The 20th International Conference on Principles and Practice of
Constraint Programming

Proceedings

ModRef 2014: The Thirteenth International Workshop on
Constraint Modelling and Reformulation
Monday September 8th, 2014
Lyon, France

Carlos Ansótegui
Preface

The Thirteenth International Workshop on Constraint Modelling and Reformulation was held in Lyon (France) on September 8th, 2014 jointly with the 20th 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 8 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 Perron for giving an invited talk entitled “OR-Tools and the minizinc challenge”, Barry Hurley for giving an invited talk entitled “Modelling, reformulation, and solving with Numberjack” and Mateu Villaret for giving an invited talk entitled “WSimply: Solving intensional WCSPs by reformulation into SMT”.

Organizers

Carlos Ansótegui University of Lleida, Spain

Program committee

Jessica Davies University of Toronto, Canada
Ivan Dotu UPC, Spain
Alan Frisch University of York, UK
Emmanuel Hebrard LAAS, CNRS
George Katsirelos INRA, Toulouse, France
Christopher Jefferson University of St. Andrews, UK
Daniel Le Berre Artois University, France
Jimmy Lee The Chinese University of Hong Kong
Yuri Malitsky Cork Constraint Computation Centre, Ireland
Joao Marques-Silva University College Dublin, Ireland
Ian Miguel University of St. Andrews, UK
Peter Nightingale University of St. Andrews, UK
Jean-Charles Regin University of Nice-Sophia Antipolis / I3S / CNRS
Ashish Sabharwal IBM Research, USA
Horst Samulowitz IBM Research, USA
Meinolf Sellmann IBM Research, USA
Roberto Sebastiani University of Trento, Italy
Peter Stuckey University of Melbourne, Australia
Mateu Villaret Universitat de Girona, Spain
## Table of Contents

Solving a Symmetric Key Cryptographic Problem with Constraint Programming ............................................. 1  
*Marine Minier, Christine Solnon and Julia Reboul*

Distraining : Modeling Trainings in Faculty of Medicine and Pharmacy of Casablanca as Constraint Satisfaction problem ................................. 14  
*Benamrane Amine, Yosra Acodad, Benelallam Imade, El Houssine Bouyakhf and Mohammed Bennani Othmani*

Towards Automatic Dominance Breaking ................................. 21  
*Christopher Mears and Maria Garcia de La Banda*

Modelling RTP-based Residential Load Scheduling for Demand Response in Smart Grids ............................................. 36  
*Shan He, Ariel Liebman, Andrea Rendl, Mark Wallace and Campbell Wilson*

Rethinking the Quest for Declarativity ................................. 52  
*Kathryn Francis*

Efficient SMT Encodings for the Petrobras Domain ................................. 68  
*Miquel Bofill, Joan Espasa and Mateu Villaret*

Encoding The Lexicographic Ordering Constraint in SAT Modulo Theories ................................. 85  
*Hani Elgabou and Alan Frisch*

Anomalies in SMT Solving, Difficulties in Modelling Combinatorial Problems ................................. 97  
*Miquel Palahí and Alan Frisch*
Solving a Symmetric Key Cryptographic Problem with Constraint Programming

Marine Minier\textsuperscript{1,2}, Christine Solnon\textsuperscript{1,3}, and Julia Reboul\textsuperscript{1}

\textsuperscript{1} Université de Lyon, INSA-Lyon, F-69621, France
\textsuperscript{2} CITI, INRIA
\textsuperscript{3} LIRIS, CNRS UMR5205
{marine.minier,christine.solnon,julia.reboul}@insa-lyon.fr

Abstract. This paper tries to sum up a starting work at the edge between Cryptography and Constraint Programming. Indeed, many cryptographic problems are solved using Branch & Bound approaches which are implemented from scratch using classical programming languages such as C. This implies quite a lot of programming work. Furthermore, these problems are NP-hard and solving them within a reasonable amount of time is still challenging.

The main goal of this paper is to investigate the capabilities of classical Constraint Programming tools for solving these problems. In this preliminary study, we focus on a particular problem coming from the symmetric key cryptography world. This particular problem could help cryptographers mount attacks called differential attacks against block ciphers. Branch & Bound approaches are not able to solve this problem within a reasonable amount of time. We thus introduce a CP model to solve it, and show that Choco is able to solve it to optimality in a few hours.

Keywords: Symmetric Key Cryptography, Block Cipher, Differential Attack, Constraint Programming, Choco.

1 Introduction

Cryptography is now a cornerstone for the security of communications. It guarantees properties such as confidentiality, integrity and signature. Whereas public key cryptography is usually built on problems well known for their hardnesses, symmetric key cryptography relies on simple operations that are iterated many times to speed up the encryption/decryption process.

The most important symmetric key primitives are hash functions that guarantee integrity and stream and block ciphers that guarantee confidentiality. Hash functions create a fixed size fingerprint from messages of arbitrary lengths. They are the most famous examples of symmetric key algorithms. During the last decade, however, many cryptanalytic results
appeared that completely break the standards MD5, SHA-0 and SHA-1 [15, 16, 14] by finding collisions, i.e., two messages having the same reduction. Following these results, SAT solvers have been used to generate such collisions [9, 5, 8] and the future hash standard Keccak [10].

Concerning cryptanalysis of stream ciphers, several papers propose different approaches. In [13], the authors propose to solve algebraic systems generated by writing the equations linking together the keybits and the outputs of a stream cipher. In [11], the authors compute the bounds of differential and linear cryptanalysis using mixed-integer linear programming. They apply their results to a particular stream cipher named Enocove-128v2. However only a few results focus on the cryptanalysis of block ciphers.

In this paper, we propose to model the new attacks proposed in [3, 7] against block ciphers using Constraint Programming (CP). We have chosen CP because our goal in this starting work is to propose a first mathematical model by means of constraints, and have a first feedback on the hardness of the problem for classical CP tools. Other tools could be used such as, for example, Integer Programming, Satisfiability Modulo Theories, or SAT, and our goal for further work is to compare different models and approaches.

This paper is organized as follows: Section 2 describes the cryptanalytic problem on block ciphers; Section 3 describes the CP model; and Section 4 gives first experimental results obtained with Choco [12].

2 Problem Statement

In this Section, we detail the general structure of a block cipher focusing on the particular case of the Advanced Encryption Standard (AES) [6]. We then describe what a differential attack is and finally introduce the chosen key differential attack model.

2.1 Block Ciphers

A block cipher is a bijective function $E$ that ciphers binary blocks (called plaintexts) of length $n$ under binary keys of length $k$ to output binary ciphertexts of length $n$:

$$E : \{0, 1\}^n \times \{0, 1\}^k \rightarrow \{0, 1\}^n$$

$$(x, K) \mapsto y = E_K(x)$$
such that $x = E^{-1}_K(E_K(x))$ for all keys $K$ and plaintexts $x$. Most of today’s block ciphers have an iterated structure. They apply what we call a round function $f$ $r$ times. The round function $f$ is parametrized by a round subkey $K_i$ which is generated from the master key $K$ using a Key Schedule (KS) algorithm. The left part of Fig. 1 gives a general overview of a block cipher.

Two famous examples of block ciphers are the Data Encryption Standard (DES) and the AES: DES was the encryption standard between 1977 and 2001, and it is an example of what we call a Feistel scheme. AES is the actual standard since 2001 and it uses an SPN (Substitution-Permutation Network) structure.

The round function $f$ of an SPN is, as indicated by its name, composed of a non-linear layer $NL$ (the substitution layer) and a linear layer $L$ (the permutation layer followed by a xor with the subkey $K_{i+1}$) so that $X_{i+1} = f(X_i, K_{i+1}) = L(NL(X_i), K_{i+1})$.

The non-linear layer $NL$ is usually composed by $t$ parallel applications of a single non-linear permutation called an S-box which acts on blocks of size $m = n/t$ bits. More precisely, $X_i$ is cut into $t$ sub-sequences, denoted
$X_{i,1}, \ldots, X_{i,t}$, and $NL(X_i)$ is the concatenation of $S(X_{i,1}), \ldots, S(X_{i,t})$, as shown on the right part of Fig. 1. The linear layer $L$ usually acts on the whole $n$-bit vector.

### 2.2 The AES

Since 2001, the standard of block ciphers is the AES [6] designed by V. Rijmen and J. Daemen. The AES ciphers blocks of length $n = 128$ bits, where each block is seen as a $4 \times 4$ matrix of bytes, under keys of length $k = 128, 192$ or $256$ bits. The number of rounds depends on the key length: $r = 10$ (resp. 12 and 14) for $k = 128$ (resp. 192 and 256). Each byte is seen as an element of the set $\{0, \ldots, 255\}$.

As explained in Section 2.1, the AES is an SPN, and its round function $f$ is composed of a non-linear layer $NL$ and a linear layer $L$. The non-linear layer calls 16 times (one for each byte, i.e., $t = 16$) a single S-box $S$ that acts at byte level (i.e. $m = 128/16 = 8$ bits).

The linear part $L$ of the AES is composed of the following 3 linear mappings:

- **ShiftRows (SR):** a linear mapping that rotates on the left all the rows except the first one of the current matrix as follows:

  \[
  \begin{array}{cccc}
  a & b & c & d \\
  e & f & g & h \\
  i & j & k & l \\
  m & n & o & p \\
  \end{array}
  \xrightarrow{SR} \begin{array}{cccc}
  a & b & c & d \\
  f & g & h & e \\
  k & l & i & j \\
  p & m & n & o \\
  \end{array}
  \]

- **MixColumns (MC):** another linear mapping represented by a $4 \times 4$ matrix chosen for its good properties of diffusion (see [4]). Each column of the input matrix is multiplied by a fixed matrix.

- **AddRoundKey (ARK):** a simple $\text{x-or}$ operation between the current block and the subkey of the round $r$ denoted by $K_r$.

Those $r$ rounds are surrounded at the top by an initial key addition with the subkey $K_0$. The last round does not contain the MixColumns operation. We omit here the details of the key schedule that may be found in [6]. Just note that each subkey $K_i$ is directly computed from the subkey $K_{i-1}$ and that most of the key schedule operations are linear and only 4 calls to the non linear S-box are made.
2.3 Differential Cryptanalysis

Differential cryptanalysis has been introduced by E. Biham et A. Shamir in 1991 [2]. The main principle of this technique is to consider plaintext pairs \((X_0, X_0')\) and to study the propagation of the initial difference between \(X_0\) and \(X_0'\) into the cipher, while going through the successive round functions \(f\). We note \(\delta X_i\) as the difference between the two plaintexts \(X_i\) and \(X_i'\) obtained after the \(i\)th round of the ciphering of \(X_0\) and \(X_0'\).

Usually, this difference is computed using the x-or operator (denoted \(\oplus\)), i.e., \(\delta X_i = X_i \oplus X_i'\).

Let us keep in mind that the round function \(f\) is composed of a linear part \(L\) and a non linear part \(NL\), i.e., \(X_i = f(X_{i-1}, K_i) = L(NL(X_{i-1}), K_i)\). The linear part \(L\) only moves the differences to some other place. Indeed, for every linear operator \(l\) we have \(l(A \oplus B) = l(A) \oplus l(B)\). So, we can easily predict how the differences are propagated from \(\delta X_i\) to \(\delta X_{i+1}\) by the linear layer \(L\).

The non linear layer has to be studied more carefully. As said before, the non linear part \(NL\) of the \(f\) function is decomposed into \(t\) parallel calls to a given S-box \(S\) which operates on \(m\) bits such that each S-box computes \(S(X_{i,j})\) for a different sub-sequence \(X_{i,j}\) of \(X_i\) with \(j \in \{1, \ldots, t\}\). Therefore, we first need to study how the S-box propagates the differences for a pair \((A, B)\) of sequences of \(m\) bits. To this aim, we evaluate the probability that the output difference \(S(A) \oplus S(B)\) is equal to \(\beta\) when the input difference \(A \oplus B\) is equal to \(\alpha\), where \(\alpha\) and \(\beta\) are sequences of \(m\) bits. This probability is denoted \(D_{\alpha,\beta}\) and is defined by

\[
D_{\alpha,\beta} = \frac{\#\{(A, B) \in \{0, 1\}^m \times \{0, 1\}^m \mid (A \oplus B = \alpha) \land (S(A) \oplus S(B) = \beta)\}}{2^m}
\]

For example, let us consider the S-box of the AES that acts on bytes (so that \(m = 8\)), and let us consider an input difference \(\alpha = 00000001\) and an output difference \(00100000\). The transition from 00000001 to 00100000 only occurs for 4 couples \((A, B)\) of inputs (or 2 pairs due to the fact that \(A \oplus B = B \oplus A\), among the 256 possible couples. Therefore, the transition probability \(D_{00000001,00100000}\) is equal to \(\frac{4}{256}\). For the AES S-box with \(m = 8\), most of the times the transition probability is equal to \(\frac{0}{256}\), or \(\frac{2}{256}\), and rarely to \(\frac{4}{256}\). Note that in the case of the AES S-box, \(S\) is a bijection so that \(A \oplus B = 0 \iff S(A) \oplus S(B) = 0\). As a consequence, \(D_{00000000,00000000} = 1\). In other words, if there is no difference in the input \(A \oplus B\), then there is no difference in the output \(S(A) \oplus S(B)\).

Then, to cross a complete non-linear layer \(NL\) at round \(i\) and study the propagation of the differences from \(X_i \oplus X_i'\) to \(NL(X_i) \oplus NL(X_i')\) (where
Xᵢ and X'ᵢ are cut into t sub-sequences Xᵢ,1,...,Xᵢ,t and X'ᵢ,1,...,X'ᵢ,t, we compute the probability of obtaining the output difference NL(Xᵢ) ⊕ NL(X'ᵢ) when the input difference is Xᵢ ⊕ X'ᵢ. This probability is:

\[ p_1(NL(Xᵢ) ⊕ NL(X'ᵢ)|Xᵢ ⊕ X'ᵢ) = \prod_{j=1}^{t} D_{α_j,β_j} \] (1)

where αᵢ = Xᵢ,j ⊕ X'ᵢ,j and βᵢ = S(Xᵢ,j) ⊕ S(X'ᵢ,j).

Finally, the probability of obtaining the output difference δXᵢ = Xᵢ ⊕ X'ᵢ (after r rounds) when the input difference δX₀ = X₀ ⊕ X₀' is:

\[ p_2(δXᵢ|δX₀) = \prod_{i=0}^{r} p_1(NL(Xᵢ) ⊕ NL(X'ᵢ)|Xᵢ ⊕ X'ᵢ) \] (2)

where Xᵢ = L(NL(Xᵢ₋₁), Kᵢ) and X'ᵢ = L(NL(X'ᵢ₋₁), Kᵢ) for all rounds \(i \in \{1,...,r\}\). We refer the reader to [2] for more details.

The goal of the attacker is to find the values of δXᵢ for \(i \in \{0,...r\}\) which maximize this probability \(p_2\).

### 2.4 Chosen Key Differential Cryptanalysis

Today, differential cryptanalysis is public knowledge, so modern block ciphers such as the AES have been designed to have proven bounds against differential attacks. However, in 1993, E. Biham proposed a new type of attack called related key attack [1] that allows an attacker to inject differences not only between the plaintexts X₀ and X₀’ but also between the keys K and K’ to try to mount more powerful attacks. The main idea behind this is to be able to derive from those attacks particular cryptanalysis in the more classical settings: the unknown key model.

We note \(Kᵢ\) and \(K'ᵢ\) the subkeys of \(K\) and \(K'\) at round \(i\) and we note \(δKᵢ\) the difference between \(Kᵢ\) and \(K'ᵢ\), i.e., \(δKᵢ = Kᵢ ⊕ K'ᵢ\). The goal of the attacker is to find the values of \(δXᵢ\) and \(δKᵢ\) which maximize the probability \(p_2\) defined by equation (2) while satisfying \(Xᵢ = L(NL(Xᵢ₋₁), Kᵢ)\) and \(X'ᵢ = L(NL(X'ᵢ₋₁), K'ᵢ)\) for every round \(i \in \{1,...,r\}\).

Two main papers [3, 7] describe results for the chosen key differential cryptanalysis of the AES and propose algorithms for finding the values of \(δXᵢ\) and \(δKᵢ\) which maximize the probability \(p_2\) in the case of the AES. In both papers, the problem is solved in two steps. In the first step, each unknown \(δXᵢ\) (resp. \(δKᵢ\)) is decomposed into a 4 × 4 matrix of bytes, and a binary variable \(ΔXᵢ[j][k]\) (resp. \(ΔKᵢ[j][k]\)) is associated with every byte \(δXᵢ[j][k]\) (resp. \(δKᵢ[j][k]\)) at row \(j\) and column \(k\) of \(δXᵢ\) (resp. \(δKᵢ\)). These
Fig. 2. First step of chosen key differential cryptanalysis with 3 rounds. Each $4 \times 4$ array represents a group of 16 binary variables, associated with a bit-vector of 16 bytes. For each round $1 \leq i \leq 3$, $\Delta X_i$ is obtained by applying AddRoundKey (ARK) on $Y_{i-1}$ and $\Delta K_i$; then $Y_i$ is obtained by applying ShiftRows and MixColumns (MC-SR) on $\Delta X_i$. Round 0 is a special case, where ARK is applied on $\Delta X_0$ and $\Delta K_0$ before applying MC-SR to obtain $Y_0$. Each subkey $\Delta K_i$ is obtained by applying KeySchedule (KS) on $\Delta K_{i-1}$.

binary variables are equal to 0 if their associated bytes are equal to $0^8$, i.e.,

$\Delta X_i[j][k] = 0 \iff X_i[j][k] = X'_i[j][k] \iff \delta X_i[j][k] = 0^8$

$\Delta K_i[j][k] = 0 \iff K_i[j][k] = K'_i[j][k] \iff \delta K_i[j][k] = 0^8$

and they are equal to 1 otherwise, i.e.,

$\Delta X_i[j][k] = 1 \iff X_i[j][k] \neq X'_i[j][k] \iff \delta X_i[j][k] \neq 0^8$

$\Delta K_i[j][k] = 1 \iff K_i[j][k] \neq K'_i[j][k] \iff \delta K_i[j][k] \neq 0^8$

The operations that transform $\delta X_0$ into $\delta X_r$ (described in Section 2.1), are translated into constraints between these binary variables, and new variables are associated with intermediate steps (between ShiftRows and AddRoundKey operations), as illustrated in Fig. 2. In this first step, the goal is to find all solutions which satisfy these constraints while maximizing the sum of a subset of the variables (see the next section for more details on the constraints and the objective function).

Note that during this first step, the non linear part $NL$ of the function $f$ is not considered. Indeed, the S-box does not introduce nor remove differences, i.e., given 2 bytes $A$ and $B$, $(A \oplus B = 0^8) \iff (S(A) \oplus S(B) = 0^8)$.

In the second step, each solution found in the first step is transformed into a solution of the initial problem, i.e., for each binary variable $\Delta X_i[j][k]$ or $\Delta K_i[j][k]$ set to 1 in the solution, we search for a byte value $\delta X_i[j][k]$ or $\delta K_i[j][k]$ different from $0^8$ so that the AES transformation rules are satisfied and the probability $p_2$ is maximized.
3 CP Model of the first step

In this paper, we propose to use Constraint Programming (CP) to solve the first step of the solution process described in Section 2.4. In this section, we describe the CP model used to solve this problem: variables, constraints, objective function, and ordering heuristics.

3.1 Variables

Let $r$ be the number of rounds. We define the following variables:

- For all $i \in [0; r]$ and for all $j, k \in [0; 3]$, $\Delta X_i[j][k]$ and $\Delta K_i[j][k]$ are the variables which are associated with the bytes $\delta X_i[j][k]$ and $\delta K_i[j][k]$, respectively.
- For all $j, k \in [0; 3]$, $\Delta X'_0[j][k]$ is the variable which is associated with the byte at row $j$ and column $k$ of the result of ARK operation on $\delta X_0$ and $\delta K_0$.
- For all $i \in [1; r]$ and for all $j, k \in [0; 3]$, $Y_i[j][k]$ is the variable which is associated with the byte at row $j$ and column $k$ of the result of SR and MC operations on $\delta X_i$ (except when $i = 0$, for which SR and MC are applied on $\Delta X'_0$).

All these variables are binary variable, which are set to 0 when the associated byte is $0^8$ and to 1 otherwise.

3.2 Constraints

The constraints correspond to the propagation of differences by the different operations of the round function $f$. As said before, the non linear part $NL$ does not imply any constraint as it neither introduces nor removes differences. The linear part $L$ implies the following constraints.

AddRoundKey. ARK is applied on $\Delta X_0$ and $\Delta K_0$ to obtain $\Delta X'_0$, during the first round, i.e.,

$$ARK(\Delta X_0, \Delta K_0, \Delta X'_0)$$

Then, for each next round $i$, it is applied on $Y_{i-1}$ and $\Delta K_i$ to obtain $\Delta X_i$, i.e.,

$$\forall i \in [1, r], ARK(Y_{i-1}, \Delta K_i, \Delta X_i)$$

ARK performs a xor operation. Let $B_1$ and $B_2$ be two bytes. If $B_1 = B_2 = 0^8$, then $B_1 \oplus B_2 = 0^8$. If $B_1 = 0^8$ and $B_2 \neq 0^8$, then $B_1 \oplus B_2 \neq 0^8$. However, if $B_1 \neq 0^8$ and $B_2 \neq 0^8$, then we cannot know if $B_1 \oplus B_2$ is equal
to 0^8 or not. When abstracting the byte domain with a binary domain which only models the fact that there is a difference or not, we obtain the following definition of the constraint \( ARK(A, B, C) \):

\[
\forall (j, k) \in [0; 3]^2, (A[j][k] + B[j][k] \neq 2) \Rightarrow (A[j][k] + B[j][k] = C[j][k])
\]

where \( A, B \) and \( C \) are \( 4 \times 4 \) binary matrices.

*ShiftRows and MixColumns.* MC-SR is applied on \( \Delta X'_0 \) to obtain \( Y_0 \), during the first round, i.e.,

\[
MC-SR(\Delta X'_0, Y_0)
\]

Then, for each next round \( i \), it is applied on \( \Delta X_i \) to obtain \( Y_i \), i.e.,

\[
\forall i \in [1, r - 1], MC-SR(\Delta X_i, Y_i)
\]

MC is a particular error correcting code that acts on separated columns while SR moves the differences at some other places. Given two \( 4 \times 4 \) binary matrices \( A \) and \( B \), the constraint \( MC-SR(A, B) \) is defined by

\[
\forall k \in [0; 3], \left( \sum_{j=0}^{3} A[j][(k + j) \% 4] = 0 \right) \Rightarrow \left( \sum_{j=0}^{3} B[j][k] = 0 \right)
\]

\[
\forall k \in [0; 3], \left( \sum_{j=0}^{3} A[j][(k + j) \% 4] > 0 \right) \Rightarrow \left( \sum_{j=0}^{3} (A[j][(k + j) \% 4] + B[j][k]) \geq 5 \right)
\]

*KeySchedule.* KS is applied at each round \( i \) to compute \( \Delta K_i \) from \( \Delta K_{i-1} \), i.e.,

\[
\forall i \in [1, r], KS(\Delta K_{i-1}, \Delta K_i)
\]

KS is most of the times composed of xor operations between some columns of the key. Given two \( 4 \times 4 \) binary matrices \( A \) and \( B \), the constraint \( KS(A, B) \) is defined by

\[
\forall j \in [0; 3], \quad (A[j][0] + A[(j + 1) \% 4][3] \neq 2) \Rightarrow (B[j][0] = A[j][0] + A[(j + 1) \% 4][3])
\]

\[
\forall j \in [0; 3], \forall k \in [1; 3], \quad (B[j][k - 1] + A[j][k] \neq 2) \Rightarrow (B[j][k] = B[j][k - 1] + A[j][k])
\]

Finally, we add the constraint that the initial plaintexts \( X_0 \) and \( X'_0 \) must be different, i.e.,

\[
\sum_{j=0}^{3} \sum_{k=0}^{3} \Delta X_0[j][k] \neq 0
\]
and that the initial subkeys $K_0$ and $K'_0$ must be different, i.e.,

$$\sum_{j=0}^{3} \sum_{k=0}^{3} \Delta K_0[j][k] \neq 0$$

### 3.3 Objective function

In order to maximize the probability $p_2$ of equation (2), we have to maximize its different factors. To this aim, we have to maximize the number of factors of equation (1) for which $\alpha = \beta = 0$ (because $D_{0,0} = 1$). This amounts to minimizing the number of $\Delta X_i[j][k]$ and $\Delta K_i[j][3]$ variables which are set to 1. Therefore the objective function to minimize is

$$\sum_{i=0}^{r} \sum_{j=0}^{3} (\Delta K_i[j][3] + \sum_{k=0}^{3} \Delta X_i[j][k])$$

### 3.4 Ordering heuristics

As we want to minimize the number of $\Delta X_i[j][k]$ and $\Delta K_i[j][3]$ variables which are set to 1, we add a variable ordering heuristic which first assign these variables, and a value ordering heuristic which first tries to assign them to 0. Of course, this heuristic does not eliminate the need for an explicit objective function, as heuristically setting a variable $x$ to 0 may force some other variables to be set to 1, thus leading to a worse solution than setting $x$ to 1. This ordering heuristically drastically improves the solution process. For example, when the number of rounds is $r = 3$, the time needed to find the optimal solution is increased from 1 second to 29 seconds when removing these ordering heuristics.

### 4 Experimental Results

We have implemented the CP model described in the previous section in Choco 3 [12]. Table 1 displays experimental results obtained with this very first CP model. It shows us that CP is able to solve the problem up to $r = 5$ in a reasonable amount of time. As the probability defined by equation (2) with $r = 5$ rounds reaches $p = 2^{-105}$, it is useless to try to solve the case for $r = 6$ because $p$ becomes greater than the maximal authorized probability $p = 2^{-128}$. Indeed, in this last case, the attack requires all the code book, i.e., all possible plaintexts. This could be no more consider as an attack because we obtain all possible ciphertexts.
Table 1. Experimental results. Each line successively displays: the number \( r \) of rounds, the time (in seconds) and number of choice points needed to find an optimal solution and prove its optimality, the time and number of choice points needed to find all optimal solutions and the number of optimal solutions.

<table>
<thead>
<tr>
<th>( r )</th>
<th>Search for one solution</th>
<th>Search for all optimal solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>1</td>
<td>3,157</td>
</tr>
<tr>
<td>4</td>
<td>377</td>
<td>8,733,900</td>
</tr>
<tr>
<td>5</td>
<td>4,036</td>
<td>66,534,689</td>
</tr>
</tbody>
</table>

Our Choco program only solves the first step, \textit{i.e.}, it assigns binary variables to 0 if the associated byte is 0, and to 1 if the associated byte is different from 0. In the second step, for each binary variable \( \Delta X[j][k] \) or \( \Delta K[j][k] \) set to 1 in the first step solution, we search for a byte value \( \delta X[j][k] \) or \( \delta K[j][k] \) different from 0 so that the different AES transformation rules are satisfied, for each round, and the probability \( p_2 \) is maximized. Note that some solutions of the first step cannot be transformed into solutions of the initial problem. At this time, this second step is solved by a program written in C. This second step basically involves the exploration of \( 2^{16} \) combinations, and it is rather quickly performed. Therefore, each time Choco finds a solution to the first step, we use our C program to search for byte values. If the C program succeeds, then we ask Choco to search for a better solution with respect to the objective function, otherwise we ask Choco to search for another solution without modifying the bound on the objective function. With this method, we retrieve the results given in [7].

5 Discussion

These first results obtained with a simple CP model encoded in Choco in a straightforward way are rather encouraging. Indeed, we have also implemented in C the Branch & Bound approach proposed in [3]. For \( r = 3 \), this approach is able to find the optimal solution in about one hour and for \( r = 4 \) in about 24 hours. We did not try to run it for \( r = 5 \).

The approach proposed in [7] is rather different. It is based on a breadth-first traversal of a graph which contains \( 2^{32} \) nodes, corresponding to the \( 2^{32} \) possible states considering that all plaintext and key bytes of \( \delta X_0 \) and \( \delta K_0 \) may contain a difference or not. This approach has an exponential memory complexity which may not scale well for other block cipher families. In particular, for the family of block ciphers called Rijndael,
the block and key size vary between 160-bit and 256-bit. In this case, the approach proposed in [7] would no longer be practical due to the size of the graph to store that becomes at least equal to $2^{36}$ and at most equal to $2^{64}$.

As future work, we plan to complete the CP model by integrating byte variables and constraints related to the second step. Indeed, many solutions found during the first step cannot be transformed into solutions during the second step because we cannot find byte values which satisfy the constraints defined by the non linear S-box. Adding these constraints should allow us to reduce the search space and improve the solution process.

We also plan to extend this preliminary work to other families of block ciphers, such as the Rijndael family. Indeed, for this family, the approach proposed in [7] cannot be used because of its exponential memory complexity.

In this preliminary work, we have used Choco to solve the first step of our problem. Our goal was to validate our CSP model, and evaluate the hardness of this problem for a classical CP library. As pointed out in the introduction, other approaches could be used and some of them should be more suited to solve this optimization problem. Therefore, we plan to compare CP with Integer Programming, Satisfiability Modulo Theories and SAT solvers. Also, as pointed out by one reviewer, one way to optimize the objective function indirectly in a single run of a CP solver is to use Limited Discrepancy Search (LDS) on the variables of the objective function (and branch on these variables first, before other variables).

Acknowledgement. Many thanks to our reviewers for their careful reading and valuable comments.

References


Modeling trainings in Faculty of Medicine and Pharmacy of Casablanca as Constraint Satisfaction Problem

Amine Benamrane¹, Yosra Acodad¹, Imade Benelallam², El Houssine Bouyakhf¹, Mohammed Bennani Othmani³

¹ LIMIARF. Faculty of Sciences Mohammed the fifth University - Agdal Rabat, Morocco
benamraneamine@gmail.com, yosra.acodad@gmail.com, bouyakhf@fsr.ac.ma
² INSEA, National Institut of Statistics and Applied Economic - Irfane Rabat, Morocco
imade.benelallam@ieee.org
³ LIM. Faculty of Medicine and Pharmacy of Casablanca - Casablanca, Morocco
m.bennani@fmpc.ac.ma

Abstract. Hospital internships are essential and central component of any medical training to acquire different clinical skills, they require a commitment from the student and the supervisor.
In fact, the second cycle of medical studies in Faculty of Medicine and Pharmacy of Casablanca (FMPC) focuses on the acquisition of these clinical and therapeutic skills, and the encountered problem in the hospital courses management is adapting the number of students to home capacity, knowing that the assignment of students to hospital services must undergo certain constraints.
In effect, as the formalism of constraint satisfaction problem (CSP) can be simply, clearly and successfully adapted to a large class of real problems, we thought to use constraint programming (CP) to succeed this distribution of students per services.
In the present paper, we introduce the formalized CSP, then we present the component of the hospital training in the curriculum of the medical student, next we present our modeling of this distribution of students in the hospital services as a CSP. Finally, we conclude and propose some perspectives.

Keywords: CSP, modeling, medical Studies, allocation

1 Introduction

The constraint paradigm is a useful and well-studied framework expressing many problems of interest in Artificial Intelligence, operations research and other areas of Computer Science.

Constraint programming (CP) [1] distinguishes between the description of the constraints involved in a problem on the one hand, and the algorithms and
heuristics used to solve the problem on the other hand. Modeling and solving problems is through a very elegant mathematical formalism, called the constraint satisfaction problem (CSP) [2].

A significant progress has been made in CP in the last decade. In fact, many real problems, as scheduling, configuration, hardware verification, graph problems, molecular biology, etc... can be formulated as constraint satisfaction problems (CSPs) in which solutions are assignments to a set of variables respecting a collection of constraints.

As formulating a good model (modeling the CSP) is of crucial importance to solving the model, we focus, in this paper, on modeling a real problem which is a hospital training.

The encountered problem is to allocate students of Faculty of Medicine and Pharmacy of Casablanca (FMPC) to some hospitals in the city, essentially to the universitary hospital complex (UHC) services while satisfying an imposed restrictions.

We note that, for solving this allocation problem, the FMPC administration already uses an application of students assignment management each year, but the obtained solution is always incomplete, then, the administration completes manually this distribution.

2 Background

2.1 Constraint satisfaction problem

The constraint programming is a very successful paradigm that can be used to model many real-world problems, as constraint satisfaction problems (CSPs), and solve them.

A model is an abstraction to a problem, that must respect the language of the solver. The quality of the model influences the quality of the solution.

A solution to a CSP is an assignments to a set of variables (each variable taking values from a certain domain), and in which there exists a collection of constraints that restrict the assignment of particular values or combination of values; solving a CSP means finding a feasible assignment of values to variables, i.e., one where all the constraints are satisfied.

Definition 1 (CSP). A Constraint Satisfaction Problem (CSP) is a tuple < X, D, C >, where:

- X is a set containing n variables \{x_1 , x_2 , ..., x_n \}.
- D is a set of domains \{D(x_1), D(x_2),..., D(x_n)\} for these variables, with each D(x_i) containing the possible values which x_i may take.
- C is a set of m constraints \{c_1 , c_2 , ..., c_m\} between variables in subsets of X. Each c_i \in C expresses a relation defining which variable assignment combinations are allowed for the variables in the scope of the constraint, vars(c_i).
A partial assignment is a set of tuple pairs, each tuple consisting of an instantiated variable and the value that is assigned to it in the current search node. A full assignment is one containing all $n$ variables.

A partial solution is a consistent partial assignment, i.e., a partial assignment in which no constraints are violated. A solution to a CSP is a full assignment such that no constraint is violated.

### 2.2 Training in FMPC

Hospital internship is an essential part of the curriculum of the future medical doctor, each student in Faculty of Medicine and Pharmacy of Casablanca (FMPC) must pass this component to pass the academic year. The administration of the FMPC must allocate students to the hospitals and to universitary hospital complex services respecting some constraints [3]. Actually, for solving this allocation problem, the FMPC administration uses, before the beginning of each year, an application of students assignment management, which is modeled using the formalism OOP (Object-oriented programming) [4] and developed with the DotNet plateform [5]. For example, for a population of 500 students and about 20 services, with an average capacity of 55 beds per service, the current application takes between 7 and 10 days to declare a solution. However, the obtained result is always incomplete (some students remain without service because of the exhaustion of all services capacity), reason why the administration completes manually the distribution by violating some constraints.

Here are some interesting definitions which will be useful later:

**Definition 2 (Period $p$).** A period $p$ is a fixed period of one month and half (6 weeks).

**Definition 3 (complementary service).** A complementary service is a service that lasts one period.

Cardiology, nephrology and dermatology services, are complementary service.

**Definition 4 (Fundamental service).** A fundamental service is a service that occupies 2 periods. Otherwise, it is a couple $<FS_1, FS_2>$, where each component $FS_i$, such as $i \in \{1, 2\}$, lasts a single period.

The pediatric service, for example, is a fundamental one. It is composed of pediatric$_1$ and pediatric$_2$, and each component takes a single period.

**Definition 5 (Obligatory service).** An obligatory service is a service whereby each student must pass.

**Definition 6 (Optional service).** An optional service is any service which is not obligatory.
Either obligatory and complementary services are not necessarily fundamental services. The problem constraints can be divided in two categories, the first one is the local constraints that manage the relationship of each student with its own six periods, and the second one is the global constraints that define the relationship between students and services in which they are assigned.

- Local constraints (concerning students):
  - each student must pass exactly six courses throughout the year (6 periods); \( C_1 \)
  - each student should not revisit a service; \( C_2 \)
  - each student must pass through a fixed number of obligatory services; \( C_3 \)
  - each student having a fundamental service \( s_i \) (part of the pair \( s_i, s_j \) ) in a given period \( t_k \) such as \( k \in [1,5] \), must be allocated in the following period \( t_{k+1} \) to the service \( s_j \). \( C_4 \)
  - each student should not have a first part of a fundamental service in the latest course (sixth period). \( C_5 \)

- Global constraints (regarding services):
  - each service is characterized by its capacity that can not be exceeded. \( C_6 \)

3 modeling the problem of distributing students in the hospital services as a CSP

3.1 Motivation

We choose to formulate the problem of distributing students in the hospital services as a Constraint Satisfaction Problem (CSP) for the following reasons:

- The problem of assigning students per services is too complex to be solved manually;
- Many requirements on this assignation are naturally translated to constraints;
- Constraint Programming techniques have shown to be successful for applications similar to this one, such as resource allocation problems, scheduling problems,... etc.

3.2 Formulation of the CSP

The goal of this paper is to model the training problem already explained as a CSP.

To do this, we will translate the different components of the problem as a set \( < Variables, Domains, Constraints > \) such that the modeling describes exactly the problem.

The modeling can be done in several ways, we propose the one we see the clearest, the simplest and the most efficient.
Variables:
a CSP variable $X^p_i$ is a period $p$ for a given student $i$, such as: $i \in [1, \text{number of students}]$ and $p \in [1, 6]$. $(C1)$ we choose variables as above because solution must be a complete assignment of all periods for all students.

Domains:
Let $D = \{s_1, s_2, ..., s_n\}$, the set of the $n$ services in $UHC$ (Universitary Hospital complex), be the domain of all CSP variables $X^p_i$.

$X^p_i = s_j$ means the student $i$ is affected for his period $p$ at the service $s_j$.

Constraints:

– Students' constraints: $\forall$ student $i$,

$C^1_i$:

\[
\forall k, l \in [1, 6] \text{ such as: } k \neq l: \quad X^k_i \neq X^l_i \quad (C2)
\]

$C^2_i$:

$S$ is the list of obligatory services and $m \in [1, \text{cardinal}(S)]$:

\[
\sum_{p=1}^{6} \text{isIn}(X^p_i, S) = m \text{ such as:}
\]

$\text{isIn}(a, A)$ is the boolean function returning 1 when $a$ is included in $A$. $(C3)$

$C^3_i$:

$\forall S =< s, s' >$ a fundamental service with the two unites $s$ and $s'$,

\[
\forall p \in [1, 5]: \quad X^p_i = s \Rightarrow X^{p+1}_i = s' \quad (C4)
\]

$C^4_i$:

$\forall S =< s, s' >$ a fundamental service with the two unites $s$ and $s'$:

$X^6_i \neq s \quad (C5)$

Ignoring constraint concerning students non obligatory courses does not negate their existence. In fact, due to their existence in variables domains and the high number of variables, a solution can not be found without using them.

Services' constraint:

$C_s$:

$\forall s \in D, \forall p \in [1, 6]: \text{occup}_p(s) \leq \text{capacity}(s) \quad (C6)$
such as:

$\text{occup}_p(s)$ is the number of students affected to the service $s$ in $p$, one of the 6 periods.

$\text{capacity}(s)$ is the capacity of the service $s$ that can’t be exceeded.
4 Implementation

To measure the feasibility and rigor of our proposed modeling, we have implemented it on the Choco 2.0 platform [6] on a dual core machine with 1.72 GHz in processor and 2GB of RAM.

1. IntegerVariable[][] Students = Choco.makeIntVarArray( "Students", NbStudents, 6, services, Options.V_ENUM); // (C1)
2. for (int i = 0; i < NbStudents; i++) {
   3.  CSPModel model = new CSPModel();
   4.  model.addConstraint(Choco.allDifferent(Students[i])); // (C2)
   5.  model.addConstraint(Choco.among(m, Students[i], ListOfObligatories)); // (C3)
   6.  for (int j = 0; j < NbOPeriods - 1; j++) {
      7.   for (int l = 0; l < ListofFundamentals1.size(); l++) {
         8.     model.addConstraint(Choco.implies(Choco.eq(Students[i][j], s1), Choco.eq(Students[i][j+1], s2))); // (C4)
         9.     if (j == latest_period) {
            10.       model.addConstraint(Choco.notMember(Students[i][j], ListofFundamentals1)); // (C5)
               }
         }
   }
11.  model.addConstraint(Choco.occupationMax(k, Capacities.get(k), VarsOfStudents)); // (C6)
12.  CSPSolver solver = new CSPSolver();
13.  solver.read(model);
14.  solver.solve();

Fig. 1: Pseudo code

Students is the matrix of services of all students for all periods, otherwise, each Student[i][j] is a period of a student which takes a value from services, created by the function makeIntVarArray of the main class Choco (line 1). The procedure loops for all students (line 2) to define for each one its local constraints, and defines a model that contains all specifications of the current problem (line 3). Next, it fills the model by all local constraints (lines 4 to 10). After adding the global constraints concerning capacities of services (line 11), all constraints are added, then, the procedure defines the CSPSolver (line 12) and pass to it the model (line 13) before launching the solving process (line 14).

5 Conclusion and perspective

In this paper, we presented the feasibility of modeling the problem of allocating students of FMPC to various hospital services in the city as a CSP, according
to the requirements of the medical training program. As we showed through this work, the CSP framework is a very flexible and powerful model for this kind of problems.

As perspective, we intend to compare the efficiency and quality of this application as a CSP with its previous one, in terms of modeling and solving, and to develop this modeling using the COP (Constraint Optimization Problem) framework to optimize the occupation of all services by affecting the entire population.

References


Towards Automatic Dominance Breaking

Christopher Mears, Maria Garcia de la Banda

Faculty of IT, Monash University, Australia
{chris.mears,maria.garciadelabanda}@monash.edu

Abstract. Constraint optimisation problems can exhibit dominance relations whose exploitation can lead to dramatic reductions in search space. We propose an automatic method to detect some of the dominance relations identified recently by Chu and Stuckey [1] for a given optimisation problem, and to construct the associated dominance breaking constraints. Experimental results show that the method is able to find different kinds of dominance relations and generate effective dominance breaking constraints.

1 Introduction

Dominance relations can be seen as a generalisation of the well-known symmetry and conditional symmetry relations for optimisation problems. Thus, dominance breaking offers similar or even greater opportunities for pruning the search space than symmetry breaking. Interestingly, not much work has been devoted to dominance relations. While there are several well-known methods to identify and exploit most kinds of symmetry relations, the identification and exploitation of dominance relations has followed a problem-specific treatment (e.g. [3, 11, 12]) that has offered little insight into possible generalisations.

Recently, Chu and Stuckey [1] proposed a generic method for manually identifying a large class of dominance relations and manually exploiting them by using dominance breaking constraints. Intuitively, the method aims at finding mappings \( \sigma \) that, under a certain condition \( \text{cond} \), are known to map solutions to better solutions. That is, if \( \text{cond} \) holds, \( \sigma(\theta) \) maps solution \( \theta \) to another solution, whose objective function value \( f(\sigma(\theta)) \) is better (smaller/greater) than that of \( f(\theta) \). Then, the negation of \( \text{cond} \) can be used as a dominance breaking constraint. Chu and Stuckey [1] observe that a dominance mapping is good if its associated dominance breaking constraint \( \neg\text{cond} \) is simple, as simpler constraints will propagate faster and prune more. Clearly, mappings that are symmetries will lead to a simpler constraint, as they always map solutions to solutions and, therefore, \( \text{cond} \) only needs to ensure that \( f(\sigma(\theta)) \) is better.

As a result we can increase the usefulness of Chu and Stuckey’s work by automating this process, that is, by developing a method to (a) automatically identify symmetries for a given problem, and (b) automatically construct the associated dominance breaking constraints. Note that these dominance breaking constraints are not symmetry breaking constraints, as the key element is for \( f(\sigma(\theta)) \) to be better. Further, the symmetries need to be detected for the inherent satisfaction problem — that is, the optimisation problem without the objective function — because otherwise \( f(\sigma(\theta)) \) must be equal to \( f(\theta) \), and cannot be better than it.
Our first attempt at automating the process focused on ensuring the automatic inference of symmetries was done very efficiently, as it needs to be done every time we search for a solution to the problem. However, the methods that were efficient were not capable of inferring useful symmetries. We thus decided to try more accurate methods that could infer symmetries for the problem model, that is, a parameterised version of the problem specification that can later be instantiated with particular input data for the parameters. While these methods can take significant time, they are scalable and, once applied, the symmetries inferred can be used for all instances of the model, thus compensating for the extra time required.

While we obtained good results using the symmetries automatically inferred for the model, we realised that more could be done as not all dominance mappings are actual symmetries. Interestingly, we have determined that some mappings associated with the dominance breaking constraints identified in [1] correspond to almost symmetries, that is, to mappings that are symmetries of the problem if a few constraints are ignored. Therefore, we have extended our automatic detection process to infer dominance constraints derived from almost symmetries. In particular, given the importance of obtaining a simple dominance constraint, we have focused on the class of almost symmetries that are obtained from eliminating a single constraint from the problem. Our experimental evaluation shows that the automatically generated dominance breaking constraints can yield significant improvement in search performance, particularly when some of our currently manual refinements become automated.

2 Background

This section provides the necessary background notation and concepts, following mainly the definitions given in [1]. Let \( \equiv \) denote syntactical identity. A variable \( v \) is a mathematical quantity capable of assuming any value from its initial domain, which is determined by its type. Given a set of variables \( V \), let \( \Theta_V \) denote the set of valuations over \( V \) where each variable in \( V \) is assigned to a value in its initial domain. A constraint \( c \) over \( V \) is defined by a set of valuations \( \text{solns}(c) \subseteq \Theta_V \). Given a valuation \( \theta \) over \( V' \supset V \), we say \( \theta \) satisfies \( c \) if the restriction of \( \theta \) onto \( V \) is in \( \text{solns}(c) \). Otherwise, we say that \( \theta \) violates \( c \). A domain \( D \) over variables \( V \) is a set of unary constraints, one for each variable in \( V \). In an abuse of notation, if a symbol \( A \) refers to a set of constraints \{\( c_1, \ldots, c_n \)\}, we will often also use \( A \) to refer to the constraint \( c_1 \land \ldots \land c_n \).

A Constraint Satisfaction Problem (CSP) is a tuple \( P \equiv (V, D, C) \), where \( V \) is a set of variables, \( D \) is a domain over \( V \), and \( C \) is a set of \( n \)-ary constraints, each \( c \in C \) defined over a subset of \( V \). A valuation \( \theta \) over \( V \) is a solution of \( P \) if it satisfies every constraint in \( D \) and \( C \). A Constraint Optimisation Problem (COP) \( P \equiv (V, D, C, f) \) extends a CSP by adding an objective function \( f \) mapping \( v \) to an ordered set, e.g., \( \mathbb{Z} \) or \( \mathbb{R} \), with the aim of finding a solution to the CSP that minimises/maximises \( f \). As for [1], we deal with finite domain problems only, i.e., where the initial domain \( D \) constrains each variable to take values from a finite set of values. Further, for brevity, we assume that all objective functions are to be minimised, and consider constraint satisfaction problems as constraint optimisation problems where \( f(\theta) = 0 \) for any valuation \( \theta \).
We will represent CSPs and COPs using the MiniZinc [10] modelling language. Our automatic detection method follows [8] in distinguishing between a CSP (or COP) model and its instance. A model is any specification that can be expressed as a MiniZinc program with at least one unspecified parameter, where a parameter can be an integer, a set of integers or a sequence of integers. An instance of a model (the actual CSP or COP) is then defined as the result of extending the model by providing values to all its parameters. Therefore, an instance also corresponds to a MiniZinc program, but one without unspecified parameters.

Dominance relations for a COP are defined as follows.

Definition 1. (from [1].) A dominance relation \( \prec \) for a COP \( P \equiv (X,D,C,f) \) is a transitive and irreflexive binary relation on \( \Theta_X \) such that if \( \theta_1 \prec \theta_2 \), then either: 1) \( \theta_1 \) is a solution and \( \theta_2 \) is a non-solution, or 2) they are both solutions or both non-solutions and \( f(\theta_1) \leq f(\theta_2) \).

If \( \theta_1 \prec \theta_2 \), we say that \( \theta_1 \) dominates \( \theta_2 \). As proved in [1], any dominated valuation can be safely discarded. This can be extended from valuations to search nodes, so that we can also discard dominated search nodes.

Definition 2. (from [1].) Let \( D_1 \) and \( D_2 \) be the domains from two different search nodes. If \( \forall \theta_2 \in \text{sols}(D_2), \exists \theta_1 \in \text{sols}(D_1) \) s.t. \( \theta_1 \prec \theta_2 \), then we define \( D_1 \prec D_2 \).

Chu and Stuckey [1] provide a method for identifying dominance relations in a COP and adding constraints to exploit them. Formally, the method has four steps [1]:

1. Find mappings \( \sigma : \Theta_X \rightarrow \Theta_X \) that are likely to map solutions to better solutions, that is, mappings for which condition \( \sigma(\theta) \prec \theta \) often holds.
2. For each \( \sigma \), find a constraint \( \text{sccond}(\sigma) \) s.t. if \( \theta \in \text{sols}(C \land D \land \text{sccond}(\sigma)) \), then \( \sigma(\theta) \in \text{sols}(C \land D) \), i.e., a constraint under which \( \sigma \) maps solutions to solutions.
3. For each \( \sigma \), find a constraint \( \text{occond}(\sigma) \) s.t. if \( \theta \in \text{sols}(C \land D \land \text{occond}(\sigma)) \), then \( f(\sigma(\theta)) \prec f(\theta) \), i.e., a constraint under which \( \sigma \) maps solutions to better valuations.
4. For each \( \sigma \), post the dominance breaking constraint \( \neg(\text{sccond}(\sigma) \land \text{occond}(\sigma)) \).

A solution symmetry of CSP \( P \equiv (V,D,C) \) is a permutation of the variable/value pairs in \( P \) that maps solutions of \( P \) to solutions of \( P \) and non-solutions of \( P \) to non-solutions of \( P \) [2]. A special case of solution symmetry, called a valuation symmetry in [5] (and syntactic symmetry in [9]), occurs when \( \sigma \) maps all the variable/value pairs of each variable to the variable/value pairs of another variable. Importantly, such symmetries always map a valuation to another valuation and, thus, they can easily be extended to map constraints to constraints, since the mapping on variables is uniform. For non-valuation symmetries this definition may not be meaningful since \( \sigma(\theta) \) may not be a valuation (e.g., might contain two different values for the same variable). Many common symmetries are valuation symmetries, including interchangeable variables, interchangeable values, or interchangeable rows or columns in matrix problems [6].

A permutation group is a set of permutations that is closed under composition and inverses. The solution (and valuation) symmetries of a CSP \( P \) form a permutation group, where each element is a permutation on the set of variable/value pairs of
Given permutations \( \{f_1, f_2, \ldots, f_n\} \) for \( P \), we denote by \([f_1, f_2, \ldots, f_n]\) the closure under composition and inverses of \( \{f_1, f_2, \ldots, f_n\} \). For a permutation group \( G \), if \([f_1, f_2, \ldots, f_n] = G\) then we say that \( \{f_1, f_2, \ldots, f_n\} \) generates \( G \), and we call \( \{f_1, f_2, \ldots, f_n\} \) a generating set of \( G \).

### 3 Automatically inferring almost symmetries

The input to our method is a MiniZinc COP model \( M_{zn} \equiv (X, D, C, f) \) and \( n \) MiniZinc input data files \( d_1, \ldots, d_n \). Given this, our method first automatically infers symmetries and almost symmetries of \( M_{zn} \). Then, it uses some of these symmetries as dominance mappings to generate dominance breaking constraints that are added to \( M_{zn} \).

This section focuses on the first task, which is achieved by following the steps shown in Figure 1.

Reading Figure 1 from the left, the first step takes the original MiniZinc model \( M_{zn} \) and relaxes it by removing each of its constraints separately, each time giving a relaxed model paired with the removed constraint. Next, the (relaxed) model is compiled using each data file, giving several FlatZinc files. Each FlatZinc file has its satisfaction core extracted, and the symmetries found for that core. Finally, the instance symmetries are lifted to the model level, so that the final result is the set of symmetries for the (relaxed) model, paired with the constraint removed at the beginning.

Each of these steps is explained in the following subsections.

#### 3.1 Generating relaxed versions of a COP model

The first step of our method is to obtain relaxed versions of the MiniZinc COP model \( M_{zn} \) by simply removing some of its constraints. Our aim is to use these relaxed versions to find almost symmetries of \( M_{zn} \), that is, symmetries of the relaxed versions that are not symmetries of \( M_{zn} \). Like any regular symmetry, an almost symmetry \( \sigma \) is a mapping that can be used to generate the dominance breaking constraint
scond(σ) → ¬ocond(σ)). The main difference is that for a symmetry the scond(σ)
component is trivially true, while for an almost symmetry the scond(σ) component is
the mapped version of the constraint(s) removed to obtain the relaxed version of $Mzn$.
Note that these almost symmetries might be very useful if they lead to simple conditions
for producing better solutions.

We have automated the generation of relaxed problems by using the structure present
in any MiniZinc model, that is, their set of constraint “items”, which typically corre-
spond to the high-level logical constraints of the problem. Assuming our input COP
model $Mzn ≡ (X, D, C, f)$ has $m$ constraint items, we generate a set of relaxed problems
$\{Mzn_1, \ldots, Mzn_m\}$, where $Mzn_i ≡ (X, C_i, D, f)$ and $C_i$ is obtained by remov-
ing constraint item $c_i$ from $C$.

**Example 1.** The following model for the blackhole solitaire problem (see Section 5.2)
has four constraints, denoted in the model as A, B, C and D:

```plaintext
1 % Card at position
2 array[1..52] of var 1..52: cardAt;
3 % Position of card
4 array[1..52] of var 1..52: position;
5
6 % Constraint A: Ace of Spades is first card
7 constraint cardAt[1] == 1;
8
9 % Constraint B: Consecutive cards match
10 constraint forall(i in 1..51)
11   ( table([cardAt[i], cardAt[i+1]], neighbours) );
12
13 % Constraint C: Link card-at and position
14 constraint inverse(cardAt, position) :: domain;
15
16 % Constraint D: A card must be played before the one under it.
17 constraint forall(i in 1..17, j in 1..2)
18   ( position[layout[i,j]] < position[layout[i,j+1]] );
```

Four relaxed models can thus be derived from it: a model with only constraints
A,B,C; one with A,B,D; one with A,C,D; and one with B,C,D.

For each relaxed problem $Mzn_i$, we remember the constraint item $c_i$ that was re-
moved from $Mzn$ to obtain $Mzn_i$. As we will see later in Section 4, $c_i$ will become
the condition for any dominance breaking constraints generated for $Mzn$. This is the
main reason to eliminate only a single constraint item, as otherwise the condition in the
resulting dominance breaking constraint would likely be too complex.

### 3.2 Extracting a CSP from a COP

Once the relaxed models are obtained, our aim is to obtain the symmetries of the original
model and also those present in the relaxed models. This will allow us to reuse
the symmetries found (and the associated dominance breaking constraints) for every
instance of the problem.
Several methods are available to obtain the symmetries of a model, e.g. [17, 16, 4, 8]. We decided to use the method of [8], since it has been shown to have good accuracy for a wide range of benchmarks. The method first uses [7] to accurately detect a generating set for the symmetries of several small instances of the given CSP model. It then lifts these generating sets of instance symmetries to the model level, obtaining a generating set of candidate model symmetries. These candidates are filtered to eliminate those that clearly do not hold for the model, thus obtaining a generating set of likely candidates. Finally, the method marks the likely candidate generators that can be formally proved to be symmetries of the model.

The above method needs to be slightly modified to eliminate the objective function $f$ from the models (and instances) being analysed. This is because, as mentioned before, the symmetries need to be detected for the inherent satisfaction problem since, otherwise, $f(\sigma(\theta))$ will be known to be equal — rather than better than — $f(\theta)$ for any solution $\theta$. Chu and Stuckey [1] assumed a specification of their COP problem $P \equiv (X, D, C, f)$ where everything regarding the objective function is tucked away in $f$ and therefore a symmetry only needs to preserve the solutions of the $(X, D, C)$ satisfaction part. In practice, however, this is often not the case.

Example 2. Consider the following COP model of the photo placement problem from the MiniZinc distribution, where each person is assigned a different position in such a way that their combined positional preferences (who they would like to be next to) are maximised:

```plaintext
1 int: n_names;
2 int: n_prefs;
3 array[1..n_prefs, 0..1] of int: prefs;
4 array[0..n_names-1] of var 0..n_names-1: pos;
5
6 constraint alldifferent(pos);
7
8 var 0..n_prefs: satisfies;
9 array[1..n_prefs] of var bool: ful;
10 constraint forall (i in 1..n_prefs) {
11   let { int: pa = prefs[i, 0], int: pb = prefs[i, 1] } in ful[i] = (pos[pb]-pos[pa] == 1 xor pos[pa]-pos[pb] == 1);
12 constraint sum (i in 1..n_prefs) (bool2int(ful[i])) = satisfies;
13
14 solve maximize satisfies;
```

The first three lines determine the parameters of the model (number of names of people, number of preference pairs given, and the actual array of preferences), while line 4 provides the array of decision variables assigning each position to the name of a person. Line 6 shows the only true constraint of the problem: an alldifferent constraint indicating the names assigned to each position need to be different. The new variables and constraints introduced from lines 8 to 13 serve only to define the objective variable satisfies, since they do not exclude any assignments to the decision variables in array pos. If we are to look for the symmetries of the satisfaction problem, we do not want these constraints to be included in the analysis, as they can eliminate some of the
symmetries available. This can be detrimental as it might eliminate symmetries that are likely to find better solutions.

Let us introduce a method to detect and eliminate “objective constraints” from a COP instance, that is, to obtain the satisfaction part of a COP instance by eliminating constraints whose only role is to define the objective function \( f \) without reducing the number of solutions in the instance. To do this we will use the FlatZinc version of the COP instance, obtained by compiling the MiniZinc instance into FlatZinc [10]. First, we replace the \texttt{solve} item with \texttt{solve satisfy}. This removes the objective variable (or, in general, the objective expression) from the file. Next, we iteratively remove variables and constraints that meet the following condition: If a variable \( x \) is used in only one constraint \( c \), and that constraint \( c \) describes \( x \), then remove both \( c \) and \( x \).

**Definition 3.** A constraint \( c \equiv \text{con}(x_1, x_2, \ldots, x_n, y) \) describes variable \( y \) if the set of solutions \( \text{solns}(c) \) projected over \( \{x_1, \ldots, x_n\} \) is \( D(x_1) \times \cdots \times D(x_n) \).

Intuitively, this occurs whenever the solutions to \( c \) place no restriction on the values of the \( x_1, \ldots, x_n \) variables. In practice, this often corresponds to the case where \( y \) functionally depends on \( x_1, \ldots, x_n \), although a non-functional relationship is also possible. Clearly, whether a constraint describes a variable or not depends in general on the domains of all variables involved. This is why we use the COP instance rather than the model.

**Example 3.** Consider the constraint \( x + y = z \), with \( D(x) = D(y) = \{0, 1\} \) and \( D(z) = \{0, 1, 2\} \). This constraint describes only variable \( z \). However, if the domain was instead \( D(x) = \{0, 1, 2\} \), \( D(y) = \{1, 2\} \) and \( D(z) = \{0, 1\} \), then the constraint would describe only variable \( x \).

The MiniZinc-to-FlatZinc compiler can detect many cases where a constraint describes a variable, and annotates the FlatZinc with this information. For other cases, we examine the domains of the variables and the constraint itself to determine whether it describes a variable.

**Example 4.** Consider again the model introduced in Example 2, instantiated with a data file. The resulting FlatZinc file has many introduced variables and constraints for calculating the objective value. However, after the elimination of the objective and iteratively removing variables and constraints as described above, only the \texttt{pos} decision variables and the \texttt{alldifferent} constraint remain.

### 3.3 Putting it all together

The process shown in Figure 1 is now complete: starting with a COP MiniZinc model \( Mzn \equiv (X, D, C, f) \) that has \( m \) constraint items \( c_1, \ldots, c_m \), and \( n \) MiniZinc input data files \( d_1, \ldots, d_n \):

1. Generate relaxed COP MiniZinc models \( Mzn_1, \ldots, Mzn_m \) by removing constraint items \( c_1, \ldots, c_m \), respectively
2. Instantiate each COP MiniZinc model (be it relaxed or not) with each of the input data files and compile to FlatZinc obtaining $Fzn_1, \ldots, Fzn_n$ COP FlatZinc instances.

3. For each $Fzn_i$, COP FlatZinc instance, extract the CSP instance $FznSat_i$ and infer its generating set of symmetries $S_i$.

4. Use the generating set of symmetries inferred for every instance $S_1, \ldots, S_n$ to infer the generating set of symmetries $S$ of either the relaxed or original model. If the latter, $S$ are almost symmetries of $Mzn_i$; if the former, $S$ are symmetries of $Mzn$.

Note that the effect of the detected symmetries in $S$ on any variables introduced by the flattening process can be ignored. That is, we are only interested in how the symmetry affects the variables in $Mzn$, because we will only apply the symmetry to those variables (see Section 4).

As mentioned before, rather than using a model-based detection approach, we could use an instance-based approach, where the symmetries are inferred only for the particular instance being considered. In this case, it would be crucial for the method to be fast, as its run-time will be an overhead over the solving time for the original COP. Clearly, complete methods (e.g. [2]) tend to be computationally expensive, even for small CSPs. Incomplete methods (e.g., [13, 14, 2, 7]) can actually be very accurate, but they can also take significant amounts of time, since the more accurate ones require building a graph $G$ where each variable/value pair in the CSP is represented by its own vertex, in such a way that each automorphism of $G$ corresponds to a solution symmetry of the CSP.

We attempted to use a simple and fast instance-based symmetry detection method, based on the approach of Puget [13]. Intuitively, we built a simple coloured graph where there is a vertex for each variable in the instance, and constraints are represented by additional vertices connected to the variables appearing in the constraint, in such a way that the inherent symmetry of the constraints is reflected in the graph. For example, the representation of the constraint $x \neq y$ would treat $x$ and $y$ identically, while the representation of $x < y$ would not. Although this method works well for certain problems, many constraints cannot be represented. For example, the $\text{inverse}(x, y)$ constraint expresses a relation between the variables in $x$ and the values in $y$; since the values are not explicitly represented in this form of graph, the symmetry is lost.

### 4 Automatically Generating Dominance Breaking Constraints

Each of the symmetries and almost symmetries found in the previous step is a mapping $\sigma$ that can be used for dominance breaking. As mentioned before, the general form of a dominance breaking constraint in $\text{second}(\sigma) \rightarrow \neg \text{ocond}(\sigma)$. For a symmetry of the original model $Mzn$, the $\text{second}(\sigma)$ is trivially true. For a symmetry of $Mzn_i$ (and, thus, an almost symmetry of $Mzn$), $\text{second}(\sigma)$ is $\sigma(c_i)$ where $c_i$ is the constraint that was removed to relax the original model $Mzn$ to obtain $Mzn_i$. In both cases, the $\text{ocond}(\sigma)$ for a COP with objective function $f$ (to be minimised) is $\sigma(f) < f$. The generation of the dominance constraint requires that the mapping $\sigma$ be applied to an arbitrary MiniZinc constraint or expression. This is why we restrict our symmetry detection to valuation symmetries. Our current implementation only considers variable symmetries; that is, permutations of the variables that induce solution symmetries.
The method to generate the dominance breaking constraints is as follows. Assume that the mapping $\sigma$ is induced from a variable symmetry. We first extend $\sigma$ to map every variable $y$ in the problem that was left unchanged by $\sigma$, to a fresh variable $y'$. Then, for every such variable $y$ and every constraint item $c$ in the model, we add $\sigma(y)$ and $\sigma(c)$ to the model, respectively. Finally, we add the dominance breaking constraint $\text{cond}(\sigma) \rightarrow \neg\text{cond}(\sigma)$. The $\text{cond}(\sigma)$ is simply the constraint removed (if any) when relaxing the problem (see Section 3.1) with occurrences of any variable $y$ replaced by $y'$. The $\neg\text{cond}(\sigma)$ is simply the constraint $\sigma(f) \geq f$, where $\sigma(f)$ is the expression $f$, again with occurrences of any variable $y$ replaced by $y'$.

Example 5. Consider again the photo placement problem introduced in Example 2. Any permutation of the pos variables is a valuation symmetry of the satisfaction part of the problem. As an example, let us use the mapping that reverses the order of the variables $\text{pos}\{3\}, \text{pos}\{4\}$, and $\text{pos}\{5\}$.

A fresh array of variables is added to represent the image of pos after the mapping. The differences from the original model are shown in bold.

```plaintext
array[0..n_names-1] of var 0..n_names-1: \text{pos\_mapped} =
    [ pos[ if 3 <= i \ and \ i <= 5
        then 3+(5-i)
        else i endif ] | i in 0..n_names-1 ];
```

That is, $\text{pos\_mapped}[1] = \text{pos}[1], \text{pos\_mapped}[2] = \text{pos}[2], \text{pos\_mapped}[3] = \text{pos}[5]$, and so on.

The constraints and variables that were eliminated as part of the CSP extraction are also duplicated, but any occurrences of variables in the CSP are replaced by their mapped versions. In this example, it results in the addition of the following variables and constraints:

```plaintext
var 0..n_names-1: \text{satisfies\_mapped};
array[1..n prefs] of var bool: \text{ful\_mapped};
constraint forall (i in 1..n prefs) {
    let { int: pa = prefs[i,0], int: pb = prefs[i,1] } 
    in \text{ful\_mapped}[i] = (\text{pos\_mapped}[pb]-\text{pos\_mapped}[pa] == 1 xor \text{pos\_mapped}[pa]-\text{pos\_mapped}[pb] == 1) ;
}
constraint sum (i in 1..n prefs) (bool2int(ful\_mapped[i])) = \text{satisfies\_mapped};
```

Finally, the dominance breaking constraint is added:

```plaintext
constraint \text{satisfies} >= \text{satisfies\_mapped};
```

While this construction can lead to a significant growth in the model, often most of the newly introduced constraint items are simple renamings of already existing items and can be immediately removed. Even those items that change may be only superficially different. For example, if the original model had a constraint $\text{alldifferent}(x)$, then the new constraint $\text{alldifferent}(x\text{\_mapped})$ which, although not syntactically the same, is wholly redundant.
Example 6. Most of the constraints added in Example 5 are for defining the $ful\text{\_mapped}$ variables. For values of $i$ where the $pa$ and $pb$ values are outside the range affected by the mapping, the constraint defining $ful\text{\_mapped}[i]$ is identical to the one that defines $ful[i]$. As a result, the new constraint can be eliminated, and $ful\text{\_mapped}[i]$ made equal to $ful[i]$.

Importantly, we cannot simply add a dominance breaking constraint for every element of the symmetry group. The reasons are twofold: first, the typical size of symmetry groups is so large that to do so would incur an enormous overhead which the reduction in search space would not recoup. Second, even if the group is small, some of the associated dominance breaking constraints might by themselves already add significant overheads. This means that even restricting ourselves to the set of generators automatically inferred by our symmetry detection method might not be ideal. We would like to add only those dominance breaking constraints that will propagate efficiently and give good search space reduction. We choose the best constraints by generating a small subset of the full symmetry group, and comparing the size of the resulting constraints. While we currently do this manually, it is relatively easy to automate this process.

5 Experimental Evaluation

We have evaluated the effectiveness of the automatically generated dominance breaking constraints on some benchmark problems. For these experiments, we use the MiniZinc 2.0 compiler and Gecode 4.2.1. All times are given in seconds.

5.1 Photo Placement

We have considered two models of the photo placement problem. The first model (referred to as photo-1) is the one included in the MiniZinc distribution (and shown in Example 2). Note that we removed the manual symmetry breaking constraint from the model to get closer to the raw problem. The second model (referred to as photo-2) is the one given by Chu and Stuckey [1], and is significantly more efficient regardless of dominance breaking constraints.

For both models, after the CSP extraction the only remaining decision variables are the position variables and the only constraint is the all-different constraint. Therefore, any permutation of the position variables is a symmetry. We consider two sets of dominance mappings: all-pairs swaps and all-pairs reversals. The swaps are mappings that interchange positions $i$ and $j$, for all $i, j$ pairs. The reversals are mappings that reverse all the subsequence of positions between position $i$ and $j$, for all $i, j$ pairs. Note that the dominance constraints resulting from the reversal mappings (shown in Example 5) are the ones manually identified by Chu and Stuckey [1] for this problem.

Example 7. The following shows the duplicated variables and constraints, and added dominance breaking constraint, for the mapping that reverses positions 1 and 2.

```plaintext
array [1..n] of var 1..n : x\_mapped =
[x [ if 1 <= i \&\& i <= 2
```
Then 2 - i + 1
else i endif | i in 1..n |
constraint all_different(x_mapped);
constraint sum(i in 1..n-1) (p[x_mapped[i], x_mapped[i+1]])
- sum(i in 1..n-1) (p[x[i], x[i+1]])
<= 0;

Let us first compare the effect on search, by showing the total number of search
nodes obtained when solving different instances of each of the two models without
dominance breaking, and with dominance breaking using swap mappings and reversal
mappings. Note that in a branch-and-bound search, the additional constraints can increase
the search effort if they prevent some solutions from being found early, and
these solutions would have imposed constraints on the objective.

<table>
<thead>
<tr>
<th>Instance</th>
<th>None</th>
<th>Swaps</th>
<th>Reversals</th>
</tr>
</thead>
<tbody>
<tr>
<td>photo1-1</td>
<td>35,774</td>
<td>44,537</td>
<td>37,132</td>
</tr>
<tr>
<td>photo1-2</td>
<td>480,109</td>
<td>809,282</td>
<td>506,268</td>
</tr>
<tr>
<td>photo2-12-1</td>
<td>4,015,993</td>
<td>1,199,867</td>
<td>798,781</td>
</tr>
<tr>
<td>photo2-12-2</td>
<td>6,405,713</td>
<td>1,574,469</td>
<td>1,102,183</td>
</tr>
<tr>
<td>photo2-14-2</td>
<td>227,036,415</td>
<td>35,153,627</td>
<td>17,583,257</td>
</tr>
</tbody>
</table>

Table 1. Node counts without dominance breaking and with two different sets of mappings.

Table 1 shows how the different dominance breaking constraints affect the size of
the search tree. Clearly, the reversal mappings consistently give a smaller search than
the swap mappings for both models. In the first model, the dominance breaking
constraints increase the node count, while in the second model they decrease it. A likely
explanation for the poor performance in the first model is that the dominance breaking
constraints propagate poorly and, therefore, we have the downside of excluding early
solutions, but not the advantage of reducing the search tree size. We are exploring ways
to automatically detect and transform the constraints to improve their effectiveness.

Next we compare the effect on running time (see Table 2). Since the reversal map-
pings are better than the swap mappings, we consider only those. To measure the effect
of excluding redundant constraints, we manually remove the \texttt{alldifferent} constraints
that are simply re-arranged versions of the original constraint. As expected, removing

<table>
<thead>
<tr>
<th>Instance</th>
<th>None</th>
<th>Reversals</th>
<th>Reversals w/o Redundant</th>
</tr>
</thead>
<tbody>
<tr>
<td>photo1-1</td>
<td>0.125</td>
<td>1.532</td>
<td>1.232</td>
</tr>
<tr>
<td>photo1-2</td>
<td>1.259</td>
<td>33.079</td>
<td>25.057</td>
</tr>
<tr>
<td>photo2-12-1</td>
<td>24.825</td>
<td>43.226</td>
<td>23.122</td>
</tr>
<tr>
<td>photo2-12-2</td>
<td>35.340</td>
<td>55.555</td>
<td>29.167</td>
</tr>
<tr>
<td>photo2-14-2</td>
<td>1626.636</td>
<td>1319.246</td>
<td>666.206</td>
</tr>
</tbody>
</table>

Table 2. Running time without dominance breaking, with dominance breaking, and with manual
removal of some redundant constraints.
the redundant constraints reduces the running time. For those instances where the dominance breaking constraints cause a large reduction in search (all instances of photo-2), there is also a reduction in running time.

5.2 Blackhole Solitaire

In the blackhole solitaire problem, a standard deck of playing cards — minus the ace of spades — is dealt into 17 piles of three cards. These cards are played one-by-one into a discard pile, which begins with only the ace of spades. Each card played must be one higher or one lower than the current top of the discard pile, and only the top card of each pile is playable. The task is to find a sequence of plays that leads to all cards being on the discard pile.

A model for this problem (shown in Example 1) is included in the MiniZinc distribution with (a) a variable for each card, whose value is the position in the sequence it is played, and (b) a variable for each position in the sequence, whose value is the card played at that position. These variables are linked by an inverse constraint. The other two significant constraints state that cards played in succession must be compatible (their face values differ by exactly one), and that each card must be played before any card underneath it in its pile.

This model has an almost symmetry: except for the top-of-pile constraint, cards with the same face value are symmetric. Relaxing the problem to remove this constraint leads to its automatic symmetry detection, that is, finding 37 pairs of cards that can be swapped and thus used as dominance mappings.

The removed top-of-pile constraint is as follows:

\[
\text{constraint } \forall (i \in 1..17, j \in 1..2) \ (\text{position}[\text{layout}[i,j]] < \text{position}[\text{layout}[i,j+1]])
\]

where \(\text{layout}[i,j]\) represents the \(j\)th card of pile \(i\), and variable \(\text{position}[\text{card}]\) represents the position in the sequence that \(\text{card}\) is to be played. To compute the second for the dominance breaking constraint for a single mapping, we must apply the mapping to this constraint. Since this is a satisfaction problem, the second imposes an arbitrary ordering on the variables to break the symmetry. This leads to the following dominance breaking constraint:

\[
\text{constraint } \forall (i \in 1..17, j \in 1..2) \ (\text{y_mapped}[\text{layout}[i,j]] < \text{y_mapped}[\text{layout}[i,j+1]]) \rightarrow \text{lex_lesseq}(y, \text{y_mapped});
\]

where \(\text{y_mapped}\) is the image of \(y\) under the specific mapping.

This constraint has two problems. First, it requires the solver to support a reified version of the lexicographical ordering constraint. Second, the left hand side of the implication is too strong, causing very weak propagation.

The first problem can be fixed by observing that the \(y\) variables are constrained to be all-different and, therefore, the lexicographical constraint can be reduced to \(y[a] < y[b]\), where \(a\) and \(b\) are the two cards being swapped by the mapping.

The second problem is more difficult to fix automatically. A single mapping affects at most two of the piles; the other piles are unchanged and so that part of the condition...
is known to be true and could be eliminated from the forall. We have manually altered the constraint to eliminate any of these redundant parts. The resulting dominance constraints are the ones manually identified by Chu and Stuckey [1] for this problem.

<table>
<thead>
<tr>
<th>Instance</th>
<th>None</th>
<th>Dominance</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackhole-9</td>
<td>346,973</td>
<td>8,221</td>
</tr>
<tr>
<td>blackhole-14</td>
<td>206,016,606</td>
<td>11,054,241</td>
</tr>
<tr>
<td>blackhole-16</td>
<td>&gt;336,752,365</td>
<td>4,015,958</td>
</tr>
<tr>
<td>blackhole-18</td>
<td>312,905</td>
<td>30,560</td>
</tr>
<tr>
<td>blackhole-20</td>
<td>916,488</td>
<td>62,131</td>
</tr>
</tbody>
</table>

Table 3. Node counts without and with dominance breaking.

Again, we first compare the effect on search reduction for several instances of the model (see Table 3). Clearly, the dominance breaking constraints cause a large reduction in search space. Without the manual modification described above, the dominance breaking constraint appears does not reduce the search space at all.

<table>
<thead>
<tr>
<th>Instance</th>
<th>None</th>
<th>Dominance</th>
</tr>
</thead>
<tbody>
<tr>
<td>blackhole-9</td>
<td>13.009</td>
<td>0.449</td>
</tr>
<tr>
<td>blackhole-14</td>
<td>3687.822</td>
<td>210.896</td>
</tr>
<tr>
<td>blackhole-16</td>
<td>&gt;7200</td>
<td>128.218</td>
</tr>
<tr>
<td>blackhole-18</td>
<td>9.123</td>
<td>1.138</td>
</tr>
<tr>
<td>blackhole-20</td>
<td>16.889</td>
<td>1.709</td>
</tr>
</tbody>
</table>

Table 4. Running time without and with dominance breaking.

Next, we compare the effect on running time (see Table 4). The running times are also greatly reduced by the dominance breaking constraints.

5.3 Talent Scheduling

The talent scheduling problem requires a set of film scenes to be arranged in some order so as to minimise actors’ wages. Each scene requires a subset of actors, and actors are paid for the duration that they are on set, which is from the time of their first scene to the time of their last scene. As a result, the aim is to find a scene schedule that minimises the amount of time the actors spend waiting.

We use the model of Chu and Stuckey [1]. As with the photo placement problem, once the objective-related constraints are eliminated by the CSP extraction, the only constraint that remains is an alldifferent constraint on the decision variables that map scenes to schedule. Therefore, the resulting CSP has symmetries indicating that all scenes can be reordered freely.

However, the automatically generated dominance breaking constraints for these symmetries — whether they swap pairs of variables, or reverse a subsequence — are
harmful for this model, as they lead to an increase in both the search space and running time. As discussed previously, the extra constraints can interfere with the finding of solutions which would otherwise help the branch and bound search. In this case, the dominance breaking constraints also bring no benefit, a phenomenon that we will illustrate with an example.

Consider an instance where scene 1 requires actors A, B and C, while scene 2 requires only actor B. It is clear that if these two scenes are filmed at times 1 and 2, then scene 2 should come first: this means that actors A and C can arrive later and need not wait during the filming of scene 2. Any solution that begins with scene 1 and then scene 2 will be dominated by the same solution where those two scenes are swapped.

Ideally, this would be detected after scene 1 and scene 2 are assigned to time slot 1 and 2 respectively, and the search would backtrack immediately. However, this is not the case, as we can see by observing the domains of the involved variables. Consider the variable that represents the wage of actor A. Before search begins, the lower bound on their on-set time is the number of scenes they are in, and the upper bound is the total number of time slots in the schedule. Under the assignment “scene 1” then “scene 2”, their upper bound does not change, but we now know that they wait for at least one time slot so their lower bound increases slightly. Under the other assignment, “scene 2” then “scene 1”, their lower bound does not change but their upper bound decreases slightly because we know they are not on set in the first time slot.

This small change in the bounds is not sufficient to cause backtracking. The vital missing piece of information is that the rest of their schedule has exactly the same cost under both assignments. Without this, it is only at the leaves of the search tree that the dominance breaking takes effect.

6 Conclusion

Dominance relations are often present in constraint optimisation problems. We have presented an approach to automatically detecting these relations by finding the symmetries of the satisfaction core of the problem. Also, the method can find conditional dominance relations by finding the “almost symmetries” of the satisfaction core.

We have also presented a method to automatically generate dominance breaking constraints for detected relations. These constraints can significantly reduce the search effort and computation time required to solve the problems. However, adding such constraints does not always improve the model; in some cases they can add overhead without benefit, or actually enlarge the search tree.

These issues raise several avenues for further work. To improve propagation, it is important to simplify the generated constraints automatically. To reduce the running time, constraints that are wholly redundant should be eliminated, perhaps by improving the canonicalisation of the constraints so that common subexpression elimination [15] can merge them.

References

1. Geoffrey Chu and Peter J. Stuckey. A generic method for identifying and exploiting dominance relations. In Michela Milano, editor, Principles and Practice of Constraint Program-
Modelling RTP-based Residential Load Scheduling for
Demand Response in Smart Grids

Shan He\textsuperscript{1,2}, Ariel Liebman\textsuperscript{1}, Andrea Rendl\textsuperscript{1,2}, Mark Wallace\textsuperscript{1,2}, Campbell Wilson\textsuperscript{1}

\textsuperscript{1} Faculty of IT, Monash University, Australia
\textsuperscript{2} National ICT Australia (NICTA) Victoria
{shan.he,ariel.liebman,andrea.rendl,mark.wallace,campbell.wilson}@monash.edu

Abstract. High electricity demand peaks and uncertain supply from renewable energy sources have a significant impact on the electricity price and the network capacity. One mechanism proposed to tackle this issue is the use of real-time pricing (RTP) at the end customer level. Here electricity retail prices are set in real-time in response to varying supply-demand conditions in a way that reduces peak demand. This way customers have an incentive to switch their consumption to times with low demand. The RTP-based Residential Load Scheduling problem (RTP-RSP) deals with scheduling the customer consumption such that the overall network consumption is balanced, the electricity price is minimized, and customer satisfaction maximized. In this work, we introduce different formulations of the RTP-RSP. We first introduce a formulation where the electricity price is assumed to be known a priori. This model is embedded in a heuristic approach that iteratively adapts the electricity price, based on the actual consumption that is computed by the models. Second, we present a two-stage stochastic optimisation model, where the electricity prices are stochastic. We evaluate both formulations on data based on real-world figures and present some preliminary results.

1 Introduction

Interconnected networks, commonly known as grids, deliver electricity from suppliers, such as power plants, to consumers. Since electricity is not stored within the grid, the electricity supply and demand must balance at any time and at any point in the network.

Over the past decades electricity demand has grown steadily, while in recent years in Australia and other countries with extreme hot weather conditions, peak demand has grown faster due to recent affordability of reverse-cycle air conditioners, leading to very high demand peaks during the day and low utilisation levels of networks. This puts considerable pressure on electricity power plants as well as on the networks themselves; higher demand requires increased high-voltage transmission network capacity and low-voltage distribution network capacity. However, reinforcing the grid for peak demand, as well as building new power plants, is a costly way to meet increased demand and can lead to a substantial increase in the electricity price. Furthermore, the supply from renewable energy sources, such as solar panels or wind turbines, is a function of

\textsuperscript{*} NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.
external factors such as weather and are out of the operators’ control. As a result, supply increases and decreases at unpredictable times, making it more difficult to balance supply and demand.

To address these issues, so-called smart grids have been proposed. Smart grids integrate Information System technologies to deliver electricity in a more efficient way. Smart grids would incorporate smart meters and smart appliances that jointly enable two-way communication between customers and the network operator. For example, smart meters record the consumption of the consumer, send it to the network, and receive the pricing information from the network.

Using smart meters, electricity service providers can encourage customers to reduce their electric consumption at certain times by passing through a real-time electricity price that reflects the degree to which the network is loaded due to changing demand. The literature refers to schemes like this as real time pricing (RTP) [1], under which a dynamic price is generated for every trading period (or timeslot), e.g. half an hourly, based on the aggregated load on the network. This means that the electricity price at a given time depends on the overall electricity consumption in the grid at that time. Customers thereby can reduce their electricity costs by shifting their consumptions to timeslots with a lower price. This process is referred to as demand response or demand management.

While a fully user interactive scheduling process is feasible, it is impractical and unnecessary under the smart grid paradigm. One of the core promises of the smart grid, is the delegation and automation of the demand response process to automated demand response systems. These systems determine the best times to schedule electric appliances, based on the price and the customer’s preferences. Customer preferences reflect the inconvenience of shifting electric consumption from a preferred time slot. Thus, the objective is to minimize the customers’ inconvenience and their electricity costs. Note that these two objectives are often in conflict: on the one hand, customers often have similar appliances and preferred consumption times (such as using the electric heating in the morning), but on the other hand, the electricity price would be higher if most customers schedule their appliances at the same time.

In this work, we consider the problem of automated demand response for many households within a smart grid. Furthermore, we present models for scheduling appliances in a single household, where the upcoming electricity price is uncertain. We present a global heuristic model where for each household we generate a model that is then integrated into an iterative ‘optimisation loop’ where in each iteration, the household’s schedule is recomputed in response to an updated price forecast, until the overall system converges to minimal overall electricity prices and customer inconvenience. First, we present a deterministic model, where we assume full knowledge of future electricity prices. Second, we propose a stochastic two-stage formulation, where the electricity prices are uncertain for the second half of the day. Furthermore, we report preliminary experimental results on both models.

This paper is organized as follows. First, we give a detailed problem description in Sec. 2. In Sec. 3, we present our iterative heuristic approach to deal with real-time pricing. We continue by introducing deterministic models for the single-household schedul-
ing problem in Sec. 4, and their stochastic variant in Sec. 5. Finally, Sec. 7 discusses our empirical evaluation and Sec. 8 concludes our work.

2 The RTP-based Residential Load Scheduling Problem

The RTP-based residential scheduling problem (RTP-RSP) is concerned with scheduling electric appliances in households under real time pricing (RTP) to manage peak demand in the grid. The objective is to minimize the overall electricity cost, and to minimize the user inconvenience for many households. The user inconvenience is derived from customer preferences and quantifies how inconvenient it is to shift the operation of an appliance from the preferred time to a different time. In this work, we assume that the user preferences are fully known for all the appliances in their households.

2.1 Formal problem formulation

In the RTP-RSP, the electric appliances are scheduled over a set of discrete time slots \( \mathbb{T} = \{1, ..., T\} \) where each time slot has the same duration \( \tau \). The set of households on the network is denoted as \( \mathbb{H} = \{h_1, ..., h_H\} \) where \( H \) denotes the number of households.

Running an electric appliance is defined as a job. Each household \( h_i \in \mathbb{H} \) has to schedule a set of \( J \) jobs \( \mathbb{J}_{h_i} = \{j_{i1}, ..., j_{ij}\} \), where job \( j_{ik} \in \mathbb{J}_{h_i} \) is the \( k \)-th job of household \( h_i \). Each job \( j_{ik} \) of household \( h_i \) is a tuple \( j_{ik} = (c_{ik}, e_{ik}, l_{ik}, q_{ik}, d_{ik}, f_{ik}) \), where \( c_{ik} \in \mathbb{R} \) is the electric consumption in kilowatt hour (kwh); \( e_{ik} \in \mathbb{T} \) is the earliest start time; \( l_{ik} \in \mathbb{T} \) is the latest end time; \( q_{ik} \in \mathbb{T} \) is the preferred start time; \( d_{ik} \in \mathbb{T} \) is the duration (the number of time slots that job \( j_{ik} \) will run over); \( f_{ik} \in [0, 1] \) is the "care factor", which represents the degree of inconvenience to customers when job \( j_{ik} \) is not scheduled at the preferred start time \( p_{ik} \), where \( '1' \) represents the maximal inconvenience.

The real time pricing is represented by a dynamic pricing table \( P \). The table contains \( K \) prices \( p_1, ..., p_K \) that each map to a consumption threshold for each time slot, \( p_i \rightarrow \{\hat{c}_{p_{i1}}, ..., \hat{c}_{p_{iT}}\} \). Thus, if the electricity consumption \( \hat{c}_i \) for time slot \( i \) goes beyond \( \hat{c}_{p_{ij}} \), then the price for time slot \( i \) corresponds to \( p_i \). An example of a dynamic pricing table is illustrated in Table 1.

A solution to the RTP-RSP is a schedule for each household, summarized in the set \( \mathbb{S} = (S_1, ..., S_H) \), where \( S_i \) is a valid schedule of electric appliances of the \( i \)-th household. Each schedule \( S_i \) corresponds to a list of start times for each job, \( S_i = (s_{i1}, ..., s_{ij}) \), where \( s_{ik} \in \mathbb{T} \) is the scheduled start time of job \( j_{ik} \) of household \( h_i \).

The objective is to minimize each household’s electricity bill and inconvenience. For each household, the electricity bill is calculated as:

\[
\forall i \in \{1, ..., H\} : \text{bill}_i = \sum_{j=1}^{T} p_a \ast b_{aj} \ast \hat{c}_j, \tag{1}
\]

\[
b_{aj} \iff \hat{c}_{pa} \leq \hat{c}_j < \hat{c}_{pa+1} \tag{2}
\]

\[
\hat{c}_j = \sum_{l=1}^{J} ((s_{il} \leq j) \land (s_{il} + d_{il} - 1 \geq j)) \ast c'_l, \tag{3}
\]
Table 1. Sample Dynamic pricing table with 7 consumption thresholds for 10 time slots. As an example, if the overall consumption reaches 10.3KWh at timeslot 1, then threshold $\hat{c}_{p3,4}$ is used to obtain the price, which is $p_3 = 15.4$.

<table>
<thead>
<tr>
<th>Electricity Price [cents]</th>
<th>Consumption thresholds [KWh]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>$\hat{c}<em>{p1,1}$ $\hat{c}</em>{p1,2}$ $\hat{c}<em>{p1,3}$ $\hat{c}</em>{p1,4}$ $\hat{c}<em>{p1,5}$ $\hat{c}</em>{p1,6}$ $\hat{c}<em>{p1,7}$ $\hat{c}</em>{p1,8}$ $\hat{c}<em>{p1,9}$ $\hat{c}</em>{p1,10}$</td>
</tr>
<tr>
<td>15.1</td>
<td>9.9 10.0 10.2 10.3 10.1 9.1 8.8 9.0 8.9 9.0</td>
</tr>
<tr>
<td>$p_2$</td>
<td>15.4 10.1 10.3 10.5 10.6 10.4 9.4 9.1 9.2 9.2</td>
</tr>
<tr>
<td>$p_3$</td>
<td>15.8 10.4 10.5 10.8 10.9 10.7 9.7 9.4 9.5 9.4 9.5</td>
</tr>
<tr>
<td>$p_4$</td>
<td>16.3 10.7 10.8 11.1 11.2 11.0 9.9 9.7 9.8 9.7</td>
</tr>
<tr>
<td>$p_5$</td>
<td>17.0 11.0 11.1 11.3 11.5 11.3 10.2 9.9 10.1 10.0 10.1</td>
</tr>
<tr>
<td>$p_6$</td>
<td>17.8 11.3 11.4 11.6 11.8 11.5 10.5 10.2 10.4 10.3 10.4</td>
</tr>
<tr>
<td>$p_7$</td>
<td>18.9 11.5 11.7 11.9 12.0 11.8 10.8 10.5 10.6 10.6</td>
</tr>
</tbody>
</table>

\[ c'_l = \frac{c_l \times 24}{T}. \]  

where $p_a$ is the price of time slot $t_j$ assuming the total consumption of that time slot is greater than $\hat{c}_{p_a,j} ; c_j$ is the total consumption of time slot $t_j ; J$ is the total number of the jobs ; $s_{ij}$ is the scheduled start time for job $j_l$ of household $h_i ; c'_l$ is the electric consumption of job $j_l$ in kilowatt per time slot duration $\tau$ (which is less than an hour).

More specifically, in our work, $\tau = 10$ minutes and $T = 144$ time slots.

The customer inconvenience is calculated as:

\[ \forall i \in \{1..H\} : \text{inconv}_i = \sum_{k=1}^{J} |q_{ik} - s_{ik}| \times f_{ik}, \]  

where $q_{ik}$ is the preferred start time for job $j_k$ of household $h_i$, $s_{ik}$ is the scheduled start time, and $f_{ik}$ is the care factor. The overall objective is to find schedules for all households such that the cost of each household is minimized:

\[ \forall i \in \{1..H\} : \text{cost}_i = \text{bill}_i + \text{inconv}_i \]  

Example 1. As an example, consider the problem of scheduling 6 electric appliances for a single household. The jobs’ parameters are summarized in Table 2. As an example, job-1 consumes 0.9 KWh and takes 5 time slots (50 minutes) to finish. It has to be scheduled between the first (0:00) and the 120th (19:50) time slot, however the customer prefers to schedule it at the 91st time slot (15:00). The care factor is 0, which means the customer does not mind to schedule the job at a different time than the preferred time. A care factor of 1, like for job-5, means that the customer would like to schedule the job as close as possible to the preferred time.

3 Iteratively Improving Electricity Price and Household Schedules

In this work we consider the interleaved objectives of minimizing the electricity price by avoiding demand peaks and minimizing the customer’s inconvenience for scheduling...
their electric appliances. This stems from the real-time pricing (RTP) scheme, where the
electricity price is calculated in real time for each time slot \( t \). In RTP, the price depends
on two different factors: the overall consumption and the electricity supply at \( t \). This
introduces two issues. First, the overall consumption depends on the household sched-
ules that we seek to compute. This makes scheduling appliances in a single household
difficult, since the schedule influences the actual price. Second, the electricity supply
is uncertain, since renewable energy sources depend on external factors such as the
weather.

We tackle these two issues in the following way. First, we introduce an \textit{iterative}
market simulator to deal with the connection between electricity price and actual con-
sumption. The market simulator is an iterative heuristic optimisation approach and dis-
cussed in detail below. Second, we consider randomly varying electricity supply as the
stochastic component of our problem, which we will discuss in Section 5. For the re-
main ing part of this section however, we assume that the electricity supply is known.

3.1 Iterative Electricity Market Simulator

The iterative market simulator emulates the electricity market and aims to converge
to an (local) optimal price via an iterative process. It employs two main components:
household energy schedulers and an electricity price calculator.

A household energy scheduler takes a fixed electricity price and computes an opti-
mal schedule for a single household, considering all its appliances and the user prefer-
ences. We will refer to this problem as the Household Load Scheduling Problem.

The price calculator takes an aggregated load schedule (the schedules of all house-
holds) as input, and uses a dynamic pricing table to calculate the actual electricity price
based on those schedules. The dynamic pricing table maps consumption thresholds to
electricity prices (for each time slot): if the electricity consumption exceeds the thresh-
old \( \hat{c}_{p2} \) at some time \( t \), then the price for that time slot is \( p_2 \).

The iterative approximation approach of the market simulator is illustrated in Fig. 1.
In the first iteration, it initializes the electricity prices to a random value, and then calls all
house energy schedulers to schedule their electric appliances against the initial price.
This way each house energy scheduler produces a load schedule. In the second step,
market simulator produces an aggregated load schedule by summing up the schedules
of each house to obtain the total consumptions of each time slot over all houses. Thirdly,
the market simulator calls the price calculator to produce a new price schedule for the
whole day, based on the aggregated load schedule.

<table>
<thead>
<tr>
<th>Job parameters</th>
<th>job-1</th>
<th>job-2</th>
<th>job-3</th>
<th>job-4</th>
<th>job-5</th>
<th>job-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Electric consumption (KWh): ( c_{ik} )</td>
<td>0.9</td>
<td>0.9</td>
<td>2.5</td>
<td>2.5</td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td>Earliest start time: ( e_{ik} )</td>
<td>1</td>
<td>1</td>
<td>85</td>
<td>85</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>Preferred start time: ( q_{ik} )</td>
<td>91</td>
<td>91</td>
<td>97</td>
<td>97</td>
<td>97</td>
<td>97</td>
</tr>
<tr>
<td>Duration: ( d_{ik} )</td>
<td>5</td>
<td>5</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Latest end time: ( l_{ik} )</td>
<td>120</td>
<td>120</td>
<td>132</td>
<td>132</td>
<td>132</td>
<td>132</td>
</tr>
<tr>
<td>Care factor: ( f_{ik} )</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0.5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\textbf{Table 2.} Jobs properties of household in Example 1
Fig. 1. Illustration of the iterative heuristic to obtain an optimal electricity price

In the second iteration, the market simulator calls all house energy schedulers to schedule its electric appliances against the new price schedule and continues with this procedure, until an iteration limit has been reached. The idea is that after several iterations, the load schedules of houses will not change, but converge to a local optimal solution for all houses.

4 Deterministic Household Load Scheduling Models

The key components of the market simulator are the household schedulers that compute an optimal schedule for each household, given the current electricity price. To apply the market simulator on real-world problems (where we schedule appliances of thousands of houses), it is vital that the Household Load Scheduling Problem is solved efficiently, in as little time as possible. Therefore, we now consider this problem and its deterministic variant.

In the deterministic variant, the household schedulers compute an optimal schedule for a single household, given a fixed electricity price table and the customer preferences. This means that we consider the electricity price a function of the demand, and assume that the demand is known. This allows us to facilitate the parameters and preprocess them to slim down the problem models. The parameter preprocessing is described in Sec. 4.1.

4.1 Parameter Preprocessing

During parameter processing we merge certain parameters to facilitate modelling. Since we assume that the electricity price is known a priori, and the customer preferences and job durations are given as a parameter, we can use them to compute the overall cost of scheduling an appliance at a specific time slot. More specifically, the cost of scheduling an appliance $i$ at time $t$ is defined as: $cost_{it} = bill_{it} + inconv_{it}$. 
Furthermore, we sort the timeslots by their price in an array that we use in the models to achieve a better search strategy. It allows us to direct the search to try out the cheapest time slots first.

### 4.2 Constraint Programming Model

We depict the Constraint Programming (CP) model in Fig. 2 in the modelling language MiniZinc [8]. First, we include the global constraints library in line 1, since we are using the global constraint `cumulative`. Then we declare the (preprocessed) problem parameters from line 3-15: the number of time slots per day (line 3), the number of jobs (line 5), the cost of scheduling a job at a specific timeslot (combining electricity price and the customer inconvenience) at line 11, the time slots ordered by their cheapness in the array `bestTimeSlots` (line 12) where `bestTimeSlots[i,1]=4` states that the cheapest time slot for scheduling job `i` is slot 4, and `bestTimeSlots[i,2]=7` states that the second-cheapest time slot for job `i` is slot 4, and so on. Further parameters are the consumption (line 7), duration (line 8), earliest start time (line 9), and latest finish time (line 10) per job, and the overall consumption limit per time slot (line 15).

Then we define the decision variables in lines 18-19. The variables `schedule` represent the timeslot (not ordered chronologically but according to their cheapness) at which each job is scheduled. For instance, `schedule[i] = 3` means that job `i` is scheduled at the 3rd cheapest time slot for that job `i`. This order corresponds to the time slot order obtained during preprocessing and can be easily converted to the actual starting times. The second set of variables, `actualCosts` (line 19), represents the overall cost for each job.

The constraints are listed from line 22-28. First, we link the actual cost to the schedule (line 22). Since the costs are ordered by cheapness and not chronologically, this reflects the actual cost of each job. Second, we impose the `cumulative` constraint on the schedule, stating that the scheduled jobs may not exceed the consumption limit. Finally, we state the search strategy (line 31) and the objective (line 33).

### 5 A Stochastic Household Load Scheduling Model

In order to model the impact on the market price due to unpredictable and uncontrollable sources of generation (renewable energy sources), we incorporate a stochastic element in our market simulator. Since the majority of these renewable sources are incorporated into wholesale markets as effectively zero variable-cost electricity generators with a stochastic output, they can be treated as subtractive modifiers of the total system demand. This is reflected as a stochastic horizontal shift in the supply curve for every timeslot of the day. The construction of this supply curve is explained in Section 7.

This means that in the stochastic variant of the Household Load Scheduling Problem, we consider the cost (that incorporates electricity price and the customer inconvenience) as uncertain, due to unpredictable fluctuations in supply. We formulate the problem as a two-stage stochastic optimisation problem where we assume that the cost for the first part of the day is known and the cost of the second part is uncertain. This is a
First step towards a multi-stage stochastic optimisation problem, where each new time slot corresponds to a stage. We leave the multi-stage formulation for future work.

The model takes $N$ cost scenarios as input, where each scenario represents a possible outcome of the cost of the second half of the day. Furthermore, each scenario is assigned a weight that represents its probability of occurring.

We model the stochastic formulation using Stochastic MiniZinc [9] that can translate the model to its scenario-based deterministic equivalent [10]. We first transformed the CP model from Sec. 4.2 into a stochastic formulation, which we present in Sec. 5.1. Since this formulation had a very poor performance, we re-formulated the model into a MIP-style stochastic model that we present in Sec. 5.2.

5.1 CP stochastic Model

We show excerpts (due to space) of the stochastic CP model in Fig. 3. We start with the parameter declaration (line 2-15), where we first declare the scenarios (line 2) and their weights (line 3). Then we divide the time slots by stage: the ones belonging to stage 1 (line 4) and to stage 2 (line 5). We also divide other parameters by stages, such as the...
Fig. 3. Stochastic Two-stage CP formulation of the Household Load Scheduling Problem

 costs and bestTimeSlots. Similarly, we divide the problem variables by stages
(line 17-20). Since we use the deterministic equivalent, the second stage variables
are also indexed by scenario.

 The constraints are given in lines 22-38. They correspond to the constraints from
Model 4.2, with the difference that the cumulative constraint is decomposed for the
stages. In the objective (line 40), we minimize the expected cost over both stages.

5.2 MIP-based stochastic Model

We show excerpts (due to space) of the stochastic MIP-based model in Fig. 4. The pa-
rameters (line 2) are the very same as in the stochastic CP model. The decision variables,
however, differ: the 0-1 variables schedule (line 4) represent the household schedule
Fig. 4. Stochastic Two-stage MIP formulation of the Household Load Scheduling Problem

```plaintext
% ------ input parameters -------------------------------------- %
% same as stochastic CP model
array[Jobs, TimeSlots1] of var 0..1: schedule_stage1;
array[Jobs] of var 0..numOfTimeSlots1: actualStarts_stage1;
array[Scenarios, Jobs, TimeSlots2] of var 0..1: schedule_stage2;
array[Scenarios, Jobs] of var 0..costLimit2: actualCosts_stage2;
% --------- variables ------------------------------------------ %
array[Jobs, TimeSlots1] of var 0..1: schedule_stage1;
array[Jobs] of var 0..costLimit1: actualCosts_stage1;
array[Scenarios, Jobs, TimeSlots2] of var 0..1: schedule_stage2;
array[Scenarios, Jobs] of var 0..costLimit2: actualCosts_stage2;
% --------- constraints ----------------------------------------- %
constraint forall(sc in Scenarios) (forall(j in Jobs) (sum(t in TimeSlots1) (schedule_stage1[j,t] + sum(t in TimeSlots2) (schedule_stage2[sc,j,t]) == 1)));
constraint forall(j in Jobs) (actualCosts_stage1[j] = sum(t in TimeSlots1) (schedule_stage1[j,t] * costs1[j,t]));
constraint forall(j in Jobs) (actualStarts_stage1[j] = sum(t in TimeSlots1) (schedule_stage1[j,t] * bestTimesSlots1[j,t]));
constraint forall(sc in Scenarios) (forall(j in Jobs) (actualCosts_stage2[sc,j] = sum(t in TimeSlots2) (schedule_stage2[sc,j,t] * costs2[sc,j,t])));
constraint forall(sc in Scenarios) (forall(j in Jobs) (actualStarts_stage2[sc,j] = sum(t in TimeSlots2) (schedule_stage2[sc,j,t] * bestTimesSlots2[sc,j,t])));
% --------- objective ---------------------------------------- %
solve minimize sum ( sc in Scenarios ) ( % expected cost
  weights[sc] * sum ( j in Jobs ) ( actualCosts_stage1[j] ) )
+ weights[sc] * sum ( j in Jobs ) ( actualCosts_stage2[sc,j] ) );
```

with respect to the cheapest time slots. This means that schedule_stage1[i,j] is `1`, if job \( i \) is scheduled to start at the \( j \)th cheapest time slot, and `0` otherwise. The reason for this is to embed the variable ordering (by cheapest timeslot) into schedule, since they are the search variables. In line 5, the variables actualStart represent the actual start times of each job, where actualStart[j] is the time slot at which job \( j \) is scheduled. Third, the variables actualCosts (line 6) contain the cost for each job with respect to when they are scheduled. All variables are divided by stage.

The constraints are given in line 11-34, where we first state that each job has to be scheduled once, either in the first or the second stage (line 11). Second, we link the actual costs with the schedule for the first stage (line 14) and the second stage (line 20), and set the actual start times for the first stage (line 17) and the second stage (line 24). Finally, we need to split the cumulative constraint into its decomposition for the first stage (line 28) and the second stage (line 31). Finally, we state the objective in line 36, where we minimize over the expected costs of both stages.
6 Related Work

A number of research projects have been conducted on automated demand response techniques [5, 11]. Some of them study the scheduling problem of a single household [6, 12] which does not guarantee to reduce the overall electricity cost in the network. Yu et al. [12] discuss a method that addresses the scheduling problem in a single house under the market with real time pricing. It considers the customer satisfaction by calculating the delay of each electric appliance as part of the cost. It is more suitable for a community in which customers have different electric consumption patterns.

Other approaches [2–4, 7] do not include a feedback loop from customers to service providers, which means that communications between them are one-way, and consumers cannot bargain with service providers in real time.

The method presented by van den Briel et al. [2] predicts the shiftable loads in a day ahead, then calculates an ideal load curve that the energy generators wish to supply. The smart meter in each household stores the ideal load curve, and derives a probability distribution from it. Whenever a customer submits a request to run an electric appliance in a specific time window, the smart meter schedules the electric appliance at a random time within the specified time window, following the probability distribution. This method simplifies the communication between service providers and consumers, and minimizes the data flow in the network. This model best suits customers who do not have strict requirements on the starting time of each job.

Chen et al. [4] use Stackelberg’s competition model to calculate an optimal solution for each household. Each household is equipped with an energy schedule controller that submits an request to the service provider when the customer wants to schedule an electric appliance. Then the service provider and the energy schedule controller form a Stackelberg game to find the best time to schedule the electric appliance. The process keeps repeating whenever a new request is submitted. This method makes good use of the two-way communication network, however, customers have limited control on when the electric appliance will be scheduled.

Mohsenian-Rad et al. [7] introduce a method that requires coordination between households to reach a global optimal solution. The households communicate with each other by broadcasting their electric appliances’ schedule, and updating the electric consumption schedules when they receive the information from other houses. This process is repeated until a global optimal solution is found (and always guarantees a global optimum). The objective of this work is closer to our research, though it best suits customers who are willing to share their electric consumption data with the others.

7 Preliminary Experimental Results

The experiments were conducted on a Macbook Pro (OS X Version 10.9.4) with a 3GHz Intel Core i7 processor using 8GB 1600 MHz DDR3 RAM and a Flash storage. The data preprocessor was implemented in Python 2.7.6, using the NumPy 1.8.1 library. The deterministic models and the stochastic model were implemented in MiniZinc 2.0. The CP model was solved using Gecode 4.2.1; the stochastic model was solved by the G12-MIP solver.
7.1 Experiment data

For the deterministic model, the input data consists of the jobs details for each household, a dynamic pricing table, and an initial price for each time slot in a day. For the stochastic model, the input data consists of the jobs details for one household, an initial price for each time slot in stage one, and an initial price for each time slot in stage two for each scenario.

We generate test data sets from a job pool of 41 jobs that are based on real world data. For instance, the electric consumption per hour of each job comes from real data published at the AusGrid website. Jobs were divided into 3 categories: jobs with limited flexibility (jobs whose care factor is 1), jobs with some flexibility (jobs whose care factor is 0.5), and jobs with full flexibility (jobs whose care factor is 0). Each experiment selects jobs from the pool to generate an instance.

The initial prices were generated from the historic aggregated electricity price for Victoria, Australia region published at AEMO website. They are illustrated in Figure 5, where scenario 1 corresponds to the initial electricity price for the deterministic model.

The price lookup table is derived from a supply curve based on a year’s historical data from the Australian National Energy Market (NEM) where we used the average relationship between wholesale market spot price. This is the price that changes every half hour, and is paid by all electricity retailers for all the bulk purchases they make on behalf of their retail customers. It is also the price that all power station operators earn for power generated in the same half hour. By fitting a simple curve to the historical data we were able to derive a representative supply-demand function. This function was then transformed by rescaling the demand to conform to the load levels in our model and the prices were rescaled to conform to average end-user retail prices.

This is a simple model and therefore is unavoidably distortive. In particular, it does not correctly reflect the way that the contribution of transmission and distribution network costs is incorporated into retail prices. In essence we assume that these network costs scale linearly as the generation costs increase. In reality these costs are fixed on an annual basis and amortised over expected energy demand using complex network

---


---

![Fig. 5](#) Initial prices for the stochastic model; scenario 1 for the deterministic model.
ModRef 2014: The Thirteenth International Workshop on Constraint Modelling and Reformulation

Fig. 6. Run times of the deterministic model for category 1

Table 3. Solutions for all data sets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Model Type</th>
<th>Jobs</th>
<th>Total cost [cent]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CP</td>
<td>1</td>
<td>119</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>103</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>85</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>1391.22</td>
</tr>
<tr>
<td>2</td>
<td>CP</td>
<td>1</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>95</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1400.92</td>
</tr>
<tr>
<td>3</td>
<td>CP</td>
<td>1</td>
<td>117</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>139</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td></td>
<td>7</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td>76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9</td>
<td>101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
<td>1408.39</td>
</tr>
</tbody>
</table>

tariff design models. However our model only needs to be a stylistic representation of the much more complex market and therefore is sufficient for our purposes.

7.2 Experiments for the deterministic model on a single household

We first tested the scalability of CP model with 12 data sets. Each data set uses 144 time slots, where each time slot has 10 minutes duration. The data sets have 10, 15, 20, 30, 40 and 50 jobs and two categories: limited flexibility (the jobs’ care factor are 1) and full flexibility (the jobs’ care factor is 0). The results of scalability experiments are illustrated in Fig. 6. The scalability is measured by run time in milliseconds (ms). The results showed that the model is scalable. Even if a household has 50 jobs to schedule, the model could find an optimal solution in 110 ms.

We then tested the solutions computed by the CP model with 3 sets of data. The data set 1 had jobs with full flexibility; the data set 2 had jobs with some flexibility; the data set 3 had jobs with limited flexibility. The experiment results are illustrated in Table 3.

7.3 Experiments for the deterministic model on multiple households

We first tested the scalability of the Iterative Optimisation Model. 10 sets of data were generated for this experiment. Each data set had 10, 20, 30, 40, 50, 60, 70, 80, 90, and 100 houses respectively. Each house randomly selected 5 to 15 jobs from the job pool. 10 sets of data were tested on a 10, 20, and 30 iteration base. The results are illustrated in Figure 7. The scalability is measured by run time in seconds (s) on a logarithmic scale. The results showed that this model needs improvement on scalability. It at least
We are interested in finding an optimal schedule for more than 1000 households.

We then tested the solutions generated by the Iterative Optimisation Model for 90 households. A data set consisting of 90 houses is generated for this experiment. Each house randomly selected 5 to 15 jobs from the job pool. The total cost, bill and inconvenience of 90 houses in each iteration were calculated. The results illustrated in Figure 8 showed that the total costs, total bills and total inconvenience oscillate at each iteration. They oscillate because at each iteration, all households always see the same price. They try to schedule the jobs at the same cheapest time slots, which make those time slots become expensive at the next iteration, and cheap again at the iteration after the next.

### 7.4 Experiments for the stochastic models

In this experiment, we tested the performances and the solutions of the stochastic models using the prices illustrated in Figure 5.
The CP stochastic model could not reach a solution within one minute time limit even on the smallest instances. The MIP stochastic model, however, could find a solution. The runtimes of the MIP stochastic model are illustrated in Figure 9 in seconds (s) on a logarithmic scale. We illustrate a solution of the stochastic model in Figure 10 that displays the schedules of 3 scenarios over the two stages. The timeslots 0-71 represent the first stage (where naturally, the consumption is the same for all scenarios), the timeslots 72-142 represent the second stage, where the consumption differs in each scenario due to their different schedules. We do not yet iteratively improve the solution, so we still experience peaks in consumption.

8 Conclusions

This paper contributes to the fast growing research on residential load management under real-time pricing. This work aims to minimize the overall electricity cost at a community level as well as inconvenience caused by shifting electric appliances, by providing a feedback loop from consumers to service providers, enabling them to bargain with service providers.

At the current stage, we are working towards finding a global optimal solution for customers in an acceptable time frame. In next steps, we will investigate on how to eliminate oscillations produced from the Iterative Optimisation Model. We will further increase the scale of the problem to 100 and 1000 houses, and study the performance of the Iterative Optimisation Model on bigger problems.
References

2. van den Briel, M., Scott, P., Thiebaux, S.: Randomized load control: A simple distributed approach for scheduling smart appliances. In: International Joint Conference on Artificial Intelligence (IJCAI). Beijing, China (August 2013)
Rethinking the Quest for Declarativity

Kathryn Francis

National ICT Australia, Victoria Research Laboratory,
The University of Melbourne, Victoria 3010, Australia

kathryn.francis@nicta.com.au

Abstract. It is generally assumed that in order to free users from the burden of deciding how problems should be solved, it is necessary to provide a more declarative problem specification language. The purpose of this paper is to challenge that assumption, arguing that instead we should treat existing languages more declaratively. Specifically, I argue that the ideal interface to CP would use a (declarative) problem definition written using native code in an existing general purpose programming language.

1 Introduction

The ultimate goal for CP, the holy grail, is that the user simply states the problem and then the computer solves it [3, 4]. This description leaves implicit an important requirement; ‘stating’ the problem must be easy. Otherwise we are at risk of creating very powerful tools that are never used.

When considering how to make problem specification easy, we need to keep in mind our target users. There is an inherent assumption in many of our tools that the problem will be defined and solved more or less in isolation. This is not realistic for widespread use. In most cases constraint solving will need to be integrated into some sort of larger program. This means both that targeting mathematicians is not especially useful, and that aiming for a problem definition interface simple enough to be used by anyone (e.g. a natural language interface) is unnecessarily ambitious. A more practical choice is to aim for an interface which is intuitive and convenient for application programmers.

The usual suggestion for how problems should be defined involves some sort of abstract, high level problem definition language. Unfortunately this does not make life easy for programmers. It would be much more convenient to allow programmers to define the problem within their own languages, ideally without introducing foreign modelling concepts such as decision variables and constraints. I call this a native interface to CP. The remainder of the paper explains this idea in detail, including illustrative examples in Haskell and Java, before discussing how we might work towards this goal.

2 Native CP interface

Combinatorial problems are difficult to solve because they consist of decisions which cannot be made sequentially. It is not possible to finalise one decision
without considering the options available for the others. This is the difficulty that constraint solving technology should eliminate for programmers.

Rather than insisting that the programmer construct a representation of the problem and send that off to a solver, imagine if they could simply ask some sort of magical oracle to make individual decisions, and use the answers to build a solution. Clearly this would be much easier. Unfortunately it is also impossible. However, it is possible to create an interface which appears to the programmer to be very close to this.

Code which combines individual decisions made by an oracle into a solution defines a pool of candidate solutions. If we also obtain code to evaluate a candidate solution (which should also be straightforward to write), this is enough to define the CSP.

A native interface to CP allows the programmer to define problems in this way and receive solutions exactly as if their own building code had been given a perfect oracle.

Essentially the programmer says:

1. This code builds a candidate solution from individual decisions.
2. This code checks if a candidate solution is feasible.
3. This code calculates the score for a solution. (optional)
4. Build the (best) solution.

Note that the actual computation happens in the last step, and the code does not specify how that step should be achieved. In this sense the interface is declarative even though it may use procedural code.

This final step could be achieved by repeatedly executing the candidate building code with a randomised oracle, each time testing the result for feasibility and score, until one is satisfied that no better solution is likely. This possible implementation allows the programmer to understand what this step means, but obviously it should not be used in practice. Instead we should treat the provided code (1-3) not as code to be executed, but as a declarative specification of the problem to be solved. The library should automatically convert that specification into a standard constraint model, solve it, and then convert the result back into the required form.

3 Show me the code!

In this section examples from two very different programming languages (Haskell and Java) are used to explain in more detail what a native interface to CP might look like and how it would be used. First the interface for each language is defined, and then we examine complete programs solving two example problems. The example problems are taken from the First International Lightning Model and Solve competition, held at CP 2013 [9].
3.1 Haskell

A Haskell version of a native interface to CP is shown in Figure 1. In Haskell we have the advantage that functions are first class. So the three pieces of code provided by the programmer (candidate-building, checking and scoring) can simply be functions which will be passed to the ‘find the best solution’ function (the interface provides three of these: satisfy, minimise and maximise).

The candidate-building function returns type DeciderState s, where s is the type of a candidate solution. It should make one or more calls to the provided decision functions (chooseBool, chooseInt and chooseOne), combining the results to build a candidate. Since DeciderState is a State this can be done using familiar do notation (as will be shown in the examples). We are using the State monad (rather than having the function simply return type s) to account for the fact that a value of type s will be just one from the pool of candidate solutions. The state within DeciderState is understood to determine which values are returned by chooseBool, chooseInt and chooseOne, and therefore which candidate is produced.

The checking function takes the candidate type s and returns a Bool which is true if the candidate is acceptable, while the scoring function takes an s and returns a value of some orderable type o.

module CP where
import Control.Monad.State

type DeciderState = State Int

satisfy :: DeciderState a -> (s -> bool) -> s
minimise :: Ord o => DeciderState s -> (s -> bool) -> (s -> o) -> s
maximise :: Ord o => DeciderState s -> (s -> bool) -> (s -> o) -> s

-- make a yes/no decision
chooseBool :: DeciderState Bool
-- choose a number between the first arg and the second
chooseInt :: Int -> Int -> DeciderState Int
-- choose one item from the given list
chooseOne :: [a] -> DeciderState a

Fig. 1: Native Haskell interface to CP

This interface is very simple and easy to understand. There is only one type and very few functions to become familiar with, all of which have a very clear meaning. The type signatures of the main functions provide obvious cues for the programmer about the code they need to write, and that code does not need to use any different syntax or special types (DeciderState is a provided type, but it is really just a State).
3.2 Java

We now consider how the same thing might be achieved in Java. Unfortunately in Java we cannot pass functions as arguments, so instead we define an interface for the programmer to implement.

A class implementing CSP must contain a candidate-building method build, and a checking method check. The build method is passed a Decider object, which provides the same decision functions as in the Haskell version (chooseBool, chooseInt and chooseOne). Instead of returning a candidate it should change the state of the CSP object itself to represent the candidate. The check method will then use this state to determine if the candidate is acceptable. For optimisation problems the programmer must implement the CSPopt interface which extends CSP with another method to calculate a score (Comparable is a standard Java interface for classes having a natural ordering).

This time the ‘find the best solution’ methods belong to a library class called Solver. The buildMinimal method (for example) can be understood to call the build method of the provided CSP object using a Decider which makes optimal decisions (those producing a state for which calling check will return true and calling score will return the smallest possible value).

```java
interface CSP {
    void build(Decider d);
    boolean check();
}

class Decider {
    boolean chooseBool();
    int chooseInt(int min, int max);
    T chooseOne(Set<T> options);
}

interface CSPopt extends CSP {
    Comparable score();
}

class Solver {
    void buildSatisfactory(CSP problem);
    void buildMinimal(CSPopt problem);
    void buildMaximal(CSPopt problem);
}
```

Fig. 2: Native Java interface to CP

Once again there are very few library classes and methods to become familiar with, and the CSP/CSPopt interface prompts the programmer to write the required code. The decision making code can be more straightforward than the Haskell version because Java has no qualms about methods returning different values each time they are called. However the inability to directly pass functions as arguments makes the semantics of buildSatisfactory/buildMinimal/buildMaximal less clear than that of their Haskell counterparts.

3.3 M-Queens

Our first example problem is M-Queens, which is an optimisation extension of N-Queens. As with N-Queens, the queens placed on the board must not be able to attack each other. The difference is that rather than aiming to place n queens on an n-by-n board, the goal is to place as few as possible while still covering every square on the board (making it impossible to add another queen).
Haskell implementation  Figure 3 shows a complete Haskell program solving M-Queens using a native interface to CP. The main function reads the parameter $n$ from standard in, calls $\text{mqueens}$ to find an optimal placement of queens for an $n$-by-$n$ board, and then prints the result to standard out. In the competition from which this example is taken, the required output format for solutions was a list (using the same syntax as Haskell lists) giving the column for the queen in each row, or 0 if no queen is placed in that row. Our program uses this same list representation for solutions to make producing the output convenient (it is sufficient to call the built-in function $\text{print}$ as shown).

main = do
  n <- readLn
  let chosenCols = mqueens n
  print chosenCols

mqueens :: \textbf{Int} \rightarrow [\textbf{Int}]
mqueens n = minimise (choosequeens n) (validqueens n) numqueens

choosequeens :: \textbf{Int} \rightarrow \textbf{DeciderState} [\textbf{Int}]
choosequeens n =
  replicateM n (chooseInt 0 n)

validqueens :: \textbf{Int} \rightarrow [\textbf{Int}] \rightarrow \textbf{Bool}
validqueens n colchoices =
  let allcoords = [(i,j) | i <- [1..n], j <- [1..n]]
      pairs = zip [1..n] colchoices
      chosencoords = filter (\(r,c) \rightarrow c>0) pairs
    in all (covered chosencoords) allcoords &&
        noclash chosencoords

covered :: [(\textbf{Int},\textbf{Int})] \rightarrow (\textbf{Int},\textbf{Int}) \rightarrow \textbf{Bool}
covered chosenpositions pos =
  any (covers pos) chosenpositions

noclash :: [(\textbf{Int},\textbf{Int})] \rightarrow \textbf{Bool}
noclash coords =
  let pairs = [(c1,c2) | c1 <- coords, c2 <- coords, c1/=c2]
      covers' (a,b) = covers a b
    in not (any covers’ pairs)

covers :: (\textbf{Int},\textbf{Int}) \rightarrow (\textbf{Int},\textbf{Int}) \rightarrow \textbf{Bool}
covers (x,y) (i,j) =
  (x == i) || (y == j) || ((x-y) == (i-j)) || ((x+y) == (i+j))

numqueens :: \textbf{Int} \rightarrow \textbf{Int}
umqueens = length . filter (>0)

Fig. 3: Haskell program solving the M-Queens problem.
The \texttt{mqueens} function simply calls \texttt{minimise} passing the three required functions as arguments.

- The candidate-building function \texttt{choosequeens} takes the parameter $n$ and uses \texttt{replicateM} (from State) and \texttt{chooseInt} (from CP) to choose $n$ numbers between 0 and $n$, returning them in a list. The number at index $i$ represents the column for the queen in row $i$ (with 0 meaning no queen in row $i$).
- The checking function \texttt{validqueens} takes the parameter $n$ and a list representing a candidate solution and checks that all board positions are covered by the chosen coordinates (using \texttt{all covered}), and no pair of chosen coordinates clash (using \texttt{noclash}). The \texttt{covers} function defines what it means for one coordinate to cover (or clash with) another; the two coordinates must have the same row or column, or the same sum or difference between row and column (as then they are on the same diagonal).
- The scoring function \texttt{numqueens} counts the number of positive numbers in its input list. This corresponds to the number of queens placed on the board.

Note that almost all of this code uses standard types and functions and can be understood independently from the CP module. In fact the CP module is only used on three lines (highlighted with arrows); there is one use of \texttt{minimize}, one use of \texttt{chooseInt}, and the \texttt{choosequeens} function returns a \texttt{DeciderState}. Minimal use of CP-specific code makes this program easy to understand (and to write) for someone familiar with Haskell. Furthermore, the checking functions \texttt{validqueens}, \texttt{covered}, \texttt{noclash} and \texttt{covers} would likely be needed for testing purposes regardless of the solving method.

\textbf{Java implementation} We now consider a Java version of the same program.

In order to demonstrate that we don’t have to build the solution in a simple format such as a list of integers, this version instead uses a \texttt{Map} from row number to column number. Only rows in which a queen is placed should be included. \texttt{Map} is a standard Java library class, but it would also be possible (in both the Java and Haskell interface) to build a candidate solution using user-defined classes. For example, another valid representation for the selected queen positions would be a \texttt{Set} of \texttt{Coordinate} objects, each \texttt{Coordinate} having a row and a column.

The majority of the code is shown in Figure 4 (for space reasons \texttt{print} is shown separately in figure 5). The \texttt{main} method retrieves the parameter $n$, constructs an \texttt{MQueens} object for this size board, then calls \texttt{buildMinimal} to set up a minimal placement before printing the result using the \texttt{print} method of \texttt{MQueens}.

To be accepted as an argument to \texttt{buildMinimal}, the \texttt{MQueens} class needs to implement the \texttt{CSPopt} interface by defining \texttt{build}, \texttt{check} and \texttt{score}.

- The \texttt{build} method iterates through the rows deciding first whether or not a queen should be placed in this row, and then if so which column should be used. The chosen column for each row is recorded in the \texttt{colForRow} map.
- The \texttt{check} method checks that every board coordinate is covered by one of the chosen queen positions (stored in \texttt{colForRow}), and that for every pair of (different) positions in this map, the queens cannot attack each other.
class QueensMain {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        MQqueens mq = new MQqueens(n);
        Solver.buildMinimal(mq);
        mq.print(System.out);
    }
}

class MQqueens implements CSPopt {
    int n;
    Map<Integer,Integer> colForRow;
    MQqueens(int n) {
        this.n = n;
        colForRow = new HashMap<Integer,Integer>();
    }
    void build(Decider d) {
        for(int row = 1; row <= n; row++)
            if(d.chooseBool()) // put a queen in this row?
                colForRow.put(row, d.chooseInt(1, n));
    }
    boolean check() {
        for(int row = 1; row <= n; row++)
            for(int col = 1; col <= n; col++)
                if(!coveredByChosen(row,col))
                    return false; // this position is not covered
        for(int r1 : colForRow.keySet())
            for(int r2 : colForRow.keySet())
                if(r1 != r2 && covers(r1, colForRow.get(r1), r2, colForRow.get(r2)))
                    return false; // this pair of queens can attack each other
        return true;
    }
    boolean coveredByChosen(int row, int col) {
        for(int qrow : colForRow.keySet())
            if(covers(qrow,colForRow.get(qrow), row, col))
                return true;
        return false;
    }
    boolean covers(int r1, int c1, int r2, int c2) {
        return (r1==r2 || c1==c2 || (r1-c1 == r2-c2) || (r1+c1==r2+c2));
    }
    Integer score() {
        return colForRow.size();
    }
}

Fig. 4: Java program solving the M-Queens problem (excluding print method).
The score method simply returns the size of the map (the number of keys), which gives the number of queens placed on the board.

The print method of MQueens (Figure 5) prints the solution in the required format. Recall that buildMinimal can be understood to call build with an optimal Decider. So when the print method is called (in main immediately after the call to buildMinimal) the colForRow map will contain optimal coordinates.

Once again almost all of the code is independent of the CP library (arrows indicate lines using library methods or classes). Comparing the Haskell and Java solutions, the Java code is unsurprisingly longer (Java is famously verbose), and the Haskell version is more reminiscent of a model written in a dedicated constraint language (with its use of not, any and all), but they are actually very similar. Which version is easier to understand would almost certainly depend on the familiarity of the reader with the two different languages. For this reason I believe we should create native interfaces for a variety of languages rather than attempting to choose a single language which is most suited to the task.

```java
void print(PrintStream ps) {
    ps.print("[");
    for(int row = 1; row <= n; row++) {
        if(row > 1)
            ps.print(",");
        if(colForRow.containsKey(row))
            ps.print(colForRow.get(row));
        else ps.print(0);
    }
    ps.println("]");
}
```

Fig. 5: Print method extracted from MQueens class in Figure 4.

### 3.4 Doubleclock

Our second example is based on the solitary card game double clock patience. Assuming a deck with \(2k\) suits and \(n\) different card numbers, the rules of the game are as follows. The cards are dealt into \(2n\) piles (of \(k\) cards), and then the top-most card in the last pile is revealed and moved to the initially empty ‘discard’ pile. The player then makes a sequence of moves, each time discarding another card, until all piles are empty. In each move the number (ignoring the suit) of the last discard determines which two piles the next card might be taken from. If the number is \(j\) then the player can choose pile \(2j - 1\) or pile \(2j\). If one of these piles is empty then the other pile must be chosen. If both are empty then the next card is taken from the last non-empty pile, incurring a restart penalty.

Our objective is to find a sequence of moves minimising the number of restarts. The input data consists of the two size parameters \(n\) and \(k\) followed by a permutation of \(1..2nk\) giving the initial layout of the cards. The first \(k\) integers
give the first pile from bottom to top, the next \( k \) integers the second pile, and so on, where integer \( j \) represents a card with number \((j - 1) \mod n + 1\). The required output is another list of integers giving the discard order.

The Haskell and Java implementations are largely the same. To make input and output straightforward we use the given integer representation for cards. Both versions essentially simulate the game, counting the number of restarts and recording the discards. At each move only allowable choices are made, so all candidate solutions will be valid.

The Haskell program (Figure 8) passes the input data straight through to the `doubleclock` function, where it is converted into the two parameters \( n \) and \( k \) and a list of list of `Int` for the piles. Then `minimise` is used to find an optimal solution. The candidate-building function is `play`, with an initial state constructed by discarding the top card from the last pile (\( 2n \)). A candidate solution produced by `play` is a pair composed of the discard list and number of restarts. We want to minimise the number of restarts, so the scoring function is `snd` (which returns the second element from a pair). The checking function is simply true.

The `play` function simulates the game. If all piles are empty then the game is finished and the current discard pile and number of restarts are returned. Otherwise we first find the available choices for the next discard, and then either perform a restart (if neither pile contains cards) or use `chooseOne` to decide on a pile and remove a card from there, before continuing with the game.

In the Java program (Figures 6 and 7) the game state is represented using a list of stacks (each stack is a pile). The state is initialised in the `DoubleClock` constructor using the input data passed in from `main`. The `build` method makes the compulsory first move and then continues making allowed moves until the game is finished (all piles are empty). When choosing a pile, if only one option is available then that pile is selected. If no options are available then the number of restarts is incremented and the last non-empty pile is selected. Otherwise `chooseBool` is used to decide which pile to select. As with the Haskell version, the `check`, `score` and `print` methods are trivial.

Once again both versions contain very little CP-specific code (all affected lines are highlighted with arrows). A competent programmer should find it straightforward to produce similar programs. The Java implementation feels more natural in this case; as a procedural language it is well suited to simulating the game playing process. This demonstrates that the choice of specification language may depend not only on the programmer’s experience but also on the nature of the problem, highlighting the need to support multiple languages.

```java
public class ClockMain {
    public static void main(String[] args) {
        DoubleClock dc = new DoubleClock(args);
        Solver.buildMinimal(dc);
        dc.print(System.out);
    }
}
```

Fig. 6: Main method for Java doubleclock program.
public class DoubleClock implements CSPopt {

    List<Stack<Integer>> piles = new ArrayList<Stack<Integer>>();
    Stack<Integer> discardPile = new Stack<Integer>();
    int restarts = 0;
    int n;

    DoubleClock(String[] args) {
        n = Integer.parseInt(args[0]);
        int k = Integer.parseInt(args[1]);
        Stack<Integer> pile = new Stack<Integer>();
        piles.add(pile);
        for (int i = 2; i < args.length; i++) {
            if (pile.size() == k) {
                pile = new Stack<Integer>();
                piles.add(pile);
            }
            pile.push(Integer.parseInt(args[i]));
        }
    }

    public void build(Decider d) {
        discardPile.push(piles.get(piles.size() - 1).pop()); // compulsory first move
        while (!gameFinished()) {
            Stack<Integer> chosenpile = choosePile(d, discardPile.peek());
            discardPile.push(chosenpile.pop());
        }
    }

    Stack<Integer> choosePile(Decider d, int lastcard) {
        int cardnum = ((lastcard-1) % n) + 1;
        Stack<Integer> pile1 = piles.get(2*cardnum-1);
        Stack<Integer> pile2 = piles.get(2*cardnum-2); // 0-based index
        if (pile1.isEmpty() && !pile2.isEmpty()) return pile2;
        if (pile2.isEmpty() && !pile1.isEmpty()) return pile1;
        if (pile1.isEmpty() && pile2.isEmpty()) {
            restarts++;
            for (int i = piles.size()-1; i >=0; i--) // find last nonempty pile
                if (piles.get(i).isEmpty()) return piles.get(i);
            return d.chooseBool() ? pile1 : pile2;
        }
    }

    boolean gameFinished() {
        for (Stack<Integer> p : piles)
            if (p.isEmpty()) return false;
        return true;
    }

    public boolean check() { return true; }
    public Integer score() { return restarts; }
    void print(PrintStream ps) { ps.print(discardPile); }
}

Fig. 7: Java CSPopt implementation for the doubleclock problem.


type Card = Int

data Gamestate = Gamestate Int [[Card]] [Card] Int

main = do
  i <- readLn
  inputdata <- readFile
  ""doubleclock"++i++".txt"
  let (discards, restarts) = doubleclock inputdata
  print discards

doubleclock :: String -> ([Card], Int)

doubleclock inputdata =
  let [n:k:cards] = map read (words inputdata)
      piles = chunksOf k cards
      initialstate = makemove (2*n) (Gamestate n piles [] 0)
  in minimise (play initialstate) (const True) snd

play :: Gamestate -> DeciderState ([Card], Int)

play gs@(Gamestate n piles discardpile restarts) =
  if all null piles
    then return (discardpile, restarts)
  else let j = cardnumber (head discardpile) n
        pileoptions = filter (notEmpty piles) [2*j, 2*j-1]
        in
          if null pileoptions
            then play (restart gs)
          else do
                chosenPile <- chooseOne pileoptions
                play (makemove chosenPile gs)

Fig. 8: Haskell program solving the doubleclock problem.
4 Advantages of a native interface

If our target users are programmers then there are significant advantages to allowing problems to be defined using existing general purpose programming languages. Programmers are already familiar with and practiced at using these languages, mature editors/IDEs exist, and integration with a wider program (which is essential for widespread use of CP) is much easier.

For many programming languages there already exist one or more CP libraries designed to facilitate the integration of constraint solving into a wider program. The following is a list of advantages of a native interface over one of these libraries. Some of these advantages also apply when comparing to dedicated problem definition languages.

4.1 No CP-specific modelling

While using a standard CP library is certainly more convenient than using a separate language, the programmer still has to learn CP-specific modelling skills. With a native interface that is not the case. The programmer does not have to learn how to write a CP model, coming to grips with decision variables and constraints. Unlike using a standard library it is not necessary to understand anything about how constraint solvers work.

Obviously the programmer still needs to choose a representation for solutions. This is modelling, but since the choice is based on the requirements of the application rather than the requirements of the solver, it is no different from the modelling already required to write the rest of the application.

4.2 Less documentation

CP libraries typically require writing code which makes heavy use of library types representing decision variables and constraints. For example, in the n-queens sample program provided with Choco [7], 16 of the 26 lines of code (excluding imports, annotations, comments, bracket-only lines and whitespace) use a library class or method. This is inconvenient for programmers who have to look up how to use each library-provided type or function. Apart from being a hassle it can also be very daunting. The Gecode [8] tutorial is more than 500 pages long. Even Numberjack [6], which is aimed more at beginners, has 36 classes with almost 70 documented members. In contrast, a native interface introduces a very small number of types and methods and should require very little documentation. This means programmers can quickly read the entire specification and not wonder what else is there that they are not aware of.

4.3 Readable code

The requirement to use special types representing decision variables also often necessitates special syntax or awkward code which obfuscates the problem definition, especially when dealing with mathematical expressions. While this can
sometimes be avoided using operator overloading, with a native interface there is no need to introduce special types in the first place, so the code is readable even in languages such as Java which do not allow operator overloading.

4.4 Reusable code

When using a native interface, code forming part of the problem definition can be re-used, for example to evaluate current practice or user-provided solutions. The data structures used to represent the input data and solution can also be re-used. For example, there is no need to invent an integer representation for cards if the rest of the card playing application uses a structured type. The same structured type can be used directly in the candidate-building, checking and scoring code. Any required conversion between solver and host language types is performed automatically.

4.5 Easier debugging and testing

Native interfaces permit natural, straightforward debugging. Incorrect solver results indicate a bug in the building or checking code. An invalid solution gives a failing test case for the checking code (it returns true for the given candidate when it should return false), while a missing solution indicates either that the building code cannot build that solution, or that the checking code erroneously returns false. Once one of these problems has been discovered, it can be corrected with no further reference to a constraint solver, using standard debugging techniques and tools (e.g. interactive debuggers or logging). Narrowing down which constraint is erroneous in a standard model is often much harder (in part because fewer tools are available).

Similarly, the code defining the constraint problem can be tested in the same manner as the rest of the program, without using a constraint solver. Individual methods/classes/functions used in the definition may have their own unit tests. Even the decision-making code can be tested independently, using a Decider/DeciderState implementation making random or predetermined decisions.

4.6 IDEs and compile-time checking

A native interface allows advanced features of IDEs such as text-prediction, automatic re-factoring, and live compilation to be directly applicable to the code defining the model. Standard static checking performed by the compiler can help prevent errors in the problem definition, with no need for error-prone string-based references to variables.

5 Implementing a native interface

It might seem as though the described interfaces are impossible to implement. This is certainly not the case, in fact in previous work [1] I have implemented
a proof-of-concept Java interface similar to the one described here. In order to understand how this is possible we need to come back to the idea of treating code written in existing languages more declaratively.

Our basic requirement is for the three pieces of code provided by the programmer (candidate-building, checking and scoring) to be translated into a constraint model. This can be achieved using symbolic execution techniques. All uncertainty in the computation stems from the decision making calls. Each of these requires the creation of one or more decision variables to represent the returned value. The rest of the code can be translated into constraints linking these values to the computed checking result and score. Note that inputs already provided to functions passed to \texttt{minimise} (for example), or the state of the program when \texttt{buildMinimal} is called, can be considered parameters as they will be known at solving time (when \texttt{minimise/buildMinimal} is actually called).

Given a model produced in this fashion, a standard constraint solver can be used to find satisfactory/optimal results for the decision-making calls. These can then be used to execute the candidate-building code, producing the result expected by the programmer. The practical details of when the translation should be performed and exactly how the correct final state should be produced will depend on the language. For an example of a possible approach for Java see [1].

While a basic translation is reasonably straightforward, the resulting models are likely to be very inefficient. The following list outlines some ideas for how a more effective translation might be achieved (also see [2]).

– Functions/methods for commonly used types like lists and sets can be translated directly as atomic operations.
– Specialised translations can be used for common code patterns (e.g. computing a sum), possibly building on pattern-detection algorithms already used for editor suggestions.
– Compiler transformations designed to produce more efficient code might be adapted to produce more effective models. These two goals are related as they both benefit from simplicity.
– The structure of the code could be used to aid the automatic detection of global constraints. Looping, recursion, and the use of higher-order functions (e.g. all) suggest that a global constraint may be relevant.
– Other structure information present in the code may also be useful. For example, nested decision making as in the Java M-Queens program (Figure 4) may benefit from a particular search strategy or other solving technique.
– Given a correct model, we can apply general model reformulation techniques.

Obviously a translation able to compete with models produced by a human expert is a very ambitious goal. I am certainly not suggesting we abandon work on dedicated modelling languages or CP libraries. However, as a starting point we can aim for a translation sufficient for small/easy problems, or we can make concessions in our interface to allow greater control over the resulting model (e.g. the plugging in of specialised constraints and search strategy). Also, while chasing our ultimate goal we will almost certainly encounter ideas which could be transferred back to the mainstream approach. Two examples are given below.
5.1 Complex decisions

The decision functions included in the examples above are sufficient to represent any problem, but in practice the interface should include more complex decisions such as choosing a permutation of a list, or a partition of a set. One might also provide more specific decisions like scheduling a set of tasks so that none overlap. The benefit is that appropriate (global) constraints can be added automatically.

Global constraints are often used to tie back together complex decisions which have been decomposed. It would be preferable to allow the direct specification of the original decision, enabling automatic symmetry breaking and straightforward experimentation with different decompositions. This is the premise behind Essence [5], which allows decision variables to represent arbitrarily complex combinatorial objects. However, it should be possible to encapsulate more constraints within a complex decision, beyond the type of the combinatorial object.

It would also be interesting to see how this idea could be incorporated in a normal CP library. Ideally the actual decision variables would be hidden behind an abstract type for the complex decision, so that different decompositions could be compared without changing the rest of the code.

5.2 Separation of decisions and consequences

A native interface provides a clear distinction between core decisions (calls to the library-provided decision functions) and their consequences (values computed from the results). This information could be used to guide the search, or avoid useless propagation. It also prevents modelling errors where the consequences are not actually functionally defined by the decisions. It might be possible to transfer some of these benefits to mainstream modelling if we could introduce a distinction between core decisions and consequences.

6 Conclusion

The results of the 2013 International Lightning Model and Solve competition served as a reminder that our tools are too hard to use. The winners solved the problems by hand, suggesting that creating an effective model was more time consuming (or daunting) than solving the problems. In order for the holy grail (that users simply define the problem and then the computer solves it) to solve our usability problems, we need to ensure that it is not still easier to solve the problem by hand than to define it in a way the computer understands.

In this paper I argue that we should be aiming for a realisation of the holy grail where constraint problems are defined natively inside general purpose programming languages. The benefit of this approach is greatly enhanced ease of use for programmers, who represent ideal target users due to their ability to embed constraint solving in wider applications.

Acknowledgments NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.
References


Efficient SMT Encodings for the Petrobras Domain

Miquel Bofill, Joan Espasa, and Mateu Villaret
Departament d'Informàtica, Matemàtica Aplicada i Estadística
Universitat de Girona, Spain
{miquel.bofill, joan.espasa, mateu.villaret}@udg.edu

Abstract. Reformulation into SAT is one of the main approaches to deterministic planning. The main idea is to successively check the satisfiability of each of the propositional formulas in a sequence $\phi_0, \phi_1, \phi_2, \ldots$ where $\phi_i$ encodes the feasibility of a plan of length $i$. For real-life applications, resources such as time, distances, or money must often be considered. Therefore, in the context of planning with resources, a natural choice is to consider satisfiability modulo theories (SMT) instead of plain SAT. In this paper we consider the application of off-the-shelf SMT solvers to the Petrobras domain, an abstraction of a real-life problem of resource-efficient transportation of goods from ports to petroleum platforms. We consider different encodings into SMT and compare with state of the art alternative approaches.

1 Introduction and Preliminaries

The problem of planning, in its most basic form, consists in finding a sequence of actions that will allow to reach a goal state from a given initial state. Although initially considered a deduction problem, it was rapidly seen that it could be better addressed by looking at it as a satisfiability (model finding) problem [14]. Many (incomplete) heuristic methods can be found in the literature to efficiently deal with this problem, most of them oriented towards finding models. Exact methods were ruled out at the beginning due to their inefficiency. However, in [15] it was shown that modern off-the-shelf SAT solvers could be effectively used to solve planning problems. In the last years, the power of SAT technology has been leveraged to planning [17], making reduction into SAT state of the art for deterministic planning.

In the deterministic planning problem actions are formalized as pairs $(p, e)$, where $p$ and $e$ are sets of literals denoting the precondition and the effects of the action, respectively. A finite set of Boolean variables determines the state at each moment, and an action $(p, e)$ is executable in state $s$ if $s \models p$. The successor state $s'$ (which can be seen either as a set of literals or as a valuation function) is defined by $s' \models e$ and $s'(a) = s(a)$ for all variables $a$ not occurring in $e$. A solution to the planning problem consists in a sequence of actions that allow to reach a goal state from a given initial state.
To solve the planning problem by reduction into SAT, a sequence of formulas $\phi_0, \phi_1, \phi_2, \ldots$ is considered, where $\phi_i$ encodes the feasibility of a plan that allows to reach the goal state from the initial state in $i$ steps, i.e., by executing $i$ actions sequentially. The solving procedure proceeds by testing the satisfiability of $\phi_0, \phi_1, \phi_2$, and so on, until a satisfiable formula $\phi_n$ is found. Then, a plan of length $n$ can be extracted from the assignment satisfying $\phi_n$. The basic idea of the encoding [16,17] for $\phi_i$ is to introduce, for each state variable $a$ and time point $t \in \{0, \ldots, i-1\}$, the formula
\[ a_{t+1} \rightarrow (a_t \lor o_1^a \lor \cdots \lor o_n^a) \]
where $o_1^a, \ldots, o_n^a$ are all the actions (operators) that have $a$ as effect, for explaining the possible reasons of the truth of $a_{t+1}$, and
\[ \neg a_{t+1} \rightarrow (\neg a_t \lor \neg o_1^a \lor \cdots \lor \neg o_m^a) \]
where $\neg o_1^a \lor \cdots \lor \neg o_m^a$ are all the actions that have $\neg a$ as effect, for explaining the possible reasons of the falsity of $a_{t+1}$. Many-valued variables must be represented in terms of several Boolean ones. In this setting, binary constraints of the form $\neg a_{1,t} \lor \neg a_{2,t}$, stating that a many-valued variable $a$ can only have one of the two values 1 and 2 at time $t$, are redundant but often crucial for efficiency.

As the number of variables, and hence the search space, rapidly grows with the number of time steps considered, a key idea to improve the performance of SAT-based planners is to consider the possibility of executing several actions at the same time, i.e., the notion of parallel plans. Parallel plans increase the efficiency not only because they allow to reduce the time horizon, but also because it is unnecessary to consider all total orderings of the actions that are performed in parallel. However, parallel plans are not intended to represent true parallelism in time, and it is usually required that a sequential plan can be built from a parallel plan in polynomial time. Two main types of parallel plans are considered: $\forall$-step plans, and $\exists$-step plans. In $\forall$-step plans, any ordering of parallel actions must result in a valid sequential plan. In $\exists$-step plans, there must exist a total ordering of parallel actions resulting in a valid sequential plan. We refer the reader to [16,18] for further details.

Although a lot of work has been devoted to the encoding of plans in propositional logic, only a few works can be found in the literature on satisfiability based approaches to planning in domains that require numeric reasoning. This is probably due to the difficulty of efficiently handling at the same time numeric constraints and propositional formulas. However, the advances in satisfiability modulo theories (SMT) [1] in the last years make worth considering this alternative.

One of the first approaches to planning with resources, the one of LPSAT [20], can indeed be considered one of the precursors of SMT, as the basic ideas of SMT (Boolean abstraction, interaction of a SAT solver with a theory solver, etc.) were already present there. A comparison between SAT and SMT based encodings for planning in numeric domains can be found in [12]: In the SAT approach, the possible values of numeric state variables is approximated, by generating a
set of values $D_t(v)$ for every numeric variable $v$, so that every value that $v$ can have after $t$ time steps is contained in $D_t(v)$. These finite domains then serve as the basis for a fully Boolean encoding, where atoms represent numeric variables taking on particular values. The authors argue that the expressivity of the SMT language comes at the price of requiring much more complex solvers than SAT solvers, and their method is very efficient in domains with tightly constrained resources, where the number of distinct values that a numeric variable can take is small.

Other approaches, related to SMT to some amount as well, have been developed more recently. In [4], a set of encoding rules is defined for spatio-temporal planning, with SMT as target formalism. In [11] a modular framework, inspired in the work of SMT, is developed for planning with resources.

In this paper we consider the application of off-the-shelf SMT solvers to the Petrobras domain, an abstraction of a real-life problem of resource-efficient transportation of goods from ports to petroleum platforms, posed as a challenge at the International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS 2012).

The rest of the paper is structured as follows. In Section 2 the Petrobras problem is presented and modelled. In Section 3 different SMT encodings for this problem are considered. Section 4 is devoted to experimental evaluation. Conclusions are given in Section 5.

2 The Petrobras Problem

The Petrobras domain was posed as a challenge problem at the International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS 2012). This domain\(^1\) is an interesting real-life problem, that lies in the border between scheduling and planning.

Generically speaking, the problem is described as the need to transport various cargos of goods and tools from two ports to various platforms located in the ocean at various distances. The strips are divided in two parts: Rio de Janeiro and Santos. The basic elements and agents of the problem are: ports, platforms, waiting areas, cargo items and vessels. The actions that can be performed are:

- **Sail**: Navigates a ship from one location to another.
- **Dock**: Docks a vessel in a port or platform.
- **Undock**: Undocks a vessel in a port or platform.
- **Load**: Loads a cargo item into the ship.
- **Unload**: Unloads a cargo item from the ship to a platform or port.
- **Refuel**: Refuels a ship at a refueling location (a port or any specified platform).

Although the proposal gives various optimization criteria, we only consider the satisfiability of the problem, minimizing the plan length.

2.1 Modelling

We model the problem with the Planning Domain Definition Language (PDDL). PDDL [10] is the language used in the International Planning Competition, and the most commonly supported input definition language of modern planning systems.

Natural Model We focus on encoding the first scenario proposed by Petrobras, where time constraints like speed, refueling rates, and docking costs are not considered. We use the PDDL language facilities to compactly encode some of the problem features, like refueling or sailing using conditional preconditions and effects.

\[
(:types strip port waiting_area platform - location
  ship cargo - object )
\]

\[
(:predicates
  (at_ ?sh - object ?where - location)
  (can_refuel ?l - location)
  (docked ?sh - ship ?where - location)
  (loaded ?c - cargo ?sh - ship)
  (is_waiting_area ?l - location) )
\]

\[
(:functions
  (current_fuel ?sh - ship) - number
  (current_load ?sh - ship) - number
  (fuel_capacity ?sh - ship) - number
  (load_capacity ?sh - ship) - number
  (current_docking_capacity ?p - location) - number
  (total_docking_capacity ?p - location) - number
  (distance ?from - location ?to - location) - number
  (weight ?c - cargo) - number
  (total_fuel_used) - number )
\]

Fig. 1. Preamble for the Petrobras domain, in PDDL.

Figure 1 depicts the types, predicates and functions used. The following actions are defined using those elements: sail, load, unload, refuel_at_port, and refuel_at_platform. One can intuitively model the actions, taking into account the following design decisions:

– The sail action (Figure 2) can only be performed if the ship is not docked, and it has enough fuel for that trip. Note that the consumed fuel depends on the current ship load.
– To perform the load and unload actions, the ship must be docked where the load is, and it must have enough free space. The current load of the ship must be modified accordingly.
– For the dock and undock operations, the position and docking capacities must be checked.

As time constraints are simplified, the refueling amount is set to either 100 or 200 liters per action, and conditional effects are used to ensure that the total fuel capacity of the vessels is not exceeded.

Model with Unconditional Constraints  As the NumReach solver of [12] does not support conditional effects, like the ones introduced with the when keyword in Figure 2, we model some parts of the problem in a different way. This will allow to make a fair comparison between our software and NumReach. In order to make the minimum changes from the natural model, we do not change any predicate or function. We only make the following changes: We split the sail action in two, namely sail_empty and sail_full, with the (unconditional) effects corresponding to the case where current_load is zero or not, respectively (see Figure 2). The refueling_at_port and refueling_at_platform actions also make use of conditional effects, so they also need to be split in two. Note that although this apparently seems a minor change, it may cause many new variables to appear in the final encoding, as more actions are present.

\[
\text{Fig. 2. Example action in the natural model, containing conditional effects.}
\]
3 SMT Encodings

We reformulate the natural and unconditional PDDL models of the Petrobras problem to SMT. We apply a preprocessing step where we identify constant function values and simplify arithmetic constraints that operate on them. For example, function values like $(/ (\text{distance } ?\text{from } ?\text{to}))$ in Figure 2 can be replaced by a numeric constant since $?\text{from}$ and $?\text{to}$ are given locations, and distances between locations are constant.

3.1 QF_LIA Encoding

In the SMT-LIB standard [2], QF_LIA stands for the logic of Quantifier-Free Boolean formulas, with Linear Integer Arithmetic constraints. This logic has a good compromise between expressivity and performance, and is the natural choice for this problem. The reformulation of our PDDL models goes as follows.

For each time step, every ground instance of a PDDL predicate and action is mapped to a Boolean variable, and every ground instance of a PDDL function is mapped to an integer variable. For example, the action $\text{dock}(\text{ship} - ?\text{loc} - \text{location})$, with three locations $P_1$, $P_2$ and $P_3$, and two ships $\text{ship}_1$ and $\text{ship}_2$, will result into six ground instances $\text{dock}(\text{ship}_1,P_1)$, ..., $\text{dock}(\text{ship}_2,P_3)$, that will be mapped to six Boolean variables $\text{dock}^{t}_{\text{ship}_1,P_1}$, ..., $\text{dock}^{t}_{\text{ship}_2,P_3}$ for each time step $t$. The Boolean variables resulting from actions will be used to denote what action is executed at each time step, and with which parameters. The Boolean and integer variables resulting from grounding the predicates and functions, respectively, will constitute the state variables. Again, a superscript $t$ is used to differentiate the variables at each time step.

Below we describe the constraints that we impose on these variables, following the notation and ideas in [16]: $O$ is the set of all actions $o$ of the form $(p,e)$, where $p$ is the action precondition, and $e$ is a set of conditional action effects of the form $(f,d)$ (where $f$ is a condition, and $d$ is a set of literals that must be set to true when $f$ holds). We define $\text{EPC}_l(o) = \bigvee \{ f \mid (f,d) \in e, l \in d \}$, that is, the effect precondition for literal $l$ in action $o$.

Since apart from Boolean variables we also deal with integer variables, we assume that we can have literals like $x \geq k$ in preconditions and like $x = x^{\text{prev}} + k$ in effects, where $x^{\text{prev}}$ denotes the previous value of $x$. Moreover, given a formula $\phi$, by $\phi^t$ we denote the same formula where all integer variables $x$ have been replaced by $x^t$, and $x^{\text{prev}}$ by $x^{t-1}$, and analogously for Boolean variables.

For each ground action $o = (p,e)$, we have the following constraints. First, its execution during time step $t$ implies that its precondition is met.

---

2 For the sake of brevity, we refer to the model with unconditional constraints as the unconditional model.

3 Here preconditions can be general propositional formulas, not just sets of literals.

4 By a ground action $(p,e)$ we refer to an action where $p$ and $e$ are built on the state variables that result from grounding a PDDL model, as explained above.
Also, each of its conditional effects will hold at the next time step if the corresponding condition holds.

\[ o^t \implies p^t \quad \forall o \in O \]  

(1)

Here we view sets \(d\) of literals as conjunctions of literals. Note that unconditional effects will have \(\top\) as condition \(f\).

Second, we need explanatory axioms to express the reason of a change in the value of Boolean state variables.

\[ a^t \land \neg a^{t+1} \implies ((o_1^t \land EPC_{\neg a}^t(o_1)) \lor \cdots \lor (o_m^t \land EPC_{\neg a}^t(o_m))) \]  

(3)

Constraints (3) and (4) are generalized for numeric state variables as follows.

\[ x^t \neq x^{t+1} \implies ((o_1^t \land EPC_{mod(x,o_1)}^t(o_1)) \lor \cdots \lor (o_m^t \land EPC_{mod(x,o_m)}^t(o_m))) \]  

(5)

where \(mod(x,o)\) is an arithmetic literal, either \(x = x^{prev} + k\), \(x = x^{prev} - k\), or \(x = k\), resulting from the translation of the PDDL expression \(\text{increase}(x,k)\). \(\text{decrease}(x,k)\) or \(\text{assign}(x,k)\), respectively, occurring in the effects of the PDDL action corresponding to \(o\).

### 3.2 Sequential Plans

The sequential encoding allows exactly one action per time step. This is achieved by imposing an exactly-one constraint on the action variables at each time step. We tested some well-known encodings, and we settled with the binary encoding (see [9]) as it gave us the best performance. The encoding introduces new variables \(B_1, \ldots, B_{\log_2 n}\), where \(n = |O|\), and associates each variable \(o_i^t\) with a unique bit string \(s_i \in \{0,1\}^{\log_2 n}\). The encoding is:

\[ \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\log_2 n} \neg o_i^t \lor \circ(i,j) \]  

(6)
\begin{equation}
\bigvee_{i=1}^{n} o_i^t
\end{equation}

where \( \odot(i,j) \) is \( B_j \) if the \( j^{th} \) bit of the bit string of \( s_i \) is 1, and \( \neg B_j \) otherwise. The binary encoding of the at-most-one constraint (6), introduces \( \lceil \log_2 n \rceil \) new variables and \( n \lceil \log_2 n \rceil \) binary clauses. Together with the at-least-one constraint (7), we obtain the desired exactly-one constraint.

### 3.3 Parallel Plans

A parallel plan follows the same idea as a sequential plan, but with the exception that at each time step a set of actions can be executed, instead of only one. Two types of parallel plans have been encoded: \( \forall \)-step plans, and \( \exists \)-step plans.

We have found a good compromise between the number of added clauses and the parallelism obtained in the Petrobras domain using the notion of affection defined in [16]. For the Boolean case, an action \( o = \langle p, e \rangle \) is considered to affect \( o' = \langle p', e' \rangle \) if, for some Boolean state variable \( a \), either of the following holds:

\begin{align*}
& a \in d \text{ for some } (f,d) \in e \land (a \text{ occurs in } f' \text{ for some } (f',d') \in e') \\
& \quad \lor a \text{ occurs negatively in } p') \quad (8) \\
& \neg a \in d \text{ for some } (f,d) \in e \land (a \text{ occurs in } f' \text{ for some } (f',d') \in e') \\
& \quad \lor a \text{ occurs positively in } p') \quad (9)
\end{align*}

That is, \( o \) affects \( o' \) if \( o \) can impede the execution of \( o' \), or change its effects. For integer variables \( x \) the constraints become:

\begin{equation}
\text{mod}(x,o) \in d \text{ for some } (f,d) \in e \\
\quad \land (x \text{ occurs in } f' \text{ for some } (f',d') \in e') \lor x \text{ occurs in } p') \quad (10)
\end{equation}

where \( \text{mod}(x,o) \), as in (5), is an arithmetic literal of the form \( x = x^{\text{prev}} + k \), \( x = x^{\text{prev}} - k \) or \( x = k \).

These precomputed affectations will be used to forbid parallel executions of incompatible actions. Notice that affection is not a symmetric relation.

**\( \forall \)-step Plans** The notion of parallelism of a \( \forall \)-step plan is defined as the possibility of ordering the actions to any total order. Therefore, at each time step \( t \) we simply add a mutex between any pair of interfering actions \( o_i \) and \( o_j \):

\begin{equation}
\neg (o_i^t \land o_j^t) \text{ if } o_i \text{ affects } o_j \text{ or } o_j \text{ affects } o_i \quad (11)
\end{equation}
∃-step Plans In ∃-step plans, a fixed (arbitrary) total ordering on the actions is imposed beforehand, and the parallel execution of two actions $o_i$ and $o_j$ such that $o_i$ affects $o_j$ is forbidden only if $i < j$:

$$\neg (o_i^t \land o_j^t) \text{ if } o_i \text{ affects } o_j \text{ and } i < j$$  \hspace{1cm} (12)

Since ∃-step plans are less restrictive than ∀-step plans, as they do not require that all orderings of parallel actions result in valid sequential plan, they sometimes allow more parallelism.

3.4 QF_UFLIA Encoding

In the SMT-LIB standard [2], QF_UFLIA stands for the logic of Quantifier-Free Boolean formulas, with Linear Integer Arithmetic constraints and Uninterpreted Functions. Uninterpreted functions have no other property than its name and arity. In other words, they are only subject to the following axiom schema of consistency: $x_1 = x'_1 \land \cdots \land x_n = x'_n \implies f(x_1, \ldots, x_n) = f(x'_1, \ldots, x'_n)$.

As the previously introduced QF_LIA encodings grows considerably with time, to the point of getting unmanageable instances, we developed a more compact encoding, using the theory of uninterpreted functions to express predicates, functions and actions. This encoding is reminiscent of the lifted causal encodings in [15].

Every defined object (ship, port, cargo, ... ) in the problem is mapped to an integer. For each function, predicate and action an uninterpreted function is declared, with each parameter being declared as an integer. Also, a new integer parameter is added to each, representing a time step. Uninterpreted functions corresponding to predicates and actions return a Boolean value, whilst the ones for functions return an integer value. Moreover, for each action, parameter and time step, a new integer variable is defined, representing the value of that parameter in the action if executed at the corresponding time step.

For example, the Boolean function $\varphi_o(x_{o,1}^t, x_{o,n}^t, t)$ determines whether action $o$ with parameters $x_{o,1}^t, \ldots, x_{o,n}^t$ is executed at time step $t$. The parameter $t$ is a constant representing the time step. It is shared between all uninterpreted functions for the actions, predicates and functions in the same time step. Contrarily, $x_{o,1}^t, \ldots, x_{o,n}^t$ are variables with finite domains, and constraints are added to restrict their possible values. Regarding predicates and functions, no new variables are defined, since their arguments will be either constants or variables occurring in some action.

We remark that, in this new setting, a state is defined by the value of the uninterpreted functions corresponding to predicates and functions, for a given value of their arguments. Equations (1) and (2) of the QF_LIA encoding are generalized here as:

$$\varphi_o(x_{o,1}^t, \ldots, x_{o,n}^t, t) \implies p_t^t \quad \forall o = \langle p, e \rangle \in O$$  \hspace{1cm} (13)

$$\varphi_o(x_{o,1}^t, \ldots, x_{o,n}^t, t) \land f_t^t \implies d_t^{t+1} \quad \forall o = \langle p, e \rangle \in O, \forall (f, d) \in e$$  \hspace{1cm} (14)
Note that this results in a much more compact encoding than if we restrict to QF_LIA, since here we are using variables as arguments of functions, and it is the SMT solver who is in charge of guessing the concrete values of the parameters of the executed actions. The considered set of actions \( O \) is now parameterized, and hence similar to that of PDDL, with actions like \( \text{dock}(x, y) \), \( \text{sail}(x, y, z) \), etc. instead of grounded actions like \( \text{dock}_{\text{ship1}, P1}, \text{dock}_{\text{ship1}, P2}, \) etc.

Equations (3), (4) and (5) are generalized as:

\[
\bigvee_{o \in \text{touch}(g)} \left( \bigwedge_{i \in 1..n, j \in 1..m} \left( \varphi_g(c_{g,1}, \ldots, c_{g,n}, t) \neq \varphi_g(c_{g,1}, \ldots, c_{g,n}, t + 1) \implies \bigwedge_{i = 1..n, j = 1..m} \left( x_{i,j}^t = c_{g,i} \right) \right) \right) \forall g \in G \quad \forall c_{g,1}, \ldots, c_{g,n} \in S_1 \times \cdots \times S_n
\]

where \( G \) is the set of predicates and functions, \( \text{touch}(g) \) is the set of actions that may modify \( g \), \( S_i \) is the domain of the \( i \)-th argument of \( \varphi_g \), and \( \text{name}(h, k) \) is the name in the PDDL model of the \( k \)-th argument of the functor \( h \). To help the reader understand the formula, we provide an example.

**Example 1.** Suppose we have the following simple PDDL problem:

- objects: \( A, B - \text{truck}, L1, L2, L3 - \text{loc} \)
- predicate: \( \text{at}(?t - \text{truck}, ?l - \text{loc}) \)
- action: \( \text{travel}(?t - \text{truck}, ?\text{from} - \text{loc}, ?\text{to} - \text{loc}) \)
- action: \( \text{refuel}(?x - \text{truck}, ?\text{where} - \text{loc}) \)
- function: \( \text{fuel}(?t - \text{truck}) - \text{number} \)

where the action \( \text{travel} \) has \( \text{(decrease (fuel ?t) 10)} \) among its effects, and the action \( \text{refuel} \) has \( \text{(increase (fuel ?x) 20)} \) as its only effect. The constraint described in (15) for the \( \text{fuel} \) function would be encoded into SMT at time step 0 as follows:

\[
\text{(\Rightarrow (distinct (fuel A 0) (fuel A 1)))}
\text{(or (and (travel x1_0 x2_0 x3_0 0) (= x1_0 A))}
\text{(and (refuel x4_0 x5_0 0) (= x4_0 A))))}
\]

\[
\text{(\Rightarrow (distinct (fuel B 0) (fuel B 1)))}
\text{(or (and (travel x1_0 x2_0 x3_0 0) (= x1_0 B))}
\text{(and (refuel x4_0 x5_0 0) (= x4_0 B))))}
\]

That is, we are saying that if the fuel of truck \( A \) (or \( B \)) has changed this should be because it has been the protagonist of some action implying a modification in its fuel, namely traveling or refueling.

Again, this is much more compact than its QF_LIA counterpart.
4 Experimental Evaluation

The experiments were run on a cluster of machines, running the CentOS operating System, equipped with Intel® Xeon® E3-1220v2 Processors at 3.10 GHz with Turbo Boost disabled, and 8GB of main memory.

To test our encodings, we created a very similar set of benchmarks to the ones used in [19]. It consists of 4 groups of generated instances, with an increasing number of cargo items, ranging from 1 to 15. Every cargo is assigned randomly to one of the two ports, and each ship is randomly docked in one of them. The groups differ in the number of ships and in the total fuel capacity of each ship:

- Group A: 3 ships with 600 liters of fuel capacity.
- Group B: 10 ships with 600 liters of fuel capacity.
- Group C: 10 ships with 800 liters of fuel capacity.
- Group D: 10 ships with 1000 liters of fuel capacity.

We compare the performance of our approach to that of NumReach [12], which approximates the reachable domains of the numeric state variables. That is, it generates a set of values $D_t(v)$ for every numeric variable $v$, so that every value that $v$ can have after $t$ time steps is contained in $D_t(v)$. Then a SAT encoding is generated, by introducing a Boolean variable $a_{v,c,t}$ for every $t$, $v$ and $c \in D_t(v)$. As we will see, this method is very sensitive to the size of $D_k(v)$.

The input language of the NumReach solver is PDDL, and it has two strategies for solving: NumReach/SAT and NumReach/SMT. NumReach/SMT is similar to NumReach/SAT, except for the encoding of the numeric variables. NumReach uses a different backend solver for each one. For the SAT approach, it uses MiniSAT or ZChaff, but we only used the latest version of MiniSAT (2.2.0) [7] in the experiments, as we couldn’t find any modern version of ZChaff. For the SMT backend, it was not possible to use any modern version of a SMT solver, and we had to restrict MathSAT 3. This is because NumReach generates the SMT instances in a file format which is different from SMT-LIB and not known by modern SMT solvers. We also could not find any documentation on the format used, and writing a converter would be costly. For these reasons and due to the poor observed performance, we do not include the results for NumReach/SMT.

During the experiments with NumReach, we found out that MiniSAT dedicated most of its solving time into simplificating the formula. So we decided to execute the same experiments in two ways: instructing MiniSAT not to simplify the input formula, and with the default options. In the tables of results we refer to both solving options as SAT and SAT w/o pre, respectively.

For each instance, we made executions for the three QF_LIA encodings: sequential, ∀-step and ∃-step, with two SMT solvers via API: Yices-1.0.38 [6] and Z3-4.3.1 [5]. The results depicted are from the Yices executions, as although it wasn’t always faster, it solved more instances than Z3. The executions were made through the APIs, because when we tried to use plain files we found that the generated files for the biggest instances were too big for the solvers, spanning some gigabytes.
We do not report results for the QF_UFLIA encoding, as they are comparable to that of the sequential QF_LIA encoding and, moreover, the extension of the QF_UFLIA encoding to parallel plans is a non-trivial task, as the encoding of the explanatory axioms relies on the premise that only one action is executed. Consequently we leave it as future work.

Nevertheless, it is worth noting that the QF_UFLIA encoding, which is much more compact, allowed us to run the solvers by feeding them the formula from a file.

Table 1. Execution times for group A in seconds, and number of time steps checked.

<table>
<thead>
<tr>
<th>Instance</th>
<th>QF_LIA encoding</th>
<th>NumReach</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
<td>∃-step</td>
<td>SAT</td>
</tr>
<tr>
<td>A1</td>
<td>2.73 (5)</td>
<td>45.27 (5)</td>
<td>22.95 (5)</td>
</tr>
<tr>
<td>A2</td>
<td>45.67 (13)</td>
<td>76.37 (8)</td>
<td>40.51 (8)</td>
</tr>
<tr>
<td>A3</td>
<td>TO (17)</td>
<td>105.44 (10)</td>
<td>55.29 (9)</td>
</tr>
<tr>
<td>A4</td>
<td>TO (17) 1727.77 (13)</td>
<td>2674.75 (12)</td>
<td>2067.24 (11)</td>
</tr>
<tr>
<td>A5</td>
<td>TO (19)</td>
<td>TO (13)</td>
<td>TO (13)</td>
</tr>
<tr>
<td>A6</td>
<td>TO (19)</td>
<td>TO (14)</td>
<td>TO (13)</td>
</tr>
<tr>
<td>A7</td>
<td>TO (19)</td>
<td>TO (14)</td>
<td>TO (13)</td>
</tr>
<tr>
<td>A8</td>
<td>TO (20)</td>
<td>TO (14)</td>
<td>TO (14)</td>
</tr>
<tr>
<td>A9</td>
<td>TO (18)</td>
<td>TO (14)</td>
<td>TO (13)</td>
</tr>
<tr>
<td>A10</td>
<td>TO (19)</td>
<td>TO (15)</td>
<td>TO (14)</td>
</tr>
<tr>
<td>A11</td>
<td>TO (20)</td>
<td>TO (14)</td>
<td>TO (14)</td>
</tr>
<tr>
<td>A12</td>
<td>TO (20)</td>
<td>TO (14)</td>
<td>TO (15)</td>
</tr>
<tr>
<td>A13</td>
<td>TO (20)</td>
<td>TO (15)</td>
<td>TO (15)</td>
</tr>
<tr>
<td>A14</td>
<td>TO (19)</td>
<td>TO (14)</td>
<td>TO (14)</td>
</tr>
<tr>
<td>A15</td>
<td>TO (21)</td>
<td>TO (15)</td>
<td>TO (15)</td>
</tr>
</tbody>
</table>

Tables 1 to 4 show the execution time in seconds and number of time steps checked for each group of instances. TO denotes that the solver could not find a plan in the given time of one hour, and the fastest solver for each instance is highlighted. Between parenthesis there is the last time step checked by the solver (which corresponds to the length of the shortest plan found for the solved instances). Note that it is not clear for us what notion of affectation or parallelism is NumReach using, so the plan lengths between correct solutions given by NumReach and our encoding for the same instance may differ. Table 5 summarizes how many instances each solving approach could finish in the given time, and among those in how many it was the fastest.

If we look at the NumReach/SAT executions, all the instances have a better solving time without simplifying the input formula. But, although we observe an speedup of more than one order of magnitude on most of the solved instances, only a few more instances can be solved without preprocessing, due to the combinatorial explosion. This indicates that the problem is inherently hard.

After analyzing the computed affectations between actions, we could see that the problem is highly parallel in the number of ships. Ships can operate inde-
Table 2. Execution times for group B in seconds, and number of time steps checked.

<table>
<thead>
<tr>
<th>Instance</th>
<th>QF, LIA encoding</th>
<th>NumReach</th>
<th>Sequential</th>
<th>∀-step</th>
<th>∃-step</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>11.47 (5)</td>
<td></td>
<td>154.01 (5)</td>
<td>79.20 (5)</td>
<td>366.29 (6)</td>
</tr>
<tr>
<td>B2</td>
<td>154.53 (10)</td>
<td></td>
<td>158.88 (5)</td>
<td>82.80 (5)</td>
<td>544.15 (6)</td>
</tr>
<tr>
<td>B3</td>
<td>TO (12)</td>
<td></td>
<td>162.81 (5)</td>
<td>86.02 (5)</td>
<td>1344.00 (7)</td>
</tr>
<tr>
<td>B4</td>
<td>TO (13)</td>
<td></td>
<td>168.30 (5)</td>
<td>89.74 (5)</td>
<td>2761.95 (8)</td>
</tr>
<tr>
<td>B5</td>
<td>TO (13)</td>
<td></td>
<td>173.05 (5)</td>
<td>93.77 (5)</td>
<td>2864.55 (8)</td>
</tr>
<tr>
<td>B6</td>
<td>TO (14)</td>
<td></td>
<td>178.21 (5)</td>
<td>96.92 (5)</td>
<td>2952.88 (8)</td>
</tr>
<tr>
<td>B7</td>
<td>TO (14)</td>
<td></td>
<td>183.16 (5)</td>
<td>101.81 (5)</td>
<td>TO (10)</td>
</tr>
<tr>
<td>B8</td>
<td>TO (14)</td>
<td></td>
<td>300.96 (7)</td>
<td>189.73 (7)</td>
<td>TO (9)</td>
</tr>
<tr>
<td>B9</td>
<td>TO (14)</td>
<td></td>
<td>TO (8)</td>
<td>TO (7)</td>
<td>TO (9)</td>
</tr>
<tr>
<td>B10</td>
<td>TO (15)</td>
<td></td>
<td>748.626 (8)</td>
<td>358.57 (7)</td>
<td>TO (9)</td>
</tr>
<tr>
<td>B11</td>
<td>TO (14)</td>
<td></td>
<td>TO (8)</td>
<td>TO (7)</td>
<td>TO (9)</td>
</tr>
<tr>
<td>B12</td>
<td>TO (14)</td>
<td></td>
<td>TO (8)</td>
<td>TO (8)</td>
<td>TO (9)</td>
</tr>
<tr>
<td>B13</td>
<td>TO (14)</td>
<td></td>
<td>TO (8)</td>
<td>TO (8)</td>
<td>TO (9)</td>
</tr>
<tr>
<td>B14</td>
<td>TO (16)</td>
<td></td>
<td>TO (9)</td>
<td>TO (8)</td>
<td>TO (9)</td>
</tr>
<tr>
<td>B15</td>
<td>TO (14)</td>
<td></td>
<td>TO (9)</td>
<td>TO (8)</td>
<td>TO (9)</td>
</tr>
</tbody>
</table>

Table 3. Execution times for group C in seconds, and number of time steps checked.

<table>
<thead>
<tr>
<th>Instance</th>
<th>QF, LIA encoding</th>
<th>NumReach</th>
<th>Sequential</th>
<th>∀-step</th>
<th>∃-step</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>11.58 (5)</td>
<td></td>
<td>154.46 (5)</td>
<td>79.05 (5)</td>
<td>510.45 (6)</td>
</tr>
<tr>
<td>C2</td>
<td>149.59 (10)</td>
<td></td>
<td>159.05 (5)</td>
<td>81.59 (5)</td>
<td>744.29 (6)</td>
</tr>
<tr>
<td>C3</td>
<td>TO (13)</td>
<td></td>
<td>163.17 (5)</td>
<td>86.21 (5)</td>
<td>1826.95 (7)</td>
</tr>
<tr>
<td>C4</td>
<td>TO (13)</td>
<td></td>
<td>168.05 (5)</td>
<td>89.74 (5)</td>
<td>TO (9)</td>
</tr>
<tr>
<td>C5</td>
<td>TO (13)</td>
<td></td>
<td>173.13 (5)</td>
<td>93.44 (5)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C6</td>
<td>TO (13)</td>
<td></td>
<td>178.25 (5)</td>
<td>97.28 (5)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C7</td>
<td>TO (14)</td>
<td></td>
<td>183.30 (5)</td>
<td>101.06 (5)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C8</td>
<td>TO (14)</td>
<td></td>
<td>298.40 (7)</td>
<td>168.80 (7)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C9</td>
<td>TO (14)</td>
<td></td>
<td>TO (8)</td>
<td>TO (7)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C10</td>
<td>TO (14)</td>
<td></td>
<td>758.50 (8)</td>
<td>351.54 (7)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C11</td>
<td>TO (14)</td>
<td></td>
<td>TO (8)</td>
<td>TO (7)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C12</td>
<td>TO (14)</td>
<td></td>
<td>TO (8)</td>
<td>TO (8)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C13</td>
<td>TO (14)</td>
<td></td>
<td>TO (8)</td>
<td>TO (8)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C14</td>
<td>TO (17)</td>
<td></td>
<td>TO (9)</td>
<td>TO (8)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>C15</td>
<td>TO (14)</td>
<td></td>
<td>TO (9)</td>
<td>TO (8)</td>
<td>TO (8)</td>
</tr>
</tbody>
</table>

pendently, with the only limitation of the docking space. This can be seen for example in instance D7: in 4 time steps, 7 cargo items are transported from the port of origin to its destination. The difference of 2 time steps between D7 and D8 is caused only by the docking space capacities.

Note also that ∃-step plans are easier to find than ∀-step plans in this domain. However, contrarily to what could be expected, in most of the cases they are not shorter. This is due to the nature of the domain: as said, ships can operate independently and hence, in many cases, requiring parallel actions to result in a
Table 4. Execution times for group D in seconds, and number of time steps checked.

<table>
<thead>
<tr>
<th>Instance</th>
<th>QF LIA encoding</th>
<th>NumReach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
<td>∀-step</td>
</tr>
<tr>
<td>D1</td>
<td>11.58 (5)</td>
<td>154.47 (5)</td>
</tr>
<tr>
<td>D2</td>
<td>139.56 (10)</td>
<td>158.66 (5)</td>
</tr>
<tr>
<td>D3</td>
<td>TO (12)</td>
<td>163.36 (5)</td>
</tr>
<tr>
<td>D4</td>
<td>TO (13)</td>
<td>168.01 (5)</td>
</tr>
<tr>
<td>D5</td>
<td>TO (13)</td>
<td>173.36 (5)</td>
</tr>
<tr>
<td>D6</td>
<td>TO (13)</td>
<td>177.89 (5)</td>
</tr>
<tr>
<td>D7</td>
<td>TO (14)</td>
<td>182.66 (5)</td>
</tr>
<tr>
<td>D8</td>
<td>TO (14)</td>
<td>302.91 (7)</td>
</tr>
<tr>
<td>D9</td>
<td>TO (14)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>D10</td>
<td>TO (14)</td>
<td>762.78 (8)</td>
</tr>
<tr>
<td>D11</td>
<td>TO (15)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>D12</td>
<td>TO (14)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>D13</td>
<td>TO (14)</td>
<td>TO (8)</td>
</tr>
<tr>
<td>D14</td>
<td>TO (17)</td>
<td>TO (9)</td>
</tr>
<tr>
<td>D15</td>
<td>TO (15)</td>
<td>TO (9)</td>
</tr>
</tbody>
</table>

Table 5. Summary of the results.

<table>
<thead>
<tr>
<th>60 instances</th>
<th>QF LIA encoding</th>
<th>NumReach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
<td>∀-step</td>
</tr>
<tr>
<td>Total solved</td>
<td>8</td>
<td>31</td>
</tr>
<tr>
<td>Faster instances</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

valid plan if putting them to any total order, is not stronger than requiring this for some fixed order.

On the other hand, under our approach, with the natural model the solver found a solution for 7 more instances than with the unconditional model. However the natural model could not be compared with NumReach (recall that it does not support conditional effects), so the results shown in Tables 1 to 5 are from the unconditional model.

Intuitively, a higher ship fuel capacity should make the problem easier, as less actions will be necessary as ships will need to refuel less often. Instead, it is interesting to note that for NumReach/SAT the groups C and D become the hardest instances. This is because with the higher numbers, state-space exploration seems to grow too large to be manageable, as we suspected.

Other recent works have provided efficient solutions to the Petrobras challenge proposal. In [19,3] various heuristic (incomplete) solvers are used to solve the Petrobras challenge under different optimization criteria. Since our natural PDDL model is very close to one of the proposed models in [19], we mimicked the generated benchmarks and compared the results. In Table 6 we can see that, with the unconditional model and the QF LIA encoding, only 3 instances less are solved than with SGPlan [13]. But if we consider the natural model, also with the QF LIA encoding, 7 more instances are solved, outperforming SGPlan.
Table 6. Number of instances solved by each approximation.

<table>
<thead>
<tr>
<th>Instance set</th>
<th>Unconditional model</th>
<th>Natural model</th>
<th>SGPlan</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
<td>∀-step</td>
<td>3-step</td>
</tr>
<tr>
<td>Group A</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Group B</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Group C</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Group D</td>
<td>2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Total</td>
<td>8</td>
<td>31</td>
<td>31</td>
</tr>
</tbody>
</table>

5 Conclusions and Future Work

In this paper we have presented several SMT encodings for the Petrobras challenge. The proposed encodings make use of SMT to tightly integrate arithmetic into the problem, where other approximations rely into making state-space exploration on the numerical variables, or loosely integrate external solvers for evaluating arithmetic constraints. Our approximation seems to be competitive with other exact and complete methods for planning with resources on this problem, and also with some incomplete (heuristic) ones. In particular, we have obtained better results than NumReach [12] and similar results to SGPlan [13]. We have seen that the method of [12], which is based on approximating the reachable domains of numeric variables, is very sensitive to the number of distinct possible values, and it is not well-suited for this real real-life problem.

Although SAT and SMT solvers have generic preprocessing steps to simplify the input formulas, we observed that for MiniSAT those were harmful for this problem. Nevertheless, it would be interesting to consider some more ad hoc preprocessing steps to help reduce the search space, based on reachability analysis, as well as operator splitting and factoring [15,8].

We have also observed a promising performance using a sequential encoding with uninterpreted functions. This encoding is more compact, and it retains most of the problem original structure. It remains to be seen if a parallelized version of this encoding could lead to better results than the encoding without functions. To the best of our knowledge, there are no works using parallelized encodings with uninterpreted functions. It should hence be studied how to generalize the standard parallel encodings to this setting.

Acknowledgements

All authors supported by the Spanish Ministry of Economy and Competitiveness through the project HeLo (ref. TIN2012-33042). Joan Espasa also supported by UdG grant (BR 2013).

The authors acknowledge Jörg Hoffmann for giving them access to the NumReach software.
References


Encoding The Lexicographic Ordering Constraint in SAT Modulo Theories

Hani A. Elgabou and Alan M. Frisch

Department of Computer Science, University of York, Heslington, York, YO10 5GH, UK
{he583,alan.frisch}@york.ac.uk
http://www.cs.york.ac.uk/

Abstract. This paper presents eight different SMT encodings of the lexicographic ordering constraint. These constraints are helpful in breaking some kinds of symmetries in combinatorial decision and optimisation problems. The encodings are obtained from the literature then translated into an SMT suitable form. We have done this using two methods, the first is by directly translating the encodings into SMT. The second starts by rewriting the encodings in MiniZinc language, flattening them into FlatZinc instances then using a tool called fzn2smt to translate them to SMT. We evaluate the encodings on a suite of instances of the Social Golfer problem, which is well known for its highly symmetric models. This shows that different encodings of the lexicographic ordering constraint perform best on different problem instances and that no one encoding is dominant.

Keywords: Global Constraints, Lexicographic Ordering, Symmetry, SAT Modulo Theories, MiniZinc, SMTLIB

1 Introduction

Modern Boolean satisfiability (SAT) solvers are essential tools in many applications including scheduling, verification, circuit design and others. The nature of some of today’s combinatorial problems makes them difficult to be represented using SAT’s Boolean formulas. Satisfiability Modulo Theories (SMT) is introduced to overcome this limitation, where problems can be encoded using logical formulas of combinations of atomic propositions and atomic expressions in one or more Theories (T). The theory part in SMT formulas enables them to naturally describe problems related to any of the SMT supported theories, like arithmetic, arrays and bit-vectors. SMTLIB is the standard modelling language in SMT.

Symmetries arise in many constraint satisfaction problem models. A common form of symmetry is the interchangeability between elements of sets of variables and the corresponding sets of values. An example is the ability to swap any two rows or columns in a Latin Square while preserving satisfiability. Breaking symmetry reduces the search space, which could improve performance.

The Lex ordering constraint enforces lexicographic ordering between two vectors of variables, which makes it useful in breaking symmetry between rows and
columns of a matrix of decision variables [1]. For example, enforcing lex on between pairs of rows of a Latin Square eliminates the interchangeability between them and the same applies for the columns.

A recent study demonstrated that SMT solvers have a competitive performance in solving Constraint Satisfaction Problems. In that study, Bofill et al. [2] built a tool called fzn2smt to translate CSP instances expressed in flatzinc into the SMT language SMTLIB1.2 standard [3], then solved them using the SMT solver Yices1 [4], which in general performed better than some well known constraint solvers.

Our main aim for this research is to explore ways of further improving this performance by studying some of the instances where fzn2smt-Yices exhibited weaker performance. We found out that some of these instances involve Global Constraints, which are constraints that could have a non-fixed number of variables and encapsulates a set of other constraints [5]. So we decided to study how fzn2smt translate some of these constraints to try to come up with a better translation. Our first study is on encodings of the lexicographic ordering global constraint.

2 Methodology

We present eight alternative decompositions of the lexicographic ordering constraint, each of which is drawn from the constraint solving literature or is a variant of such. We evaluate seven of the decompositions in solving a suite of instances of the Social Golfers Problem (SGP) (Problem 010 in CSPLib [6]) with the Yices2 [4] SMT solver. The eighth decomposition is not evaluated because, as we will see, it is impractical to do so.

Using MiniZinc [7] and fzn2smt we have produced SMT encodings of the SGP instances as follows. MiniZinc provides a default decomposition that can be used for each of its global constraints, including the lexicographic ordering constraint. Indeed, the MiniZinc decomposition of lexicographic ordering is one of the seven we have evaluated. The other six encodings are produced by replacing the MiniZinc default decomposition with one of the alternatives.

We use a MiniZinc model of the SGP in which row and column symmetry is broken by constraining the rows to be in increasing lexicographic order and the columns to be in increasing lexicographic order. With one of the seven decompositions in place, an SGP instance is translated to FlatZinc by the MiniZinc system. The resulting FlatZinc is translated to SMT by fzn2smt. The SMT code generated by this process follows the SMTLIB 1.2 standard and uses linear integer arithmetic logic as provided by the QF_LIA SMT theory.

We shall use the term mzn2smt to refer to the pipeline of translating a MiniZinc specification to SMT in two steps, first using MiniZinc to produce FlatZinc and then passing this through fzn2smt to produce SMT. We chose this pipeline at this stage of our research just for convenience and we are aware of some of its possible drawbacks. For instance, some constraints could be naturally represented in SMT, but when they go through the translation process they get
broken into smaller ones. That is why we performed two sets of benchmarks, one is for the mzn2smt translation and the other is of the direct translation from each encoding to SMT.

3 Encodings for Lex Ordering Constraints

This section presents eight different encodings for the lex Ordering Constraint. Throughout, we consider a non-strict lex constraint between two vectors $A$ and $B$ of finite-domain variables. Both vectors are considered to be of length $n$. We write such a constraint as $A \leq_{\text{lex}} B$. Each of the following subsections presents a decomposition of the lex constraint followed by the result of passing it through the mzn2smt pipeline. $T_1[i], T_2[i], \ldots$ are auxiliary Boolean arrays introduced by mzn2smt. The index $i$ of these arrays ranges between 1 and $n$. The generated SMTLIB code does not literally contain arrays; we use the notation as a clean way of naming a set of $n$ distinct SMT variables.

3.1 The AND Decomposition Encoding [8]

This encoding decomposes lex constraint into a conjunction of smaller constraints as shown in the following formula, and because of that it is known as AND Decomposition.

$$A[1] \leq B[1] \land \left( \bigwedge_{i=1}^{n-1} \left( \bigwedge_{j=1}^{i} (A[j] = B[j]) \rightarrow (A[i+1] \leq B[i+1]) \right) \right)$$

$A[1] \leq B[1]$ is there because the first values of the two vectors are the most significant values to compare. The rest of the formula is self explanatory.

A strict ordering can be obtained by changing the formula $(A[i+1] \leq B[i+1])$ to $(A[i+1] < B[i+1])$.

After translation using mzn2smt:

$$A[1] \leq B[1] \quad (1)$$
$$1 \leq i \leq n-1 \quad T_1[i] \iff (A[i] = B[i]) \quad (2)$$
$$1 \leq i \leq n-1 \quad T_2[i] \iff (A[i+1] \leq B[i+1]) \quad (3)$$
$$1 \leq i \leq n-2 \quad T_3[i] \iff \bigwedge_{j=1}^{i+1} T_1[j] \quad (4)$$
$$\neg T_1[1] \lor T_2[1] \quad (5)$$
$$1 \leq i \leq n-2 \quad \neg T_3[i] \lor T_2[i+1] \quad (6)$$

Number of constraints generated by this encoding is $4n - 4$. 

4 Encoding the Lexicographic Ordering Constraint in SAT Modulo Theories

3.2 The AND Decomposition Encoding using Common Sub-expressions Elimination

This encoding, which we call AND CSE, is similar to the previous one and produces a similar formula too. The difference is, in this encoding we use a Boolean array to eliminate common sub-expressions in the formula as presented in line (9). The purpose of this encoding is to compare performance between using the nested loops as in line (4) in the previous encoding and this approach.

The resulting formula after eliminating common sub-expressions using the Boolean array $X[i]$:  


$$1 \leq i \leq n-2 \quad X[i+1] \leftrightarrow (X[i] \land (A[i+1] = B[i+1]))$$  

$$1 \leq i \leq n-1 \quad X[n] \rightarrow (A[n+1] \leq B[n+1])$$

A strict ordering can be obtained by changing the formula $(A[i+1] \leq B[i+1])$ in line (3) to $(A[i+1] < B[i+1])$.

After translation using mzn2smt:


$$1 \leq i \leq n-2 \quad T_1[i] \leftrightarrow (A[i+1] = B[i+1])$$  

$$1 \leq i \leq n-1 \quad T_2[i] \leftrightarrow (A[i+1] \leq B[i+1])$$  

$$1 \leq i \leq n-2 \quad X[i+1] \leftrightarrow (X[i] \land T_1[i])$$  

$$1 \leq i \leq n-1 \quad \neg X[i] \lor T_2[i]$$

Number of constraints generated by this encoding is $4n - 4$.

3.3 The OR Decomposition Encoding [9]

This encoding, known as OR Decomposition, decomposes lex constraint into a formula of smaller constraints dis-joined together, as shown below:

$$(A[1] < B[1]) \lor$$

$$\bigvee_{i=1}^{n-1} \left( \bigwedge_{j=1}^{i} (A[j] = B[j]) \land (A[i+1] < B[i+1]) \lor \right.$$  

$$\bigwedge_{i=1}^{n} (A[i] = B[i])$$

A strict ordering can be obtained by removing $\bigwedge_{i=1}^{n} (A[i] = B[i])$ from the above formula.
Encoding the Lexicographic Ordering Constraint in SAT Modulo Theories

After translation using mzn2smt:

\[ \begin{align*}
1 \leq i \leq n & \quad T_1[i] \iff (A[i] = B[i]) \\
1 \leq i \leq n & \quad T_2[i] \iff (A[i] < B[i]) \\
1 \leq i \leq n - 1 & \quad T_3[i] \iff \bigwedge_{j=1}^{i} T_1[j] \land T_2[i + 1] \\
T_3[n] & \iff \bigwedge_{i=1}^{n} T_1[i] \\
(n - 1) \bigvee_{i=1}^{n-1} T_3[i] & \lor T_2[1] \lor T_3[n]
\end{align*} \]

Number of constraints generated by this encoding is \(3n + 1\)

3.4 The OR Decomposition Encoding using Common Sub-expressions Elimination

This version of OR decomposition, called OR CSE, uses a Boolean array to eliminate common sub-expressions from the formula. \(X[i]\) is a Boolean array with an index range of 1\(\text{to}n\), this array is used to eliminating common sub-expressions as shown in the following formula.

\[ \begin{align*}
(A[1] < B[1]) \lor \\
(n - 1) \bigvee_{i=1}^{n-1} (X[i] \land (A[i + 1] < B[i + 1])) \lor X[n] \land \\
1 \leq i \leq n - 1 & \quad (X[i] \land (A[i + 1] = B[i + 1])\)
\end{align*} \]

A strict ordering can be obtained by removing \(X[n]\) from the above formula.

After translation using mzn2smt:

\[ \begin{align*}
1 \leq i \leq n & \quad X[1] \iff (A[1] = B[1]) \\
1 \leq i \leq n - 1 & \quad T_2[i] \iff (A[i + 1] = B[i + 1]) \\
1 \leq i \leq n & \quad T_3[i] \iff (A[i] < B[i]) \\
1 \leq i \leq n - 1 & \quad X[i + 1] \iff (X[i] \land T_2[i]) \\
1 \leq i \leq n - 1 & \quad T_4[i] \iff (X[i] \land T_3[i + 1]) \\
(n - 1) \bigvee_{i=1}^{n-1} T_4[i] & \lor T_3[1] \lor X[n]
\end{align*} \]

Number of constraints generated by this encoding is \(5n - 2\)
Encoding the Lexicographic Ordering Constraint in SAT Modulo Theories

3.5 Arithmetic Lex Encoding [8]

Another way of encoding lex constraint is using arithmetic constraint. This constraint compares the sum of the values of two vectors with each value multiplied by a factor that represents the significance of the values. We assume all the variables in A and B have a domain of 1 to n.

\[
\sum_{i=1}^{n} A[i] \times d^{n-i} \leq \sum_{i=1}^{n} B[i] \times d^{n-i}
\]

mzn2smt translation produces the same formula above.

This encoding is limited by the size of data type used to represent domains of values, for example, if \( A[1] \times d^{n-1} \) exceeds the maximum value that can be stored in a 32-bit integer this would cause an arithmetic overflow and a system error in computers.

A strict ordering can be achieved by changing \( \leq \) to \( < \). Number of constraints generated by this encoding is 1.

3.6 Harvey Lex Encoding [8]

This encoding is presented by [8] who attribute it to Warwick Harvey. The general formula as presented by the source is

\[
\]

To remove the ellipsis and encode the decomposition in Minizinc, we introduce \( X[i] \), a Boolean array used to eliminate common sub-expressions, where \( i \) is an index with possible values from 1 to \( n - 1 \).

\[
X[1]
\]

\[
X[n] = (A[n] < (B[n] + 1))
\]

\[
0 \leq i \leq n - 2 \quad X[n-i-1] = A[n-i-1] < (B[n-i-1] + Bool2Int(X[n-i]))
\]

We get a strict version by changing \( B[n] + 1 \) to \( B[n] + 0 \) in the above formula.

Translation from MiniZinc to SMT using mzn2smt produces the following.\( \text{int}[i] \) is an integer array introduced by fzn2smt to encode the Bool2Int function of MiniZinc. \( \text{int}[i] \) has a domain of 0 to 1 and a size of 1 to \( n \)

\[
X[1]
\]

\[
X[n] \leftrightarrow ((A[n] - B[n]) \leq 0)
\]

\[
1 \leq i \leq n \quad \text{int}[i] \leq 1
\]

\[
1 \leq i \leq n \quad \text{int}[i] \geq 0
\]

\[
1 \leq i \leq n - 1 \quad X[i + 1] \to (\text{int}[i] = 1)
\]

\[
1 \leq i \leq n - 1 \quad \neg X[i + 1] \to (\text{int}[i] = 0)
\]

\[
1 \leq i \leq n - 1 \quad X[n-i] \leftrightarrow ((A[n-i] - B[n-i] - \text{int}[i]) \leq -1)
\]

The translated Harvey encoding generates \( 5n - 1 \) constraints.
3.7 Alpha Lex Encoding [9]

We call this encoding Alpha, because it uses a Boolean array as an index to track the relations between values. This Boolean array is called \( \alpha[i] \) and behaves as follows:
- if \( \alpha[i] = 1 \) then \( A[j] = B[j] \) for all \( 1 \leq j \leq i \leq n \)
- and
- if \( \alpha[i] = 1 \) and \( \alpha[i+1] = 0 \) then \( A[i+1] < B[i+1] \)

This makes all values from \( A[i] \) to \( B[i] \) held, and equal to 0 from the first occurrence of \( A[i] < B[i] \) till the end of vectors. This encoding could be changed to a strict lex by adding the constraint \( \alpha[n+1] = 0 \).

\( \alpha \) is a Boolean matrix so \( \alpha[i] = 1 \) is equivalent to \( \alpha[i] = true \) and \( \alpha[i] = 0 \) is equivalent to \( \alpha[i] = false \).

\[
\begin{align*}
\alpha[0] &= 1 \\
0 \leq i &\leq n-1 \quad \alpha[i] = 0 \Rightarrow (\alpha[i+1] = 0) \\
1 \leq i &\leq n \quad (\alpha[i] = 1 \Rightarrow (A[i] = B[i])) \\
0 \leq i &\leq n-1 \quad ((\alpha[i] = 1) \land (\alpha[i+1] = 0)) \Rightarrow (A[i+1] < B[i+1]) \\
0 \leq i &\leq n-1 \quad (\alpha[i] = 1 \Rightarrow (A[i+1] \leq B[i+1]))
\end{align*}
\]

After translation using mzn2smt:

Here we use \( \alpha' \) instead of \( \alpha \) because the mzn2smt translator changed the range of \( \alpha \) from \( 0 \leq i \leq n \) to \( 1 \leq i \leq n+1 \)

\[
\begin{align*}
\alpha'[0] &= 1 \\
1 \leq i &\leq n+1 \quad (T_1[i] \Leftrightarrow \neg \alpha'[i+1]) \\
1 \leq i &\leq n \quad (T_2[i] \Leftrightarrow (A[i] = B[i])) \\
1 \leq i &\leq n \quad (T_3[i] \Leftrightarrow (A[i] \leq B[i])) \\
1 \leq i &\leq n \quad (T_4[i] \Leftrightarrow (A[i] \geq B[i])) \\
1 \leq i &\leq n \quad (T_5[i] \Leftrightarrow (\alpha[i] \land T_1[i+1])) \\
1 \leq i &\leq n \quad (T_6[i] \Leftrightarrow (\neg T_1[i] \lor T_1[i+1])) \\
1 \leq i &\leq n \quad (T_7[i] \Leftrightarrow (\neg T_3[i] \lor T_3[i])) \\
1 \leq i &\leq n \quad (T_8[i] \Leftrightarrow (\neg T_4[i])) \\
1 \leq i &\leq n \quad (T_9[i] \Leftrightarrow (\neg \alpha'[i+1] \lor T_2[i]))
\end{align*}
\]

Number of constraints generated by this encoding is \( 9n+2 \)

3.8 Alpha M Encoding [7]

This decomposition, which we call Alpha M, is the default decomposition used by Minizinc [7]. Like the previous Alpha encoding it uses a binary array as a bookkeeping mechanism for relations between the corresponding values in both vectors. The index of the Alpha array ranges from 1 to \( n+1 \).
8 Encoding the Lexicographic Ordering Constraint in SAT Modulo Theories

\[ \alpha[1] = 1 \]
\[ 1 \leq i \leq n \quad \alpha[i] = ((A[i] < B[i]) \lor \alpha[i + 1]) \land (A[i] \leq B[i]) \]

\( \alpha[i] = 1 \) makes sure that \( A[i] \leq B[i] \) is true. We obtain a strict version by adding \( \alpha[n + 1] = 0 \) to the constraints.

After translation using mzn2smt:

\[ \alpha[1] \] (52)
\[ 1 \leq i \leq n \quad T_1[i] \Leftrightarrow (A[i] \leq B[i]) \] (53)
\[ 1 \leq i \leq n \quad T_2[i] \Leftrightarrow (A[i] < B[i]) \] (54)
\[ 1 \leq i \leq n \quad T_3[i] \Leftrightarrow (T_2[i] \lor \alpha[i + 1]) \] (55)
\[ 1 \leq i \leq n \quad \alpha[i] \Leftrightarrow (T_3[i] \land T_1[i]) \] (56)

Number of constraints generated by this encoding is \( 4n + 1 \)

4 The Social Golfers Problem

The Social Golfers Problem is a computational problem of partitioning a set of golfers into \( g \) groups of size \( s \) in each of \( w \) weeks such that no two players meet more that once in the same group. An instance of the Social Golfer problem is usually denoted \( g - s - w \), which stand for number of groups, the group size and number of weeks. We use \( m = g \ast s \) to denote the number of players.

The table below shows one possible solution to the instance 3-2-3, where rows and columns represent players and weeks respectively, and each value in the table denotes a group number. So as an example column 2 can be interpreted as follows; In Week 2, Player 1 and Player 3 meet in the first group, Player 2 and Player 5 meet in the second group and Player 4 and Player 6 meet in the third.

<table>
<thead>
<tr>
<th>Week 1</th>
<th>Week 2</th>
<th>Week 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Player

Player 1

Player 2

Player 3

Player 4

Player 5

Player 6

The Social Golfer is known for its highly symmetric models. We use the lex constraint to break two groups of symmetries in the problem; symmetries in weeks and symmetries in players. The MiniZinc model that we used for the problem maps Players and Weeks to groups in an array as above. Symmetry among the players is broken by constraining the rows to be in lex increasing
order and symmetry among the weeks is broken by constraining the columns to be in lex increasing order.

The model that we used for the SGP is a modified model created by H. Kjellerstrand [10] and it has two constraints: The first is to make all groups contain \( s \) players, while the second is to make sure that each two players play together at most once in each week.

\( \text{Schedule}[i] \) is a two dimensional integer array that holds the weekly assignment of players to groups. Each group has exactly \( s \) players:

\[
1 \leq \text{group} \leq g \quad 1 \leq \text{week} \leq w \quad \left( \sum_{\text{player}=1}^{m} \text{Bool2Int}(\text{Schedule}[\text{player}, \text{week}] = \text{group}) \right) = s
\]

Where \( \text{Bool2Int}() \) is Boolean to integer converter function and \( m = \text{number of players} \), which equals to \( g \times s \).

Each pair of players only meets at most once

\[
1 \leq pa \leq m \quad 1 \leq pb \leq m \\
1 \leq wa \leq w \quad 1 \leq wb \leq w \\
\text{where } pa \neq pb \land wa \neq wb
\]

\[
(\text{Schedule}[pa, wa] \neq \text{Schedule}[pb, wa]) \lor \\
(\text{Schedule}[pa, wb] \neq \text{Schedule}[pb, wb])
\]

For any two distinct players, \( pa \) and \( pb \), and any two distinct weeks, \( wa \) and \( wb \), players \( pa \) and \( pb \) cannot play in the same group in both week \( wa \) and \( wb \).

From the assignment array \( \text{Schedule}[i] \) it is clear that symmetries can happen between weeks and between players. To break symmetry between weeks we put lex constraint ordering between each two neighbouring columns and the same is done for players.

Lex constraint on weeks:

\[
1 \leq \text{week} \leq w - 1 \quad [\text{Schedule}[\text{player}, \text{week}] \mid \text{player} \in 1..m] \leq_{\text{lex}} \\
[\text{Schedule}[\text{player}, \text{week} + 1] \mid 1\text{player} \in 1..m]
\]

Lex constraint on players:

\[
1 \leq \text{player} \leq m - 1 \quad [\text{Schedule}[\text{player}, \text{week}] \mid \text{week} \in 1..w] \leq_{\text{lex}} \\
[\text{Schedule}[\text{player} + 1, \text{week}] \mid \text{week} \in 1..w]
\]

5 Benchmarks

The MiniZinc implementation includes a set of libraries to decompose global constraints and made to be called from MiniZinc models. We created a MiniZinc global library for each of the seven lex decompositions, then we called them from the Social Golfer MiniZinc code. We modified a MiniZinc model created by
H. Kjellerstrand [10] for the problem by removing his implementation of symmetry breaking code and adding a symmetry breaking based on lexicographical orderings constraint. All benchmarks were run on a Windows PC with Intel i7 1.8Ghz processor and 8GB of RAM and using Yices 2.2.1 as an SMT solver. We ran 30 samples for each instance, each sample is created by choosing a random ordering of the constraints from a uniform distribution over all orderings of the sample SMT file. Each figure in the following two tables represents an average of 30 timings for each encoding on each instance. We only used satisfiable instances obtained from [11]. We ran two sets of benchmarks, the first is for samples translated to SMT using the mzn2smt pipeline, while the second is for the same samples but translated directly from the each of the 7 presented encodings in this paper. Both sets of benchmarks share the same code for the SGP, the difference is only in the code related to different the lex constraint encodings.

Table 5 shows the timings in seconds for each of the encodings on instances of the SGP produced by mzn2smt. As it can be seen from these results the OR encoding takes the lead by a very narrow margin, it also seems that using CSE did not made any improvement to the timings of AND and OR encodings.

<table>
<thead>
<tr>
<th>Instances</th>
<th>AND</th>
<th>AND</th>
<th>OR</th>
<th>OR</th>
<th>Alpha</th>
<th>Alpha M</th>
<th>Harvey</th>
</tr>
</thead>
<tbody>
<tr>
<td>G-S-W</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5-3-5</td>
<td>0.27</td>
<td>0.25</td>
<td>0.28</td>
<td>0.26</td>
<td>0.25</td>
<td>0.27</td>
<td>0.33</td>
</tr>
<tr>
<td>5-3-6</td>
<td>1.40</td>
<td>1.79</td>
<td>1.70</td>
<td>2.02</td>
<td>1.60</td>
<td>1.59</td>
<td>1.94</td>
</tr>
<tr>
<td>5-3-7</td>
<td>12.53</td>
<td>13.08</td>
<td>11.07</td>
<td>13.94</td>
<td>13.47</td>
<td>14.77</td>
<td>16.71</td>
</tr>
<tr>
<td>6-3-5</td>
<td>0.27</td>
<td>0.27</td>
<td>0.25</td>
<td>0.26</td>
<td>0.26</td>
<td>0.29</td>
<td>0.32</td>
</tr>
<tr>
<td>6-3-6</td>
<td>1.63</td>
<td>1.64</td>
<td>1.70</td>
<td>1.79</td>
<td>1.69</td>
<td>1.55</td>
<td>1.87</td>
</tr>
<tr>
<td>6-3-7</td>
<td>6.77</td>
<td>6.72</td>
<td>7.14</td>
<td>7.32</td>
<td>6.10</td>
<td>6.84</td>
<td>6.81</td>
</tr>
<tr>
<td>6-4-4</td>
<td>0.63</td>
<td>0.63</td>
<td>0.62</td>
<td>0.63</td>
<td>0.66</td>
<td>0.64</td>
<td>0.68</td>
</tr>
<tr>
<td>6-4-5</td>
<td>5.32</td>
<td>4.82</td>
<td>5.01</td>
<td>4.53</td>
<td>4.97</td>
<td>4.88</td>
<td>4.89</td>
</tr>
<tr>
<td>8-4-4</td>
<td>1.08</td>
<td>0.98</td>
<td>1.11</td>
<td>1.17</td>
<td>1.09</td>
<td>1.14</td>
<td>1.20</td>
</tr>
<tr>
<td>8-4-5</td>
<td>10.88</td>
<td>10.80</td>
<td>10.42</td>
<td>9.73</td>
<td>10.76</td>
<td>9.63</td>
<td>10.80</td>
</tr>
<tr>
<td>8-4-6</td>
<td>85.45</td>
<td>87.43</td>
<td>79.19</td>
<td>90.63</td>
<td>84.18</td>
<td>93.65</td>
<td>83.29</td>
</tr>
</tbody>
</table>

| Arithmetic mean | 11.48 | 11.67 | 10.77 | 12.03 | 11.37 | 12.29 | 11.71 |
| Geometric mean  | 2.66  | 2.67  | 2.65  | 2.77  | 2.66  | 2.73  | 2.97  |

Table 1. Solution timings (in Seconds) for instances of The SGP using fzn2smt translation

Table 5 is for the directly translated samples. Here Alpha encodings takes a marginal lead while Harvey encoding came last by a relatively wide gap. Compared to the mzn2smt translation results, the only improvement is in AND encoding results.
6 Conclusion and Future Work

Although the averages of the results provide no clear judgement about which encoding is better, from individual results some conclusions still can be made. For example, in 5 on the instance 5-3-7 Alpha encoding performed better than OR CSE, while on the instance 8-4-6 it was the opposite. So conjoining both encodings into a single one might perform better on both instances. This variability also could be a good property for building a portfolio of encodings, where an instance tackled using multiple encodings in parallel and get solved first by its most efficient encoding.

This research is still a work in progress and for the next stages we plan to answer some of the questions that have arisen. For instance, the question of the effect of conjoining two or more encodings and examining which SMT encodings obtain Generalized Arc Consistency and its impact on performance. Also the main question of our study, which is; Can we do better by directly translating from MiniZinc instead of going through the mzn2smt?. We are also planning to study other global constraints and our next nominee is the value precedence constraint, which is useful in breaking value symmetries.

References

Encoding the Lexicographic Ordering Constraint in SAT Modulo Theories


Anomalies in SMT Solving:
Difficulties in Modelling Combinatorial Problems

Alan M. Frisch\textsuperscript{1} and Miquel Palahi\textsuperscript{2}
\textsuperscript{1} University of York, UK
\textsuperscript{2} Universitat de Girona, Spain

Abstract Much research on modelling combinatorial problems for a solver compares alternative models of a problem in an attempt to understand the characteristics of good models. The aim is to discover principles and heuristics that in the future would guide someone to develop good models or to select the more effective model from a set of alternatives. For many years this methodology has been moderately successful in studying modelling for SAT solvers and for finite-domain constraint solvers. Our attempts to apply this methodology to SMT solving have been hindered by the apparently erratic behaviour of SMT solvers. This paper presents four of the more extreme anomalies that we have encountered in our work. Of course we label these phenomena as anomalies because we have no explanation for them and they run counter to our intuitions. We bring these anomalies to light in an attempt to motivate the research community to try to develop a better understanding of SMT solving and modelling.

1 Introduction

SAT modulo theory (SMT) solvers combine the powerful technology of modern SAT solvers with specialised reasoning that supports particular theories. SMT was originally developed for verification problems and has proven to be highly effective in that domain. Some recent effort has been invested in solving combinatorial decision and optimisation problems with SMT, particularly exploiting specialised reasoning over linear integer arithmetic.

The most significant study [1] of using SMT for combinatorial problems developed an automatic system to translate constraint problem instances expressed in FlatZinc to SMT. Over a suite of 294 instances of 32 problems the solve times of the Yices 2 SMT solver—predominantly using QF\textsubscript{LIA}, the theory of quantifier free linear integer arithmetic—were competitive with those of leading finite-domain constraint solvers.

The strong performance of the SMT solver is both striking and highly promising when one considers the context in which the comparison was made. Firstly, the source language of the problem instances, FlatZinc, is designed primarily with finite-domain constraint solvers in mind; this could disadvantage the SMT
solver. Secondly, the efforts of the Girona group focused mostly, though not exclusively, on exploring the breadth of SMT’s capabilities rather than on how best to encode combinatorial problems in SMT.

The line of research we have been pursuing is complementary to that of the Girona group. We aim to better understand how best to model combinatorial problems in SMT, free from the language of FlatZinc and particular FlatZinc problem encodings. Our work towards this goal has used two two methodologies that have proven effective for both finite-domain constraint solvers and SAT solvers. One is to discover how best to encode particular problems and from this draw generalisations that could be applied to modelling other problems. The other approach is to consider how best to encode particular components that commonly arise in combinatorial problems, such as finite domains or certain constraints. In both cases the goal is form general principles or patterns that can be used in the future to build effective encodings of other problems.

Disappointingly and surprisingly our attempts to identify patterns and generalisations were hampered by erratic behaviour of SMT solvers. An encoding that worked well for one solver would be poor for another. Changes to encodings that practitioners of finite-domain constraint programming would expect to be improve performance, such as tightening bounds on the objective function, sometimes proved to hamper performance.

This paper reports on the anomalous behaviour of SMT solvers using QF_LIA that has impeded our progress towards constructing an understanding of how best to model combinatorial problems for SMT solvers. The work reported here is solely concerned with SMT using the QF_LIA theory and we refer to this combination as SMT(QF_LIA). We investigate four SMT(QF_LIA) solvers and use all with their default settings.

Problem instances and data associated with this research can be found at http://www.cs.york.ac.uk/aig/constraints/SMT/.

2 Anomalies in Representing Finite Domains

The first two anomalies arise in representing finite integer domains and considering two kinds of propagation.

Throughout we consider the representation of a variable $x$ with domain $D = \{d_1, \ldots, d_n\}$ where $d_1 \leq d_2 \leq \cdots \leq d_n$. We use $\overline{D}$ to denote $\{d' | d_1 < d' < d_n \text{ and } d' \notin D\}$.

To simplify the presentation we use the following schemas:

- $ALO(x) \stackrel{\text{def}}{=} \bigvee_{d \in D} x = d.$
- $BOUND(x) \stackrel{\text{def}}{=} (d_1 \leq x \leq d_n).$
- $NEG(x) \stackrel{\text{def}}{=} \bigwedge_{d \in \overline{D}} \neg(x = d).$

In all the experiments of Anomalies 1 and 2 we have used the following four SMT(QF_LIA) solvers: Yices 1.0.35 [6], Yices 2.1.0, Z3-4.1 [5] and MathSat 5.1.10 [2]. The input formulas for the first three solvers are written using
SMT-Lib 1.2 and using SMT-Lib 2 for the fourth. The experiments were run on a cluster of Intel® Xeon™ CPU@3.1GHz machines, with 8GB of RAM, under 64-bit CentOS release 6.3 (kernel 2.6.32).

2.1 Anomaly 1

Anomaly 1 arises in a test to measure how fast different finite domain representations are in recognising unsatisfiability when all of the domain values are eliminated.

Here we consider the domain of variable $x$ to be the contiguous set of values $1..n$. The domain values are all eliminated by asserting the following formula:

$$\text{UNSAT}_\text{ALL}(x) \overset{\text{def}}{=} \bigwedge_{d \in D} \neg(x = d)$$

We conjoin this with each of three domain representations resulting in three test cases

1. $\text{ALO}(x) \land \text{UNSAT}_\text{ALL}(x)$
2. $\text{BOUND}(x) \land \text{UNSAT}_\text{ALL}(x)$
3. $\text{BOUND}(x) \land \text{ALO}(x) \land \text{UNSAT}_\text{ALL}(x)$

We measured the solve time of each of these formulas on each of the four SMT($\text{qf}_{\text{lia}}$) solvers. Each of Figures 1–4 shows the run time as a function of domain size, $n$, of one solver on the three domain encodings. The first three of these figures show that each of Yices 1, Yices 2 and MatSat 5 perform similarly on the three encodings. Indeed, in each case two of the curves overlay each other almost perfectly. MathSat 5 is remarkable in that the solve time is almost independent of both domain size and encoding.

The anomaly occurs with Z3; as domain size increases the bounds-only encoding performs far worse than the other two encodings, which behave almost identically. Notice that the scale in Fig. 4 goes over 100 seconds, whereas the scales in the other plots go to only 10 seconds. More remarkably, with the bounds-only representation Z3 performs search. For example, with domain size 10000 it reports having made 5843 decisions and 5460 conflicts, while in the ALO representations does not report any information about decisions or conflicts.

In contrast, the other solvers don't report having made any decisions as expected, but they report different number of conflicts. On all problem instances Yices 1 reports 1 conflict, Yices 2 does not report any statistic when the instance is unsatisfiable and MathSat 5 reports 1 conflict with the bounds-only representation and 0 conflicts in the ALO representations. Finally, MathSat 5 is the only solver reporting calls to theory solvers, calling 5001 times the linear arithmetic theory with the domain size 10000. A summary of the reported conflicts and calls to the linear arithmetic theory solver by the SMT solvers can be found in Table 1.
2.2 Anomaly 2

Anomaly 2 arises in a test to measure how fast different domain representations are in recognising that a variable has a particular value because all but one of it’s domain values are eliminated.

Here we consider the domain of variable $x$ to be the first $n$ odd natural numbers, $\{1, 3, 5, \ldots, 2n - 1\}$. All but one of the domain values are eliminated by asserting the following formula:

\[
ONLY1(x, v) \overset{\text{def}}{=} \bigwedge_{d \in D \backslash \{v\}} \neg(x = d)
\]

We have evaluated the performance of four solvers in determining the satisfiability of $ONLY1(x, d_n)$ when conjoined with each of four domain represent-
Table 1. Anomaly 1: Number of conflicts and number of calls to the linear arithmetic theory solver (between parenthesis) for instance with domain size 10000 for each SMT solver. n/a means that the solver has not reported any information.

<table>
<thead>
<tr>
<th></th>
<th>Bounds</th>
<th>ALO</th>
<th>Bounds + ALO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yices 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Yices 2</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Z3</td>
<td>5460</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>MathSat 5</td>
<td>(5001)</td>
<td>0 (0)</td>
<td>0 (0)</td>
</tr>
</tbody>
</table>

The main anomaly here is that the solve time of Z3 does not increase monotonically with \( n \). Furthermore, Z3 performs search in solving this simple problem and the metrics that quantify this search also do not increase monotonically. As shown in Figure 6, their non-monotonic behaviour tracks that of the runtime measurement. In contrast, the three other solvers report a constant number of conflicts and decisions; on all instances Yices 1 and Yices 2 report 0 conflict and MathSat 5 reports 1 conflict. MathSat makes \( 2n + 3 \) calls to the QF_LIA theory solver on instances with domain size \( n \); the other solvers do not report this statistic.

### 3 Anomalies in representing the Pedigree Reconstruction Problem

The second two anomalies arise in representing the Pedigree Reconstruction Problem (PRP)[4], which involves identifying relatives amongst a group \( G \) of individuals from genetic marker data.

In particular, the goal here is to find the maximum likelihood pedigree. A pedigree for \( G \) assigns to each individual \( i \) in \( G \) a parent set, which takes one of three forms:

- \( j, k \), where \( i, j \) and \( k \) are distinct members of \( G \). This indicates that the parents of \( i \) are \( j \) and \( k \).
- \( j \), where \( i \) and \( j \) are distinct members of \( G \). This indicates that \( j \) is a parent of \( i \) and the other parent of \( i \) is not in \( G \).
- \( \emptyset \), which indicates that neither parent of \( i \) is in \( G \).
For each member \( i \in G \), the genetic markers of the members of \( G \) determine a probability distribution over all possible parent sets of \( i \). In typical problem instances most potential parent sets of \( i \) have probability zero.

The probability of a pedigree is the product of the probability that each \( i \in G \) has the parent set assigned by the pedigree.

Every assignment of individuals to parent sets is not a valid pedigree. Sexual reproduction imposes two constraints:

**acyclicity:** No individual can be an ancestor of itself, and

**gender consistency:** The two parents of an individual have opposite gender.

The genetic marker data does not identify the gender of the individuals, so gender consistency requires that a gender could be assigned to each individual so that the parents of every individual have opposite genders. As an example, an assignment in which \( a \) and \( b \) have a child, \( b \) and \( c \) have a child, and \( a \) and \( c \) have a child is not gender consistent.

Summing up, the pedigree reconstruction problem is: Given \( G \) a finite set of individuals and for each \( i \in G \) a probability distribution over all possible parent sets of \( i \), find a maximum likelihood assignment \( A \) of a parent set to each \( i \in G \) such that \( A \) is acyclic and gender consistent.

Following Cussens [4] we simplify our model by assuming that we are given not the probability of each parent set but rather the log of that probability. This enables us to use a linear objective function: the sum of the logarithm of the probability of each assigned parent set. This value is to be maximised.

Table 2 shows the basic model written in SMT\( (QF_{LIA}) \) of the PRP. The model represents the individuals of \( G \) by the integers 1..\( n \). For each individual
For each individual $i$ the model has a decision variable $parentset_i$ such that assigning it $p$ indicates the decision to assign individual $i$ the parentset $pps_i[p]$. The decision variable $globalValue$ is the objective value to be maximised. It is the sum of values of $localValue_i$ ($1 \leq i \leq n$) — see constraint (c3) — where $localValue_i$ is constrained by (c5) to be equal to $logLikelihood_i[parentset_i]$. Constraints (c1) and (c4) bound the $parentset_i$ and $globalValue$ variables, in effect giving each a finite domain. Constraint (c2) imposes an upper bound on the $localValue_i$ variables; the use of a lower bound is an issue discussed below.

Acyclicity is enforced by associating a generation number, the decision variable $gen_i$, with each individual $i$. The generation number of an individual is constrained to be one greater than the maximum generation number of its parents. This is stipulated in (c7) if $i$ has two parents and in (c8) if $i$ has one parent. Constraint (c6) bounds each $gen_i$, giving it a finite domain.

Finally, gender consistency is enforced by associating a gender, the boolean decision variable $female_i$ with each individual. The only constraint is that if an individual has two parents then those parents must have opposite gender. This is stipulated in the last conjunct of (c7).
Given

\[ n: \text{positive integer} \]
\[ k_i: \text{positive integer} \quad (1 \leq i \leq n) \]
\[ pps_i[1..k_i]: \text{set maxsize 2 drawn from 1..n} \quad (1 \leq i \leq n) \]
\[ logLikelihood_i[1..k_i]: \text{positive integer} \quad (1 \leq i \leq n) \]

**Decision Variables**

\[ \text{parentset}_i: \text{int} \quad (1 \leq i \leq n) \]
\[ \text{gen}_i: \text{int} \quad (1 \leq i \leq n) \]
\[ \text{localValue}_i: \text{int} \quad (1 \leq i \leq n) \]
\[ \text{globalValue}: \text{int} \]
\[ \text{female}_i: \text{bool} \quad (1 \leq i \leq n) \]

**Constraints**

(c1) \( 1 \leq \text{parentset}_i \land \text{parentset}_i \leq k_i \quad (1 \leq i \leq n) \)

(c2) \( \text{localValue}_i \leq \max_{j \in [1..k_i]} logLikelihood_i[j] \quad (1 \leq i \leq n) \)

(c3) \( \text{globalValue} = \sum_{i=1}^{n} \text{localValue}_i \)

(c4) \( \text{globalValue} \geq 0 \land \text{globalValue} \leq \sum_{i=1}^{n} \max_{j \in [1..k_i]} logLikelihood_i[j] \)

(c5) \( \text{not} (\text{parentset}_i = j) \lor \text{localValue}_i = logLikelihood_i[j] \quad (1 \leq i \leq n, 1 \leq j \leq k_i) \)

(c6) \( 0 \leq \text{gen}_i \land \text{gen}_i \leq n \quad (1 \leq i \leq n) \)

(c7) \( \text{not} (\text{parentset}_i = r) \lor \text{gen}_i - \text{gen}_p \geq 1 \land \)
\( \text{not} (\text{parentset}_i = r) \lor \text{gen}_i - \text{gen}_p \geq 1 \land \)
\( \text{not} (\text{parentset}_i = r) \lor \text{gen}_i - \text{gen}_p = 1 \lor \text{gen}_i - \text{gen}_{p'} = 1 \land \)
\( \text{not} (\text{parentset}_i = r) \lor \text{not} (\text{female}_p = \text{female}_{p'}) \)
\( (1 \leq i \leq n, 1 \leq j \leq k_i, pps_i[j] = \{p, p'\}) \)

(c8) \( \text{not} (\text{parentset}_i = r) \lor \text{gen}_i - \text{gen}_p \geq 1 \quad (1 \leq i \leq n, 1 \leq j \leq k_i, pps_i[j] = \{p\}) \)

**Objective**

maximize \( \text{globalValue} \)

**Table 2. Basic SMT(qf lia) model of the pedigree reconstruction problem**
The basic model imposes used upper and lower bounds on the variables \textit{parentset} and \textit{gen}, but only an upper bound on the variable \textit{localValue}. This is because during our experiments we have detected surprising behaviour when we use a lower bound on that variable. Therefore, to study this behaviour, we have defined two variants of constraint (c2):

- \((c2^+)\) when in addition to \((c2)\) we also use the tighter lower bound:

  \[
  \text{localValue}_i \geq \min_{j \in \{1..k\}} \text{logLikelihood}_i[j]
  \]

- \((c2^0)\) when in addition to \((c2)\) we also use a weaker lower bound:

  \[
  \text{localValue}_i > 0
  \]

In some of the experiments we extended the model to use three additional constraints to represent the domain of the variables:

- \((c9)\) DomainALO(\textit{parentset}): \(\bigvee_{j \in \{1..k\}} \text{parentset}_i = j \ (1 \leq i \leq n)\)
- \((c10)\) DomainALO(\textit{gen}): \(\bigvee_{j=0..n} \text{gen}_i = j \ (1 \leq i \leq n)\)
- \((c11)\) DomainALO(\textit{lc}): \(\bigvee_{j \in \{1..k\}} \text{localValue}_i = \text{logLikelihood}_i[j]\)

To simplify our study we replace the PRP optimisation problem with the corresponding decision problem. In particular, from each PRP instance we generate the hardest satisfiable instance and the hardest unsatisfiable instance. We precompute the global value, \(c^*\), of the optimal solution and create two benchmark problem instances: a satisfiable instance in which the objective is replaced with the constraint \(\text{globalValue} \geq c^*\) and an unsatisfiable instance in which the objective is replaced with the constraint \(\text{globalValue} > c^*\).

Notice that in both the satisfiable and unsatisfiable instances the constraints \((c10)\) and \((c11)\) are implied. To fully appreciate Anomaly 4 it is important to bear in mind that because these are implied constraints they prune no solutions.

All of our experiments use the same suite of 100 PRP instances generated randomly by pedsim [3], configured to use genetically realistic random distributions. Such instances are known to be harder to solve than those generated using random distributions. All of our instances contain 46 individuals.

In all the experiments of Anomalies 3 and 4 we have used the same four SMT(\textit{qf_lia}) solvers used in Anomalies 1 and 2. But the experiments were run on a slightly different computer, Intel\textsuperscript{\textregistered} Core\textsuperscript{TM} i5 CPU\textsuperscript{\textregistered}2.66GHz, with 3GB of RAM, under 32-bit openSUSE 11.2 (kernel 2.6.31).

### 3.1 Anomaly 3

Anomaly 3 arose through a surreptitious observation. Solve time can be reduced if we replace constraint \((c3)\) with

\[
(c3') \quad \text{globalValue}_2 = \sum_{i=1}^{n} \text{localValue}_i \land \text{globalValue}_2 = \text{globalValue}
\]

where \textit{globalValue} is a new integer variable.
This anomaly only appears in MathSat 5 and Yices 1 solvers when we are also using \((c_2^+)\) instead of \((c_2)\), and in the case of Yices 1 when we are also using ALO for the variables (constraints \((c_9), (c_{10})\) and \((c_{11})\)). It appears only in the satisfiable instances and it does not appear in the unsatisfiable ones. Table 3 and Table 4 show the mean time and median time with and without the extra variable. In both cases the extra variable reduces the median time about 20%.

<table>
<thead>
<tr>
<th></th>
<th>Mean time (s)</th>
<th>Median time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>non ex.</td>
<td>5.1283</td>
<td>2.085</td>
</tr>
<tr>
<td>extra</td>
<td>4.0067</td>
<td>1.56</td>
</tr>
</tbody>
</table>

**Table 3.** Anomaly 3: MathSat 5 solves the 100 instances about 20% faster with an extra variable in the basic model with tighter lower bound.

<table>
<thead>
<tr>
<th></th>
<th>Mean time (s)</th>
<th>Median time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>non ex.</td>
<td>1.2569</td>
<td>1.005</td>
</tr>
<tr>
<td>extra</td>
<td>1.0584</td>
<td>0.82</td>
</tr>
</tbody>
</table>

**Table 4.** Anomaly 3: Yices 1 solves the 100 instances about 20% faster with an extra variable in the basic model with tighter lower bound and ALO.

One would expect that adding this extra variable would have only a trivial affect on solution time. However, the two scatter plots, Figure 7 and Figure 8, show that the effect of adding an extra variable can drastically increase or decrease the solving time of an instance. More precisely, in Figure 7 when adding the extra variable there is a variability on solving one instance from 52.9 times faster (decreasing the solving time from 35.92 to 0.68) to 32.4 times slower (increasing the solving time from 0.86 to 27.87). In Figure 8 this variability is smaller, ranging from 4.6 times faster (decreasing the solving time from 5.1 to 1.11) to 2.0 times slower (increasing the solving time from 1.51 to 3.02).

### 3.2 Anomaly 4

The last anomaly arises in test to measure in the basic model which way to represent the lower bound is the best for Z3 solver: without any lower bound \((c_2)\), with a tighter lower bound \((c_2^+)\) or with a lighter lower bound \((c_{2^0})\). Again the first impression is that \(c_2^+\) has to be the best option, but in practise the best option is \(c_2\), that is not using any lower bound. This is shown at Table 5 where we can see the mean and median times of solving the 100 instances for the three experiments. The first two columns are the solving times for the satisfiable instances and the two last columns are the times for the unsatisfiable instances. We can see that solving the basic model with \(c_2\) is about 5 to 6 times faster than with \(c_2^+\) or \(c_{2^0}\) in the satisfiable case. A similar phenomenon happens with the unsatisfiable instances, where the basic model with \(c_2\) is about 3 to 4 times faster then the other two representations.

We want to note that this anomaly disappears if we add ALO to represent localValue variable domain (adding constraint \(c_{11}\)).

Finally, we present four scatter plots comparing \(c_2\) with \(c_2^+\) and \(c_{2^0}\) in Figure 9 and Figure 10 respectively for the satisfiable case and in Figure 11 and
Figure 7. Anomaly 3: scatter-plot of MathSat 5 with and without extra variable in the basic model with tighter lower bound. 57 blue crosses and 43 red crosses.

Figure 8. Anomaly 3: scatter-plot of Yices 1 with and without extra variable in the basic model with tighter lower bound and ALO. 53 blue crosses, 43 red crosses and 12 black crosses.

<table>
<thead>
<tr>
<th></th>
<th>Satisfiable</th>
<th>Unsatisfiable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean time (s)</td>
<td>Median time (s)</td>
</tr>
<tr>
<td>$c^2^+$</td>
<td>29.94</td>
<td>31.73</td>
</tr>
<tr>
<td>$c^2^-$</td>
<td>35.43</td>
<td>34.64</td>
</tr>
<tr>
<td>$c^2$</td>
<td>7.06</td>
<td>5.59</td>
</tr>
</tbody>
</table>

Table 5. Anomaly 4: Z3 solves the satisfiable instances about 5 to 6 times faster in the basic model without any lower bound compared to using any lower bound, and about 3 to 4 times faster for the unsatisfiable instances.

Figure 12 respectively for the unsatisfiable case. What is interesting of these plots is that in all the cases $c^2$ is better in all the instances (especially for the unsatisfiable case). More precisely, $c^2$ is better than $c^2^+$ between 1.18 and 18.90 times in the satisfiable case and between 1.38 and 9.46 times better in the unsatisfiable case, and $c^2$ is better than $c^2^0$ between 1.17 and 22.56 times in the satisfiable case and between 1.54 and 9.71 times better in the unsatisfiable case.

4 Conclusions

The behaviours of the SMT($qf_{lia}$) solvers presented in this paper are anomalous in that they contradict our expectations, which have been formed from our experience with modelling problems for CP solvers. In Anomaly 1 we see that a simple, obvious representation of finite domains is problematic for one SMT($qf_{lia}$) solver, though not for three others.

Anomaly 2 arises in considering parameterised problem instances in which solving larger instances requires a superset of the reasoning involved in solving
**Figure 9.** Anomaly 4: scatter-plot of $Z_3$ with tighter lower bound ($c^+$) and without lower bound ($c$) in the basic model for the satisfable case.

**Figure 10.** Anomaly 4: scatter-plot of $Z_3$ with lower bound greater than 0 ($c^0$) and without lower bound ($c$) in the basic model for the satisfable case.

**Figure 11.** Anomaly 4: scatter-plot of $Z_3$ with tighter lower bound ($c^+$) and without lower bound ($c$) in the basic model for the unsatisfable case.

**Figure 12.** Anomaly 4: scatter-plot of $Z_3$ with lower bound greater than 0 ($c^0$) and without lower bound ($c$) in the basic model for the satisfable case.
smaller instances. Nonetheless, for one solver the solution time is not monotonic in the size of the problem instance.

In Anomaly 3 a trivial change to a model, that would have little to no effect on a CP solver, can greatly reduce the solve time of one instance while greatly increasing that of another instance. Though we would expect any change in performance to be a very small negative one, we see an average improvement of 20% with two of the solvers.

Finally, Anomaly 4 is perhaps the most baffling to anyone trying to build better models. Here we see a case where tightening bounds on a variable impedes performance even in unsatisfiable instances.

In seeking an explanation of these anomalies one must start with the observation that the behaviour of SMT solvers can be chaotic; a small change in the model can result in a large, seemingly random change in the behaviour of the solver. For example, it is well known that changing the order of the constraints in an SMT model can greatly increase or decrease the solution time.

This chaos is certainly present in Anomaly 3 where we see the addition of an extra variable can greatly increase or decrease solution time. However, we believe that beneath the noise there is still a significant difference in the average solution time. To assert this with greater confidence we would need to gather a large sample size and conduct a statistical test.

It is difficult to see how chaos could play a significant role in the other anomalies, especially so in Anomaly 4 where the tighter bounds increases solution. This increase is observed on 100 out of 100 satisfiable instances and 100 out of 100 unsatisfiable instances. And this behaviour is repeated for two forms of tighter bounds. So the phenomenon is robust, being observed 100 times out of 100 in each of 4 different settings. The anomaly arises in spite of chaos, not because of chaos.

Perhaps there are explanations for all these behaviours, but, from what we currently know, developing an understanding of what makes a good SMT model appears to be an enormous challenge. Anyone working on SMT modelling of combinatorial problems needs to proceed cautiously, expect the unexpected and be prepared for difficulties.

In the future we hope to work with the developers of SMT solvers to try to develop an understanding of how modelling choices affect solver performance and to consider whether SMT solvers can be tuned to perform better on combinatorial problems. In future experiments we plan to reduce the affect of chaos by averaging over large samples of instances that have their constraints randomly ordered.

**Acknowledgement**

We thank James Cussens and Mark Bartlett for explaining the pedigree reconstruction problem and supplying us with their problem instance generator. We are grateful to the referees of this paper for their valuable feedback. Miquel Palahi has been supported by the Spanish Ministry of Science and Innovation (project
TIN2012-33042) the Universitat de Girona (grant BR 2010) and the Economy and Knowledge Department (Generalitat de Catalunya) and SUR (BE-DGR 2012).

References