the efficiency of our new method on weighted MaxSAT and MaxSMT instances.

References

1. I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. A New Look at BDDs for Pseudo-Boolean Constraints. *Journal of Artificial Intelligence Research (JAIR)*, 45:443–480, 2012.
2. S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, June 1978.
3. D. Allouche, S. de Givry, and T. Schiex. Toulbar2, an open source exact cost function network solver. Technical report, INRIA, 2010.
4. C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. A Proposal for Solving Weighted CSPs with SMT. In *Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (ModRef 2011)*, pages 5–19, 2011.
5. C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. Satisfiability Modulo Theories: an Efficient Approach for the Resource-Constrained Project Scheduling Problem. In *Proceedings of the 9th Symposium on Abstraction, Reformulation and Approximation (SARA 2011)*, pages 2–9, 2011.
6. C. Ansótegui, M. Bofill, M. Palahí, J. Suy, and M. Villaret. Solving weighted CSPs with meta-constraints by reformulation into Satisfiability Modulo Theories. *Constraints*, 18(2):236–268, 2013.
7. C. Ansótegui, M. L. Bonet, and J. Levy. Solving (Weighted) Partial MaxSAT through Satisfiability Testing. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*, volume 5584 of *LNCS*, pages 427–440. Springer, 2009.
8. C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.
9. M. Bofill, M. Palahí, J. Suy, and M. Villaret. SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format. In *Proceedings of the Eighth International Workshop on Constraint Modelling and Reformulation (ModRef 2009)*, pages 30–44, 2009.
10. M. Bofill, M. Palahí, J. Suy, and M. Villaret. Solving constraint satisfaction problems with SAT modulo theories. *Constraints*, 17(3):273–303, 2012.
11. E. Boros and P. L. Hammer. Pseudo-Boolean optimization. *Discrete Applied Mathematics*, 123(1-3):155–225, 2002.
12. A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani, and C. Stenico. Satisfiability Modulo the Theory of Costs: Foundations and Applications. In *Proceeding of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2010)*, volume 6015 of *LNCS*, pages 99–113. Springer, 2010.
13. A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. A Modular Approach to MaxSAT Modulo Theories. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT 2013)*, volume 7962 of *LNCS*, pages 150–165. Springer, 2013.
14. S. de Givry, M. Zytnicki, F. Heras, and J. Larrosa. Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence (IJCAI 2005)*, pages 84–89, 2005.
15. L. M. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

16. B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper available at http://yices.csl.sri.com/tool-paper.pdf, August 2006.
17. N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2(1-4):1–26, 2006.
18. E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58(1-3):21 – 70, 1992.
19. S. ichi Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *Proceedings of the 27th ACM/IEEE Conference on Design Automation (DAC 1990)*, pages 52–57, 1990.
20. J. Larrosa and T. Schiex. Solving Weighted CSP by Maintaining Arc-Consistency. *Artificial Intelligence*, 159(1–2):1–26, 2004.
21. P. Meseguer, F. Rossi, and T. Schiex. Soft constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 9. Elsevier, 2006.

Improving the Maintenance Planning of Heavy Trucks using Constraint Programming

Tony Lindgren[1,2], Håkan Warnquist[2], and Martin Eineborg[2]

[1] Department of Computer and System Sciences
Stockholm University, Stockholm, Sweden
[2] Scania CV, Service Support Solutions
Södertälje, Sweden

Abstract. Maintenance planning of heavy trucks at Scania is presently done using static cyclic plans where each maintenance occasion contains a fixed set of components. Using vehicle operational data gained from on-board sensors we will be able to predict at which intervals each component needs to be maintained. However, dynamic planning is needed to take this new knowledge into account. Another benefit using dynamic planning is that vehicle owners can influence maintenance plans with regard to their business. For this reason we have implemented a prototype of an automated maintenance planner based on constraint programming techniques. The planner has successfully been tested on vehicles belonging to Scania's internal haulage contractor. In this paper we will describe the planner and what we have learned using and developing it as well as ongoing work on how the planner will be developed further.

1 Introduction

Scania Commercial Vehicles (Scania) is a manufacturer of heavy trucks, coaches and engines for industrial and marine usage. This paper is concerned with Scania's ongoing effort of improving its maintenance service offer. Better maintenance planning is beneficial for the customers because they can utilize their Scania products in a more efficient manner and it makes Scania more competitive. To achieve better maintenance planning, we have developed a new maintenance planner that uses constraint programming techniques.

Customers are currently offered services such as Repair and Maintenance Contracts enabling the customer to fixate the operating costs of the vehicle [12]. When the vehicle manufacturer has full responsibility for vehicle repair and maintenance, the cost of the repair and maintenance contract can be reduced by customizing the maintenance planning for each individual vehicle. To achieve this customization, more information regarding the current and predicted status of the vehicle is needed. This information can be obtained by employing techniques for Integrated Vehicle Health Management (IVHM) which is an area interested in improving the safety, availability and reliability of vehicles [1, 7].

Using vehicle operational data gained from on-board sensors we can predict when and how often a component needs to be maintained. The components'

individual maintenance requirements make it possible to create more efficient maintenance schedules than using the present scheduling method where, three modules consisting of fixed sets of components are scheduled for maintenance with a preset periodicity [11]. The maintenance of an individual component is referred to as a maintenance point. When the maintenance points can be scheduled freely with irregular periodicity, the task of creating an efficient maintenance plan becomes too difficult for a human planner and there is need for an automated planner.

As a Proof of Concept (PoC), we have implemented a maintenance planner prototype using finite domain constraint programming techniques and evaluated it on the haulage contractor responsible for driving goods to Scania's factories with promising results. We will also report ongoing work with the next generation of the maintenance planner based on the lessons learned from the PoC with the haulage contractor.

2 The Problem

A customer that utilizes a Repair and Maintenance contract wants to maximize the availability of the vehicle and Scania as an issuer of the contract wants to minimize the maintenance costs. Downtime is when the vehicle is intended for use but not available. This time is costly for the customer because of loss of profit. We want a maintenance plan where each component is maintained sufficiently often to prevent components from breaking down and that has minimal interference with the vehicle's intended use. The customer cost of a maintenance plan is dependent on the downtime, the number of maintenance occasions, the part costs, and the time spent at the workshop.

The maintenance need of a component depends on how the vehicle is operated by the owner. A vehicle that is operating with heavy loads may need oil changes more frequently than one that is operating with lighter loads. The better we can predict wear, the more correct maintenance intervals we can use for each component. The vehicle has an internal network of connected computers for controlling functions in the vehicle. The computers collect data about the vehicle which can remotely be sent to a central server for further processing with the purpose of computing the required maintenance intervals.

The inputs to the maintenance planner are the maintenance point intervals of each component and optional user preferences in form of when maintenance can be done and when it cannot be done. The output is a maintenance schedule listing the dates for each maintenance occasion and, for each occasion, a list of components that should be maintained, see Figure 1.

3 Current Solution

Today the maintenance plan for a vehicle is set when the vehicle is sold. This is typically done by the seller together with the buyer by selecting one of a set of

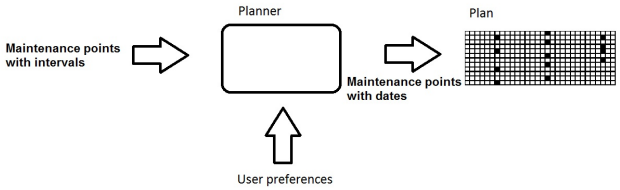


Fig. 1. Inputs and outputs of the maintenance planner.

predefined maintenance plans that best matches the vehicle specifications and the buyers intended usage.

The predefined maintenance plans are developed and maintained by skilled personnel having knowledge about both the products and the customer's usage. Vehicle usage is divided into six typical applications types. For each application type and vehicle specification, a cyclic maintenance plan is given as the number of kilometers between maintenance occasions with fixed maintenance protocols. Maintenance is always done in a cycle of S-M-S-L occasions, where S = Small, M = Medium, and L = Large are different maintenance modules for maintaining different sets of components.

There are a number of problems with the way maintenance plans are created today:

- Much responsibility is put on the salesperson to know the product as well as the customer's usage of the product.
- Once created the plans are seldom updated even if the application of the vehicle changes. Thus, it is possible that the maintenance a vehicle receives does not correspond to its needs.
- Although the fixed S, M, and L modules make it convenient to plan, they contain maintenance points that do not need to be grouped together with the effect that components are maintained more than necessary.
- The current maintenance plans are coarse in the sense that the precision in the type of application must be fitted into one of the six types of application. Therefore the experts dictating when maintenance ought to be done, use a safety margin given the uncertainty of the actual usage of a particular vehicle. This has the consequence that plans are not individualized to the degree that they could be.

4 An Automated Maintenance Planner

We have used Constraint Programming to create an automated maintenance planner which has been installed and used by two workshops servicing 20 ve-

hicles belonging to the Scania Transport Laboratory which is a Scania-owned transport company responsible for transporting goods between Scania's factories in Europe.

4.1 Motivation

The work load of the trucks is high and usage of around 16 hours a day is not unusual. To avoid interference with the daily operation of the trucks, a requirement from the fleet planner was that the trucks could only be maintained every forth week for a maximum of four hours. Such requirements together with previously mentioned goals of minimizing maintenance costs and offering better services to customers was the main driving force for developing this maintenance planner. The prototype was created to gain knowledge of how a solution could be implemented and what aspects are critical for Scanias customers.

The planning problem is too complex to be solved manually. We therefore chose to formulate the problem as a constraint satisfaction problem because many of the requirements on the plan are naturally translated into constraints and also because Constraint Programming techniques has historically been successful for applications similar to this. For example see [10, 8, 5, 2, 6].

4.2 Formulation of the Constraint Satisfaction Problem

The maintenance planning of a single vehicle is formulated as an independent Constraint Satisfaction Problem (CSP). A solution is a plan for all maintenance points with a resolution of one week and a limited horizon. In the Scania Transport Laboratory PoC each vehicle had around 80 maintenance points that needed to be scheduled 52 weeks ahead.

Each variable in the CSP corresponds to a maintenance point that needs to be scheduled in time. Where the Dm_i refers to the domain of the i:th maintenance point. The latest completion time (lct) of a maintenance point refers to its calculated maintenance interval. To reduce the solution space, a constraint is added that dictates the minimum maintenance interval of each maintenance point, i.e. earliest start time, (est). The est is user defined and typically between one third to half of the calculated maintenance interval as shown in equation 1. This also ensures an offset between two maintenance occasions are no closer than the est. In all equations, I refer to the set of maintenance point variable indexes.

$$EarliestStartTime : \forall_i \in I : est_i = lct_i - \frac{lct_i}{offsetParam \in [2,3]} \tag{1}$$

$$Domain : \forall_i \in I : Dm_i = [est_i, lct_i] \tag{2}$$

The usage of est_i affects the i:th maintenance points domains as in equation 2. Maintenance point dependency chains are defined by assigning a *starting variable* which has a domain value between est_i and lct_i. Each variable in these dependency chains corresponds to an occasion of a maintenance point and the

value corresponds to the time when the maintenance should occur. Successive maintenance points are then created until the planning horizon is reached. In the dependency chains each new variable gets a domain with a earliest plan starting time mpv_j that is equal to or greater than the preceding maintenance point variable $mpv_i + est_j$ and a latest plan completion time for mpv_j that is less than or equal to $mpv_i + lct_j$, shown in equation 3 and equation 4.

$$EarliestPlanStartTime : \exists_{i,j} \in I : mpv_j \geq mpv_i + est_j \tag{3}$$

$$LatestPlanCompletionTime : \exists_{i,j} \in I : mpv_j \leq mpv_i + lct_j \tag{4}$$

After the maintenance point dependency chains have been created, then, if the user has defined certain periods when maintenance can be done (*cbd*), or when it cannot be done (*cnbd*), these periods are handled as show below.

$$CanPeriod : \forall_i \in I : Dm_i = Dm_i \cap cbd \tag{5}$$

$$CannotPeriod : \forall_i \in I : Dm_i = Dm_i \setminus (Dm_i \cap cnbd) \tag{6}$$

In practice this means that *cnbd* periods are excluded from the variable domains. If *cbd* is defined, then these periods constitute the variables domain. Each dependency chain is related to one maintenance point which means that for our PoC there are around 80 dependency chains.

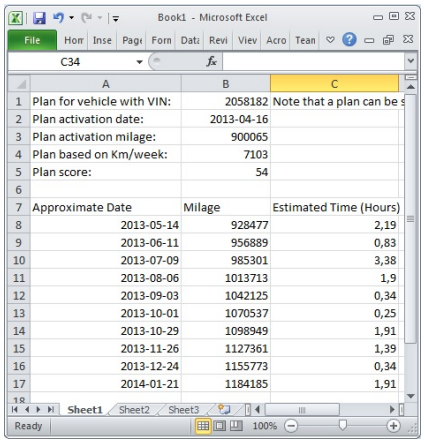
Input and Output. The input consists of the periodicity of each maintenance point expressed in kilometers and the times when each maintenance point was last maintained. The periodicity is then converted into weeks based on the expected number of kilometers the vehicle will be used per week which can either be set by the user or be learned from previous vehicle behavior.

If it exists, the solver finds a maintenance plan that satisfies all the constraints. This plan is then presented to the user as an Excel worksheet listing the dates, expected mileages, and durations of all the scheduled maintenance occasions, see Figure 2. The dates are approximate because the planner only plans with a resolution of one week. The exact date within that week must be set in dialogue between the fleet owner and workshop. The maintenance planner can also output the maintenance protocols that shall be used for each occasion.

Heuristics and Propagators. The solver in the clp(FD) library is set such that it will return the first maintenance plan it finds that satisfies all the constraints. This means that the search heuristics are important for the behavior of the planner. The user can select between two search heuristics.

The first heuristic uses the built in parameters of the clp(FD) library so that the variable, not yet assigned, with the smallest domain is chosen and the maximum value of its domain is selected.

The second heuristic is specific for this problem formulation. Variables to assign are selected in the same way as the first heuristic, but the value selection is different. If assigned values corresponding to different maintenance points lie



	A	B	C
1	Plan for vehicle with VIN:	2058182	Note that a plan can be s
2	Plan activation date:	2013-04-16	
3	Plan activation milage:	900065	
4	Plan based on Km/week:	7103	
5	Plan score:	54	
6			
7	Approximate Date	Milage	Estimated Time (Hours)
8	2013-05-14	928477	2,19
9	2013-06-11	956889	0,83
10	2013-07-09	985301	3,38
11	2013-08-06	1013713	1,9
12	2013-09-03	1042125	0,34
13	2013-10-01	1070537	0,25
14	2013-10-29	1098949	1,91
15	2013-11-26	1127361	1,39
16	2013-12-24	1155773	0,34
17	2014-01-21	1184185	1,91

Fig. 2. An example of the maintenance plan of a vehicle.

within the domain of the selected variable, we select the assignment with the highest value. If such a value does not exist, the variable is assigned to the highest value in its domain like the first heuristic. The motivation for this heuristic is that we want to co-locate maintenance points in time so that we get as few maintenance occasions as possible.

Pseudo-code for the value selection heuristics are shown in Algorithm 1. The search heuristic takes a constraint store as input and returns a modified constraint store. The function ValSel uses the constraint store and the variable with the smallest domain to assign a value to the variable using the function GetBestValue. The function GetBestValue uses the constraint store and the variable with the smallest domain and returns either the highest value of the intersection between the selected variable and any other assigned variable or, if no such intersection exists, returns the highest value in the domain of the selected variable. The function Max returns the highest value of a domain and the functions GetNextAssigned iterates over assigned variable in constraint store and returns false when all has been shown. The function Intersect returns the elements in the intersection and the function FirstBound returns true if it is the first time its argument variable is assigned and false otherwise.

A new propagator was implemented for the controlling the maximum time of each occasion. This propagator is executed whenever a variable is assigned a value. Each maintenance point has a standard time associated with it, i.e. the time for the mechanic to complete the maintenance point task. The propagator has a week-time-list where it keep track of current summarized work time for each week up to the horizon. For each new assignment this list is updated, and each variable that is not assigned, is checked one at a time, if the summarized standard time exceed the time limit or not. If the time limit is breached, the week is removed from the variable's domain. If the propagator cannot exclude a

Algorithm 1 Search heuristic for few maintenance occasions

```
Inputs: constraintStore, selVar
Outputs: constraintStore

function ValSel(constraintStore, selVar)
    tVal ← GetBestValue(constraintStore, selVar)
    if FirstBound(tVal) then selVar ← tVal
    else
        selVar ≠ tVal
        selVar ← GetBestValue(constraintStore, selVar)
    end if
    return constraintStore
end function

function GetBestValue(constraintStore, selVar)
    tempVal ← 0
    maxVal ← 0
    while var ← GetNextAssigned(constraintStore) do
        intSect ← Intersect(selVar, var)
        if ∅ ≠ intSect then
            tempVal ← Max(intSect)
            if tempVal > maxVal then maxVal ← tempVal
        end if
    end while
    if maxVal > 0 then
        return maxVal
    else
        return Max(selVar)
    end if
end function
```

value from any variable's domain the propagator fails. The pseudo-code for the propagator are shown in Algorithm 2.

The input is the constraint store, maximum time, week-time-list and the assigned week, the output is a possibly modified constraint store. The function SumAssignedStandardTimes uses week-time-list and the assigned week to update the week-time-list with the standard time of the corresponding maintenance point. The function GetNextUnAssigned iterates over the constraint store and returns the next not assigned variable. The function Intersect returns the intersection between the variable and week. GetStTime returns the standard time for the variable, i.e. a maintenance point standard time. GetNext iterates over a week-time-list and if no value are left return false. ReVal removes the current week from the variable domain and, finally the function NoMoreUnAssigned returns true if no more unassigned values are left to iterate over in the constraint store.

Algorithm 2 Maximum time propagator

Inputs: $constraintStore, week, weekTimeL, maxTime$
Output: $constraintStore$

$weekTimeL' \leftarrow$ SumAssignedStandardTimes($weekTimeL, week$)
repeat
  $var \leftarrow$ GetNextUnAssigned($constraintStore$)
  $intSect \leftarrow$ Intersect($week, var$)
  if $\emptyset \neq intSect$ **then**
    $varWeekTime \leftarrow$ GetStTime(var)
    if $maxTime < varWeekTime$ **then**
      $var' \leftarrow$ ReVal($var, week$)
      if $var' = \emptyset$ **then return** $Fail$
    end if
  end if
until NoMoreUnAssigned($constraintStore$)
return $constraintStore$

4.3 Implementation

The maintenance planner is implemented in SICStus Prolog [14] using the clp(FD) library for Constraint Logic Programming over Finite Domains [3]. Users interact with the planner through a simple command prompt interface providing functions for setting certain constraints, creating maintenance plans, and outputting maintenance protocols. A screenshot from the user interface is shown in Figure 3. The user has the ability to create new maintenance plans, update existing ones, view maintenance history, and set various settings.

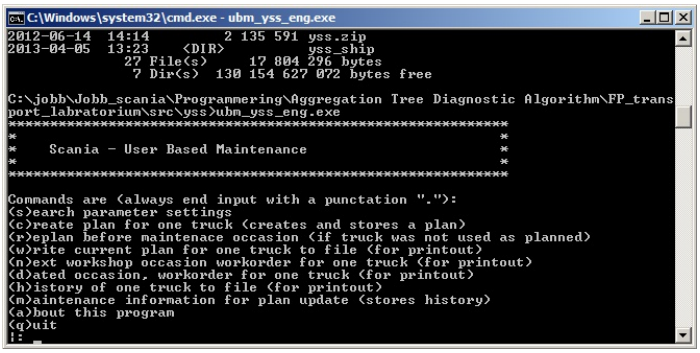


Fig. 3. Main menu of the user interface.

4.4 Typical Usage Pattern

During the PoC at the Scania Transport Laboratory, the typical usage of the maintenance planner was as following:

- One week before a scheduled maintenance occasion, the workshop planner checks using telemetry the actual mileage of the vehicle and regenerates the maintenance plan. If the mileage is lower or higher than expected, fewer or more maintenance points needs to be done at this occasion.
- With the content of the next maintenance occasion fixed, the workshop planner prints out the maintenance protocol and orders the parts needed for the occasion as specified by the protocol.
- When the vehicle is at the workshop, a mechanic performs maintenance according to the protocol.
- After the maintenance, the workshop planner reports back into the system which maintenance points that were addressed and creates a new updated plan for the vehicle. When the new maintenance plan is created the workshop planner may use the current mileage per week or manually set it to a new value if a different driving behavior is anticipated in the future.

5 Results

The vehicles that participated in the study were all of similar type and had similar driving patterns. Table 1 shows a comparison between S, M, and L plans and the automated planner for a representative vehicle from the Scania Transport Laboratory. This vehicle is a long haulage truck that has the expected usage of 6 760 km per week. The interval between maintenance occasions with the S, M and L program for such a vehicle is once every 90 000 km, or once every 13.3 weeks. For each maintenance occasion we have reported the standard time for completing all maintenance points scheduled for the occasion. Despite that the S, M, L program neither respects the periodicity or the time limit constraints

the sum of all standard times is higher. This is because the intervals for certain maintenance points could be stretched further when it no longer has to fit within the S, M, and L modules. The gain is potentially much larger, because for this study only a handful of the maintenance points had their intervals re-evaluated while most were the same as in the S, M, L program (90 000, 180 000, or 360 000 km).

Table 1. Comparing standard method and new automated planner.

	Standard	New Automated Planner
Visits	8	11
Total Time	33.4	28.9

The experiment with the haulage contractor was intended as a PoC of generating maintenance plans automatically at a much finer granularity than before. The haulage contractor requirements on the maintenance plan that could not be satisfied with the previous planning method. For example, the largest maintenance module, L, takes longer than the required maximal four hours of stand still.

6 Thoughts on the Next Generation Planner

The maintenance planner should, in the future, be extended to do optimization instead of only returning the first solution. Hence we have started doing some experiments using this set up. This section is dedicated to describing our current findings in this experimental work.

The maintenance planning problem, as formulated, has few constraints associated with it. Therefore, we must resort to using search while exploring possible solutions. Using multi-core parallel search will let us explore a larger part of the solutions space in a smaller amount of time than a single-core solution.

We choose to use the Gecode [13] toolkit for exploring a possible multi-core solution. The reasons for this choice is that Gecode has built-in support for parallel search, using a variant of work stealing [4] for handling and assigning computational tasks to idle threads without more involvement from the user than selecting the number of threads to be used. Gecode also has an excellent performance record, winning the MiniZinc Challenge [15, 9].

Our intention is to use branch-and-bound techniques for optimization. So far we have implemented an objective function to test the branch-and-bound approach. The objective function allows users to set their preferences for few maintenance occasions or minimizing waste using weights, where waste refers to not utilizing maintenance point intervals fully:

$$cost = noOccassions \times occasionsWeight + waste \times wasteWeight. \quad (7)$$

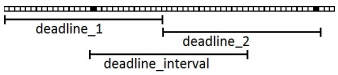


Fig. 4. Deadline interval constraint for the matrix problem formulation.

When using a multi-core solution, it is important how much memory a particular problem formulation needs, because it will need multiple copies of the constraint store. For our problem description we have identified two different problem formulations. We will describe them and run experiment focusing on their respective memory and CPU time needs.

Matrix Problem Formulation. One way to formulate the problem is to create a matrix with the same number rows as there are maintenance points and the same number of columns as there are days within the planning horizon.

- Variables. For each cell in the matrix a constraint variable is defined with domain between zero or one. The assigned value at a specific row and column denotes whether the maintenance point associated to the row should be performed during the day associated to the column. A one means that the maintenance point should be performed and a zero means that it should not.
- Constraints. Maintenance point intervals for a row are regulated by a constraint that all cells in the row up to the deadline interval should sum to 1. Hence only one maintenance occasion should occur within interval, as is shown in equation 8. This constraint is repeated for each interval up to the planning horizon.

$$SumToOne : \sum_{i=0}^{interval} cell_i = 1 \tag{8}$$

This is however not enough to ensure that the plans are correct because it can be the case that a solution ends up with a distance between two one's (1) that is greater than the interval. This is illustrated in Figure 4. Here two constraints are set, where each of their interval must sum to one but still we can end up with a plan that violates the deadlines.

One way of correcting this is to add a constraint which keep track of the number of zeros in a row and make sure that the length of the zeros do not exceed the deadline. This constraint is illustrated in equation 9, where zerosInRow takes the i:th row and returns the maximum consecutive number of zeros and, maxZeros is the limit for consecutive zeros on this row.

$$zerosInRow(row_i) < maxZeros_i. \tag{9}$$

Unable to express this constraint as an extensional constraint, we implemented a new propagator. The propagator uses the notion of 'blocks' of zeros,

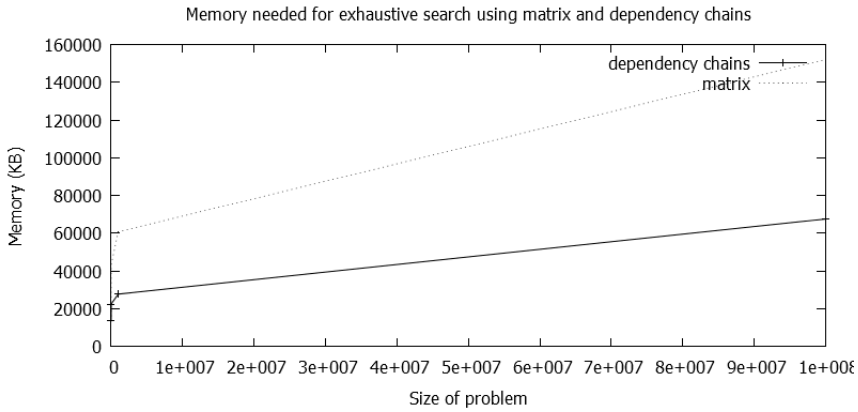


Fig. 5. Comparison of memory needs.

which can be merged if two blocks are adjacent to each other, creating a bigger block. New blocks are created when a new zero is assigned that has no adjacent blocks. The propagator first checks the constraint store so that no deadline interval is breached. Then it propagate a one if there exists a block with the same length as the interval minus one. With this propagator the planner behaves as desired.

As with the previous planner, the user can specify the minimal interval with a parameter.

Dependency Chains. The second problem formulation is the same as the one used for the first single-core maintenance planner. Each variable corresponds to an occasion of a maintenance point and the value corresponds to the time when this occasion should occur.

Experiment. We have implemented both formulations of the problem in Gecode 3.7.3 and examined their CPU usage and memory needs when conducting an exhaustive search for a planning problem with one maintenance point and a maintenance interval of 10 days. The planning horizon varied from 20 days to 80 days, which causes the total number of possible solutions to vary from 100 to 100 million.

They executed on a quad-core Intel i7-2760QM processor running at 2.4 GHz. The comparison of the memory need is shown in Figure 5 and the comparison of the CPU times is shown in Figure 6.

As expected, the matrix problem formulation consumes more memory and more CPU time to complete the exhaustive search problems. In this experiment we had only one maintenance point and the maximum planning horizon was 80 days. In a realistic setting we have almost a hundred maintenance points and we need a planning horizon of up to 365 days. Thus in our application using the

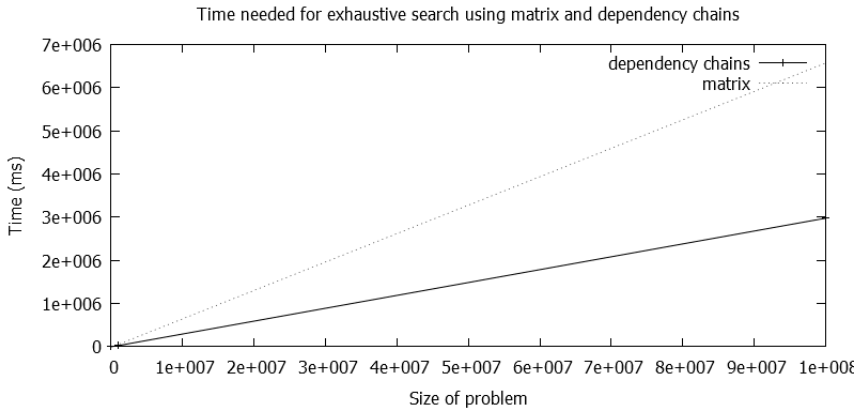


Fig. 6. Comparison of CPU time.

dependency chain problem formulation is preferable to using the matrix problem formulation.

7 Discussion

Scania gained much experience from developing a maintenance planning system for its internal haulage contractor, both in terms of what functionality a workshop planner needs and how constraint programming can be used to realize this functionality. Based on this new knowledge, work has begun with the next generation of the maintenance planner, as already mentioned. Apart from improved user interface we will implement more ways for the user of the planner to influence what constitutes good plans. Here we are considering more constraints and optimization parameters. Furthermore, the group of intended users of the planner is expanded to include fleet planners and sales personnel. We also want to do more exact planning and have decided to use a time scale of days instead of weeks for this next generation of the planner.

Development The development of the planning prototype was not more difficult than a single developer familiar with constraint programming techniques could design and implement the entire application in a few months. However, more effort was required from the experts that had to set maintenance intervals since this was made manually. The intention is that in the future these intervals will be set automatically based on data.

When the application was delivered to the workshops, a Scania engineer created the first plan for all the vehicles based on a default expected mileage. The users at the workshops were only supposed to use the application to read out maintenance plans and protocols and to input the actual times and mileages when each maintenance point is performed. Early in the experiment, it became

evident that a re-plan functionality was needed to make sure that the maintenance plan was correct at the designated date for maintenance. Usually the workshop planner re-planned the schedule for a vehicle scheduled a couple of days before the planned maintenance occasion. After this functionality had been added we saw that sometimes planned maintenance occasions could be avoided due to less vehicle usage than predicted. In some cases it was the opposite and more maintenance was needed than in the previous plan.

Another appreciated functionality that was added later was the possibility to output a list of consumable goods (e.g. oil quantities and part numbers of filters) together with the maintenance protocol. This made it possible for the workshop to make sure that all necessary products were in place in time for the maintenance occasion. This was not as important with the fixed maintenance protocols since the list of consumable goods are always the same.

Release to Customers Because of the rather primitive user interface, education of the managers responsible for planning at the workshop, was important for the users to understand how the system worked. As a part of this, users were encouraged to create simulated maintenance plans using the system.

One user at the workshops was assigned as superuser. The superuser and developer had regular meetings where they could discuss problems and questions regarding the application. The superuser could collect opinions about the system from the other users and also educate them. During the first two months many changes to the application were made because of feedback from the users.

Initially the program was unstable and would crash if fed with illegal input or if the constraints were set so that no valid plan existed. Also there were problems with the dynamic creation of maintenance point variables causing unnecessary maintenance points to pile up at the end of the plan. This had no real consequence for the performance of the plans because the remainder of the plan was correct and the unnecessary maintenance points would always be pushed ahead whenever the maintenance plan was re-planned. However it looked bad and confused the users. All data such as previous maintenance and mileages were saved in Prolog using its program state. This prevented users from correcting erroneous input. Instead users were instructed to store and keep copies of previous entire program states to do roll-back upon if the system contained information that did not align with reality.

Once these and other problems were corrected and the users had become acquainted with the program, we started getting positive feedback from the users. Some users even preferred the command prompt interface over a graphical user interface because interaction with the program was really fast.

Apart from the obvious fact that we could not satisfy the requirement of having fixed maintenance occasions every fourth week with a time limit of four hours, using the standard maintenance program, a preliminary comparison show that gains in time and money can be made with the new planning system.

8 Conclusion

The possibility to individually plan each maintenance point allows us to create more efficient maintenance plans that also consider the needs of the vehicle owner. Previously, many of these needs have been disregarded. A purpose of the PoC was to gain a better understanding of these needs and the potential gain of creating an automated maintenance planner based on constraint programming techniques.

The PoC was developed with limited resources, but the testers at the Scania Transport Laboratory were tolerant and put up with the initially buggy planner and gave back precious feedback on how to develop it further. This way of working is fine when, as in this case, the test group is a small group of users belonging to a subsidiary company. However, it would not be feasible for a test on a larger scale. We learned that, even with a primitive maintenance planner such as this PoC, the maintenance costs can be significantly reduced and user preferences that previously were ignored now can be regarded.

The PoC maintenance planner showed us that a planner based on constraint programming techniques is a good way to go. Constraint programming is a good framework for expressing and solving combinatorial problems for which human capabilities are not sufficient to cope with. The maintenance planning problem, as it has been formulated so far, has not been very constrained and had more characteristics in common with a search problem. However, if more user constraints are added, the planning problem may well prove to move from a problem with a dense distribution of solutions to one where they are sparse. Then the benefit of constraint programming may become even greater.

For example, we may want to consider when and where there is a workshop that can perform maintenance on a vehicle. This may make it necessary to extend the planning to multiple vehicles. Also the fleet planner may have its requirements on where the vehicles must be at given times and how far and fast they can travel.

Methods for estimating the remaining useful life of components to create predictive models for maintenance is currently under investigation at Scania. The maintenance interval of a component would then change dynamically, which could affect the stability of maintenance plans. This may not be a desired behavior of the planner and therefore the maintenance planner must support constraints regulating how an existing plan may change when new input arrives.

For all these possible extensions to the maintenance planning problem, a solution based on constraint programming appears the most promising.

9 Acknowledgments

This work has been funded by Scania CV AB and the Vinnova program for Strategic Vehicle Research and Innovation (FFI).

References

[1] Ian K Jennions et. al. *Integrated Vehicle Health Management: Perspectives on an Emerging Field*. Ed. by Ian K Jennions. SAE, 2011.

[2] J. Christopher Beck and Philippe Refalo. "A Hybrid Approach to Scheduling with Earliness and Tardiness Costs". In: *Annals OR* 118.1-4 (2003), pp. 49–71.

[3] Mats Carlsson, Greger Ottosson, and Björn Carlson. "An open-ended finite domain constraint solver". In: *Programming Languages: Implementations, Logics, and Programs*. Ed. by Hugh Glaser, Pieter Hartel, and Herbert Kuchen. Vol. 1292. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 191–206.

[4] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. "Confidence-based Work Stealing in Parallel Constraint Programming". In: *Fifteenth International Conference on Principles and Practice of Constraint Programming*. Ed. by Ian Gent. Vol. 5732. Lecture Notes in Computer Science. Lisbon, Portugal: Springer-Verlag, Sept. 2009, pp. 226–241.

[5] Tom Creemers et al. "Constraint-Based Maintenance Scheduling on an Electric Power-Distribution Network". In: *Proc.of the Third International Conference on the Practical Application of Prolog*. Paris, 1995, pp. 135–144.

[6] Safaai Deris, Sigeru Omatu, and Hiroshi Ohta. "Timetable planning using the constraint-based reasoning". In: *Computers & Operations Research* 27.9 (2000), pp. 819 –840. ISSN: 0305-0548.

[7] Jon Dunsdon and Mark Harrington. "The Application of Open System Architecture for Condition Based Maintenance to Complete IVHM". In: GE Aviation.

[8] Sami Gabteni and Mattias Grönkvist. "Combining column generation and constraint programming to solve the tail assignment problem". In: *Annals OR* 171.1 (2009), pp. 61–76.

[9] MiniZinc organization. *MiniZinc Challenge*. Apr. 2013. URL: `http://www.minizinc.org/`.

[10] José Palma et al. "Scheduling of maintenance work: A constraint-based approach". In: *Expert Syst. Appl.* 37.4 (Apr. 2010), pp. 2963–2973. ISSN: 0957-4174.

[11] Scania CV. *Scania Fixed Price Repair programme extended*. Apr. 2013. URL: `http://www.scania.co.uk/about-scania/media/press-releases/2009/fixed-price-repair-programme-extended.aspx`.

[12] Scania CV. *Scania Repair & Maintenance contract*. Apr. 2013. URL: `http://www.scania.com/products-services/services/workshop-services/repair-maintenance-contract/`.

[13] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling and Programming with Gecode*. 2010. URL: `http://www.gecode.org/doc-latest/MPG.pdf`.

[14] *SICStus Prolog User's Manual*. Intelligent Systems Laboratory, Swedish Institute of Computer Science. 2013.

[15] Peter J. Stuckey, Ralph Becket, and Julien Fischer. “Philosophy of the MiniZinc challenge”. In: *Constraints* 15.3 (July 2010), pp. 307–316. ISSN: 1383-7133.

Modeling Distributed Information: Send+More=Money

Andrés F. Barco

Department of Electronics and Computing Science
Pontificia Universidad Javeriana - Cali
Cali, Colombia, South America
anfelbar@javerianacali.edu.co

Abstract. Concurrent behavior is present in most information and communication technologies, from Web Services to Social Networks and Cloud Computing. For these systems, the knowledge representation of the involved agents are of crucial importance for an accurate description and modeling of their behavior. Further, the distributed nature of information forces to analyze system constraints in both processing and storage capabilities. On this regard, we present simple implementations of the send+more=money puzzle using several scenarios in which the information is distributed among a set of agents that may know certain information while ignoring other. We use a constraint-based interpreter to model the puzzle according with spatial and epistemic specifications. We use three well-know examples to illustrate agent interaction.

Introduction

Concurrent behavior is present in most information and communication technologies, from Web Services to Social Networks and Cloud Computing. For these systems, the knowledge representation of the involved agents are of crucial importance for an accurate description and modeling of their behavior. Indeed, reasoning about other agents' knowledge is a fundamental aspect on behavioral approaches, e.g., game theory [10], artificial intelligence [12] and logic [3]. In each scenario, see [3, 7] for some examples, where epistemic interactions takes place, the sharing of knowledge, e.g., commercial strategies, connections, logins and passwords, allows different properties to emerge.

On this regard, we present simple implementations of the send+more=money puzzle using several scenarios in which the information is distributed among a set of agents that may know certain information while ignoring other. Special attention should be paid to the representation of knowledge and distributed information. We use a constraint-based interpreter [4] to model the puzzle according with spatial and epistemic specifications. The interpreter is based on two novel process calculi that use spatial and epistemic modal operators as its underlying logic [11]. In consequence, the interpreter allow to play with agents that have different processing and storage constraints. The distributed nature of information enables a large family of problems to be specified in the calculi and

to be simulated by the interpreter. The addressed puzzle shows the potential expressive power of the calculi and its modeling benefits. Along the paper we use three well-know examples to illustrate agent interaction.

The paper is structured as follows. The section 1 is dedicated to a brief description of the puzzle, the spatial and epistemic ccp calculi and the tool we use. Next, we describe the notion of inconsistency confinement which reinforces the spatial model. Section 3 illustrates the concept of distributed knowledge among a set of rational agents. We present the implementation of a spatial and epistemic send+more=money in section 4. The last section is dedicated to some remarks and related work. Finally, a bibliography is included.

1 Preliminaries

In this section we present puzzle specification along with the mathematical model and the programming environment we use for the implementation. As the paper does not focus on the model nor the tool but in modeling, the read interested in more information should look it in the references. The reader should know, however, that it is not mandatory to deepen in the mathematical foundations, a basic understanding of constraint programming will suffice.

1.1 The puzzle

The well-known puzzle send+more=money was created almost a century ago. The man who created it was Henry Dudeney [6], a recognized mathematician of his time. Dudeney argued that the simplest puzzle should not be passed over without careful attention. He considered in particular arithmetic problems, like the send+more=money, to be the most interesting ones. Indeed, mathematicians and logicians reason about the underlying true behind the puzzles using different representation and intuitions. Nonetheless, although the send+more=money puzzle was not conceived to be studied within a particular model, it seems that the the problem fits neatly with the declarative nature of the constraint programming paradigm. The basic description of the puzzle is shown next. The specification is taken from the Oz Programming Interface webpage [1].

Send+more=money The Send More Money problem consists of finding distinct digits for the letters D, E, M, N, O, R, S, Y such that S and M are different from zero (no leading zeros) and the equation SEND+ MORE=MONEY is satisfied. The unique solution of the problem is 9567+1085= 10652.

1.2 Spatial and epistemic concurrent constraint calculi

The traditional concurrent constraint programming viewpoint is not well equipped for the modeling of distributed information. This is due to the local computing constraints agents may have, i.e., both local processing and storage. Spatial ccp

and Epistemic ccp [11] extend concurrent constraint programming calculus in order to capture some notions of spatiality and knowledge that have not been addressed by other extensions. We encourage the reader to study the theoretical model created by Knight et al. as described in [11] to get a better understanding of the calculi. In the following, we describe the language of constructions terms.

Let P and Q be two spatial-epistemic ccp processes. The language of construction terms as:

$$P, Q := \mid tell(c) \mid ask(c) \rightarrow P \mid P \parallel Q \mid [P]_i \mid X$$

Assuming that for each process variable X there exists a process definition, possibly recursive, of the form $X \stackrel{def}{=} P$.

Intuitively, each agent i has his own local store $[\cdot]_i$ where processes and other agents' stores may reside. Consequently, the spatial construct $[P]_i$ represents the process P within the store of agent i. Ask and parallel semantics remains as in classic ccp. However, the application of tell semantics has two different meaning.

Essentially, a tell operation may be seen as spatial or epistemic. Used as a spatial operator, a tell only adds partial information to the agent's store. This is akin to the notion of belief; agent i may believe that it is raining while agent j believes it is not. Consequently, no spatial tell operation can cause the overall computation to fail, i.e., $[tell(false)]_i \neq false$ and $[tell(c)]_i \sqcup [tell(d)]_j \neq false$ even when $c \sqcup d = false$[1]. On the other hand, tell operator in its epistemic forms allows to represent facts. Thus, $[tell(c)]_i \sqcup [tell(d)]_j = false$ when $c \sqcup d = false$. Intuitively, the process $[tell(c)]_i$ is that c is added to the knowledge of agent i. Some interesting properties of the epistemic tell operator are that; a) after $[tell(c)]_i$ is executed, the store entails c (for any constraint c); meaning that if an agent knows something then it is true; b) $[tell(c)]_i$ is idempotent; meaning that agent i knows that he knows c; and c) after $[[tell(c)]_j]_i$ is executed, c will be in i's store. This is because if i knows that he knows c, he can conclude c from this fact. The details of spatial and epistemic constraint systems, along with their operational and denotational semantics, can be found in [11].

1.3 K-stores: Sccp and Eccp Interpreter

In essence, an interpreter is a computer program that executes expressions of a particular programming language. Such program, often called the *defined* language, is built using other programming language, also called the *defining* language [13]. An useful characterization for programming languages is related with the underlying logic the language is based on. Most programming languages and interpreters are based in the well-studied logic of the λ-calculus [2]. This calculus is universal in the sequential computation, i.e., any computable function can be expressed using the calculus. However, most of real-life system do not exhibit a sequential behavior but a concurrent and reactive one. Thus, using the λ-calculus for the modeling of concurrent-reactive systems is not likely.

[1] The symbol $\sqcup$, least upper bound, represent the join of information.

The `K-stores` interpreter is a Prolog implementation of the operational semantics of the spatial and epistemic ccp calculi allowing the programmer to simulate distributed information systems [4]. Its main feature consists of an implementation of a spatial (distributed) store that allows epistemic information in it. In particular, it implements the S4 epistemic logic axiomatic system [3, 9]. The system supports the specification of (named) processes along with the ccp classic primitives, namely, ask and tell operations. The interpreter is built with the SWI-Prolog programming environment [15] and uses its constraint logic programming module as its underlying constraint system. For further information about this tool refer to [4].

2 Important Properties

Distributed Knowledge `K-stores` allows the programmer to simulate real-life scenarios. Scenarios are characterized by having spatial properties or epistemic ones. Perhaps, a given system exhibits spatial and epistemic properties at the same time, however, our decision is to define programs specified either epistemically or spatially.

It is worth noting the difference between spatial and epistemic information. If an agent makes a spatial tell operation in his own store, the information added does not affect other agent's stores. On the contrary, an epistemic tell will, potentially, change other stores. For instance, if the agent A knows that agent B knows ϕ, then B must know ϕ. This is due that epistemic structure S4 does not allow people to know false statements [3, 9]. Conversely, if agent A knows that X is equal to 0 and agent B knows that X is equal to 1, an incosistency is raised.

In this section we give some intuition about possibles scenarios in a give problem. Such intuition will allow us to describe the distributed information that may be present in a system. We use the example called *The logicians* [7] because it makes evident the distributed nature of information and that agents can gain information independently of each other. In scenarios such as this, any given agent has his own processing and storage constraints. Consequently, all agents, represented with different stores, use a particular (possibly) disjoint set of constraints to draw its own conclusions.

The logicians The logicians Alice and Bob are sitting in their windowless office wondering whether or not it is raining outside. Now, none of them actually knows, but Alice knows something about her friend Carol, namely that Carol wears her red coat only if it is raining. Bob does not know this, but he just saw Carol, and noticed that she was wearing her red coat.

The fact that is raining can become knowledge for Alice and Bob if they make the right assumptions and question. This is due that the fact "it is raining" is distributed knowledge among Alice and Bob. In short, an event ϕ is distributed knowledge among a group of agents G if and only if ϕ follows from the knowledge intersection of all individual agents in G. Semantically, ϕ is distributed knowledge

among G if and only if ϕ is true in all worlds that every agent in G considers possible [14]. Distributed knowledge can be modeled as $D_G(\phi) = \bigwedge_{i \in G} K_i$

Where $K_i\phi$ means the event that "agent i knows ϕ". The result of the operator K over an agent can be seen as a projection over the state of possible worlds that shapes the agent behavior. This notion is found materialized in most distributed information systems as social networks.

Isolation One of the most important intuitions of the spatial ccp calculus is the isolation of information, also referred to inconsistency confinement in [11]. A graphical representation may help us understand the relevance of store isolation to represent agents' knowledge. In particular we want to reason about agents that reason about other agent's knowledge. The next example, taken from [5], and its representation shows us the importance of first order knowledge and higher order knowledge. The graphic makes clear that knowledge can be modeled using the constraint programming paradigm which is one of our goals.

Hats puzzle Three people, Adam, Ben and Clark are siting in such a way that Adam can see Ben and Clark, Ben can see Clark and Clark can see no one. Without seeing the color, a white hat or a red hat is put in the head of each agent. Suppose that all three hats are red. Then, with open eyes, they are asked whether they know what the colors of their hats are. In this setup all agents answers are negative. However, when an agent makes the next announcement "there is at least one of you wearing a red hat" the result is quite different. They are asked again which color they are wearing. Then Adam answers that he does not know. Then Ben answers that he does not know either. Finally Clark says that he knows the color of his hat. What color is Clark's hat?

The possible state of knowledge for the puzzle, shown in fig 1, may be represented[2] as a set of vertices [9]. Each corner of the cube is defined by three values. All of them may be either `W` or `R` representing whether the hat of the agent i is white or red, respectively. Each of those vertices in the cube models a given state of affairs in the agents' minds. Thus, the first cube is the moment when no body has answer the questions; they think that any combination of hats' colors is possible.

An agent is sure about his hat color in a given state, if none of their vertices connects with another vertex in which the color of his hat is different. It is worth noting that Clark's knowledge is unambiguous; Clark knows that his hat is actually red. Public announcements of his two friends enabled Clark to find the answer. Distributed knowledge is generated only when Adam and Ben answer the question aloud. We leave the reasoning of this puzzle to the reader.

[2] The graphical representation is an adaptation of one presented in [9].

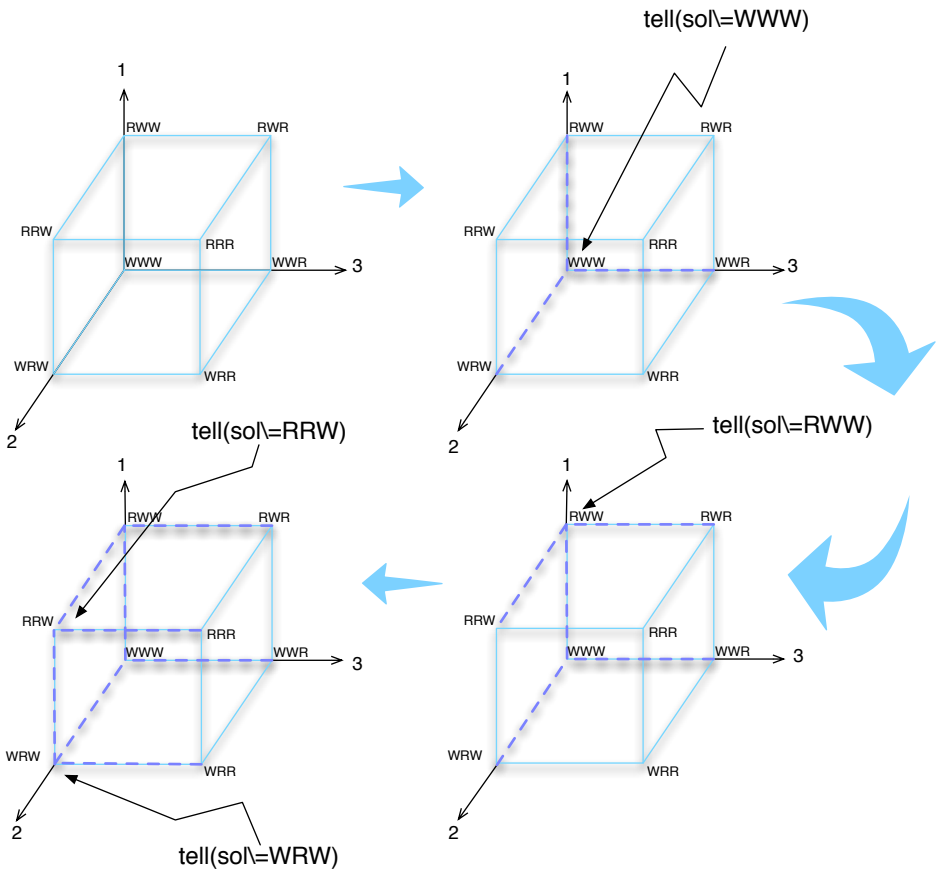


Fig. 1. Evolution of the puzzle graphical representation.

3 Spatial and epistemic send+more=money

Bearing in mind the send+more=money puzzle we describe above, we create program specifications of the puzzle involving spatial hierarchies and knowledge. The distributed nature of the spatial and epistemic ccp language implies that any agent is running in a separate computer node or processor core. Although the interpreter does not allow execution on different nodes, it is easy to map the specification in a programming environment that allows parallel execution of programs. In addition, in order to understand the programs, we need to keep in mind that any tell operation with an agent as argument translates into a tell operartion in that agent store (this resembles post operations in social networks) and that the semantics of askI is asking some information in the owned (nested) store (see [4] for further details).

Scenario 1: The first attempt to solve the problem uses a spatial division. Two agents are trying to reach a solution. Agent `katherine` ignores that all variables are pairwise distinct, whereas agent `andrew` does not know that `s` and `m` can not be zero.

```
parallel([
  spatial(katherine, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10,
        E*1],
              #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell( M #\= 0), tell(S #\= 0),
    tell(solve([S,E,N,D,M,O,R,Y]))]),
  spatial(andrew, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10,
        E*1],
              #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell(all_different([S,E,N,D,M,O,R,Y])),
    tell(solve([S,E,N,D,M,O,R,Y]))])
]).

Knowledge for agent katherine --> {9,0,0,0,1,0,0,0}
Knowledge for agent andrew --> {2,8,1,7,0,3,6,5}
```

In this specification, agents have different information about the state of affairs, i.e., the complete set of constraints. Thus, neither agent is able to reach the valid answer.

Scenario 2: In the second specification the agent `katherine` gives information (post) to `andrew` about the constraints she knows. The only new information is that `m` can not be zero. Thus, `andrew` reach a new and valid conclusion.

```
parallel([
  spatial(katherine, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10,
        E*1],
              #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell(M #\= 0),
    tell(andrew, tell(M#\=0)),
    tell(solve([S,E,N,D,M,O,R,Y]))]),
  spatial(andrew, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10,
        E*1],
              #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell(all_different([S,E,N,D,M,O,R,Y])),
    tell(solve([S,E,N,D,M,O,R,Y]))])
]).

Knowledge for agent katherine --> {9,0,0,0,1,0,0,0}
Knowledge for agent andrew --> {9,5,6,7,1,0,8,2}
```

Scenario 3: The third case presents an agent that has some local representation, nested store, of other agent's knowledge. In this specification, `katherine` beliefs that `andrew` knows certain set of constraint. Using that beliefs, she can reach the valid conclusion.

```
parallel([
  spatial(katherine, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    spatial(andrew, [tell([S,M] ins 0..9), tell( M #\= 0),
        tell(S #\= 0)]),
    askI('andrew:own', M#\=0 -> M#\=0),  askI('andrew:own', S
        #\=0 -> S#\=0),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10,
        E*1],
          #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell(all_different([S,E,N,D,M,O,R,Y])),
    tell(solve([S,E,N,D,M,O,R,Y]))])
]).

Knowledge for agent katherine --> {9,5,6,7,1,0,8,2}
Knowledge for nested:katherine's representation of andrew -->
     {1..9,1..9}
```

Scenario 4: In this specification, and the next one, we use an epistemic viewpoint to solve the problem. In this case, both `katherine` and `andrew` reach different assignments for the variables. Thus, an inconsistency among stores is raised.

```
parallel([
  epistemic(katherine, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10,
        E*1],
              #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell( M #\= 0), tell(S #\= 0),
    tell(solve([S,E,N,D,M,O,R,Y]))]),
  epistemic(andrew, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10,
        E*1],
              #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell(all_different([S,E,N,D,M,O,R,Y])),
    tell(solve([S,E,N,D,M,O,R,Y]))])
]).

Global store failed: inconsistency among stores
aborting execution...
```

Scenario 5: Finally, this epistemic scenario models the case where `katherine` has a local epistemic representation of `andrew`. The information in it, the fact that all variables are pairwise distinct, enables `katherine` to find the valid answer to the puzzle.

```
parallel([
  epistemic(katherine, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    epistemic(andrew, [tell([S,M] ins 0..9), tell(
        all_different([S,E,N,D,M,O,R,Y]))]),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10,
        E*1],
          #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell( M #\= 0), tell(S #\= 0),
    tell(solve([S,E,N,D,M,O,R,Y]))])
]).

Knowledge for agent katherine --> {9,5,6,7,1,0,8,2}
Knowledge for nested:katherine's representation of andrew -->
     {1..9,1..9}
```

4 Conclusions

Social networks, mainly web-based networks, are a growing field of research because of their complexity and ubiquity. In such multi-agent systems information flows in huge magnitudes from client to client. Two fundamental features in these networks are the private data locality and the (possibly constrained) public information posting that is allowed for agents. Furthermore, information exchange may take place within a subset of agents inside the network. Moreover, different information (knowledge) shared inside the network may become common knowledge throughout the entire network (e.g. worldwide disasters).

Such distributed information and knowledge in systems with concurrent behavior are too complex to understand using an unique language or model [8, 7]. Nonetheless, modeling tools , such as the `K-stores` interpreter, can be used to simulate processes that have their own local storage and may gain information by means of a shared medium, such as cloud computing and social networks. We presented some implementations of the send+more=money puzzle using spatial and epistemic interactions. We show how distributed information among a set of agents may reach solutions or inconsistency depending on locality properties, thus, illustrating the modeling properties of the sccp and eccp calculi and the `K-stores` tool. A more elaborated study should include the implementation of the puzzle using various processing nodes.

We acknowledge that in order to solve a distributed problem an agent needs to know all constraints over the variables. Nonetheless, the proposed modeling capabilities are enough to represent evident real life scenarios in which no agent has all information. Moreover, there exists a global store where all information is posted, given the chance to reach an inconsistency or a solution. However, we chose not to search for a solution in that global store because it does not belong to an agent with reasoning capabilities. Instead, we want the global store to help us to respect the epistemic logic axioms. More effort should be put on different views of such global store.

References

1. Oz Programming Interface. `http://mozart-oz.org`. Accessed: 2013-07-10.
2. S. Kamal Abdali. A lambda-calculus model of programming languages - i. simple constructs. *Comput. Lang.*, 1(4):287–301, 1976.
3. Alexandru Baltag, Lawrence S. Moss, and Slawomir Solecki. The logic of public announcements, common knowledge, and private suspicions. In *Proceedings of the 7th conference on Theoretical aspects of rationality and knowledge*, TARK '98, pages 43–56, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.
4. A. Barco, S. Knight, and F. Valencia. K-stores an spatial and epistemic concurrent constraint interpreter. In *Functional and Constraint Logic Programming - 21th International Workshop, WFLP 2012, Nagoya, Japan, May 29th, Proceedings*, 2012.
5. Luc Bovens and Wlodek Rabinowicz. The puzzle of the hats. *Synthese*, 172:57–78, 2010. 10.1007/s11229-009-9476-1.
6. Henry Dudeney. Send more money. *Strand Magazine*, pages 68–79, July, 1924.
7. R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.
8. H. Garavel. Reflections on the future of concurrency theory in general and process calculi in particular. Research Report RR-6368, INRIA, 2007.
9. J. Geanakoplos. Common knowledge. In *Proceedings of the 4th conference on Theoretical aspects of reasoning about knowledge*, pages 254–315. Morgan Kaufmann Publishers Inc., 1992.
10. Herbert Gintis. Rationality and common knowledge. *Rationality and Society*, 22(3):259–282, August 2010.
11. Sophia Knight, Catuscia Palamidessi, Prakash Panangaden, and Frank Valencia. Spatial and epistemic modalities in constraint-based process calculi. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 – Concurrency Theory*, volume 7454 of *Lecture Notes in Computer Science*, pages 317–332. Springer Berlin Heidelberg, 2012.
12. Marc Ponsen, Pieter Spronck, Héctor Muñoz Avila, and David W. Aha. Knowledge acquisition for adaptive game ai. *Sci. Comput. Program.*, 67:59–75, June 2007.
13. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, volume 2 of *ACM '72*, pages 717–740, New York, NY, USA, 1972.
14. Floris Roelofsen. Distributed knowledge. *Journal of Applied Non-Classical Logics*, 17(2):255–273, 2007.
15. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. Swi-prolog. *Computing Research Repository*, abs/1011.5332, 2010.