Proceedings of SymCon'02

The Second International Workshop on Symmetry in Constraint Satisfaction Problems

A workshop at the Eighth International Conference on Principles and Practice of Constraint Programming (CP'02)

Sunday 8 September 2002

Cornell University, Ithaca, NY, USA

Edited by:

Pierre Flener and Justin Pearson
Uppsala University
Department of Information Technology
Box 337
751 05 Uppsala
Sweden

Preface

There are many constraint satisfaction problems that have a great deal of symmetry. For example, when colouring a graph, many of the solutions are related by rotating the colours. Similarly, when solving a tournament timetabling problem, it does not matter in which order the participants play the first games: many of the solutions are related by permuting the participants. Symmetry in constraint satisfaction problems can be exploited to increase the efficiency of search. This can be done by avoiding exploring paths that have already been shown to be a dead-end in a symmetrical part of the search tree, or by adding constraints so that (ideally) only one assignment per equivalence class is enumerated. This process is often referred to as *symmetry breaking* or *symmetry reduction*.

This workshop focuses on the analysis and development of techniques to detect and exploit symmetry in constraint satisfaction problems. It is the second workshop in a series that started with SymCon'01 at CP'01 in Paphos, Cyprus (see http://www.csd.uu.se/~pierref/astra/SymCon01/).

These proceedings bring together a number of recent papers, extended abstracts, and work-in-progress reports on the topic of symmetry. It is hoped that this snapshot of current research will act as a catalyst for further research. These proceedings are also on-line, at http://www.csd.uu.se/~pierref/astra/SymCon02/.

Uppsala, 6 August 2002

Pierre Flener and Justin Pearson
Uppsala University, Sweden
Programme Chairs

Programme Committee

Accepted Papers and Schedule

Papers (to be presented at the workshop)

Statements of Interest (not to be presented at the workshop) (full papers and presentations at the CP'02 main conference)

Symmetry-Breaking Constraints for Matrix Models

Zeynep Kiziltan[1] and Barbara M. Smith[2]

[1] Department of Information Science, Uppsala University, Sweden.
Zeynep.Kiziltan@dis.uu.se
[2] School of Computing and Engineering, University of Huddersfield, England.
b.m.smith@hud.ac.uk

Abstract. Many CSPs can be effectively represented and efficiently solved using matrix models, in which the matrices may have symmetry between their rows and/or columns. Eliminating all such symmetry can be very costly as there are in general exponentially many symmetries. Cost-effective methods have been proposed to break much of the symmetry, if not all. In this paper, we continue with this line of research, and propose several symmetry-breaking constraints. Experimental results confirm their value.

1 Introduction

Symmetry in a CSP model is an important issue as the exploration of symmetric but essentially equivalent branches in a search tree may significantly slow down the search process. This has interested many researchers in recent years, and several techniques have been developed to address the issue of eliminating symmetry in CSP models.

An important class of symmetries in constraint programming arises from matrices of decision variables where any two rows and any two columns can be interchanged [2]. For example, in the 2-d matrix model of the social golfers problem[1], the weeks are indistinguishable, and so are the groups. Any solution is thus symmetric to another obtained by interchanging any two rows representing two periods, and/or any two columns representing two weeks. In attempting to find all solutions[2], an exponential number of search states will be visited unnecessarily if row and column symmetries are not eliminated.

On the other hand, eliminating all symmetries is not so easy, since the effort required may also be exponential. In such a case, removing all symmetries is not cost-effective: the overhead introduced is as bad as the wasted search effort with no symmetry breaking. Methods to reduce significantly the row and column symmetry in matrix models with only a polynomial effort have been proposed. For instance, we can lexicographically order both the rows and columns [2],

[1] prob010 at www.csplib.org
[2] Note that symmetry elimination is beneficial in exhaustive search as opposed to in first-solution search [6].

or we can apply Symmetry Breaking During Search (SBDS) [5] on the columns and combine this with adding symmetry-breaking constraints on the rows before search starts. Such constraints must not get in the way of permuting the columns. One strategy is to insist that the vectors are ordered by their sums [7].

In this paper, we first explore other symmetry-breaking constraints that can be posed on a matrix model so as to remove a significant amount of row and column symmetry. We then empirically show the effectiveness of the proposed constraints in comparison with the related work.

2 Related work

In recent years, many techniques have been developed towards eliminating symmetry in CSPs models. One can for instance specify some constraints that are satisfied by a subset of the solutions within a set of equivalent symmetric solutions of a CSP model. Ideally, only one solution in an equivalence class satisfies these constraints, and thus all symmetries are eliminated. These constraints are called 'symmetry-breaking' constraints. There are at least two ways of posting symmetry-breaking constraints in constraint programming: by adding symmetry-breaking constraints to the model before search starts (e.g. [1]), or during search (e.g. SBDS [5]). In the rest of this paper, we will refer to the former when we talk about adding symmetry-breaking constraints.

In [2], it is shown that if a 2-d matrix model with row and column symmetry has a solution then it has a solution with the rows and columns ordered lexicographically. Hence, imposing lexicographic ordering constraints on both the rows and the columns (called double-lex) does not remove any unique solutions. The lexicographic ordering constraints are symmetry-breaking constraints and are posted between every adjacent pair of rows or columns. For an $n \times n$ matrix model with row and column symmetry, $O(n)$ symmetry-breaking constraints are posted. Whilst imposing lexicographic ordering constraints on the rows (columns) breaks all row (column) symmetry, double-lex does not break all row and column symmetries. However, experimental results show that double-lex is in practice effective at dealing with row and column symmetries.

In theory, SBDS can be used to eliminate all row and column symmetries of a matrix model. However, since, an $n \times m$ matrix has $n!m! - 1$ symmetries other than identity, breaking all symmetries would require as many SBDS functions, so SBDS can only be used for small matrices (e.g. 3×3 and 4×4). Therefore, in [7], using a subset of the full SBDS functions is proposed, to reduce rather than eliminate row and column symmetries in large matrices. Since row symmetry alone (or column symmetry alone) can entirely be eliminated by row (column) transpositions, a promising subset is the row transpositions *and* the column transpositions, which require only $O(n^2)$ SBDS functions for an $n \times n$ matrix. Adding all combinations of a row transposition and a column transposition to the row and column transpositions would eliminate even more symmetry. This, however, increases the number of SBDS functions to $O(n^4)$.

In [4], an implementation of SBDS combined with the GAP system (Groups, Algorithms and Programming) is presented, which allows the symmetries of a CSP to be represented by the generators of the group. This is a promising approach, but the experiments reported show that the current implementation is much slower than double-lex ordering to solve a BIBD problem represented by a 6×10 matrix.

In [7], combining SBDS with adding symmetry-breaking constraints is explored. One can for instance use column transpositions in SBDS to remove the column symmetry (called col-trans). This would require only $O(n^2)$ SBDS functions. To reduce the row symmetry, one can add constraints that order the rows by their sums. This method (called col-trans+row-sum) does not break all symmetries because it does not consider combinations of row and column permutations. However, it is very practical for reducing symmetry in large matrices.

3 Symmetry-breaking constraints

The effect of col-trans can also be achieved by posing lexicographic constraints on the columns (called col-lex), because either of col-trans or col-lex breaks all column symmetry. Col-trans+row-sum thus removes the same amount of symmetry as col-lex combined with row-sum (col-lex+row-sum). There are other factors which might make one of these strategies preferable to the other: this is discussed in Section 5.

The following theorem is given in [2]:

Theorem 1. *Given a 2-d matrix model where the row sums are all different, ordering its rows by their sums as well as its columns lexicographically breaks all row and column symmetry.*

If a 2-d matrix model with row and column symmetry has a solution, then it has a solution with the rows ordered by their sums. Now any two rows may not be permuted even if any column permutation follows this row permutation. We can now order the columns lexicographically as the row sums are invariant to column permutations. Hence, all symmetry is broken.

What happens when some row sums are equal? In this case, even if we order the rows by their sums and the columns lexicographically, we do not break all symmetry because any two rows with the same sum can be swapped, and then the columns can be lexicographically ordered. We now show that when some row sums are repeated, we can order the rows by their sums and the columns lexicographically, *and* order the rows having the same sum lexicographically. This will remove more row and column symmetry.

Theorem 2. *If a 2-d matrix model with row and column symmetry has a solution, then it has a solution with the rows ordered by their sums and the columns ordered lexicographically, as well as the rows with equal sum ordered lexicographically.*

Proof. We can order the rows of any matrix by their sums. Now the rows having different sums may not be permuted by Theorem 1. On the other hand, the rows with equal sum can freely be permuted. Hence, imposing an ordering on the rows by their sums makes the model have *partial* row symmetry[3]. We reduce partial row symmetry, and column symmetry by imposing a lexicographic ordering on every subset of the rows that can be permuted (in this case the rows with equal sums), and a lexicographic ordering on the columns [2]. □

The method in Theorem 2, called col-lex+row-sum(+row-lex), reduces to double-lex if the row sums are all the same. If the row sums are all different, then it reduces to col-lex+row-sum, which in that case breaks all symmetry [2]. This method thus combines the power of double-lex when the row sums are all the same, with the power of row-sum constraints when the row sums are all different. When the row sums are neither all the same nor all different, col-lex+row-sum(+row-lex) eliminates more symmetry than col-lex+row-sum. For instance, the solution $\begin{pmatrix} 0\,0\,1 \\ 1\,1\,0 \\ 0\,1\,1 \end{pmatrix}$ is symmetric to $\begin{pmatrix} 0\,0\,1 \\ 0\,1\,1 \\ 1\,1\,0 \end{pmatrix}$, and is eliminated by col-lex+row-sum(+row-lex) but not by col-lex+row-sum.

However, note that col-lex+row-sum(+row-lex) can also leave symmetry in a 2-d matrix model. Consider a 3×3 0/1 matrix model that has both row and column symmetry. The solutions $\begin{pmatrix} 0\,0\,1 \\ 0\,1\,0 \\ 1\,0\,1 \end{pmatrix}$ and $\begin{pmatrix} 0\,0\,1 \\ 0\,1\,0 \\ 1\,1\,0 \end{pmatrix}$ are symmetric solutions that have rows ordered by their sums, columns ordered lexicographically, as well as the rows with equal sums ordered lexicographically.

We can also treat each row of the matrix as a multiset, i.e. as a set with repetitions, and insist that each row, as a multiset, should be no greater than the rows below it. We can, for instance, imagine the values in rows 1 and 2 sorted in descending order: the largest value in row 1 must be no greater than the largest value in row 2; if they are the same, we compare the 2nd largest values, and so on. Multiset ordering of the rows is stronger than row-sum ordering because non-identical rows with the same sum may be different when considered as multisets. Hence, combining multiset row ordering with col-lex (called col-lex+row-multiset) would reduce more symmetry than col-lex+row-sum does. For instance, the matrix $\begin{pmatrix} 0\,1\,2\,3 \\ 0\,3\,3\,3 \\ 1\,2\,3\,3 \end{pmatrix}$ is symmetric to $\begin{pmatrix} 0\,1\,2\,3 \\ 1\,2\,3\,3 \\ 0\,3\,3\,3 \end{pmatrix}$, and is eliminated by col-lex+row-multiset but not by col-lex+row-sum.

We can achieve multiset ordering by assigning a weight to each value, summing the weights along each row and constraining the sums to be non-decreasing. Since we first order the rows in increasing order of maximum element, and consider other elements in the rows only if there is a tie, the weight should increase with the value. We want to ensure that, for instance, for any possible value k,

[3] A matrix model has partial row (resp. column) symmetry iff strict subset(s) of the rows (resp. columns) of one of its matrices are indistinguishable [2].

a row containing say one element with value k and $n-1$ 0s has greater weight than a row in which each of the n elements is $k-1$, where n is the number of columns. A suitable weighting assigns the weight n^r to the value r. Thus the first row in the example has total weight $n^k + n - 1$ and the second row has weight n^k. The ordering can be implemented by introducing new constrained variables w_{ij} such that $w_{ij} = n^k$ iff $x_{ij} = k$. Then the constraint that row i is not greater than row $i+1$, both considered as multisets, is: $\sum_j w_{i,j} \leq \sum_j w_{i+1,j}$.

With this implementation, multiset ordering appears to be no more expensive than row-sum ordering. However, it will not be possible to implement it in this way unless n^l is manageable, where l is the largest possible value in the matrix. Moreover, if there are only two possible values in the domains, col-lex+row-multiset is equivalent to col-lex+row-sum.

It is possible to improve col-lex+row-multiset even further. We can imagine that two non-identical rows may have the same weighted sum, so that multiset ordering is not able to distinguish between them. We may, however, distinguish them by ordering them lexicographically. We can easily show that we do not lose any solutions by this method.

Theorem 3. *Given a 2-d matrix model where the row weighted-sums are all different, ordering its rows by their weighted-sums as well as its columns lexicographically breaks all row and column symmetry.*

Proof. The proof of theorem 1 can easily be adapted for weighted-sums. □

Theorem 4. *If a 2-d matrix model with row and column symmetry has a solution, then it has a solution with the rows ordered by their weighted-sums and the columns ordered lexicographically, as well as the rows with equal weighted-sum ordered lexicographically.*

Proof. The proof of theorem 2 can easily be adapted for weighted-sums. □

With this new method, called col-lex+row-multiset(+row-lex), if the row weighted-sums are all the same then we will end up with double-lex. Also, when the domain size is 2, the strategy will specialise into col-lex+row-sum(+row-lex) which is stronger than col-lex+row-sum. If the row weighted-sums are all different then we will end up with col-lex+row-multiset, which breaks all symmetry when the row weighted-sums are all different.

This method thus combines the power of double-lex when the row weighted-sums are all the same, with the power of col-lex+row-sum(+row-lex) when the domain size is 2, and with the power of weighted-sum constraints when the row weighted-sums are all different. Hence, col-lex+row-multiset(+row-lex) eliminates more symmetry than col-lex+row-multiset. For instance, the matrix $\begin{pmatrix} 0\,1\,2\,3 \\ 3\,3\,3\,0 \\ 0\,3\,3\,3 \end{pmatrix}$ is symmetric to $\begin{pmatrix} 0\,1\,2\,3 \\ 0\,3\,3\,3 \\ 3\,3\,3\,0 \end{pmatrix}$, and is eliminated by col-lex+row-multiset(+row-lex) but not by col-lex+row-multiset. However, note that col-lex+row-multiset(+row-lex) can also leave symmetry in a 2-d matrix model.

	3x3 matrices					4x4 matrices	
Strategy	2 vals	3 vals	4 vals	5 vals	6 vals	2 vals	3 vals
SBDS	36	738	8240	57675	289716	317	90492
no symmetry-breaking	512	19683	262144	$\geq$ 1.5M	$\geq$ 1.5M	65536	$\geq$ 1.5M
double-lex	45	1169	14178	102251	520017	650	250841
col-lex+row-sum	42	1007	11174	75715	368154	567	190671
col-lex+row-sum(+row-lex)	39	832	9264	63829	316329	420	120281
col-lex+row-multiset	42	863	9128	61555	302386	567	136665
col-lex+row-multiset(+row-lex)	39	804	8710	59716	296337	420	109545
row & col. transpositions +combinations	36	786	8985	63052	315428	353	114966

Table 1. Number of matrices found using different symmetry-breaking strategies.

4 Experimental results

We have carried out some experiments to compare different ways of reducing symmetry in a matrix model with row and column symmetry. In the experiments, we ignore any constraints on the matrix to be constructed. We only specify the size of the matrix and the domain size of the elements in the matrix, and then want to find a set of matrices such that no two can be generated from each other by permuting the rows and/or columns. We here consider small matrices only, so that we can compare the results with the results of SBDS, which eliminates symmetry entirely but is applicable, at present, only to small matrices. Table 1 shows how many solutions each technique finds, in comparison with the number of distinct solutions given by SBDS, and also in comparison with no symmetry-breaking. Note that M stands for a million.

Note that using SBDS does not hinder the variable and value ordering; if we order the matrices in any symmetry equivalence class according to the variable and value ordering being used for the search, then the first matrix in each equivalence class is the one that will be found. On the other hand, if we are adding symmetry-breaking constraints to the model before search, we have to be more careful in choosing the search strategy: the constraints should be designed with a variable and value ordering in mind (or v.v.); otherwise, the first matrix in an equivalence class according to the ordering may conflict with the symmetry-breaking constraints.

The lexicographic ordering constraints discussed in [2] are consistent with a variable ordering which considers the top row, left to right, 2nd row, left to right, and so on, and the values in ascending order. This ordering has been used for the experiments described here.

As seen in Table 1, col-lex+row-sum(+row-lex) breaks more symmetry than double-lex and col-lex+row-sum. Col-lex+row-multiset is a better strategy than col-lex+row-sum(+row-lex) when there are more than 3 possible values for each element of the matrix. On the other hand, col-lex+rom-sum(+row-lex) is a better strategy than col-lex+row-multiset when the domains of the matrix elements

have 2 or 3 values. Table 1 shows that by improving col-lex+row-multiset by lexicographically ordering equivalent rows under multiset ordering, we obtain better results than any other approximation technique mentioned so far.

Another possibility for reducing symmetry is to use SBDS with row transpositions, column transpositions, and combinations of one of each (just the transpositions without the combinations is a poor strategy, as shown in [7]). For 3×3 matrices with 2 or 3 possible values, and for 4×4 matrices with 2 values, row and column transpositions plus combinations gives better results than col-lex+row-multiset(+row-lex). For the other domains, col-lex+multiset-row(+row-lex) is the best approximation technique. As the matrix size enlarges, SBDS functions may not be manageable, so col-lex+row-multiset(+row-lex) may be preferable. However, whether multiset ordering is practicable on larger matrices depends on whether it can be implemented efficiently: the method described here would only be feasible for small matrices and a small number of values. In such a case, col-lex+row-sum(+row-lex) would be a practical approximation technique.

5 Discussions

From the experiments presented here, the best approximation to eliminating symmetry in matrices with more than three possible values, of those we have tried, is multiset ordering on one dimension as well as lexicographic ordering on the vectors that are equivalent under multiset ordering, combined with *either* lexicographic ordering *or* SBDS using just the transposition symmetries on the other dimension. Both strategies will result in the same number of matrices being generated. Which of them is the better choice in practice probably depends on other factors, such as the efficiency of the implementation of the multiset ordering and lexicographic ordering constraints, or the effect of enforcing GAC on them. Frisch et al. in [3] propose a linear time global consistency algorithm for maintaining GAC on lexicographic ordering between two vectors. One factor that favours SBDS is that whichever solution is found first in an equivalence class, symmetry-breaking constraints added during search eliminate the symmetric solutions. On the other hand, symmetry-breaking constraints added to the model implicitly specify which solution is to be found, before search starts. This may slow down the search, as the specified solution may not be found easily in the search tree.

Whether a strategy involving multiset ordering is actually practicable depends on whether the ordering can be implemented efficiently: the method described here would only be feasible for small matrices and a small number of possible values. For large matrices, row-sum ordering with lexicographic ordering on the rows with equal row-sums, combined with *either* lexicographic ordering *or* SBDS using just the transposition symmetries on the columns, or the same strategy with rows and columns reversed, would be a practical approximation technique.

The experiments reported omit any problem constraints. Even though the symmetry-breaking constraints mentioned do not interfere with problem con-

straints, the effect of reducing symmetry could be very problem dependent. For instance, if the rows of a matrix are constrained to have the same sum (e.g. the matrix model of BIBD[4]), then row-sum ordering will not be effective, and thus col-lex+row-sum will specialise into col-lex, and col-lex+row-sum(+row-lex) will specialise into double-lex. Likewise, if the rows of a matrix are constrained to have the same number of occurrences of every possible value (e.g. the matrix model of the sports scheduling problem[5]), then every row will be identical when seen as a multiset. This will reduce col-lex+row-multiset to col-lex, and col-lex+row-multiset(+row-lex) to double-lex.

Acknowledgements

We would like to thank Alan Frisch for suggesting multiset ordering, and also Pierre Flener, Brahim Hnich and Toby Walsh for valuable discussions.

References

1. J. Crawford, G. Luks, M. Ginsberg, and A. Roy. Symmetry breaking predicates for search problems. In *Proc. of KR'96, the 5th Int. Conf. on Knowledge Representation and Reasoning*, pp. 148–159, 1996.
2. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetry in matrix models. In *Proc. of CP'2002*. Springer, 2002.
3. A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proc. of CP'2002*. Springer, 2002.
4. I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: symmetry breaking during search. In *Proc. of CP'2002*. Springer, 2002.
5. I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In *Proc. of ECAI'00, the 14th European Conf. on AI*, pp. 599–603. IOS Press, 2000.
6. S. Prestwich. First-solution search with symmetry breaking and implied constraints. In *Proc. of Formul'01 , the CP'01 Workshop on Modelling and Problem Formulation*, 2001.
7. B.M. Smith and I.P. Gent. Reducing symmetry in matrix models: SBDS vs. constraints. Technical report APES-31-2001. Available from http://www.dcs.st-and.ac.uk/∼apes/reports/apes-31-2001.ps.gz, 2001.

[4] prob028 at www.csplib.org
[5] prob026 at www.csplib.org

Constraint Programming with Multisets

Zeynep Kiziltan[1] and Toby Walsh[2]

[1] Department of Information Science, Uppsala University, Uppsala, Sweden
Zeynep.Kiziltan@dis.uu.se
[2] Cork Constraint Computation Center, University College Cork, Ireland
tw@4c.ucc.ie

Abstract. We propose extending constraint solvers with multiset variables. That is, variables whose values are multisets. Such an extension can help prevent introducing unnecessary symmetry into a model. We identify a number of different representations for multiset variables, and suggest primitive and global constraints on multiset variables. Surprisingly, unlike finite domain variables, decomposition of global constraints on multiset variables often does not hinder constraint propagation. We also study in detail the multiset ordering constraint. This constraint is useful for breaking symmetry between multiset variables. We show how it can be enforced using a simple lexicographical ordering constraint.

1 Introduction

Dealing efficiently and effectively with symmetry is one of the major difficulties in constraint programming. Symmetry occurs in many scheduling, assignment, routing and supply chain problems. In addition to the symmetry inherently present in a problem, modelling may introduce additional and, in some cases, unnecessary symmetry. Consider, for example, the template design problem (prob002 in CSPLib) in which we assign designs to printing templates. As there are a fixed number of slots for designs on each template, we can model this problem with a variable for each slot, whose value is the design assigned to this slot. However, slots on a template are essentially indistinguishable. This model therefore introduces an unnecessary symmetry, namely the permutations of the slots. A "better" model would remove this symmetry by having a variable for each template, whose value is the multiset of designs assigned to that template. It is a multiset, not a set, as the designs on a template are often repeated. This second model still introduces a symmetry as the templates are indistinguishable. However, as we show in this paper, orderings on multisets can be used to break such symmetries.

Set variables have been incorporated into all the major constraint solvers (see, for example, [6, 8]). It is therefore surprising that there is little if no work on multiset variables in constraint programming. Our goal is to correct this imbalance. The paper is structured as follows. We first introduce some notation and formal background (Section 2). We then discuss how to represent multiset variables (Section 3) and constraints on them (Section 4). In Section 5 we introduce

the multiset ordering, and discuss how to enforce this as a constraint (Section 6). Such a constraint is useful for breaking symmetry in a range of problems. Finally, we end with future work and conclusions.

2 Formal background

We will need vectors, sets and multisets. A vector is an ordered list of elements, written $\langle x_0, \ldots x_{n-1} \rangle$. A set is an unordered list of elements without repetition, written $\{x_0, \ldots x_{n-1}\}$. A multiset is an unordered list of elements in which repetition is allowed, written $\{\!\{x_0, \ldots x_{n-1}\}\!\}$. To simplify notation, we assume that the elements of vectors, sets and multisets are integers drawn from some finite domain $[0, d)$. Basic operations on sets generalize to similar operations on multisets. We let $occ(x, M)$ be the number of occurrences of x in the multiset M. Multiset union, intersection, difference, and equality are defined by the follow identities: $occ(x, M \cup N) = occ(x, M) + occ(x, N)$, $occ(x, M \cap N) = \min(occ(x, M), occ(x, N))$, $occ(x, M - N) = \max(0, occ(x, M) - occ(x, N))$, and $M = N$ iff $occ(x, M) = occ(x, N)$ for all x. For example, if $M = \{\!\{0, 1, 1\}\!\}$ and $N = \{\!\{0, 0, 1, 2\}\!\}$ then $M \cup N = \{\!\{0, 0, 0, 1, 1, 1, 2\}\!\}$, $M \cap N = \{\!\{0, 1\}\!\}$, $M - N = \{\!\{1\}\!\}$, and $M \neq N$. By ignoring the order of elements in a vector, we can view a vector as a multiset. For example, the vector $\langle 0, 1, 0 \rangle$ can be viewed as the multiset $\{\!\{0, 0, 1\}\!\}$. We will abuse notation and write $\{\!\{\boldsymbol{m}\}\!\}$ for the multiset view of the vector $\boldsymbol{m}$.

We will need the strict lexicographical ordering relation on d-bit vectors. Formally, $\boldsymbol{m} <_{\text{lex}} \boldsymbol{n}$ iff $m_{d-1} \leq n_{d-1}$; $m_{d-2} \leq n_{d-2}$ when $m_{d-1} = n_{d-1}$; $\ldots$; $m_0 < n_0$ when $m_{d-1} = n_{d-1}$, $m_{d-2} = n_{d-2}$, $\ldots$, and $m_1 = n_1$. Note we are treating the dth element of the vectors as the high bit.

3 Representing multisets

There are a number of ways we can represent variables whose values are multisets of values. The most naive method would be to have a finite domain variable whose values are all the possible multisets. However, this will be computationally intractable as the number of possible multisets can in general be exponential.

3.1 Bounds representation

One type of representation for multiset variables generalizes the upper and lower bound representation used for set variables in [6]. For each multiset variable, we maintain two multisets: a least upper and a greatest lower bound. The least upper bound is the smallest multiset containing all those values that can be in the multiset, whilst the greatest lower bound is the largest multiset containing all those values that must be in the multiset. Given a multiset variable M, we write $lub(M)$ and $glb(M)$ for the least upper and greatest lower bound respectively. For reasoning with such a representation, we can introduce the notion of

multiset bounds consistency (BC). Given a constraint C over multiset variables $X_1, \ldots X_n$, we write $sol(X_i)$ for the set of multiset values for X_i which can be extended to the other variables to satisfy the constraint. That is,

$$sol(X_i) = \{m_i \mid C(m_1, \ldots, m_n) \ \wedge \ \forall j \ . \ glb(X_j) \subseteq m_j \subseteq lub(X_j)\}$$

Definition 1. *A constraint over multiset variables $X_1 \ldots X_n$ is BC iff for each $1 \leq i \leq n$, we have*

$$lub(X_i) = \bigcup_{m \in sol(X_i)} m \quad \text{and} \quad glb(X_i) = \bigcap_{m \in sol(X_i)} m$$

A set of constraints is BC iff each constraint in the set is BC.

The rules given in [6] for constructing bounds consistent upper and lower bounds for set variables should easily adapt to multiset variables. This representation is compact but carries the penalty of not being able to represent all forms of disjunction. Consider, for example, a multiset variable M with two possible multiset values: $\{\!\{0\}\!\}$ or $\{\!\{1\}\!\}$. To represent this, we would need $lub(M) = \{\!\{0, 1\}\!\}$ and $glb(M) = \{\!\{\}\!\}$. However, this representation also permits M to take the multiset values $\{\!\{\}\!\}$ and $\{\!\{0, 1\}\!\}$.

3.2 Occurrence representation

Set variables can also be represented by their characteristic function (a vector of d Boolean variables, each of which indicates whether a particular value is in the set or not). A straightforward generalization to multisets is the dual occurrence vector. Each multiset variable M can be represented by a vector $\boldsymbol{m}$ of d finite domain variables with $m_i = occ(i, M)$. For reasoning with such a representation, we can apply consistency techniques like generalized arc consistency (GAC) or bounds consistency (BC) to the variables in these vectors. This representation is also compact but carries the penalty of not being able to represent all forms of disjunction. Consider again the example of a multiset variable M with two possible multiset values: $\{\!\{0\}\!\}$ or $\{\!\{1\}\!\}$. To represent this, we would need an occurrence vector with $m_0 = \{0, 1\}$ (that is, the value 0 can occur zero times or once) and $m_1 = \{0, 1\}$ (that is, the value 1 can occur zero times or once). Like the bounds representation, this also permits M to take the multiset values $\{\!\{\}\!\}$ and $\{\!\{0, 1\}\!\}$.

3.3 Fixed cardinality

Multiset variables of a fixed cardinality are common in a number of problems. For example, the template design problem can be modelled as finding a multiset of designs for each template which is of a fixed cardinality. In such situations, we can represent each of the finite elements of the multiset with a variable whose values are possible values for this multiset element. It may appear that this representation introduces symmetry into the problem (via permutations of these

variables). This is not the case as we will be posting (non-binary) constraints on these variables which ignore their permutation. Of course, this may make it harder work to define and post constraints. To channel efficiently between the occurrence vector representation and this representation, we can use Regin's global cardinality constraint [9]. In addition, we can post the constraint that the sum of the values taken by the variables in the occurrence vector is the multiset cardinality. This representation is also compact but again carries the penalty of not being able to represent all forms of disjunction. Consider, for example, a multiset variable M of cardinality 3 with two possible multiset values: $\{\!\{0,0,0\}\!\}$ or $\{\!\{1,1,1\}\!\}$. To represent this, we would need three variables: $M_1 = \{0,1\}$, $M_2 = \{0,1\}$ and $M_3 = \{0,1\}$. Each finite domain variable represents one of the possible elements of the multiset. However, this representation also permits M to take the multiset values: $\{\!\{0,0,1\}\!\}$, and $\{\!\{0,1,1\}\!\}$.

If the multiset variables are not of a fixed cardinality but there are upper bounds on their maximum cardinality, we can use a similar representation to the fixed cardinality representation. We need, however, to introduce an additional value which represents no value being assigned to a particular finite domain variable. For example, suppose we have a multiset variable of maximum cardinality n drawing elements from $(0, d)$. We can represent this with n variables, each with a finite domain $[0, d)$. The additional value 0 represents no assignment. With the multiset ordering constraint (see Section 6), this additional value will be reasoned with transparently.

3.4 Comparing the representations

We can compare the expressivity of the different representations. We say that one representation is **as expressive** than another if it can represent the same sets of multiset values, **more expressive** if it is as expressive and there is one set of multiset values that it can represent that the other cannot, and **incomparable** if neither representation is as expressive as the other.

Theorem 1.

1. *The occurrence representation is more expressive than the bounds representation.*
2. *The fixed cardinality representation is incomparable to both the bounds and the occurrence representation.*

Proof. 1. Clearly the occurrence representation is as expressive as the bounds. Consider a multiset variable M with two possible multiset values: $\{\!\{\}\!\}$ or $\{\!\{0,0\}\!\}$. This can be represented exactly with the occurrence variable $m_0 = \{0,2\}$. By comparison, a bounds representation would need $lub(M) = \{\!\{\}\!\}$ and $glb(M) = \{\!\{0,0\}\!\}$, and this permits M to take the additional multiset value $\{\!\{0\}\!\}$.

2. Consider a multiset variable M of cardinality 2 with six possible multiset values: $\{\!\{0,1\}\!\}$, $\{\!\{0,2\}\!\}$, $\{\!\{0,3\}\!\}$, $\{\!\{1,1\}\!\}$, $\{\!\{1,2\}\!\}$, or $\{\!\{1,3\}\!\}$. The fixed cardinality representation can represent this exactly with two finite domain variables $M_1 =$

$\{0,1\}$ and $M_2 = \{1,2,3\}$. Both the bounds and the occurrence representations of this set of multiset values would also permit M to take the additional multiset value $\{\!\{2,3\}\!\}$.

Consider a multiset variable M of cardinality 2 with three possible multiset values: $\{\!\{0,1\}\!\}$, $\{\!\{0,2\}\!\}$, or $\{\!\{1,2\}\!\}$. In the bounds representation, we need $lub(M) = \{\!\{0,1,2\}\!\}$ and $glb(M) = \{\!\{\}\!\}$. The only two element multisets between these bounds are exactly $\{\!\{0,1\}\!\}$, $\{\!\{0,2\}\!\}$, or $\{\!\{1,2\}\!\}$ as required. Similarly with an occurrence representation, we need $m_0 = m_1 = m_2 = \{0,1\}$. The only two element multisets between these bounds are again the required ones. A fixed cardinality representation cannot, on the other hand, represent this set of multiset values exactly. We would need two finite domain variables with, say, $M_1 = \{0,1\}$ and $M_2 = \{1,2\}$. These would permit M to take the additional two element mutliset value $\{\!\{1,1\}\!\}$. □

Note that if we restrict the occurrence representation to maintain just bounds on the number of occurrences of a value in the multiset then we obtain a representation that is equally expressive as the original multiset bounds representation. We can also compare the amount of pruning constraint propagation performs on the different representations.

Theorem 2.

1. *BC on the occurrence representation is equivalent to BC on the bounds representation;*
2. *GAC on the occurrence representation is strictly stronger than BC on the occurrence representation;*
3. *BC on the bounds or occurrence representation is incomparable to BC on the fixed cardinality representation*
4. *GAC on the occurrence representation is incomparable to GAC on the fixed cardinality representation;*

Proof. 1. Suppose that a bounds representation of a constraint is BC. Consider any multiset variable in this constraint, X with bounds $lub(X)$ and $glb(X)$. We can construct an equivalent occurrence representation. Suppose $a_{\max} = occ(a, lub(X))$ and $a_{\min} = occ(a, glb(X))$. Then we let the finite domain variable X_a in the occurrence vector have a domain $[a_{\min}, a_{\max}]$. Consider $X_a = a_{\max}$. Then, from the definition of BC of the bounds representation and the generalized multiset union operator, there must be a satisfying solution to the constraint in which $occ(a, X) = a_{\max}$. If there are several, we choose one non-deterministically. This solution is support for the bounds consistency of the occurrence representation. A similar argument holds for $X_a = a_{\min}$. Hence, BC of the bounds representation implies BC of the occurrence representation. The proof reverses directly.

2. It is clearly at least as strong. To show strictness, consider 3 multiset variables, M_1 to M_3 with domains containing the multiset values $\{\!\{\}\!\}$, or $\{\!\{0,0\}\!\}$. Suppose we have a constraint that $distinct(M_1, M_2, M_2)$. This constraint is not GAC (indeed, it has no solution). In the bounds representation, we have $lub(M_i) = \{\!\{0,0\}\!\}$ and $glb(M_i) = \{\!\{\}\!\}$, and the constraint is BC.

3 & 4. Consider seven multiset variables M of cardinality 2, each with six possible multiset values: $\{\!\{0,1\}\!\}$, $\{\!\{0,2\}\!\}$, $\{\!\{0,3\}\!\}$, $\{\!\{1,1\}\!\}$, $\{\!\{1,2\}\!\}$, or $\{\!\{1,3\}\!\}$. The fixed cardinality representation can represent these values exactly. However, the occurrence representation of these multiset values would also permit the additional multiset value $\{\!\{2,3\}\!\}$. Suppose that we have a constraint that all seven multiset variables are distinct. The occurrence representation is GAC (and hence BC) but the fixed cardinality representation is not BC (and hence not GAC).

Consider four multiset variables M of cardinality 2 with three possible multiset values: $\{\!\{0,1\}\!\}$, $\{\!\{0,2\}\!\}$, or $\{\!\{1,2\}\!\}$. The bounds and occurrence representations can represent these values exactly. However, the fixed cardinality representation would permit each multiset variable to take the additional mutliset value $\{\!\{1,1\}\!\}$. Suppose that we have a constraint that all four multiset variables are distinct. The fixed cardinality representation is GAC (and hence BC) but the bounds and occurrence representation are not BC (and hence the occurrence representation is not GAC). □

Since BC on the bounds representation is equivalent to BC on the occurrence representation, in the rest of this paper we will write BC without specifying which representation is used. In addition, when we write GAC on multiset variables, we will assume the occurrence representation unless otherwise indicated.

4 Multiset constraints

We need to support a number of different constraints on multiset variables.

4.1 Primitive constraints

Multiset expressions are made from multiset variables, ground multisets including the empty multiset $\{\!\{\}\!\}$, and the function symbols: $\cup$, $\cap$ and $-$. Multiset constraints are constructed from the inclusion relation, $\subseteq$ and its negation, $\not\subseteq$. Multiset equality, strict inclusion, and membership constraints can all be implemented as defined relations:

$$\begin{aligned} M = N &\implies M \subseteq N \;\&\; N \subseteq M \\ M \subset N &\implies M \subseteq N \;\&\; N \not\subseteq M \\ x \in M &\implies \{\!\{x\}\!\} \subseteq M \end{aligned}$$

Another interesting extension is to graded constraints [6]. An example of a graded constraints is the multiset cardinality function $|M|$. A graded function, f is a mapping onto the positive integers that satisfies the property $M \subset N$ implies $f(M) < f(N)$. A graded constraint restricts f to be within some integer interval.

Such primitive constraints can be implemented via equality and inequality constraints on the associated occurrence vectors as follows:

$$\begin{aligned} M \subseteq N &\implies m_i \leq n_i \text{ for all } i \\ M \not\subseteq N &\implies m_i > n_i \text{ for some } i \end{aligned}$$

$$
\begin{aligned}
M \cup N &\implies m_i + n_i \\
M \cap N &\implies \min(m_i, n_i) \\
M - N &\implies \max(0, m_i - n_i) \\
l \leq |M| \leq u &\implies l \leq \sum_i m_i \leq u
\end{aligned}
$$

4.2 Global constraints

We can also define global (or non-binary) constraints on multiset variables. An important question about such global constraints is whether decomposition hurts. Consider a global constraint on finite domain variables like the all-different constraint. This can be decomposed into binary not-equals constraints. However, such decomposition tends to hinder constraint propagation. For instance, GAC on an all-different constraint is strictly stronger than arc-consistency (AC) on the decomposed binary not-equals constraints [5]. Surprisingly, similar results often do *not* hold for a number of global constraints involving multiset variables. This is good news. We can provide global constraints on multiset variables to help users compactly specify models. However, we do not need to develop complex algorithms for reasoning about them as is often the case with finite domain variables. We can simply decompose such global constraints into primitive constraints. The following results also map over to global constraints on set variables, where sets are implemented either via characteristic functions, or upper and lower bounds. Decomposition of global set constraints thus also often does not hinder constraint propagation. In the rest of this section, we give results to show that decomposition on the occurrence representation often does not hinder GAC. Very similar results can be given to show that decomposition on the occurrence or bounds representation does not hinder BC.

Disjoint constraint The constraint $disjoint([M_1, \ldots, M_n])$ ensures that the multiset variables are pairwise disjoint. This global constraint can be decomposed into the binary constraints: $M_i \cap M_j = \{\!\{\}\!\}$ for all $i \neq j$. Such decomposition does not appear to hinder constraint propagation.

Theorem 3. *GAC (resp. BC) on a disjoint constraint is equivalent to AC (resp. BC) on the binary decomposition.*

Proof. Clearly GAC (resp. BC) on a disjoint constraint is as strong as AC (resp. BC) on the decomposition. To show the reverse, suppose that the binary decomposition is AC (resp. BC). If the disjoint constraint is not GAC or BC then there must be at least two multiset variables, M_i and M_j with a value k in common. That is, $m_{ik} \geq 1$ and $m_{jk} \geq 1$. However, in such a situation, the decomposed constraint $\min(m_{ik}, m_{jk}) = 0$ would neither be AC nor BC. □

Partition constraint The constraint $partition([M_1, \ldots, M_n], M)$ ensures that the multiset variables, M_i are pairwise disjoint and union together to give M. By

introducing new auxiliary variables, it can be decomposed into binary and union constraints of the form: $M_i \cap M_j = \{\!\{\}\!\}$ for all $i \neq j$, and $M_1 \cup \ldots \cup M_n = M$. Decomposition again causes no loss in pruning.

Theorem 4. *GAC (resp. BC) on a partition constraint is equivalent to GAC (resp. BC) on the decomposition.*

Proof. Clearly GAC (resp. BC) on a partition constraint is as strong as GAC (resp. BC) on the decomposition. To show the reverse, by Theorem 3, we need focus just on the union constraints. Suppose that the decomposition is GAC (resp. BC). If the partition constraint is not GAC or BC then there must be one value k that does not occur frequently enough in the upper bounds of the multiset variables. But, in this case, the decomposed constraint (which is equivalent to $\sum_{i=1}^{n} m_{ik} = m_k$) would neither be GAC nor BC. □

This result continues to hold even if the union constraint is decomposed into the set of ternary union constraints by introducing new auxiliary variables: $M_1 \cup M_2 = M_{12}, M_{12} \cup M_3 = M_{13}, \ldots, M_{1n-1} \cup M_n = M$. We can also consider the non-empty partition constraint which ensures we have a partition and that each multiset variable is not the empty multiset. Decomposition now appears to hinder constraint propagation.

Theorem 5. *GAC (resp. BC) on a non-empty partition constraint is strictly stronger than GAC (resp. BC) on the decomposition.*

Proof. Clearly it is at least as strong. To show strictness, consider 3 multiset variables with $glb(M_1) = glb(M_2) = glb(M_3) = \{\!\{\}\!\}$, $lub(M_1) = lub(M_2) = \{\!\{1, 2\}\!\}$ and $lub(M_3) = \{\!\{1, 2, 3\}\!\}$. The decomposition is both GAC and BC. However, enforcing GAC or BC on the non-empty partition constraint gives $glb(M_3) = lub(M_3) = \{\!\{3\}\!\}$. □

Distinct constraint Consider the constraint $distinct([M_1, \ldots, M_n])$ which ensures that all the multisets are distinct from each other. This decomposes into pairwise not equals constraints: $M_i \neq M_j$ for all $i \neq j$. Decomposition in this case appears to hinder constraint propagation.

Theorem 6. *GAC (resp. BC) on a distinct constraint is strictly stronger than AC (resp. BC) on the decomposition.*

Proof. Clearly it is at least as strong. To show strictness, consider a distinct constraint on 3 multiset variables with $glb(M_1) = glb(M_2) = \{\!\{\}\!\}$, $lub(M_1) = lub(M_2) = \{\!\{0\}\!\}$, $glb(M_3) = \{\!\{0\}\!\}$, and $lub(M_3) = \{\!\{0, 0\}\!\}$. The decomposition is both AC and BC. But enforcing GAC or BC on the distinct constraint gives $glb(M_3) = lub(M_3) = \{\!\{0, 0\}\!\}$. □

5 Ordering multisets

We will often want to order multiset variables. For example, in the template design problem we can break the symmetry between templates by insisting that

their multiset values are ordered. Another application (suggested by Alan Frisch and since also considered in [7]) is for breaking row and column symmetry in matrix models [1]. Many problems can be modelled by matrices of decision variables in which the rows and/or columns are indistinguishable and can be permuted. Whilst in theory we can eliminate such symmetry using techniques like SBDS [3], in practice there are often too many symmetries to deal with them exhaustively. One strategy is to use SBDS to eliminate all column symmetry, and then add symmetry breaking constraints to eliminate some (but perhaps not all) the row symmetry. Since we can no longer freely permute the columns, whatever symmetry breaking constraints we add must be invariant to column permutation. We can, for instance, insist that the row sums are non-decreasing [4]. A stronger symmetry breaking constraint is to insist that the rows, when viewed as multisets, are ordered. This is stronger because two vectors may have the same sum, whereas two vectors treated as multisets are never equal unless they are identical. For instance, the vectors $\langle 3, 3, 2, 1\rangle$ and $\langle 3, 3, 3, 0\rangle$ have the same sum, but when viewed as multisets they are different. Ordering constraints on multisets are therefore very useful.

We write $\max(M)$ for the maximum element of a multiset M. This presuppose a total ordering on elements of the multiset. Such an ordering induces a total ordering on multisets. Formally, $M \prec_m N$ iff $x = \max(M)$, $y = \max(N)$ and:

$$x < y \ \vee \ (x = y \ \wedge \ M - \{\!\{x\}\!\} \prec_m N - \{\!\{y\}\!\})$$

That is, either the largest element in M is smaller than the largest element in N, or they are the same and we eliminate this occurrence and recurse. This is called the **multiset ordering**. We can derive almost identical results if we weaken the ordering to include multiset equality (that is, for the ordering relation $M \preceq_m N$ defined by $M = N$ or $M \prec_m N$). As the following theorem shows, the multiset ordering is equivalent to the lexicographical ordering on the associated occurrence vectors. As we have efficient algorithms for reasoning about occurrence constraints [9] and about lexicographical ordering constraints [2], this may be the most effective route to reasoning about multiset ordering constraints.

Theorem 7. *$M \prec_m N$ iff $\boldsymbol{m} <_{\text{lex}} \boldsymbol{n}$.*

Proof. Suppose $M \prec_m N$. There are two cases. In the first case, $\max(M) < \max(N)$. Then $n_{\max(N)} > 0$ and for all $i \geq \max(N)$, we have $m_i = 0$. Hence, $\boldsymbol{m} <_{\text{lex}} \boldsymbol{n}$. In the second case, $\max(M) = \max(N) = a$. Then either $occ(a, M) = occ(a, N)$ or $occ(a, M) < occ(a, N)$. In the first subcase, we can delete all occurrences of a from M and N and recurse. In the second subcase, $m_a < n_a$ and for all $i > a$ we have $m_i = n_i = 0$. Hence, $\boldsymbol{m} <_{\text{lex}} \boldsymbol{n}$. The proof reverses easily. □

One special case that will concern us is ordering 0/1 vectors viewed as multisets. This occurs when we are dealing with row and column symmetry in a matrix model of Boolean decision variables. As the following theorem demonstrates, the multiset ordering then reduces to ordering the vector sum. As bounds consistency techniques will reason about such sums quickly and effectively, we need not consider multiset orderings when dealing with 0/1 vectors.

Theorem 8. *On 0/1 vectors* $\boldsymbol{m}$ *and* $\boldsymbol{n}$, $\{\!\{\boldsymbol{m}\}\!\} \prec_m \{\!\{\boldsymbol{n}\}\!\}$ *iff* $\sum_i m_i < \sum_i n_i$.

Proof. Suppose $\{\!\{\boldsymbol{m}\}\!\} \prec_m \{\!\{\boldsymbol{n}\}\!\}$. There are two cases. In the first case, $1 \notin \{\!\{\boldsymbol{m}\}\!\}$ and $1 \in \{\!\{\boldsymbol{n}\}\!\}$. Then $(\sum_i m_i = 0) < (\sum_i n_i > 0)$. In the second case, $1 \in \{\!\{\boldsymbol{m}\}\!\}$ and $1 \in \{\!\{\boldsymbol{n}\}\!\}$. Then $occ(1, \{\!\{\boldsymbol{m}\}\!\}) < occ(1, \{\!\{\boldsymbol{n}\}\!\})$ and $\sum_i m_i < \sum_i n_i$. The proof again reverses easily. □

6 Multiset ordering constraints

How do we enforce multiset ordering constraints? If multisets are being represented by occurrence vectors then by Theorem 7 we can simply post a lexicographic ordering constraint on these occurrence vectors and use the linear GAC algorithm given in [2]. If, however, multisets are being represented using the fixed cardinality representation, we would first have to channel into occurrence vectors with cardinality constraints. This would need to occur when, for example, we are symmetry breaking in a matrix model by viewing rows or columns as fixed cardinality multisets. Such channeling can be done efficiently using the polynomial GAC algorithm for cardinality constraints given in [9]. However, as the following theorem demonstrates, such channelling hinders constraint propagation.

Theorem 9. *GAC on a multiset ordering constraint represented using the fixed cardinality representation is strictly stronger than GAC applied simultaneously to a lexicographical ordering constraint on the occurrence vectors and to cardinality constraints between the fixed cardinality and occurrence representations.*

Proof. Clearly it is as strong. To show strictness, consider two multiset variables in the fixed cardinality representation, each of which contains four elements:

$$M_1 = \{\!\{\{1,2\},\{1,2\},\{2\},\{2\}\}\!\}$$
$$M_2 = \{\!\{\{1,2\},\{1,2\},\{0,1,2\},\{0,1\}\}\!\}$$

A multiset ordering constraint, $M_1 \prec M_2$ on this representation is not GAC since the third finite domain variable in M_2 needs to be reduced to give:

$$M_2 = \{\!\{\{1,2\},\{1,2\},\{1,2\},\{0,1\}\}\!\}$$

The corresponding occurrence vectors are:

$$\boldsymbol{m_1} = \langle\{0\},\{0,1,2\},\{2,3,4\}\rangle$$
$$\boldsymbol{m_2} = \langle\{0,1,2\},\{0,1,2,3,4\},\{0,1,2,3\}\rangle$$

Enforcing GAC on the lexicographical ordering constraint gives:

$$\boldsymbol{m_1} = \langle\{0\},\{0,1,2\},\{2,3\}\rangle$$
$$\boldsymbol{m_2} = \langle\{0,1,2\},\{0,1,2,3,4\},\{2,3\}\rangle$$

This does not allow us to prune any values from the multisets. GAC on the fixed cardinality representation therefore does more pruning.

Note that we also know that the multisets are of cardinality 4. We could therefore have a combined lexicographical and vector sum constraint to ensure this. Enforcing GAC on such a combined constraint gives:

$$\boldsymbol{m_1} = \langle\{0\}, \{1, 2\}, \{2, 3\}\rangle$$
$$\boldsymbol{m_2} = \langle\{0, 1\}, \{0, 1, 2\}, \{2, 3\}\rangle$$

Even though we now know that the value 0 can only occur at most once in M_2, we cannot say where. We still therefore cannot do as much pruning as GAC on the fixed cardinality representation. □

For multisets of fixed cardinality n, multiset ordering constraints $M \prec_m N$ can be enforced via the arithmetic constraint $n^{m_0} + \ldots n^{m_{n-1}} < n^{n_0} + \ldots n^{n_{n-1}}$ [7]. It is not hard to show that BC on such a constraint is equivalent to GAC on the original multiset ordering constraint. However, such an arithmetic constraint is only feasible for small n.

7 Future work and conclusions

We have proposed that constraint solvers be extended with multiset variables. That is, variables whose values are multisets. Such an extension will help prevent introducing unnecessary symmetry into models. We have identified a number of different representations for multiset variables, and suggested a set of primitive and global constraints on multiset variables. Surprisingly, unlike finite domain variables, decomposition of global constraints on multiset variables often does not hinder constraint propagation. We also studied in detail the multiset ordering constraint. This constraint is useful for breaking symmetry between multiset variables, and also breaking row and column symmetry in matrix models. We have shown how a multiset ordering can be enforced by lexicographically ordering the associated vectors of value occurrences. Although channelling into the occurrence representation hinders constraint propagation, we conjecture that the impact will not be great, and that this will prove an effective way of enforcing multiset ordering constraints. We are currently running experiments to test this thesis.

References

1. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetry in matrix models. In *Proc. of CP'2002*. Springer, 2002.
2. A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proc. of CP'2002*. Springer, 2002.
3. I. Gent and B. Smith. Symmetry breaking in constraint programming. In *Proc. of ECAI-2000*, pp 599–603. IOS Press, 2000.
4. I. Gent and B. Smith. Reducing symmetry in matrix models: Sbds vs. constraints. Technical Report APES-31-2001, APES Research Group, 2001. Available from http://www.dcs.st-and.ac.uk/ apes/apesreports.html.

5. I. Gent, K. Stergiou, and T. Walsh. Decomposable constraints. *Artificial Intelligence* 123:133–156, 2000.
6. C. Gervet. Conjunto: constraint logic programming with finite set domains. In *Proc. of the 1994 International Symposium on Logic Programming*, pp 339–358. MIT Press, 1994.
7. Z. Kiziltan and B.M. Smith. Symmetry-breaking constraints for matrix models. In *Proc. of SymCon'02, the CP'02 Workshop on Symmetry in Constraints*, 2002.
8. T. Müller and M. Müller. Finite set constraints in Oz. In *Proc. of 13th Logic Programming Workshop*, pp 104–115. Technische Universität München, 1997.
9. J. Régin. Generalized arc consistency for global cardinality constraints. In *Proc. of AAAI'96*, pp 209–215. AAAI Press/The MIT Press, 1996.

Supersymmetric Modeling for Local Search

Steven Prestwich

Cork Constraint Computation Centre
University College, Cork, Ireland
s.prestwich@cs.ucc.ie

Abstract. A previous paper showed that adding binary symmetry breaking constraints to a model can degrade local search performance. This paper shows that even unary constraints are not necessarily beneficial because they can transform solutions into local minima. Furthermore, performance can be improved by the reverse of symmetry breaking: using a *supersymmetric* model with extra symmetry. Some improved results for the highly symmetrical Social Golfer problem are presented.

1 Introduction

Considerable research has been devoted to symmetry breaking techniques for reducing search space size. Probably the most popular approach is to add constraints to the problem formulation, so that each equivalence class of solutions to the original problem corresponds to a single solution in the new problem. This avoids the need to modify search algorithms, which are often rather complex, and is the only option available to a researcher using SAT solvers or other algorithms downloaded from the Internet. (Dynamic approaches to symmetry breaking such as [3] are not considered in this paper.)

Symmetry breaking is usually applied when complete backtrack search is to be performed and its benefits are often considerable. However, it is well known that it does not necessarily improve search for a single solution. A previous paper [4] showed that adding binary symmetry breaking constraints to SAT models can have a negative effect on local search for clique, set cover and block design problems. This was not merely overhead caused by extra constraint processing; the number of search steps needed to find a solution was also increased. Several researchers have demonstrated unexpected effects when combining two or more techniques: backtracking has been shown to interact badly with the normally beneficial techniques of domain pruning [6], arc consistency preprocessing [7] and removal of inconsistent or redundant domain values or subproblems [2]. The effects of combining techniques can be surprising and unpredictable, so it should not be a complete surprise that local search and symmetry breaking do not always combine well. However, in our experiments local search performance was *almost always* degraded by symmetry breaking, suggesting that the effects are not isolated anomalies.

This paper investigates the effect of *unary* symmetry breaking constraints on local search performance. These might be expected to benefit any search

algorithm but Section 2 shows that, in principle, they can have a negative effect on a variety of algorithms. Section 3 shows that this can occur on real problems, using a hybrid local search algorithm and an ILP model of the Social Golfer problem. Section 4 investigates the following question: if reducing symmetry in a model is bad for local search, might *increasing* symmetry help?

2 Symmetry breaking and local minima

First we show that adding unary symmetry breaking constraints can have a negative effect on local, backtrack and hybrid search. Consider the SAT problem

$$\overline{a} \vee b \quad \overline{a} \vee c \quad a \vee \overline{b} \quad a \vee \overline{c}$$

There are two solutions: [a=T, b=T, c=T] and [a=F, b=F, c=F]. Suppose a problem modeler realises that every solution to a problem has a symmetrical solution in which all truth values are negated, as in this example. Then a simple way to break symmetry is to fix the value of any variable by adding a new clause such as

$$a$$

Denote the first model by M and the model with symmetry breaking by M_s.

Now suppose we apply a local search algorithm such as GSAT [9] to the problem. GSAT starts by making a random truth assignment to all variables, then repeatedly flipping truth assignments to try to reduce the number of violated clauses. In model M the state [a=F, b=F, c=F] is a solution, but in M_s the added clause a is violated; moreover, any flip leads to a state in which *two* clauses are violated. In other words this state has been transformed from a solution to a *local minimum*. (The unary clause does not preclude this as a random first state, nor does it necessarily prevent a randomized local search algorithm from reaching this state.) In contrast M has no local minima: any non-solution state contains either two T or two F assignments, so a single flip leads to a solution (respectively TTT or FFF). GSAT will actually escape this local minimum because it makes a "best" flip even when that flip increases the number of violations, but examples can be constructed with deeper local minima to defeat this heuristic. The point is that a new local minimum has been created, and local minima generally degrade local search performance by requiring more noise.

Unit propagation applied to the added clause gives a reduced problem

$$b \quad c$$

which contains no local minima; will unary constraints always benefit search algorithms with unit propagation? Consider a Davis-Logemann-Loveland algorithm [1] (backtracking with unit propagation) applied to another SAT problem

$$a \vee b \vee c \vee d \quad a \vee \overline{b} \vee c \vee d \quad \overline{a} \vee b \vee c \vee d \quad \overline{a} \vee \overline{b} \vee c \vee d \quad \overline{c} \vee \overline{d}$$

There are eight solutions:

	1	2	3	4	5	6	7	8
a:	T	T	T	T	F	F	F	F
b:	T	T	F	F	T	T	F	F
c:	T	F	T	F	T	F	T	F
d:	F	T	F	T	F	T	F	T

Suppose a problem modeler realises that each solution has a symmetric version in which the values of c and d are exchanged. To exclude solutions 1, 3, 5 and 7 a unary constraint

$$\overline{c}$$

may be added. Applying unit propagation and removing redundant constraints gives a reduced problem

$$a \vee b \vee d \quad a \vee \overline{b} \vee d \quad \overline{a} \vee b \vee d \quad \overline{a} \vee \overline{b} \vee d$$

However, even though the unary constraint has been propagated, the backtracker can still move part of the way towards one of the excluded solutions. Assign d=F (as in excluded solutions 1, 3, 5 and 7) and apply unit propagation:

$$a \vee b \quad a \vee \overline{b} \quad \overline{a} \vee b \quad \overline{a} \vee \overline{b}$$

No empty clause has been derived so the algorithm will not backtrack at this point. However, this state cannot be extended to a solution because all combinations of a and b assignments lead to an empty clause. Only after a further variable assignment will backtracking occur. Adding the unary symmetry breaking constraint $\overline{c}$ has increased the number of backtracks needed to find a solution.

This phenomenon is well known for backtrackers and was a motivation for approaches such as Symmetry Breaking During Search [3]. However, the example also applies to a hybrid local search algorithm: Saturn [5], which performs local search in a space of partial assignments, uses unit propagation to prune the search space, and attempts to minimize the number of unassigned variables. For Saturn adding the unary symmetry breaking constraint transforms the state [c=F, d=F] from a partial solution to a local minimum.

To summarize: adding a unary symmetry breaking constraint may increase the number of search steps needed to find a solution, whether we use local search, backtracking or a hybrid such as Saturn. But does this effect occur in practice or are these contrived examples mere curiosities? The next section provides evidence that it does occur.

3 Experiments on the Social Golfer problem

As a symmetrical problem we choose the Social Golfer problem which is studied extensively in [10]. In a golf club there are 32 social golfers, each of whom play golf once a week and always in groups of 4. The problem is to find a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion. This

can be generalized to P golf players playing in G groups of size S for W weeks, where $G \times S = P$. An example is shown in Figure 1 for instance 3-3-4 (G-S-W). The related Kirkman's Schoolgirls problem corresponds to instance 5-3-7. These problems have a huge number of symmetries because players, groups, weeks and variable names may all be permuted.

week	group 1	group 2	group 3
1	1,2,3	4,5,6	7,8,9
2	1,4,7	2,5,8	3,6,9
3	1,6,8	2,4,9	3,5,7
4	1,5,9	2,6,7	3,4,8

Fig. 1. A schedule for Social Golfer instance 3-3-4

3.1 A model

A pure 0/1 integer linear program (ILP) model is as follows. Define a 0/1 variable v_{pgw} which is 1 if and only if player p plays in group g in week w. The constraints are as follows. Each group contains exactly S players:

$$\sum_{p=1}^{P} v_{pgw} = S$$

where $1 \leq g \leq G$ and $1 \leq w \leq W$. Each player plays in exactly one group per week:

$$\sum_{g=1}^{G} v_{pgw} = 1$$

where $1 \leq p \leq P$ and $1 \leq w \leq W$. Finally the "sociability constraints": no two players can play in the same group as each other more than once. Define auxiliary variables $s_{pp'w}$ ($1 \leq p < p' \leq P$, $1 \leq w \leq W$) to denote that in week w players p and p' play in the same group:

$$v_{pgw} + v_{p'gw} \leq 1 + s_{pp'w}$$

where $1 \leq p < p' \leq P$, $1 \leq g \leq G$ and $1 \leq w \leq W$. Now the sociability constraints can be expressed on the s:

$$\sum_{w=1}^{W} s_{pp'w} \leq 1$$

where $1 \leq p < p' \leq P$ and $1 \leq w < w' \leq W$. Symmetry breaking can be applied by adding two sets of constraints to the model. We can fix the groups in the first week:

$$v_{ij1} = 1$$

where $1 \leq i \leq P$ and $j = (i-1)/S + 1$ rounded down to the nearest integer. We can also fix player 1 to be in group 1 in every subsequent week:

$$v_{11w} = 1$$

where $2 \leq w \leq W$. This is Walser's model as described in CSPLib. [1]

3.2 Results

If adding unary symmetry breaking constraints can create local minima, this should require additional noise in a local search algorithm. We test this using the Saturn local search algorithm, which has been extended from SAT to pure 0/1 ILP models. Saturn has an integer noise parameter B and we examine the effect of varying B on its performance. Performance is measured as the number of search steps (non-systematic backtracks, denoted by "bt") required to find a solution, taking medians over 1000 runs. The use of search steps filters out overheads caused by processing the extra symmetry breaking constraints, and the use of medians rather than means reduces distortion by outliers. The results for two instances, which are representative of more extensive experiments, are shown in Figures 2 and 3.

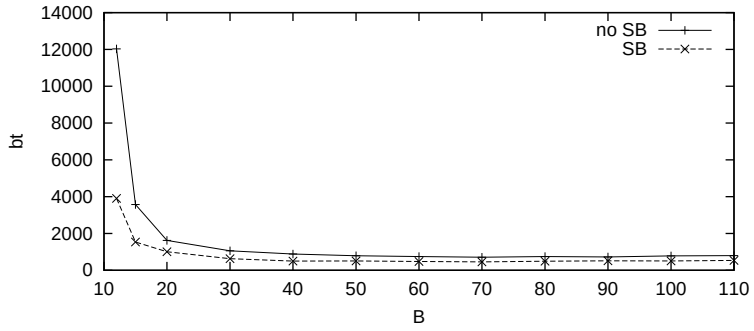

Fig. 2. Results for instance 5-4-3

For easy instances such as 5-4-3 (Figure 2) the added constraints consistently improve performance. We believe that this is simply because the number of search variables has been reduced via unit propagation on the unary constraints. For harder instances such as 6-4-5 (Figure 3) the results are more complex. The optimum noise level has increased, which is evidence for extra local minima. However, the search effort at the optimum noise levels is similar in both cases. This may be the positive effect of fewer search variables balancing the negative effect of extra local minima. These results are not a strong argument against

[1] http://www.csplib.org, problem 10

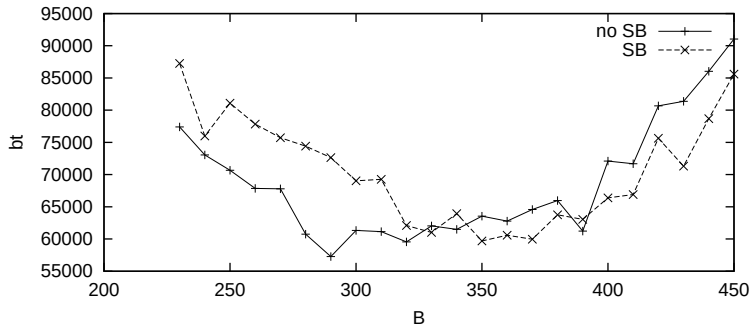

Fig. 3. Results for instance 6-4-5

using symmetry breaking with local search, but there may be other problems on which the negative effect is greater. Moreover, extra constraints increase runtime overheads, so avoiding them seems preferable.

Local search on symmetric models can be very effective. Saturn was applied to the model without symmetry breaking, and at the time of writing it has found the longest schedules for several large instances: 9-5-6, 9-6-5, 9-8-3, 9-9-3, 10-5-7, 10-7-5, 10-8-4, 10-9-3 and 10-10-3, each taking a few seconds or minutes. The optimal 7 week solution for Kirkman's Schoolgirls problem was found in a few seconds, a 7 week solution for the instance with 8 groups of 4 golfers was found in approximately 1 minute, and an 8 week solution was found after approximately 6 hours. On the other hand, Saturn cannot reproduce some solutions found by other algorithms, for example 8-4-9. Current results for the Social Golfer are summarized on a web page. [2]

4 The opposite of symmetry breaking

These results suggest a novel approach to modeling for local search, mentioned in [4] but not tested: *adding* symmetry. We might call this reversal of symmetry breaking *supersymmetric modeling*.

As an example, consider the usual definition of a Golomb Ruler: an ordered sequence of integers $0 = x_1 < x_2 < \ldots < x_m = \ell$ such that the $m(m-1)/2$ differences $x_j - x_i$ $(j > i)$ are distinct, where ℓ is the permitted ruler length. The constraint $|x_2 - x_1| < |x_m - x_{m-1}|$ is usually added to break a reflective symmetry. Finding a ruler with given m and ℓ is a CSP. Several models are given in [11] and we use their *binary/ternary* model. Define variables $x_1 \ldots x_m$ each with domain $\{0, \ldots, \ell\}$, and auxiliary variables d_{ij} where $1 \leq i < j \leq m$. Impose ordering constraints $x_i < x_{i+1}$ where $1 \leq i \leq m-1$. Impose ternary constraints $d_{ij} = x_j - x_i$ and binary constraints $d_{ij} \neq d_{i'j'}$ where $i < i'$, or $i = i'$ and $j < j'$. Also impose unary constraints $x_1 = 0$ and $x_m = \ell$, and a symmetry

[2] http://www.icparc.ic.ac.uk/~wh/golf/

breaking constraint $d_{12} < d_{m-1,m}$. This CSP can be SAT-encoded using the direct encoding.

To obtain a supersymmetric model we remove the symmetry breaking constraint and the ordering constraints on the x_i, merely constraining them to be distinct: $x_i \neq x_j$ where $1 \leq i < j \leq m$. The binary constraints $d_{ij} \neq d_{i'j'}$ are as above, but because the x_i are no longer ordered the ternary constraints are modified to $d_{ij} = |x_i - x_j|$. The symmetry breaking constraint $d_{12} < d_{m-1,m}$ is not used but the unary constraints $x_1 = 0$ and $x_m = \ell$ are (these decisions are somewhat arbitrary but gave best results). Though a solution to this problem is no longer necessarily a Golomb ruler, one can be derived in polynomial time by sorting the x_i into ascending order.

Figure 4 shows mean results over 50 runs for Golomb rulers with various lengths ℓ and numbers of marks m, and the model M either "natural" (N) or supersymmetric (S). Best results for Walksat [8] are reported using noise values from 5% to 95% in steps of 5%; and best results for Saturn with noise values {5, 10, 15, 20, 25, 30, 40, 50, 70, 90, 110, 130, 150} (less tuning effort was required at higher noise levels).

			Walksat		Saturn	
m	ℓ	M	flips	sec.	back.	sec.
4	6	S	139	0.002	126	0.002
4	6	N	492	0.005	1467	0.020
5	13	S	397	0.033	1751	0.35
5	13	N	1042	0.058	8460	1.71
5	11	S	564	0.023	1534	0.19
5	11	N	1509	0.050	8435	1.12
6	21	S	1897	0.35	12688	9.0
6	21	N	4579	0.75	68250	49.0
6	19	S	2390	0.30	14101	8.3
6	19	N	3007	0.33	111128	66.5
6	17	S	3736	0.31	36304	17.0
6	17	N	11233	0.82	166549	81.5

Fig. 4. Results on Golomb rulers

On these problems Walksat is much faster than Saturn. However, both consistently perform better on the supersymmetric models, both in terms of steps and execution times, showing that supersymmetry can improve local search performance. We have not yet found a successful way to apply this approach to the Social Golfer problem: a first attempt (details omitted for space reasons) introduced many new variables and gave inferior results. But we believe that the approach is potentially useful and hope to find other examples in future work.

Acknowledgment The Cork Constraint Computation Centre is supported by Science Foundation Ireland.

References

1. M. Davis, G. Logemann, D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM* vol. 5, 1962, pp. 394–397.
2. E. C. Freuder, P. D. Hubbe, D. Sabin. Inconsistency and Redundancy do not Imply Irrelevance. *Proceedings of the AAAI Fall Symposium on Relevance*, New Orleans, LA, USA, 1994.
3. I. P. Gent, B. Smith. Symmetry Breaking During Search in Constraint Programming. *Proceedings of the Fourteenth European Conference on Artificial Intelligence*, 2000, pp. 599–603.
4. S. D. Prestwich. First-Solution Search with Symmetry Breaking and Implied Constraints. *CP'01 Workshop on Modeling and Formulation*, Cyprus, 2001.
5. S. D. Prestwich. Randomised Backtracking for Linear Pseudo-Boolean Constraint Problems. *Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, le Croisic, France, 2002, pp. 7–20.
6. P. Prosser. Domain Filtering Can Degrade Intelligent Backjumping Search. *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, Chambery, France, Morgan Kaufmann 1993, pp. 262–267.
7. D. Sabin, E. C. Freuder. Contradicting Conventional Wisdom in Constraint Satisfaction. *Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 874, Springer-Verlag 1994, pp. 125–129.
8. B. Selman, H. Kautz, B. Cohen. Noise Strategies for Improving Local Search. *Proceedings of the Twelfth National Conference on Artificial Intelligence*, AAAI Press 1994, pp. 337–343.
9. B. Selman, H. Levesque, D. Mitchell. A New Method for Solving Hard Satisfiability Problems. *Proceedings of the Tenth National Conference on Artificial Intelligence*, MIT Press 1992, pp. 440–446.
10. B. Smith. Reducing Symmetry in a Combinatorial Design Problem. *Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Ashford, Kent, England, 2001, pp. 351–359.
11. B. Smith, K. Stergiou, T. Walsh. Modeling the Golomb Ruler Problem. Research Report 1999.12, University of Leeds, England, June 1999 (presented at the IJCAI'99 Workshop on Non-binary Constraints).

Symmetry Breaking via Dominance Detection for Lookahead Constraint Solvers

Igor Razgon and Amnon Meisels

Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, 84-105, Israel
{irazgon,am}@cs.bgu.ac.il

Extended Abstract

1 Introduction

A general method for detecting symmetric choice points during search was recently proposed in [3]. The method achieves symmetry breaking by detecting the dominance of search subspaces and eliminating the need to explore them.

Let us sketch the principle of this method applied to a search algorithm returning the first solution found. Assume that a search algorithm currently explores some node P of the search tree. Let S be the remaining search space induced by P [1]. Let P' be some node of the search tree that has already been rejected by the algorithm and S' be the search space induced by P'. If $S \subseteq S'$ than P can be rejected without further search in S. We say that P' dominates P.

The essence of *dominance checking* methods is that before starting search over the search space S induced by the current node P, the algorithm checks whether P is dominated by some previously considered node and if so, P is rejected without further search.

A search algorithm that explores different value choices on variables can potentially eliminate values that are dominated by values

[1] when we say "induced", we mean that S is obtained as the result of application of some procedure maintaining consistency with the current node of the search tree

of the same variable that were already rejected. The present paper proposes a complete search algorithm for constraint satisfaction problems (CSPs) that utilizes lookahead techniques for dominance detection. Lookahead techniques like forward checking (FC) check constraints with all unassigned variables and produce domains of values that change dynamically during search [9]. Our proposed algorithm utilizes the dynamic nature of domains of values of future variables to detect dominance and thereby eliminate values that are dominated by former (rejected) values.

Consider a backtrack based constraint solver, that utilizes some lookahead procedure *Proc*. This means that the solver runs *Proc* after every assignment of a value to a variable to achieve some level of consistency for the remaining constraint network of unassigned variables. In the case of forward checking *Proc* achieves node-consistency, in the case of MAC it achieves arc-consistency [9].

Let P be the current partial solution at some moment of the execution of the solver. Denote by F_P the set of domain values of all unassigned variables for the partial solution P, after application of *Proc*. Let us call F_P the set of *future values* of P.

Let V be the current variable, val_1 and val_2 be values from the current domain of V, and denote by $\langle V, val \rangle$ the assignment of val to V. The dominance detection property can be formulated as follows: if $P \cup \{\langle V, val_1 \rangle\}$ has been rejected and
$F_{P \cup \{\langle V, val_2 \rangle\}} \subseteq F_{P \cup \{\langle V, val_1 \rangle\}}$ then the assignment $\langle V, val_2 \rangle$ can be rejected without further search. In other words, if the assignment $\langle V, val_1 \rangle$ *dominates* the assignment $\langle V, val_2 \rangle$ given the current partial solution P and $\langle V, val_1 \rangle$ is inconsistent, then $\langle V, val_2 \rangle$ is also inconsistent.

Section 2 presents the proposed method of dominance checking. For constraint networks of n variables and maximal domain size of m, the proposed method has an $O(nm^2)$ running time and needs memory bounded by $O(n^2m^2)$.

For dominance detection on a CSP search algorithm the order in which values are checked is important. When values are ordered in decreasing order of future compatible domain sizes, the checking of dominance becomes easier. A new dominance relation that orders values is presented in Section 2. The *minimal dominating set*

construction heuristic requires $O(tm + m \log m)$ time where t is the complexity of the lookahead procedure of the solver.

Section 3 presents preliminary experimental results of FC with minimal dominating set construction for random CSP's. The results demonstrate that the proposed approach is promising. Further developments of the proposed algorithm are discussed.

2 Dominance Detection for Domain Shrinking

2.1 A Dominance Checking for CSP's

Definition 1. *Let P be the current partial solution. Let val be a value from the domain of some unassigned variable. If $val \in F_P$, we say that val is consistent with P.*

Given this definition, we can reformulate the dominance definition as follows.

Definition 2. *Let P, V be the current partial solution and the current variable to be assigned. An assignment $\langle V, val_1 \rangle$ is dominated by assignment $\langle V, val_2 \rangle$ given P if every domain value of an unassigned variable consistent with $P \cup \{\langle V, val_1 \rangle\}$ is also consistent with $P \cup \{\langle V, val_2 \rangle\}$.*

The last definition provides a simple method for dominance checking on constraint networks. Let $\{val_1, \ldots, val_k\}$ be the already rejected subset of the current domain of the current variable. To perform the dominance checking for val_{k+1} we simply check whether val_{k+1} is dominated by either $\langle V, val_1 \rangle, \langle V, val_2 \rangle, ..., \langle V, val_k \rangle$ given P.

The number of values consistent with $P \cup \{\langle V, val_{k+1} \rangle\}$ is $O(nm)$. The number of previously rejected values is $O(m)$. Assuming that the complexity of a consistency check is $O(x)$ we obtain the complexity of the method as $O(nm^2x)$. To decrease the complexity of the proposed method we have to decrease the complexity of the consistency check.

In order to implement an $O(1)$ consistency check method one needs to maintain a data structure for the current partial solution. One possibility is to use a *consistency array* C_P. Let val and val' be a value of the current variable and a value of some other unassigned

variable respectively. Then, $C_P[val][val'] = 1$ if $P \cup \{\langle V, val\rangle\}$ is consistent with val', otherwise $C_P[val][val'] = 0$. To check whether $\langle V, val\rangle$ is dominated by some previous assignment of V, it has to be compared with C_P entries for previous assignments. All entries are computed during the execution of the lookahead procedure at no additional cost. Given this data structure a consistency check is simply reading an entry from the array and takes time $O(1)$.

Note that C_P is deleted only when C_P becomes irrelevant (some of its assignments are deleted or changed). This condition implies that a number of consistency arrays are maintained in the memory simultaneously.

Assume $\{\langle V_1, val_1\rangle, \langle V_2, val_2\rangle, ..., \langle V_k, val_k\rangle\}$ is the partial solution. Then the following consistency arrays corresponding to the growing subsets of the partial solution are maintained simultaneously in memory: $C_{\{\langle V_1,val_1\rangle\}}, C_{\{\langle V_1,val_1\rangle,\langle V_2,val_2\rangle\}}, C_{\{\langle V_1,val_1\rangle,...,\langle V_k,val_k\rangle\}}$. That is, there are $O(n)$ consistency arrays in the general case. Every one of these arrays takes $O(nm^2)$ memory. So all arrays together take $O(n^2m^2)$ memory.

The proposed dominance checking method needs $O(nm^2)$ time and $O(n^2m^2)$ memory.

2.2 The Minimal Dominating Set

The order of assignment of domain values may greatly affect the effectiveness of the dominance detection methods. Let $\{val_1, val_2, \ldots, val_m\}$ be the order of assignments of values to the current variable V enumerated in the order of their consideration.

Assume that $\langle V, val_i\rangle$ is dominated by $\langle V, val_{i+1}\rangle$ for all i, $1 \leq i < m$. This means that in spite of the fact that the domain contains a large number of symmetries they will not be discovered by dominance checking methods. In this subsection we demonstrate how it is possible to overcome this drawback.

Let us define the notion of a *dominating subset* of a domain. Every value in a domain either belongs to the dominating subset or is dominated by one of its members. A *minimal dominating subset* is a dominating subset of a domain of the minimal size.

Lemma 1. *Let P be the current partial solution. Let V be the current variable, and $\{val_1, val_2, \ldots, val_m\}$ be the set of values of the current domain of V enumerated in the order of their instantiation.*

If $|F_{P\cup\{\langle V,val_i\rangle\}}| \geq |F_{P\cup\{\langle V,val_{i+1}\rangle\}}|$ for all $i = 1, \ldots, m-1$, then any dominance checking method (say, the one presented in the previous subsection) will explore a minimal dominating subset of V

Roughly speaking, to consider a dominating subset of minimal size we have to instantiate values in a decreasing order of their sizes of future subspaces. Note that given such an order, no value can dominate a value considered before it unless the sizes of their sets of future values are equal.

Proof.

Assume that the lemma does not hold. Let DS be the dominating subset explored by the proposed method. Let also DS_{min} be a minimal dominating subset of the domain of the current variable and $|DS_{min}| < |DS|$.

Order the values of DS and DS_{min} in the order proposed in the lemma and consider their maximal common prefix. Let val^{DS} and $val^{DS_{min}}$ be the values following this prefix in DS and DS_{min} respectively.

Consider three possible cases:

1. $|F_{P\cup\{\langle V,val^{DS}\rangle\}}| < |F_{P\cup\{\langle V,val^{DS_{min}}\rangle\}}|$
2. $|F_{P\cup\{\langle V,val^{DS}\rangle\}}| > |F_{P\cup\{\langle V,val^{DS_{min}}\rangle\}}|$
3. $|F_{P\cup\{\langle V,val^{DS}\rangle\}}| = |F_{P\cup\{\langle V,val^{DS_{min}}\rangle\}}|$

In the first case, $val^{DS_{min}}$ has been considered already by the proposed method and has not been included in the dominating subset DS. We deduce that $val^{DS_{min}}$ is dominated by a value from the common prefix of DS and DS_{min}. That is, DS_{min} is not minimal which contradicts our initial assumption.

In the second case val^{DS} is not included in DS_{min}. But val^{DS} is not dominated by any value from the common prefix of the considered sets, because otherwise it would not have been included in DS. In addition, val^{DS} is not dominated by any other value in DS_{min}, because all other values have sizes of sets of future values less than val^{DS} (by our ordering condition). So val^{DS} does not belong to DS_{min} and is not dominated by any value from it. Therefore,

DS_{min} is not a dominating subset of the considered domain which contradicts our assumption.

In the third case two subcases are possible:

- $F_{P\cup\{\langle V,val^{DS}\rangle\}} = F_{P\cup\{\langle V,val^{DS_{min}}\rangle\}}$
- $F_{P\cup\{\langle V,val^{DS}\rangle\}} \neq F_{P\cup\{\langle V,val^{DS_{min}}\rangle\}}$

In the first subcase we can replace $val^{DS_{min}}$ by val^{DS} in DS_{min} contradicting our assumption about the maximality of the common prefix. In the second subcase DS_{min} should contain some value val' such that $F_{P\cup\{\langle V,val^{DS}\rangle\}} = F_{P\cup\{\langle V,val'\rangle\}}$. We can change the order of val' and $val^{DS_{min}}$ without violating our ordering condition. In this way we reduce the second subcase to the first one considered above. □

To order values of the considered domain by the presented lemma, we have to compute $|F_{P\cup\{\langle V,val\rangle\}}|$ for every value of the current domain of the current variable V. Therefore, we apply the lookahead procedure $Proc$, $O(m)$ times. If the complexity of $Proc$ is $O(y)$, the complexity of this method is $O(ym + m\log m)$.

Note, that the ordering of domain values of the current variables in the proposed order is a good search heuristic even without consideration of symmetry breaking [2].

3 Realization and Testing

To check the effectiveness of the proposed method, we implemented two versions of FC. The first version has the value ordering proposed in Lemma 1. The second version performs dominance checking in addition to value ordering. Both algorithms use the fail-first variable ordering heuristic [5]. We compared these two algorithms on a set of randomly generated problems characterized by their constraints density p_1 and tightness p_2 [6].

Our preliminary results show that instances of randomly generated problems with a large number of symmetries are mainly concentrated in regions with either low density or low tightness. More formally, these regions are characterized by either $p_1 \leq 0.3$ or ($p_2 \leq 0.3$ and $0.3 < p_1 < 0.8$). For difficult problems in these regions, the FC version with dominance detection outperforms the version without it. For easy problems (without backtracks), the dominance detection

method is not applied at all, that is the computation effort is the same for both these algorithms. The other set of constraint problem is characterized by a small number of symmetries or by their total absence. Dominance detection increases running time of the solver when applied to these problems. But this additional time is not great, because the proposed dominance checking method frequently returns a negative answer in an early stage of its work

Selected results of our experiments are demonstrated in Table 1 and illustrate our observations. All problems have 20 variables and 12 values in all domains. The first 2 columns of Table 1 give the density and tightness of problem instances. The 3rd column presents the number of constrain checks (CCs) for the proposed method and the 4th column presents the CCs performed by FC with just ordering of values. The results in the table are the averages of 10 runs for the same problem parameters.

p_1	p_2	symm-cc	unsymm-cc
0.2	0.1	28595	28595
0.2	0.2	23056	23056
0.2	0.3	17980	17980
0.2	0.4	14081	14081
0.2	0.5	10713	10528
0.2	0.6	29817	33661
0.2	0.7	38402	54512
0.2	0.8	11782	13788
0.2	0.9	5694	4970

Table 1. Comparison of FC's with and without Domiance Checking

These results in Table 1 can be divided into the following three groups

1. Both versions do not execute backtracks. Therefore, the dominance checking mechanism is not applied at all (first four rows of the table).
2. The version with dominance checking outperforms the version without it, by discovering dominances. This group of results occurs for instances with p_2 varying from 0.6 to 0.8.
3. The dominance detection does not succeed to reduce the computational effort ($p_2 = 0.5, 0.9$). Note, that in this case the addi-

tional cost of dominance checking is small relatively to the benefit we gain for the instances of the second group

Based on our experiments, we conclude that domain shrinking methods based on construction of minimal dominating subsets are promising for a subregion of $\langle p_1, p_2 \rangle$ and need further investigation. These methods may be useful for CSP's of large size with low density, which are frequently difficult to solve by complete algorithms [8].

The most important step of the further investigation is to find a successful variable ordering heuristic. Ideally, such a heuristic should be successful for search as well as for construction of small dominating subsets. Heuristics based on constrainedness evaluation [1, 4, 7] seem to be a good choice for this purpose. According to our preliminary experiments, symmetries are concentrated in regions with low constrainedness.

Yet another direction in the development of the proposed method is to find a more effective algorithm for ordering of values in the proposed form of Lemma 1. A naive method applying the lookahead procedure m times can be replaced by a more sophisticated approach.

References

1. C. Bessiere, A. Chmeiss, L. Sais *Neightborhood-Based Variable Ordering Heuristics for the Constraint Satisfaction Problem*, Proceedings of CP2001, pp.565-570, Paphos, 2001.
2. D. Frost, R. Dechter, *Look-ahead value ordering for constraint satisfaction problems*, Proceedings of IJCAI-95, pp. 572-578, Montreal, Canada, August 1995.
3. T.Fahle, S.Schamberger, M. Sellmann *Symmetry Breaking*, Proceedings of CP2001, pp. 93-107, Paphos, 2001.
4. I. P. Gent, E. MacIntyre, P. Prosser, T. Walsh *The Constrainedness of Search*, AAAI/IAAI96, Vol. 1, pp. = 246-252, 1996.
5. R.M. Haralick, G.L.Eliott *Increasing tree search efficiency for constraint satisfaction problems*,Artificial Intelligence, 14:263–313, 1980.
6. P. Prosser *Binary constraint satisfaction problems: some are harder than others* , Proc. ECAI-94, pp.95-99, 1994.
7. I. P. Gent, E. MacIntyre, P. Prosser, B.M. Smith, T. Walsh *An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem* Proccedings of CP96, pp. 179-193, 1996.
8. B. Smith, S. Grant, *Where the Exceptionally Hard Problems Are*, CP95 Workshop on Really Hard Problems Cassis, September 1995.
9. E.Tsang *Foundations of Constraint Satisfaction*, Academic Press Limited, 1993,

Symmetry Breaking for Boolean Satisfiability: The Mysteries of Logic Minimization

Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah
The University of Michigan, Ann Arbor
{faloul, imarkov, karem}@umich.edu

Abstract

Boolean Satisfiability solvers improved dramatically over the last seven years [14, 13] and are commonly used in applications such as bounded model checking, planning, and FPGA routing. However, a number of practical SAT instances remain difficult to solve. Recent work pointed out that symmetries in the search space are often to blame [1]. The framework of symmetry-breaking (SBPs) [5], together with further improvements [1], was then used to achieve empirical speed-ups.
For symmetry-breaking to be successful in practice, its overhead must be less than the complexity reduction it brings. In this work we show how logic minimization helps to improve this trade-off and achieve much better empirical results. We also contribute detailed new studies of SBPs and their efficiency as well as new general constructions of SBPs.

1 Introduction

Many search, synthesis and optimization problems arising in algorithmic applications exhibit symmetries. The presence of multiple, symmetric solutions may lead to degeneracy and slow down known algorithms for such problems. Symmetries can make it more difficult to conclude that a a given instance of a search problem has no solutions - because symmetric sub-instances may be independent. However, once the symmetries are identified, it is often easy for people to "mod out" by symmetry and simplify the problem at hand. Of course, when the number of symmetries is high, even simple bookkeeping requires a computer program.
In this work we study the Boolean SATisfiability problem (CNF-SAT) - one of the most important in Computer Science - in the presence of symmetry. Previous work contributed the framework of symmetry-breaking predicates that proceeds as follows.

1. The symmetries are identified using a reduction to the Graph Automorphism problem.
2. Symmetry-breaking predicates (SBP) are produced.
3. A SAT-solver is applied to the conjunction of the original formula and symmetry-breaking predicates.

A reduction to Graph Automorphism that detects all permutational symmetries was proposed in [5]. That work also contributed the first general construction of SBPs for single permutations in tabular form. The authors pointed out that constructing SBPs for every symmetry is often impractical, and developed the concept of *symmetry-tree* that reduces the number of clauses added to the original CNF formula. Yet it does not always avoid exponential number of added clauses.
The symmetry-breaking framework was further improved in [1] which contributed

- a new reduction to Graph Automorphism that detects all permutational symmetries, phase shifts and their compositions (we call them *mixed symmetries*),
- empirical evaluation of symmetry-breaking performed only for generators of the symmetry group,
- more efficient symmetry-breaking predicates for mixed symmetries in cycle notation,
- strong empirical evidence that symmetry-breaking is practically useful when best available SAT-solvers are used.

In this paper we further improve the symmetry-breaking framework for CNF-SAT using logic minimization to simplify SBPs and empirically improve runtime of SAT-solvers. We contribute new constructions of full and partial symmetry-breaking predicates of smaller size, produced in many cases by explicit logic optimization.
Analyses of symmetry-breaking include orbit counts and estimates of efficiency of symmetry-breaking predicates. Additionally, we provide some justification for breaking the symmetries of only the generators. A number of related questions are still open however.
The remaining part of the paper is organized as follows. Section 2 presents the necessary definitions and notation. Section 3 covers previous work. More efficient SBP constructions are given in Section 4, and a theoretical discussion of symmetry-breaking by generators in Section 5. We show experimental results in Section 6, and the paper concludes in Section 7.

2 Definitions and Notation

Intuitively, a symmetry of a discrete object is a transformation, e.g., permutation, of its components that leaves the object intact. For example, the six permutational symmetries of the equilateral triangle can be thought of as permutations of its vertices. Symmetries are studied in abstract algebra in terms of *groups* [6].

A *group* is a set with a binary associative operation (often thought of as multiplication) defined on it such that there is a *unit* element and every element has a unique inverse. A group is called Abelian if the operation is commutative, e.g., the *cyclic* group of order *k* consists of powers of *g* modulo *k*. In general, a set of group elements such that any other group element can be expressed as their product is called a *generating set*. Any irredundant generating set is no greater than the binary logarithm of the group size and can be much smaller. A *subgroup* of a group is a subset that is closed under the operation.

The symmetric group $S(\Omega)$ on a finite set Ω is the group of all permutations of Ω. If $|\Omega| = n$, the group is commonly denoted by S_n. A *homomorphism* from group *G* to group *H* is a mapping such that a product of two group elements is mapped to the product of their images, and such a mapping is called an *isomorphism* when its inverse exists and is a homomorphism. A group *G acts* on a set Ω when a homomorphism is given from *G* to $S(\Omega)$. For an element of Ω, its *G-orbit* is the set of elements of Ω to which it can be mapped by elements of *G*. Orbits define an equivalence relation on Ω.

Permutations of Ω, often denoted by lower-case Greek letters, can be written in *tabular form* where the elements of Ω are written in the first row and their images in the second row. For example, the image of element *i* under the permutation π will be denoted i^{π} and written below *i*. We also use *cycle notation*, which can be produced from the tabular notation by (i) constructing directed edges from elements of Ω to their images, and (ii) listing the disjoint cycles of this directed graph. Single-element cycles are implicit and never listed, and two-element cycles are called *transpositions*. For example, (12)(456) can denote a permutation that swaps elements 1 and 2, maps 4 to 5, 5 to 6 and 6 to 4. Cycle notation is preferable to tabular notation for *sparse* permutations that map most of elements of Ω to themselves. The set of elements that are not mapped to themselves is called the *support* of the permutation. The *cycle type* of a permutation is a sequence of integers n_i for $i \in \{2, \ldots, k\}$; n_i is the number of *i*-cycles in the permutation.

A symmetry (*automorphism*) of a graph is a permutation of its vertices that maps edges to edges. If vertices are labeled by integers (*colored*), one may additionally require that symmetries preserve labels.

Consider the set of Boolean variables $x_1, \ldots, x_n$. A *literal* is either a variable or its negation. A *clause* is a disjunction of literals, e.g., $(x_3 + \overline{x_6} + x_7)$, and a *CNF formula* is a conjunction of clauses, e.g., $(x_3 + \overline{x_6} + x_7)(x_{10} + \overline{x_{11}})(x_{20})$. A *binary* clause has two literals and can be viewed as an implication between variables, e.g., $(x_{11} \Rightarrow x_{10})$. An instance of the CNF-SAT decision problem is given by a CNF formula, the question is whether the formula can be satisfied by an assignment of Boolean values to the variables.

We will assume a total ordering of variables $x_1, \ldots, x_n$ and consider the induced lexicographic ordering of the 2^n truth assignments, i.e., 0-1 strings of length *n*. We now assume that a group acts on the set of literals, subject to the *Boolean consistency* constraint, which requires that if $(a \rightarrow b)$ then $(\bar{a} \rightarrow \bar{b})$ for any literals *a* and *b*. Such an action unambiguously induces a corresponding action on the set of truth assignments. We focus on orbits of this action. The *lex-leader* of an orbit is defined as the lexicographically smallest element. A *lex-leader predicate* (LL-predicate) for the action is a Boolean function on $x_1, \ldots, x_n$ that evaluates to true only on lex-leaders of orbits.

Consider a permutation on the set of literals. Given a CNF formula, we can permute literals in it, potentially changing the formula. A permutation of literals is a symmetry of a given CNF formula if Boolean consistency is observed and the formula is preserved under the permutation (in other words, every clause must map into a clause with the same polarities of literals). In particular, we consider simultaneous negations of sets of variables (*phase-shifts*) and compositions of permutations and phase-shifts (*mixed symmetries*). Given a CNF formula, we consider its group of symmetries and its corresponding action on truth assignments. A symmetry-breaking predicate (SBP) is a Boolean function that evaluates to true on at least one element of each orbit of the group of symmetries. In this work, we will consider SBPs that are expressed by CNF formulae, and the *size* of an SBP is the number of literals its CNF involves. Observe that adding an SBP to the original CNF formula does not affect the satisfiability, but restricts the possible solutions to those selected by the SBP.

A *full* SBP is an SBP that selects exactly one element of each orbit, otherwise we call an SBP *partial*. A *lex-leader SBP* (LL-SBP) is an SBP that selects lex-leaders only. An LL-SBP is a full SBP. For SBPs that are not full, it is often important that they select lex-leaders, among other elements. We call such SBPs *partial lex-leader SBPs* (PLL-SBPs).

3 Previous Work

3.1 The Natural LL-SBP of Crawford et al.

Let $x = (x_1, x_2, \ldots, x_n)$ be a vector of Boolean variables, $\Pi = \{\pi_1, \ldots, \pi_m\}$ be a permutation group acting on x. A PLL-SBP $\mathrm{PLL}(\pi)$ is defined for every group element, and their conjunction is an LL-SBP for Π [5]. The construction relies on the ordering $x_1 < x_2 < \ldots < x_n$:

$$\mathrm{PLL}(\pi) = \mathrm{AND}_{1 \le i \le n} C(\pi, i) \quad (3.1)$$

$$C(\pi, i) = [\mathrm{AND}_{1 \le j \le i-1}(x_j = x_j^{\pi})] \rightarrow (x_i \le x_i^{\pi}) \quad (3.2)$$

Introducing auxiliary variables $e_j = (x_j = x_j^{\pi})$, this predicate yields a CNF formula with $5n$ clauses and $0.5n^2 + 13.5n$ literals. The SBP for the entire group is:

$$\mathrm{LL}(\Pi) = \mathrm{AND}_{1 \le i \le m}(\mathrm{PLL}\pi_i) \quad (3.3)$$

This predicate may contain redundant clauses, in particular some $C(\pi, i)$ may be tautologies and $\mathrm{PLL}(\pi)$ for different permutations may have identical clauses. Moreover, there are often exponentially many symmetries, and the natural LL-SBP for the entire group is infeasible. Crawford et al. construct a *symmetry tree* to prune "redundant" symmetries, but that may still yield exponential-sized LL-SBPs. Additionally, they have shown NP-completeness of identifying lex-leaders in certain circumstances. A more recent work by Luks and Roy [9] studies the complexity LL-SBPs for Abelian permutation groups and shows that, in general, LL-SBPs may have to be exponentially large. However, careful re-ordering of variables enables polynomial-sized formulas.

3.2 Further Improvements

3.2.1 CNF Symmetries via Graph Automorphism

While CNF symmetries are not limited to permutations of variables, we build, for a given CNF formula, a graph such that the group of CNF symmetries is isomorphic to the group of graph automorphisms. A simple construction represents every clause by a vertex of color 2, and every variable by two vertices of color 1 (one for positive literals and one for negative) connected by *Boolean consistency edges*. Every literal in the CNF formula then is represented by a bi-partite edge. An improved construction treats binary clauses differently. It leaves out their clausal vertices and connects their literal vertices by double-edges. Since some graph automorphism programs do not allow double-edges, the work in [1] uses a model with single edges which can result in spurious graph automorphisms (one-sided error) if the original CNF formula contains binary clauses forming circular chains of implications. Fortunately, this rarely happens in CNF applications and spurious graph symmetries can be easily tested for. In our experiments, no spurious symmetries were detected.

The graph automorphism problem is believed to be outside P, yet not NP-complete. Common algorithms finish in linear time if the only symmetry is trivial, and polynomial time is guaranteed in the bounded-degree case [2,3]. Finding CNF symmetries is often easier than solving SAT, and excellent software is available [11, 12].

3.2.2 Breaking Symmetries By Generators

Breaking *all* symmetries may not speed up search because there are often exponentially many of them and their PLL-SBPs may be redundant [5]. It is not necessary either. Breaking *enough* symmetries, whose SBPs are short CNF clauses, may provide a better trade-off.

Irredundant generators are good candidates for symmetries to be broken because they cannot be expressed in terms of each other, which minimizes redundancies. A potential concern is that breaking generators alone may lead to the selection of more than one element from some orbits, an example with $|\Omega| = 4$ was provided to us by Eugene Goldberg. Yet, such partial symmetry breaking may be the best option because its overhead is small. When building LL-SBPs for different generators, one must use the same variable order. An LL-SBP for a given symmetry generator can be constructed as the natural LL-SBP for the cyclic group generated by it, or a PLL can be used.

3.2.3 Breaking Symmetries by Cycles

For a given permutation, choosing an SBP with fewer literals is also important for empirical success. To this end, the construction of small SBPs from [1] is based on the cycles of a permutation (with subsequent chaining) instead of the entire permutation in tabular form. This eliminates redundancies in the SBPs of individual permutations and is convenient because generators returned by graph automorphism programs are often sparse, i.e., involve very few variables. Additionally, one can create a catalog of LL-SBPs or PLL-SBPs for small cycle lengths. This improves upon the natural LL-SBP construction which entails a conjunction over non-trivial powers of a given permutation.

A major problem with cycle-based constructions arises when they are applied to symmetry generators and may influence the order of variables. LL-SBPs for all generators must use the same order of variables, but not necessarily the original order. In some applications the original order of variables works, and in many an appropriate order can be found quickly.

4 New Constructions for Single Perms

The main idea behind our new constructions is to decompose a permutation into cycles, construct SBPs for individual cycles and then chain them together.

4.1 Extensions and Improvements of The Natural LL-SBP Construction

We describe in this section a) an extension to Crawford's construction of the LL-SBP for a single permutation to enable the handling of mixed symmetries, and b) a simplification of the LL-SBP that yields a linear-sized CNF formula.

Without loss of generality, every symmetry that is a composition of permutations and pure phase shifts can be decomposed into a product of a permutation and a phase-shift acting after the permutation. That is because the group of phase shifts is *normal* in the group of compositions, i.e., the composition of an arbitrary permutation, an arbitrary phase shift and the permutation's inverse is necessarily a phase shift (not necessarily the same as the original). Such mixed symmetries can be incorporated in the construction of the LL-SBP by allowing a variable x_i to be mapped into either its image under the permutation x_i^π or the negation of that image $\overline{x_i^\pi}$ induced by the phase shift action.

Crawford's construction in (3.1) and (3.2) for the LL-SBP of a single-permutation can be simplified to yield a CNF formula whose size is linear, instead of quadratic, in the number of variables. To facilitate the following derivation let $l_i = (x_i \le x_i^\pi)$, $g_i = (x_i \ge x_i^\pi)$, and $g_0 \equiv 1$. Noting that $e_i = l_i g_i$, Crawford's LL-SBP can now be expressed as:

$$(g_0 \to l_1)(g_0 l_1 g_1 \to l_2)\ldots(g_0 l_1 \ldots l_{n-1} g_{n-1} \to l_n) \quad (4.1)$$

Factoring out the common prefix g_0 yields:

$$g_0 \to [l_1 \cdot (l_1 g_1 \to l_2)\ldots(l_1 g_1 \ldots l_{n-1} g_{n-1} \to l_n)] \quad (4.2)$$

which simplifies further, through absorption, to:

$$g_0 \to [l_1 \cdot (g_1 \to l_2)\ldots(g_1 l_2 \ldots l_{n-1} g_{n-1} \to l_n)] \quad (4.3)$$

The recursive structure of the formula is now revealed by comparing (4.1) and (4.3). Let $p_1, \ldots, p_n$ be a sequence of predicates defined by:

$$\begin{aligned} p_i \;=\; & (g_{i-1} \to l_i) \cdot (g_{i-1} l_i g_i \to l_{i+1}) \\ & \ldots(g_{i-1} l_i g_i \ldots l_{n-1} g_{n-1} \to l_n) \end{aligned} \quad (4.4)$$

and let $p_{n+1} = 1$. Then,

$$p_i \;=\; g_{i-1} \to l_i p_{i+1} \qquad i \;=\; 1, \ldots, n \quad (4.5)$$

Note that predicate p_1 represents the entire formula (4.1). The satisfiability of (4.1) can, thus, be determined by checking the satisfiability of the following equivalent, but simpler formula:

$$(p_1)(p_1 = g_0 \to l_1 p_2)\ldots(p_n = g_{n-1} \to l_n p_{n+1}) \quad (4.6)$$

One final simplification replaces the equalities in (4.6) with implications since we are only interested in satisfying each of the predicates. We thus obtain:

$$(p_1)(p_1 \to g_0 \to l_1 p_2)\ldots(p_n \to g_{n-1} \to l_n p_{n+1}) \quad (4.7)$$

The CNF representation of (4.7) consists of $2n$ 3-literal and $2n$ 4-literal clauses for a total size of $14n$ literals.

4.2 Orbits of Cyclic Symmetries

While the cyclic group of size n naturally acts on the set 1..n, this action can be extended to the Boolean cube of 2^n truth assignments. Orbits of this action are called *necklaces* in combinatorics, and their number can be found using the celebrated counting theorem by Polya [6, Theorem 14.5]:

$$B(n) \;=\; (1/n)\sum_{d|n} 2^d \phi(n/d) \quad (4.8)$$

where the sum is taken over all divisors of *n*. $\phi(m)$ is the *Euler's totient function*, i.e., the number of positive integers not exceeding *m* which are relatively prime to *m* (1 is counted as being relatively prime to all numbers). The first several values (for *n=2,3,4,5*) of $B(n)$ are 3,4,6,8 and can be verified by direct counting. For prime *p*, $\phi(p) = p-1$ and thus the number of orbits is $(2(p-1)+2^p))/p$. Indeed, every orbit contains *p* elements, except for the two one-element orbits *00...0* and *11...1*. In general, orbit sizes must divide cycle length, and the number of necklaces is lower-bounded by $(2(n-1)+2^n))/n$. Thus it grows exponentially. The asymptotic estimate $\Theta(2^n/\mathrm{n})$ for the number of necklaces can be produced using $\phi(n/d) \le n-1$. Therefore the efficiency of full SBPs for single cycles is not bounded, at least as far as reductions in search space are concerned.

4.3 Full Symmetry-Breaking for Single Cycles

If for every literal in a given cycle, its complementary literal is not in the cycle, the natural LL-SBP construction applies and can be improved as described above. Otherwise, we must apply the extended natural LL-SBP construction.

Lemma 4.1 If a symmetry of a CNF formula has a cycle of length n that contains literals a and $\bar{a}$, then

1. Every variable participating in the cycle, has both positive and negative literal in the cycle.
2. n must be even.
3. The distance between every pair of complementary literals is $n/2$.
4. Any contiguous sub-word of length $n/2$ determines the remaining part of the cycle.

Proof: 1. The literal a can map to any literal b in the cycle, therefore by Boolean consistency $\bar{a}$ must map to $\bar{b}$, and $\bar{b}$ must be in the cycle. 2. Follows from 1. To prove 3. notice that the hop-distance from a to $\bar{a}$ in the cycle must equal that from $\bar{a}$ to a (by Boolean consistency), and the two add up to n. 4. Any contiguous sub-word of length $n/2$ contains exactly one literal of every variable participating in the cycle and unambiguously determines where it maps (the last literal in the word must map to the complement of the first literal); the images of literals not in the sub-word are unambiguously determined by Boolean consistency.

Minimal LL-SBPs can be created for single-cycle permutations by constructing the Boolean function that selects the lex-leader of each orbit and minimizing this function using standard logic minimization procedure ESPRESSO[10]. Alternatively, a natural PLL-SBP can be created according to (3.1) for each permutation in the cyclic group generated by the given cycle; the minimal LL-SBP for the cycle is now obtained by minimizing the conjunction of these formulas. A listing of minimal LL-SBPs for k-cycles of various length is give in table Table 1. It is interesting to note that the LL-SBP for cycles whose length is less than 6 is composed entirely of binary clauses. We produced minimal LL-SBPs for cycles of length up to 20. Such LL-SBPs contain clauses of increasing lengths, but also k binary clauses. Since longer clauses are less effective during search, one may prefer to use a PLL-SBP that consists only of binary clauses.

4.4 Partial SBPs For Single Cycles

This is motivated by the desire to create strong predicates that can speed up search.

Theorem 4.2 A partial LL-SBP for the n-cycle is possible with $2n-2$ binary clauses.

Proof: Suppose we have a symmetry which is an n-cycle on variables $x_1, \ldots, x_n$. A set of binary clauses is

Table 1: Minimal LL-SBPs for k-cycles

Perm	LL-SBP
(x_1x_2)	$(\bar{x_1} + x_2)$
$(x_1x_2x_3)$	$(\bar{x_1} + x_2)(\bar{x_2} + x_3)$
$(x_1x_2x_3x_4)$	$(\bar{x_1} + x_2)(\bar{x_1} + x_3)(\bar{x_2} + x_4)(\bar{x_3} + x_4)$
$(x_1x_2x_3x_4x_5)$	$(\bar{x_1} + x_2)(\bar{x_1} + x_3)(\bar{x_2} + x_4)$ $(\bar{x_2} + x_5)(\bar{x_3} + x_5)$
$(x_1x_2x_3x_4x_5x_6)$	$(\bar{x_1} + x_2)(\bar{x_1} + x_3)(\bar{x_1} + x_4)$ $(\bar{x_3} + x_6)(\bar{x_4} + x_6)(\bar{x_5} + x_6)$ $(\bar{x_2} + x_3 + x_4)(\bar{x_2} + \bar{x_3} + x_5)$

defined by a partial order on variables, and we choose the partial order in which $x_1 \leq$ other vars and $x_n \geq$ other vars. We need to show that such a set of clauses picks at least one representative from each necklace class, in particular that lex-leaders satisfy every clause. The former is accomplished by giving an algorithm that for an arbitrary truth assignment finds a rotationally symmetric assignment that satisfies every clause. We then modify the algorithm to yield lex-leaders only, and show that they satisfy all clauses as well.

For an arbitrary truth assignment, consider two cases: (1) two variables are assigned different values, (2) otherwise. In case (2) all values are the same, i.e., all 0s or all 1s. These two truth assignments satisfy all clauses. In case 1, we can find two "neighboring" variables x_i and x_{i+1} that are assigned different values. Moreover, we can even find two "cyclically-neighboring" variables x_i and $x_{(i \bmod n)+1}$ such that the left variable =1 and the right variable =0, e.g., $x_1 = 1, x_2 = 0$ and $x_n = 1, x_1 = 0$ are valid examples. Any such assignment can be further rotated (by applying the cyclic symmetry) to an assignment with $x_n = 1, x_1 = 0$, which satisfies all clauses regardless of the values assigned to other variables.

We now need to show that every lex-leader satisfies the constructed SBP. Indeed, assume a lex-leader distinct from 000...0 such that $x_1 = 0, x_n = 0$. Then rotate the truth assignment in the direction from x_n to x_1 until $x_n = 1$. Each such rotation must lead to a *lexicographically smaller* representative. Contradiction.

This proof suggests a technique of identifying partial symmetry-breaking predicates. The key is finding a "canonical form" to which any truth assignment can be

reduced (not necessarily uniquely) by rotations. In the example above, the canonical form was defined by particular values of only two variables. More refined canonical forms can be defined by inequalities, more involved arithmetic predicates, etc.

4.5 k-Cycle Chains

In this section we deal with collections of cycles of the same length. We first assume that no cycles in this collection include complementary literals.

Step one. Break the first cycle.

Step two. Add a chaining predicate. A chaining predicate is an AND of sub-predicates. Each sub-predicate is of the form $A_i(\dots) \Rightarrow B_i(\dots)$. The variable *i* traverses all divisors of the cycle length, excluding itself. The A_i predicate depends on the variables of the broken cycle, and B_i depends on all variables of the remaining cycles. A_i is true when the truth assignment is in an orbit of length *i*. For example A_1 checks that all variables have identical values. A_2 is only needed for even cycles, it checks that all even-numbered variables have the same values and that all odd-numbered variables do. In general, A_i checks that any two variables with indices having the same remainder modulo *i* must have equal values. Thus the number of clauses is linear in cycle length. B_i is a symmetry-breaking predicate for power-*i* of the remaining cycles. It must be produced recursively. Observe that recursion is guaranteed to terminate because at every call either (i) the size of cycles decreases, OR (ii) the number of cycles decreases,... OR both. Note that the size of cycles cannot increase. It will stay the same if and only if the cycle length is prime, in which case the number of cycles decreases. The number of cycles can increase, but this can only happen when cycle length is composite, in which case the size of cycles must decrease.

Special cases may be simplified further simplified. For example, for cycles of prime length, we only have *i=1*, and chaining is particularly simple. In particular, cycles of length two appear very often, and an example is given in the next Section. In the general case, one can construct a PLL-SBPs by only including $A_1(\dots) \Rightarrow B_1(\dots)$, or using any subset of *i*'s.

As mentioned above, the number of orbits of an *n*-cycle is $o(2^n)$, thus the efficiency of symmetry-breaking is not limited in this case. This is not true, however, for *n* 2-cycles. The efficiency of symmetry-breaking is limited by 50% because the efficiency of every new cycle is 2x smaller than the efficiency of the previous cycle. For example, for one 2-cycle, there are 3 lex-leaders out of 4 assignments. For two 2-cycles, there are 4+2*3=10 lex-leaders out of 16 assignments, for three 2-cycles, there are 16+2*10=36 lex-leaders out of 64 assignments. For *k* 2-cycles, the number of lex-leaders is $P(k) = 4^{k-1} + 2P(k-1)$, and therefore, by induction. $0.5 \times 4^k \le P(k) \le 0.75 \times 4^k$. Moreover, as *k* increases $P(k) \to 0.5 \times 4^k$. In other words, *k* 2-cycles, asymptotically remove 50% of the solution space.

We now generalize our techniques to handle complementary literals. Our method of handling chains of cycles of length two must be extended to cycles of the form $(a\bar{a})$. Since a cycle of this form implies an SBP that breaks exactly 50% of the solution space, we should stop right there and ignore all other 2-cycles. In other words, when we have a chain of 2-cycles, we must first scan it for same-literal cycles. To chain longer cycles with complemented literals, we conjoin implications as described earlier.

4.6 Arbitrary Cycle Types

Constructions of SBPs for multiple cycles of the same length (described above) can be used as subroutines in SBP constructions for arbitrary permutations as follows.

Algorithm: Start with an empty SBP.

Step 1. Produce an SBP for all cycles of the shortest length *k*. Conjoin it to the previously accumulated SBP.

Step 2. Compute the *k*-th power of the permutation.

Step 3. If non-trivial cycles are left, start over (at Step 1) with the k-th power.

Example. Consider *(ab)(cd)(efg)(hijklm)*.

An LL-SBP for the permutation *(ab)(cd)*, e.g., $(a + \bar{b})(a + b + x)(\bar{a} + \bar{b} + x)(\bar{x} + c + \bar{d})$ must be conjoined with n LL-SBP for the square of the original permutation *-(efg)(hjl)(ikm)*.

Theorem 4.3 Suppose that Step 1 in the algorithm above produces partial LL-SBPs. Then the algorithm, too, yields partial LL-SBPs.

Proof: First, we describe the order of variables for which the constructed SBP is a partial LL-SBP.

1. Variables from earlier cycles must appear earlier.
2. The relative order of variables from cycles of the same length must be that for which SBP from Step 1 are LL-SBPs.

The remaining part of the proof describes lex-leaders of orbits of the given permutation (with respect to the given ordering) and verifies that the constructed SBP evaluate to true on them.

In order to produce a lex-leader of a given truth assignment, we need to apply the permutation as many times as it takes. Now, suppose that the shortest cycles have length *k*. Since variables from those cycles go first, we first need to apply the permutation (less than *k* times) to choose lex-leaders for that part of the truth assignment. From now on, the values of those variables must be

fixed. Conveniently, applying the permutation Nk times (N is an integer) does not change the values of those variables. In other words, we need to find out how many times the k-th power must be applied to find a lex-leader in terms of the next batch of variables.
Because we are considering partial LL-SBPs for the first batch of variables independently of other variables, partial LL-SBPs can be used without modifications. The rest is by induction.
Observe that even when SBPs produced at step 1are LL-SBPs, the SBP constructed by the algorithm may not be full. In particular, it may select more than lex-leaders from orbits where the first batch of variables have equal values. That is because on such orbits the second batch of variables can be subject to arbitrary powers of the permutation. The same applies to further batches of variables and can be remedied by adding more symmetry-breaking clauses similar to those used in Section 4.5. In particular, for cycles of length k we need to distinguish orbits whose lengths are less than k, which leads to complications for non-prime k. However, we believe that this extended construction yields LL-SBPs.

5 Symmetry-breaking by Generators

The LL-SBP construction of Crawford et al. entails conjoining symmetry-breaking predicates built for each element of the symmetry group. While this maximally reduces search space, the set of added clauses may be overwhelmingly large. Therefore, the total search time may be increased. This section discusses the idea of conjoining only symmetry-breaking clauses of symmetry-generators - a potentially more practical alternative.

5.1 Generators and Subgroups

Lemma 5.1 Given a permutation, a symmetry-breaking predicate that distinguishes exactly one element in every orbit (i.e., fully breaks a given symmetry) also fully breaks every symmetry generated by this permutation, i.e., every power of this permutation.

Proof: The orbits of one permutation and the group it generates are identical.

We now consider groups with more than one generator. Each generator is modeled by a cyclic sub-group.

Lemma 5.2 Consider a group G and its subgroups $\{H_i\}$ For an element of a G-orbit to be a lex-leader, it is necessary that it be the lex-leader in each of its H_i-orbits.

Proof: For every subgroup H of group G, G-orbits can be decomposed into a disjoint union of H-orbits. Therefore, for an element to be a lex-leader in its G-orbit, it must be a lex-leader in its H-orbit.

Corollary 5.3 A conjunction of (not necessarily full) lex-leader SBPs for sub-groups is a valid lex-leader SBP. However, it may not be a full SBP, and therefore not an LL-SBP.
Consider a group G and its subgroups $\{H_i\}$. Suppose that we are given SBPs for all subgroups. A conjunction of those SBPs may not be a valid SBP because there may be no representative in a given G-orbit that is picked as a representative in all of its H_i-orbits.

Lemma 5.4 A special case of Lemma 4.2 happens when all subgroups are cyclic and generated by generators of G. Lemmas 4.1 and 4.2 then imply that on every G-lex-leader, full SBPs for individual generators must evaluate to true. Therefore, a conjunction of SBPs for generators is a valid SBP.

Theorem 5.5 Consider a group G and its subgroups H and K, such that $G=HK$ and H normal. Observe that K acts on H-orbits by multiplication on the left. We require that this action map H-lex-leaders to H-lex-leaders. In this case, the conjunction of full lex-leader SBPs for H-orbits and K-orbits is a full lex-leader SBP for G-orbits.

Proof: Take an arbitrary G-orbit and its element p that is the lex-leader of its H-orbit and its K-orbit. Let q be the lex-leader in p's G-orbit. Find a $g \in G, h \in H, k \in K$ such that $g(p)=q$ and $g=hk$. Then $q=h(k(p))$ and $k(p)$ is in the same H-orbit as q. Since q is the lex-leader of its H-orbit, either $h=e$ or $k(p)$ is not an H-lex-leader. The latter is impossible because p is an H-lex-leader and the action of K by multiplication on the left preserves H-lex-leaders. On the other hand, if $h=e$, then $q=k(p)$ and p is in the same K-orbit as q. However, both p and q are lex-leaders of their K-orbits. Therefore $k=e$ and $p=q$.

Corollary 5.6 Consider two permutations π_1 and π_2 with *disjoint support*. The conjunction of full lex-leader SBPs for π_1 and π_2 is a full lex-leader SBP for the Abelian group generated by those permutations.

5.2 Requirements for Variable Ordering

As mentioned in Section 3.2.3, SBPs built for individual permutations in terms of cycle notation require, in general, different variable orderings. Therefore such SBPs for symmetry generators may be incompatible. Instead, the natural LL-SBP construction can be used for cyclic

groups generated by symmetry generators or, if the order of a generator is large, the natural PLL-SBP construction can be used.

For some structured CNF formulas, such as hole-*n* instances defined below, the original order of variables is compatible with cycle-based SBPs for all generators. Otherwise, careful analyses of supports of symmetry generators from a given set may still allow using more efficient LL-SBPs or PLL-SBPs for some generators.

For example, if all supports are disjoint, then it is easy to construct an overall variable ordering consistent with cycle-based SBPs. More generally, one can build the *support intersection graph* of a generating set where each generator is represented by a vertex and edges connect vertices when the supports of generators have non-empty intersections. A maximal independent set (MIS) in this graph flags generators for which cycle-based SBPs can be used. This introduces a partial variable order, which can be arbitrarily extended to an order of all variables so that the natural LL-SBPs or PLL-SBPs can be used for remaining generators.

5.3 Symmetric Groups and The Pigeonhole Principle

Lemma 5.7 [9] A full lex-leader SBP for the symmetric group S_n can be constructed by conjoining SBPs for *n-1* generating transpositions *(12), (23),...,(n-1 n)*. This SBP contains *n-1* binary clauses.

Proof: Because S_n can map any linear order on *n* variables to any order, its orbits over truth assignments are distinguished only by the number of zeros. Thus, there are *n+1* orbits, and lex-leaders can be chosen by requiring that the values of the variables form a non-decreasing sequence. This entails *n-1* binary clauses $(x_1 \le x_2)(x_2 \le x_3)\ldots(x_{n-1} \le x_n)$.

Lemma 5.8 When, in addition to permutational symmetries, we allow all phase-shift symmetries and their compositions, a full lex-leader SBP exists with n one-literal clauses.

Proof: It can be seen that the number of zeros is not an orbit invariant anymore because the group acts transitively on the Boolean cube, which becomes one orbit. The lex-leader is therefore the truth assignment *000...0*, and the LL-SBP consists of *n* one-literal clauses.

Recall that the pigeon-hole principle states that *n+1* objects (pigeons) cannot be assigned to *n* slots (holes). This is easily proven by induction. However, the pigeon-hole principle can also be phrased as an unsatisfiable instance of Boolean satisfiability (hole-*n* instances) and proven by SAT-solving algorithms for specific values of *n*, in which case induction is unavailable. The *n(n+1)* indicator-variables encode assignments of pigeons to holes. $n^2(n+1)/2$ binary mutual-exclusion clauses form *n* families - one per hole, and ensure that no two pigeons are in the same hole. Another family consists of *n+1* *n*-literal clauses, one per pigeon, ensuring that every pigeon is assigned to least one hole. It has been proven [4] that no polynomial-length resolution proofs of the pigeon-hole principle exist. Because of this, the dominant SAT-solving frameworks due to Davis-Putnam (DP) and Davis-Longeman-Loveland (DLL) cannot solve the hole-*n* CNF instances, regardless of implementation. This agrees well with empirical results for best available SAT-solvers, such as Chaff [13]. However, short induction-less proofs of the pigeon-hole principle can be constructed using symmetry [8]. Observe that the group of symmetries of the hole-*n* instance is a Cartesian product of S_n and S_{n+1} because “all holes are symmetric” and, independently, “all pigeons are symmetric”.

Theorem 5.9 For a hole-*n* instance, the conjunction of LL-SBPs for symmetry sub-groups S_n and S_{n+1} is <u>not</u> an LL-SBP. As a consequence, it is <u>not true</u> in general that a conjunction of LL-SBPs of groups *G* and *H* is an LL-SBP of $G \times H$.

Proof: We order the variables in the “hole-major” order, i.e., in *n* groups of *n+1*. Observe that hole-symmetries permute those groups, while pigeon-symmetries simultaneously permute variables inside each group. By applying pigeon-symmetries to an arbitrary truth assignment, we can permute the values of any two variables within the first group of n+1, and if their values are equal, we can permute the two variables having the same offsets in the second group, etc. Thus one can sort variables in the first group (zeros first), but this will bipartition variables in other groups. One can then sort each partition, which will create further partitions. Independently, hole-symmetries allow lexicographically sorting the *n* groups. An LL-SBP for pigeon-symmetries selects lex-leaders with respect to the above “pigeon-sort”. An LL-SBP for hole-symmetries selects lex-leaders with respect to the above “hole-sort”. The conjunction of such LL-SBPs will pick truth assignments that are both. However, some of those are not lex-leaders with respect to the Cartesian-product group. We give an example for hole-2, which has six variables, divided into two groups of three. The truth assignment (011,100) is a lex-leader with respect to both pigeon-sort and hole-sort. Yet, it can be mapped to a lex-smaller assignment - (001,110) - by applying the pigeon-symmetry that sorts variables 3-6 and the only non-trivial hole-symmetry.

Table 2: Search runtimes of CNF-SAT instances with and without PLL-SBPs. The symmetry detection runtime and the number of symmetry generators are also included.

	Symmetries		Chaff [13] Runtime (sec)				Speedup of NEW		
	Finding (sec)	#Generators	Orig	Craw	DAC	NEW	Orig	Craw	DAC
Urq3_5	0.27	29	1000	1000	0.01	0.01	>100K	>100K	1
Urq4_5	1.64	43	1000	1000	0.01	0.01	>100K	>100K	1
Urq5_5	14.6	72	1000	1000	0.01	0.01	>100K	>100K	1
Urq6_5	70	109	1000	1000	0.02	0.02	>50K	>50K	1
Urq7_5	186	143	1000	1000	0.02	0.02	>50K	>50K	1
hole7	0.01	13	0.33	0.05	0.01	0.01	33.10	5	1
hole8	0.01	15	1.05	0.08	0.01	0.01	151	11.48	1.43
hole9	0.23	17	3.94	0.13	0.01	0.01	393.70	12.87	1
hole10	0.35	19	21.0	0.22	0.01	0.01	2104	22.08	1
hole11	0.33	21	207	0.32	0.02	0.01	19463	30.12	1.54
hole12	0.42	23	1000	0.50	0.02	0.01	>100K	50.14	2
hole15	1.32	29	1000	1.39	0.04	0.02	44262	61.52	1.56
hole20	10.3	39	1000	5.98	0.09	0.04	25000	149.5	2.23
hole30	189	59	1000	52.23	0.30	0.16	6250	326.4	1.88

Table 3: Size of SPBs using the natural PLL-SBP, the construction from [1], and the proposed approach

	Natural PLL-SBP[5]			Construction from [1]			NEW		
	#V	#C	#Lits	#V	#C	#Lits	#V	#C	#Lits
Urq3_5	0	0	0	0	29	29	0	29	29
Urq4_5	0	0	0	0	43	43	0	43	43
Urq5_5	0	0	0	0	72	72	0	72	72
Urq6_5	0	0	0	0	109	109	0	109	109
Urq7_5	0	0	0	0	143	143	0	143	143
hole7	728	3640	30212	84	433	1517	143	366	808
hole8	1080	5400	53460	112	575	2074	179	278	1068
hole9	1530	7650	89505	144	737	2734	1302	466	1364
hole10	2090	10450	143165	180	919	3503	199	578	1696
hole11	2772	13860	220374	220	1121	4387	241	702	2064
hole12	3588	17940	328302	264	1343	5392	287	838	2468
hole15	6960	34800	929160	420	2129	9193	449	1318	3896
hole20	16380	81900	3660930	760	3839	18508	799	2358	6996
hole30	54870	274350	26255295	1740	8759	51013	1799	5338	15896
Total	89998	449990	31710403	3924	20251	98717	5398	12638	36652

6 Experimental Results

In this section, we empirically show the advantage of using smaller SBPs. The experiments were performed on an AMD Athlon 1.2 GHz machine with 1 GB of RAM running Linux. The runtime limit for all experiments was set to 1000 seconds. The benchmarks included hole-*n* instances and artificially constructed randomized Urquhart (Urq) instances [17]. We used the best available back-track SAT solver Chaff [13]. Since Chaff is randomized, its runtime varies, and we averaged all results over ten independent runs. Cursory analysis of distributions of results reveals normal-looking distributions and suggests that the averages we report are, indeed, representative.

Table 2 lists symmetry detection runtimes and the number of symmetry generators. We use the reduction to graph automorphism from [1] which detects a wider range of symmetries than that from [5] (the latter construction does not find any symmetries in the Urq instances, as seen from Table 3). The graphs are then passed to GAP/GRAPE/NAUTY [16,15,12]. Table 2 also compares SAT-solving runtimes for the original CNF instance and the instances augmented with symmetry-breaking predicates using the natural PLL-SBPs [5], the cycle-based SBPs [1], and the proposed approach. Clearly, the addition of SPBs significantly reduces the search runtime, and the proposed approach leads to the greatest savings in runtime.

Table 3 compares the size of predicates produced by constructions from [5], [1] and this work. Our construction entails the smallest number of variables, clauses, and literals for all instances analyzed.

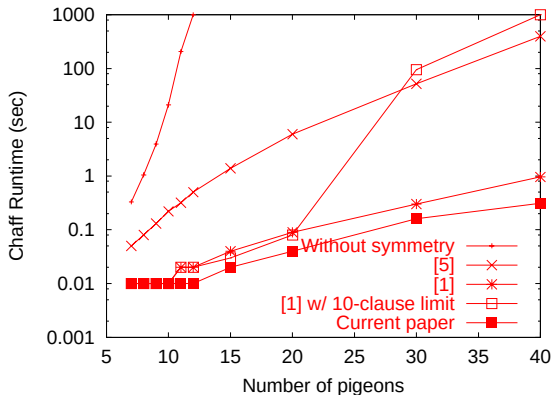

Figure 1: Comparison of runtimes for different SBP constructions for instances of the Pigeonhole principle.

7 Conclusions and Future Work

In this work we extended and improved the framework of symmetry-breaking predicates for solving Boolean Satisfiability by using logic minimization to construct more efficient CNF representations of symmetry-breaking predicates. Our constructions of symmetry-breaking predicates are in terms of cycles of permutations, which is particularly convenient when existing software for the graph automorphism problem is used. The proposed techniques lead to empirical speed-ups in back-track search for the best available SAT solvers, e.g., as shown in Figure 1.

Additionally, we gave new analyses of symmetry-breaking, including (i) estimates of efficiency of symmetry-breaking predicates, (ii) justification of symmetry-breaking by generators, and (iii) counterexamples to intuitive conjectures. Future work includes further efficiency improvement of symmetry-breaking predicates, more systematic studies of proposed constructions and empirical evaluation on applications arising the field of Electronic Design Automation.

We are pursuing improved algorithms for symmetry detection, e.g., along the lines of [1], where new techniques for opportunistic symmetry detection were proposed.

Acknowledgements. This work is funded by an Agere Systems/Semiconductor Research Corporation graduate fellowship and the DARPA/MARCO Gigascale Silicon Research Center (GSRC).

8 References

[1] F. Aloul, A. Ramani, I. Markov, K. Sakallah, “Solving Difficult SAT Instances in the Presence of Symmetries,” *DAC 2002.* pp. 731-736.

[2] L. Babai and E. M. Luks, ``Canonical Labeling of Graphs", *ACM Symp. on the Theory of Computing*, April 1983, pp. 171-183.

[3] L. Babai, “Automorphism Groups, Isomorphism, Reconstruction”, Chapter 27, pp. 1447-1541, In (R. L. Graham, M Grötschel and L. Lovász, eds, Handbook of Combinatorics, vol. 2, MIT Press, 1995).

[4] P. Beame, R. Karp, T. Pitassi and M. Saks, “The efficiency of Resolution and Davis-Putnam Procedure”, to appear in *SIAM Journal on Computing*. http://www.cs.washington.edu/homes/beame/papers/resj.ps..

[5] J. Crawford, M. Ginsberg, E. Luks and A. Roy, “Symmetry-breaking predicates for search problems”, *5th Intl Conf. Principles of Knowledge Representation and Reasoning (KR’96)*, Cambridge, MA, pp. 148-159.

[6] I. M. Gessel and R.P. Stanley, “Algebraic Enumeration”, Capter 21, p. 1059, In (R. L. Graham, M Grötschel and L. Lovász, eds, Handbook of Combinatorics, vol. 2, MIT Press, 1995).

[7] M. Hall Jr., ``The Theory of Groups", McMillan, 1959.

[8] B. Krishnamurthy, ``Short proofs for tricky formulas", *Acta Informatica*, 22:327-- 337, 1985.

[9] E. Luks and A. Roy, ``Symmetry Breaking in Constraint Satisfaction", *Intl. Conf. of Artificial Intelligence and Mathematics*, Ft. Lauderdale, FL, Jan 2-4, 2002.

[10] P. McGeer, J. Sanghavi, R. K. Brayton and A. Sangiovanni-Vincentelli, “ESPRESSO-Sgnature: A New Exact Minimizer for Logic Functions”, *IEEE Trans. on VLSI*, vol. 1, no. 4, pp. 432-440, December 1993.

[11] B. D. McKay, “Practical Graph Isomorphism”, *Congressus Numerantium*, 30 (1981), pp. 45-87.

[12] B. D. McKay, “Nauty user’s guide” (version 1.5), Technical report TR-CS-90-02, Australian National University, Computer Science Department, ANU, 1990. http://cs.anu.edu.au/~bdm/nauty/

[13] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang and S. Malik, “Chaff: Engineering an Efficient SAT Solver”, *DAC 2001*.

[14] J. P. M. Silva and K. A. Sakallah, “GRASP: A New Search Algorithm for Satisfiability”, *IEEE Trans. On Computers*, vol. 48, no. 5, May 1999.

[15] L. H. Soicher, “GRAPE: A System For Computing With Graphs and Groups”, *in "Groups and Computation"* (L. Finkelstein and W.M. Kantor, eds), *DIMACS Ser. in Discr. Math. & Theor. Comp. Sci.* 11, pp. 287-291.http://www.gap-system.org/~gap/Share/grape.html

[16] E. L. Spitznagel, “Review of Mathematical Software, GAP”, *Notices Amer. Math. Soc.*, 41 (7), (1994), pp. 780-782. http://www.gap-system.org

[17] A. Urquhart, “Hard Examples for Resolution”, *JACM*, vol. 34, 1987.

[18] A. Urquhart, ``The Symmetry Rule in Propositional Logic", 1996.

Breaking All the Symmetries in Matrix Models: Results, Conjectures, and Directions

Pierre Flener and Justin Pearson

Department of Information Technology, Uppsala University
Box 337, 751 05 Uppsala, Sweden
PierreF@csd.uu.se, justin@docs.uu.se

1 Introduction: Matrix Models and Symmetry

A *matrix model* is a constraint program that contains one or more matrices of decision variables. For example, a natural model of the round robin tournament scheduling problem (prob026 in CSPlib [4]) has a 2-dimensional (2-d) matrix of variables, each of which is assigned a value corresponding to the match played in a given week and period [15]. In this case, the matrix is obvious in the modelling of the problem: we need a *table* of fixtures. However, many other problems that are less obviously defined in terms of matrices of variables can be effectively represented and efficiently solved using matrix models [7]. In this paper, we focus on matrix models with just one matrix of decision variables.

Symmetry is an important aspect of matrix models. A *symmetry* is a bijection on decision variables that preserves solutions and non-solutions. Two variables are *indistinguishable* if some symmetry interchanges their rôles in all solutions and non-solutions. Symmetry often occurs because groups of objects within a matrix are indistinguishable. For example, in the round robin tournament scheduling problem, weeks and periods are indistinguishable. We can therefore permute any two weeks or any two periods in the schedule. That is, we can permute any two rows or any two columns of the associated matrix, whose index sets are the weeks and periods.

A *row* (*column*) *symmetry* of a 2-d matrix is a bijection between the variables of two of its rows (columns) that preserves solutions and non-solutions. Two rows (columns) are *indistinguishable* if their variables are pairwise indistinguishable due to a row (column) symmetry. The rotational symmetries of a matrix are neither row nor column symmetries. A matrix model *has total row* (*column*) *symmetry* iff all the rows (columns) of its matrix are indistinguishable. These definitions can be extended to matrices with any number of dimensions. A *symmetry class* is an equivalence class of assignments, where two assignments are *equivalent* if there is some symmetry mapping one assignment into the other. In group theory, such equivalence classes are referred to as *orbits*.

2 Breaking Symmetry

There are a number of ways of dealing with symmetry in constraint programming. We here only consider techniques that remove symmetries with *symmetry-*

breaking constraints by adding them to the model *before* search starts, e.g., [14, 3]. Other techniques break symmetries by adding constraints *during* search, e.g., [1], the symmetry-breaking during search framework (SBDS) [11], and the global cut framework (GCF) [8].

A common method to break symmetry is to impose a constraint that orders the symmetric objects. To break all the row (column) symmetries, one can order the rows (columns) lexicographically. The rows (columns) in a 2-d matrix are *lexicographically ordered* if each row (column) is lexicographically smaller (denoted $\leq_{lex}$) than the next, if any. Each orbit has a matrix where the rows and columns are lexicographically ordered [6]. However, lexicographically ordering the rows and columns does *not* break all the compositions of the row and column symmetries. Consider a 3×3 matrix of $0/1$ variables, x_{ij}, such that $\forall i\ x_{i1} + x_{i2} + x_{i3} \geq 1$ and $\sum_{ij} x_{ij} = 4$. This model has total row and column symmetry. The following solutions have lexicographically ordered rows and columns:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

These solutions are symmetric, as one can move from one to the other by swapping the first two rows and the last two columns. Swapping any rows or columns *individually* breaks the lexicographic ordering.

3 Breaking All the Symmetries

It is always possible to break all the symmetries. We now illustrate a method by an example and discuss its application.

3.1 A Method

A *group*, $G = (X, \circ)$, is a set of elements, X, where $\circ$ is an associative binary operation such that there exists exactly one element $id \in X$ having the property that for all elements $x \in X$, we have that $id \circ x = x \circ id = x$, and further for every element $x \in X$, there exists a unique element x^{-1} such that $x \circ x^{-1} = x^{-1} \circ x = id$. Given a set of elements, Ω, we will be interested in groups whose elements are bijective functions on Ω, that is *permutations*. The operation $\circ$, in such groups, will be function composition. If such a set of permutations is closed under taking inverse functions, then the set is a group. We will write permutations on finite sets as products of cycles. A *cycle* $(x, f(x), f^2(x), \ldots, f^n(x))$ gives the action of the permutation f on x; the last item in the cycle $f^n(x)$ asserts that $f^{n+1}(x) = x$. Modulo the order of the cycles and the starting point in each cycle, a permutation on a finite set has a unique cyclic decomposition. When writing permutations, unit cycles are often omitted. For example, a function f on the integers $1 \ldots 6$ such that $f(1) = 2$, $f(2) = 3$, $f(3) = 1$, $f(4) = 5$, $f(5) = 4$, and $f(6) = 6$ would be denoted $(1, 2, 3)(4, 5)$. Given an element, x, of a group, the *order* of x is the

smallest integer n such that $x^n = id$. For example, the order of $(1,2,3)(4,5)$ is 6. The group $\mathrm{Sym}(q)$ is the set of all permutations on the set $\{1,\ldots,q\}$.

In [3], a method of breaking symmetry by adding constraints is introduced. Essentially, for each symmetry of the problem a lexicographic ordering constraint is added, forcing only one of the symmetric solutions to be picked.

We now illustrate all this on a running example that has a maybe small matrix but is sufficiently illustrative.

Example 1. The group of all the row and column symmetries of a 3×2 matrix

$$\begin{pmatrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \end{pmatrix}$$

can be generated by the following 3 permutations, permuting the first two columns, the last two columns, and the two rows, respectively:

$$(1,2)(4,5) \qquad (2,3)(5,6) \qquad (1,4)(2,5)(3,6)$$

This group contains the following 12 permutations:

Permutation	Name	Order
()	id	1
(1,2)(4,5)	$P_{c_{12}}$	2
(2,3)(5,6)	$P_{c_{23}}$	2
(1,4)(2,5)(3,6)	$P_{r_{12}}$	2
(1,6,2,4,3,5)	P_δ	6
(1,5,3,4,2,6)	P_σ	6
(1,4)(2,6)(3,5)	P_{α_1}	2
(1,5)(2,4)(3,6)	P_{α_2}	2
(1,6)(2,5)(3,4)	P_{α_3}	2
(1,3)(4,6)	$P_{c_{13}}$	2
(1,2,3)(4,5,6)	$P_{c_{123}}$	3
(1,3,2)(4,6,5)	$P_{c_{132}}$	3

Omitting the identity permutation (), these symmetries can be broken by the following 11 constraints:

$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_2,x_1,x_3,x_5,x_4,x_6] \qquad (c_{12})$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_1,x_3,x_2,x_4,x_6,x_5] \qquad (c_{23})$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_4,x_5,x_6,x_1,x_2,x_3] \qquad (r_{12})$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_6,x_4,x_5,x_3,x_1,x_2] \qquad (\delta)$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_5,x_6,x_4,x_2,x_3,x_1] \qquad (\sigma)$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_4,x_6,x_5,x_1,x_3,x_2] \qquad (\alpha_1)$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_5,x_4,x_6,x_2,x_1,x_3] \qquad (\alpha_2)$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_6,x_5,x_4,x_3,x_2,x_1] \qquad (\alpha_3)$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_3,x_2,x_1,x_6,x_5,x_4] \qquad (c_{13})$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_2,x_3,x_1,x_5,x_6,x_4] \qquad (c_{123})$$
$$[x_1,x_2,x_3,x_4,x_5,x_6] \leq_{lex} [x_3,x_1,x_2,x_6,x_4,x_5] \qquad (c_{132})$$

Due to the meaning of the lexicographic ordering where:

$$[x_1, x_2, x_3, \ldots] \leq_{lex} [y_1, y_2, y_3, \ldots]$$

is defined to be:

$$(x_1 \leq y_1) \wedge \ (x_1 = y_1 \rightarrow x_2 \leq y_2) \wedge \ (x_1 = y_1 \wedge x_2 = y_2 \rightarrow x_3 \leq y_3) \wedge \ \ldots \quad (1)$$

and due to the right-hand vector here always being a permutation of the left-hand one, the elements at the positions corresponding to the last indices in each cycle (including the unit cycles) can be deleted in both vectors. By this token, the constraints above can be *internally* simplified to the following:

$$\begin{aligned}
[x_1, x_4] &\leq_{lex} [x_2, x_5] && (c_{12}) \\
[x_2, x_5] &\leq_{lex} [x_3, x_6] && (c_{23}) \\
[x_1, x_2, x_3] &\leq_{lex} [x_4, x_5, x_6] && (r_{12}) \\
[x_1, x_2, x_3, x_4, x_5] &\leq_{lex} [x_6, x_4, x_5, x_3, x_1] && (\delta) \\
[x_1, x_2, x_3, x_4, x_5] &\leq_{lex} [x_5, x_6, x_4, x_2, x_3] && (\sigma) \\
[x_1, x_2, x_3] &\leq_{lex} [x_4, x_6, x_5] && (\alpha_1) \\
[x_1, x_2, x_3] &\leq_{lex} [x_5, x_4, x_6] && (\alpha_2) \\
[x_1, x_2, x_3] &\leq_{lex} [x_6, x_5, x_4] && (\alpha_3) \\
[x_1, x_4] &\leq_{lex} [x_3, x_6] && (c_{13}) \\
[x_1, x_2, x_4, x_5] &\leq_{lex} [x_2, x_3, x_5, x_6] && (c_{123}) \\
[x_1, x_2, x_4, x_5] &\leq_{lex} [x_3, x_1, x_6, x_4] && (c_{132})
\end{aligned}$$

The first two constraints now indeed reflect the indistinguishability of the first two columns and the last two columns, respectively, whereas the third constraint reflects the indistinguishability of the two rows. The last three constraints are actually (logically) implied and can be eliminated. Indeed, the constraint c_{13} arises from the indistinguishability of the first and third columns, and is a logical consequence of the constraints c_{12} and c_{23}, due to the transitivity of $\leq_{lex}$. Also, the constraints c_{123} and c_{132} are logical consequences of the same constraints for the same reason, as the corresponding permutations of all the three columns can also be ruled out by requiring the entire columns to be (lexicographically) ordered from left to right. Furthermore, Frisch and Harvey [9] have manually established that the constraints δ and σ can be further simplified *in the context of the others*. All this leaves us now with the following 8 constraints:

$$\begin{aligned}
[x_1, x_4] &\leq_{lex} [x_2, x_5] && (c_{12}) \\
[x_2, x_5] &\leq_{lex} [x_3, x_6] && (c_{23}) \\
[x_1, x_2, x_3] &\leq_{lex} [x_4, x_5, x_6] && (r_{12}) \\
[x_1, x_2, x_3] &\leq_{lex} [x_6, x_4, x_5] && (\delta)
\end{aligned}$$

$$[x_1, x_2, x_3, x_4] \leq_{lex} [x_5, x_6, x_4, x_2] \qquad (\sigma)$$
$$[x_1, x_2, x_3] \leq_{lex} [x_4, x_6, x_5] \qquad (\alpha_1)$$
$$[x_1, x_2, x_3] \leq_{lex} [x_5, x_4, x_6] \qquad (\alpha_2)$$
$$[x_1, x_2, x_3] \leq_{lex} [x_6, x_5, x_4] \qquad (\alpha_3)$$

The last 6 constraints now (at least) require the first row to be lexicographically smaller than any permutation of the second row. We do not know whether this is a coincidence. The constraint σ cannot be further simplified [9].

In general, an $m \times n$ matrix has $m! \cdot n! - 1$ compositions of row and column symmetries except identity, generating thus a super-exponential number of $\leq_{lex}$ constraints. Hence this approach seems not always practical, even after eliminating the $m! - m + n! - n$ constraints that are implied due to the transitivity of $\leq_{lex}$. Despite encouraging experimental results [6] with just the constraints induced by the generator symmetries and in the presence of actual problem constraints, we are looking for special cases where all or most compositions of the row and column symmetries can be broken by a polynomial (and even linear) number of $\leq_{lex}$ constraints (see [6] for other results).

3.2 Implied Constraints

There may be further implied $\leq_{lex}$ constraints than we can predict so far based on the transitivity of $\leq_{lex}$, but we do not know whether the minimum set of non-implied $\leq_{lex}$ constraints is of polynomial size.

A *support set* of an implied constraint C is a set of constraints $\{c_1, \cdots, c_n\}$ not containing C such that $c_1 \wedge \ldots \wedge c_n \rightarrow C$. We here look for support sets for $\leq_{lex}$ constraints among the other $\leq_{lex}$ constraints. *Detecting* a support set of an implied constraint is not easy. Various methods can be adapted, such as circuit minimisation or integer linear programming (ILP). Indeed, $\leq_{lex}$ constraints can be encoded as linear inequalities. For instance, the constraint σ is equivalent to:

$$125 \cdot x_1 + 25 \cdot x_2 + 5 \cdot x_3 + x_4 \leq 125 \cdot x_5 + 25 \cdot x_6 + 5 \cdot x_4 + x_2 \qquad (2)$$

when the domain size is 5. However, the standard syntactic detection criteria from ILP listed in [12] fail. We have yet to try the new syntactic criteria in [12]. *Eliminating* implied constraints is also difficult, as there are usually many support sets. Furthermore, support sets may overlap.

3.3 Domain-Dependent Implied Constraints

An interesting observation is that the number of implied $\leq_{lex}$ constraints grows as the domain size of the decision variables shrinks. For instance, in Example 1, the five constraints δ, σ, α_1, α_2, and α_3 are implied when the domain size is 2, and up to four of them can be collectively eliminated, say all except δ or α_1. If the domain size is 3, then σ is no longer implied, and up to three of the other four can be collectively eliminated, say all except δ. If the domain size is 4, then none of these constraints is implied.

Experimentally, up to 6×6 matrices, we tested the following conjecture:

Conjecture 1. For a domain of size 2, it suffices to add the $\leq_{lex}$ constraints induced by the order 2 permutations.

For domain size 3, this conjecture is not true since, as just seen, the constraint σ is necessary, but is of order 6.

Unfortunately, we already determined even the number of order 2 permutations to be super-polynomial. Indeed, for an $m \times n$ matrix, it is $f(m) \cdot f(n) - 1$, where f is the sequence:

$$1, 1, 2, 4, 10, 26, 76, ...$$

This is sequence A000085 in the *On-Line Encyclopedia of Integer Sequences* [13] and has the following recurrence relation:

$$\begin{aligned} f(0) &= f(1) = 1 \\ f(n) &= f(n-1) + (n-1) \cdot f(n-2) \qquad \text{for } n > 1 \end{aligned}$$

and the following closed form:

$$f(n) = \sum_{k=0}^{n/2} \frac{n!}{(n - 2 \cdot k)! \cdot 2^k \cdot k!}$$

The value $f(q)$ is the number of order 2 permutations in the group $\mathrm{Sym}(q)$. Since the group of row and column symmetries on an $m \times n$ matrix is isomorphic to the product of $\mathrm{Sym}(m)$ and $\mathrm{Sym}(n)$, this justifies the formula $f(n) \cdot f(m) - 1$, as identity is counted twice.

Even if the conjecture is true, there still remain implied $\leq_{lex}$ constraints induced by the order 2 permutations. For example, c_{13} is induced by an order 2 permutation but is implied, as seen above. It is not known whether eliminating implied $\leq_{lex}$ constraints induced by the order 2 permutations leaves a polynomial number of $\leq_{lex}$ constraints.

It would be interesting to characterise which $\leq_{lex}$ constraints are necessary for which domain sizes.

3.4 Other Directions

As Frisch and Harvey [9] have shown, there are *contextual* simplifications to the $\leq_{lex}$ constraints that we cannot mechanise yet.

Another (by us yet unexplored) direction is to experimentally determine a polynomial number of $\leq_{lex}$ constraints that break "most" of the symmetries, possibly taking into account the effect of the problem constraints. Indeed, our experiments (with small matrices, up to 4×4) show that on the pure problem (enumerating all matrices modulo total row and column symmetry, though in the absence of any actual problem constraints), many of the super-polynomial number of $\leq_{lex}$ constraints just eliminate a handful of solutions, whereas a few of them eliminate hundreds or thousands of solutions. A characterisation of the more effective $\leq_{lex}$ constraints will be a useful achievement.

		with all the 35 constraints	without 15 implied constraints before internal simplifications	after internal simplifications
domain size = 4 (8,240 matrices)	Boolean $\leq_{lex}$	11.0"	5.8"	2.1"
	linear $\leq_{lex}$	8.3"	4.5"	1.6"
domain size = 5 (57,675 matrices)	Boolean $\leq_{lex}$	61.0"	31.8"	12.4"
	linear $\leq_{lex}$	49.6"	26.7"	10.0"
domain size = 6 (289,716 matrices)	Boolean $\leq_{lex}$	269.0"	139.0"	56.1"
	linear $\leq_{lex}$	227.0"	122.6"	46.5"

Table 1. The effects of constraint eliminations and internal simplifications

3.5 Experiments

We experimented, under GNU Prolog on a Sun SPARC Ultra station 10, with two different implementations of $\leq_{lex}$, namely a Boolean one based on (1) and an ILP one using linear inequalities as in (2). The objective was to enumerate all 3×3 matrices modulo total row and column symmetry, though in the absence of any actual problem constraints. Transitivity of $\leq_{lex}$ gives 6 implied constraints among 35 (irrespective of the domain size), and we experimentally detected another 9 implied constraints (for domain sizes from 4 up to at least 6), which can even be eliminated together as they have non-overlapping support sets within the remaining constraints.[1] Table 1 summarises the experiments, giving run times in seconds. Whichever the implementation of $\leq_{lex}$, both the constraint eliminations and then the internal simplifications pay off, together giving a five-fold time reduction for solving, irrespective of the domain size.

4 Conclusions

We have investigated some special cases where symmetry breaking constraints are implied or can be internally simplified. The constraints that can be removed due to implication depend on the domain size of the matrix problem. In some cases, these implied constraints actually slow down the constraint solver.

All approaches would benefit from an efficient means of automatic symmetry detection. However, symmetry detection has been shown to be graph-isomorphism complete in the general case [2]. Therefore, it is often assumed that the symmetries are *declared* by the user. However, symmetries are often *introduced* while formulating CSPs in today's rather low-level constraint programming languages. Indeed, the latter usually lack high-level data structures, such as sets and relations, where element order is irrelevant, but whose lower-level

[1] Surprisingly, even the constraint ordering the first two rows is implied for domain sizes up to at least 6, but we did not eliminate it in our experiments, as 8 of the other 9 implied constraints may well be implied irrespective of the domain size and even look as if more general results could eliminate them.

implementations in terms of lists or matrices make the element order seemingly relevant, so that symmetry-breaking constraints become necessary to compensate for this. The compiler of a relational constraint modelling language, such as the proposal in [5], should thus be aware of the symmetries it introduces.

Also, there is a need for efficient methods for establishing generalised arc consistency on sets of $\leq_{lex}$ constraints operating on the same matrix.

Acknowledgements. This work is partially supported by grant 221-99-369 of VR (the Swedish Research Council) and by institutional grant IG2001-67 of STINT (the Swedish Foundation for International Cooperation in Research and Higher Education). We also thank Alan M. Frisch, Warwick Harvey, the referees, as well as the members of the APES and ASTRA research groups, for their helpful discussions.

References

1. R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *Proc. of CP'99*, J. Jaffar (ed), LNCS 1713, pp. 73–87. Springer-Verlag, 1999.
2. J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In *Proc. of AAAI'92 workshop on tractable reasoning*, 1992.
3. J. Crawford, G. Luks, M. Ginsberg, and A. Roy. Symmetry breaking predicates for search problems. In *Proc. of KR'96*, pp. 148–159, 1996.
4. *CSPlib, a Problem Library for Constraints*, at www.csplib.org.
5. P. Flener. Towards relational modelling of combinatorial optimisation problems. In: Ch. Bessière (ed), *Proc. of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, at www.lirmm.fr/~bessiere/nbc_workshop.htm, 2001.
6. P. Flener, A.M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, J. Pearson, and T. Walsh. Symmetry in matrix models. In *Proc. of SymCon'01*, at www.csd.uu.se/~pierref/astra/SymCon01/, 2001. Extended paper in P. Van Hentenryck (ed), *Proc. of CP'02*, LNCS 2470, Springer-Verlag, 2002.
7. P. Flener, A.M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, and T. Walsh. Matrix modelling. In *Proc. of Formul'01*, at www.dcs.gla.ac.uk/~pat/cp2001/, 2001.
8. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proc. of CP'01*, T. Walsh (ed), LNCS 2239, pp. 77–92. Springer-Verlag, 2001.
9. A.M. Frisch. Slides for the SymCon'01 presentation of [6], at www.cs.york.ac.uk/aig/projects/implied/docs/SymMxCP01.ppt, 2001.
10. *The GAP Group – Groups, Algorithms, and Programming*, at www.gap-system.org.
11. I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In *Proc. of ECAI'00*, W. Horn (ed), pp. 599–603. IOS Press, 2000.
12. J.-L. Imbert and P. Van Hentenryck. Redundancy elimination with a lexicographic solved form. In *Annals of Mathematics and AI*, 17(1–2):85–106, 1996.
13. *On-Line Encyclopedia of Integer Sequences*, at www.research.att.com/~njas/ sequences/.
14. J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proc. of ISMIS'93*, LNAI 689, pp. 350–361. Springer-Verlag, 1993.
15. P. Van Hentenryck, L. Michel, L. Perron, and J.-C. Régin. Constraint programming in OPL. In *Proc. of PPDP'99*, LNCS 1703, pp. 97–116. Springer-Verlag, 1999.

Group-graphs associated with Row and Column Symmetries of Matrix Models: some observations

Zeynep Kiziltan[1] and Michela Milano[2]

[1] Department of Information Science, Uppsala University, Uppsala, Sweden
Zeynep.Kiziltan@dis.uu.se
[2] DEIS, University of Bologna, Bologna, Italy
mmilano@deis.unibo.it

Abstract. The effect of symmetry-breaking constraints is often evaluated empirically. In order to understand which symmetric configurations are removed by a set of constraints, we have to understand the underlying structure of the symmetry group in concern. A class of symmetry that frequently occurs in constraint programming is the row and column symmetries of a matrix model. In this paper, we study these symmetries from a structural viewpoint, and show how the graph of the associated group can be built. The graph can help us to understand the effect of certain symmetry-breaking constraints posted on a matrix model, though some questions remain open. This paper is a preliminary study on the relations between group properties and the corresponding graph and symmetry-breaking constraints.

1 Introduction

Symmetries are ubiquitous in many Constraint Satisfaction Problems (CSPs). Symmetry in a CSP can involve the variables, the values, or both, and map each search state (e.g., a partial assignment, a solution and a failure) to an equivalent one. An exhaustive search method spends time in visiting equivalent states if symmetries are not eliminated [5].

Symmetries of a CSP form a group. Thus, we can exploit results coming from the *group theory* to understand the structure of each particular symmetry. In addition, each group has a corresponding graph which again can help for this purpose.

A class of symmetries that frequently occur in constraint programming is the row and column symmetries of a matrix model [2]. A matrix model is a constraint program that contains one or more matrices of decision variables. Many CSPs can be easily represented by matrix models [3] in which the matrices may have symmetry between their rows and/or columns. Such symmetries are referred to as *row and column symmetries*. Two matrices are symmetric if one can be obtained from the other by row and/or column permutations.

In this paper, we analyse the group describing the row and column symmetries, in order to understand their underlying structure. The elements of the

group are all the symmetric matrices of a given matrix. To obtain all permutations, only two generators for rows, and two for columns should be considered: the flip of the first two rows (resp. columns), and a shift that leads the first row (resp. column) to the last position. We have observed that the resulting group-graph has a very interesting structure.

Our ultimate aim is to provide some considerations for studying, from a structural point of view, which configurations are removed when we add different sets of symmetry-breaking constraints to a matrix model so as to remove row and column symmetries. Thus, this work can be seen as a starting point for a deeper insight on this topic. In general, approaches proposing symmetry-removal algorithms or constraints experimentally evaluate the effectiveness of the method. Here, we propose a more formal perspective which could possibly be used to compare different approaches from a structural point of view, and could help to devise new algorithms and symmetry-breaking constraints.

This paper is a preliminary step towards this more general and ambitious aim, and we think it deserves more investigation. We here restrict our focus on small square matrices (3×3), but our observations could be generalized for bigger matrices.

2 Groups and their graphs

Recently, group theory has been used to describe the symmetries of a CSP, and to reduce the effort to remove them (e.g., [4]). A group is a tuple $G = (S, Op)$ where S is a set and Op is a closed binary operation over S. A group has the following properties:

- the operation Op is associative, i.e., for any $x, y, z \in S,\ (x\ Op\ y)\ Op\ z = x\ Op\ (y\ Op\ z)$.
- there is a neutral element I, i.e., for any $x \in S,\ x\ Op\ I = I\ Op\ x = x$.
- each element x in S has an inverse x^{-1}, i.e., for any $x \in S,\ x\ Op\ x^{-1} = x^{-1}\ Op\ x = I$.

As an example, we consider the permutation of n elements. We have a group $G = (S, Op)$ (called *permutation group*), where S contains $n!$ elements, each corresponding to a permutation of n elements, i.e., a bijective mapping from S to itself. The operation Op is the function composition, if we consider functions as the elements of S. It can easily be shown that the function composition is closed and associative, the neutral element is the identity permutation, and every permutation has an inverse. In a permutation group, we can consider a minimal set of permutations whose compositions gives all possible permutations. Permutations belonging to this minimal set are called *generators*. Each generator has a *period*, i.e., the number of applications of the generator to a permutation so as to obtain itself.

For the permutation group, we have two generators. The first is identified as f and corresponds to the flip of the elements in positions 1 and 2. The second is

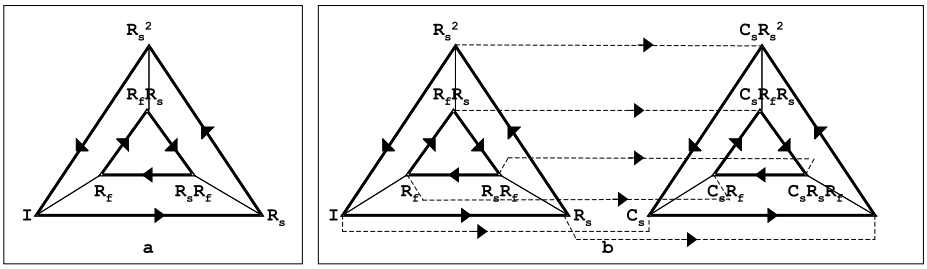

Fig. 1. a. Group-graph of row symmetries. **b.** Group-graph of row symmetries plus a column movement.

identified as s and corresponds to a shift, carrying the first element to the last position.

Every group has a corresponding graph, where a vertex corresponds to a configuration (i.e., an element of the group), and an arc represents the application of a generator to a configuration. Two configurations A and B are characterised by a distance: the number of generator applications transforming A to B.

As a further example, the size of the permutation group of 3 elements is 6 (=3!), and thus the graph associated with this group has 6 vertices. Each vertex has 2 outgoing and 2 incoming arcs. Each arc corresponds to a generator. Each generator has a period, i.e., the number of applications of the generator to a permutation so as to obtain itself. The period of f is 2, and the period of s is 3.

3 The group describing row and column symmetries

In this section, we use group theory to understand the structure of the row and column symmetries of a matrix model. This is a preliminary study: in fact, we restrict ourselves to small matrices (3×3) but we believe this study can be extended for larger matrices.

In a 3×3 matrix, we have 9 variables $I = [X_1, \ldots, X_9]$ ordered from the top left corner to the bottom right one. Let us now consider only the row symmetry. We have 3 rows $[X_1, X_2, X_3]$, $[X_4, X_5, X_6]$, and $[X_7, X_8, X_9]$, subject to permutation. The set of all possible row permutations forms a group $GR = (S_r, \circ)$. This group has the same structure (permutation) of the one described in Section 2: S_r is the set of configurations symmetric to the identity matrix I, and the operation $\circ$ is the function composition.

The row symmetry group has two generators: the flip of the first two rows R_f, and the row shift R_s that leads the first row to the last position. The period of R_f is 2, while the period of R_s is 3. Each configuration in S_r can be identified by various composition of the 2 generators applied to the identity matrix I. Figure 1.a reports the corresponding graph which has 6 (=3!) vertices. Each vertex has 2 outgoing and 2 incoming arcs. Each arc has a direction. Note that in the figure, a bi-directional arc (corresponding to R_f and its inverse) is replaced by

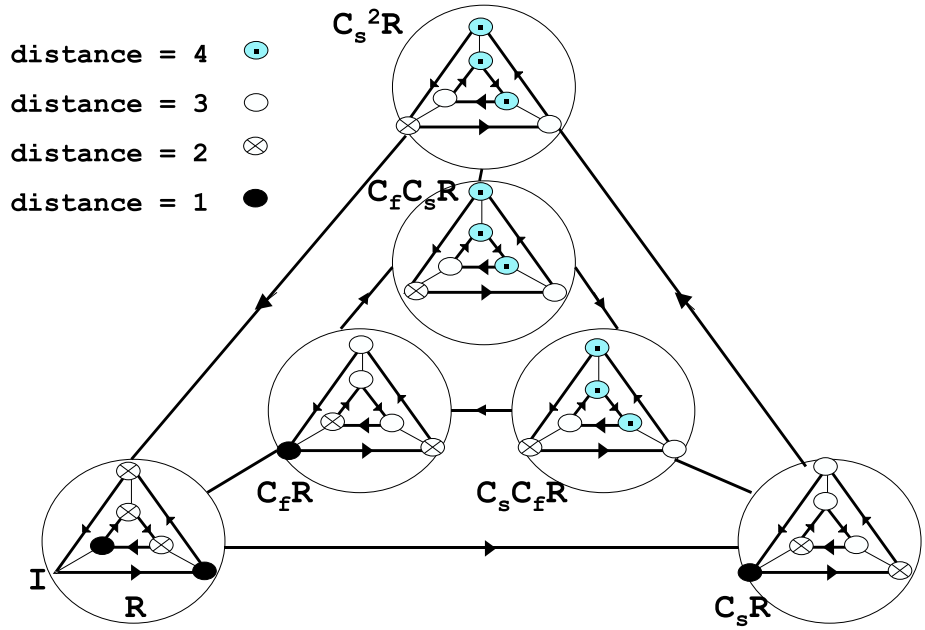

Fig. 2. Group-graph of the row and column symmetries of a 3×3 matrix.

an undirected arc. The configurations are labelled[1] in the figure as I, R_s, $R_s \circ R_s$ (hereinafter referred to as R_s^2), R_f, $R_s \circ R_f$, $R_f \circ R_s$. Note that the configuration $R_s \circ R_f$ can also be obtained by $R_f \circ R_s^2$.

We now consider the column symmetry. We have 3 columns $[X_1, X_4, X_7]$, $[X_2, X_5, X_8]$, and $[X_3, X_6, X_9]$, subject to permutation. Similar to the row symmetry case, the set of all possible column permutations forms a group with two generators: the column flip C_f and the column shift C_s. The size of the group is 6 (=3!) and its elements are I, C_s, C_s^2, C_f, $C_s \circ C_f$, $C_f \circ C_s$. Clearly, each of these permutations can be applied to each vertex of the group-graph in Figure 1.a. For instance, if we apply the permutation C_s to each vertex of the graph in Figure 1.a, then we obtain another graph, depicted in Figure 1.b. In this graph, whilst the leftmost triangle represents all the row permutations of the identity matrix I, the rightmost triangle represents all the row permutations of the matrix C_s obtained by applying a shift on the columns of the matrix I.

Since we have in total 6 column permutations, we obtain the group-graph of the row and column symmetries of a 3×3 matrix by applying all the 6 column permutations to each vertex of the graph in Figure 1.a. The resulting graph has 6 vertices (called *meta-vertices*), each of which is a graph with 6 vertices representing all the row permutations of a column permutation of the identity

[1] Vertices are labelled by the sequence of the generators applied to the identity matrix I.

matrix I, as depicted in Figure 2. The leftmost vertex of every meta-vertex is obtained from I by permuting its columns. In the rest of this paper, we will refer to every column permutation of the identity matrix I as the identity of the corresponding meta-vertex.

Each vertex in the graph has a distance with respect to the starting point, i.e., the minimum number of generators applied to reach the vertex from the identity matrix I. In Figure 2, each meta-vertex is labelled with the generator sequence to reach the vertex from the initial meta-vertex R, and each vertex is labelled with its distance.

4 Equivalence classes

In an equivalence class of matrices, any two matrices are symmetric, i.e., any matrix can be obtained from the other via row and/or column permutations. In Figure 2 we see that every 3×3 matrix has 36 symmetric configurations. Does this mean that the size of every equivalence class is 36? This is the case if the values in the matrix are all different. However, this is not always the case if there are repeating values in the matrix, because in a such case two symmetric matrices are not necessarily distinct. For instance, given $I = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$, the matrix R_f is identical to I.

Although not every equivalence class has necessarily the same size, it is possible to know how big the equivalence classes can be. For instance, the columns of the matrix $I = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$ are all the same. Hence, any column permutation leaves the matrix unchanged. That is, in the group-graph associated with row and column symmetries, all the meta-vertices fall into the (leftmost) meta-vertex R that represents the row permutations of I. Moreover, two rows of I are the same, which means that a matrix within a meta-vertex falls into the one obtained by swapping those rows. This can be visualised in Figure 3. The size of the equivalence class containing the matrix I is thus $36/(2!3!)$, which is 3.

If a 3×3 matrix I has only the values 0 and 1 then there are 9 possible scenarios of the group-graph, giving rise to 9 possible size for the equivalence class C_I containing the matrix I:

1. Rows are all the same, and columns are all the same: in this case, all the meta-vertices fall into the meta-vertex R representing the row permutations of I. Also, all matrices within a meta-vertex fall into the identity matrix of the meta-vertex. Hence, $|C_I| = 36/(3!3!) = 1$.
2. Only 2 rows are the same, and columns are all the same: in this case, all the meta-vertices fall into R. Also, a matrix within a meta-vertex falls into the one obtained by swapping those rows. Hence, $|C_I| = 36/(2!3!) = 3$.
3. Rows are all the same, and only 2 columns are the same: this case is analogues to the previous case.
4. Only 2 rows are the same, and only 2 columns are the same: in this case, a meta-vertex falls into the one obtained by swapping those columns. Also, a

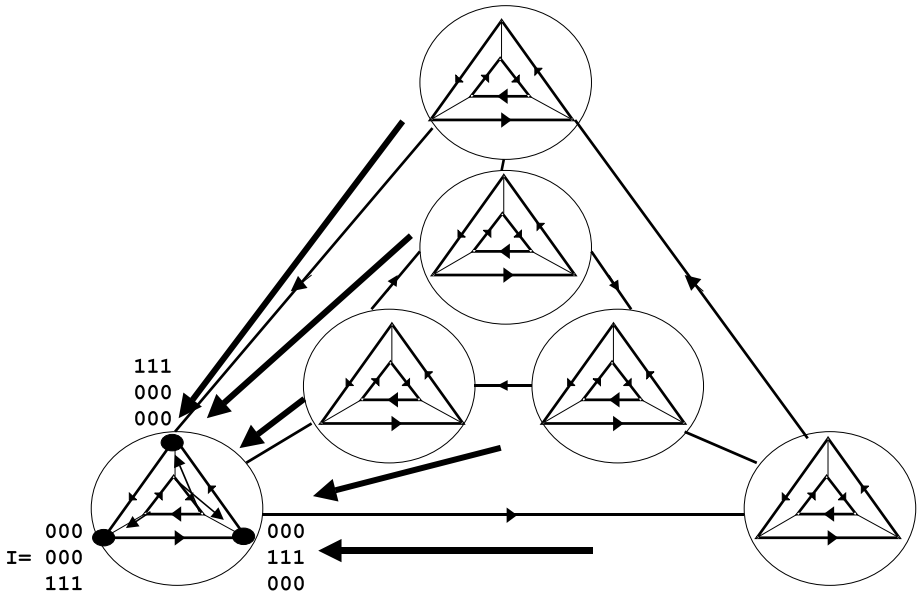

Fig. 3. Symmetric configurations of the matrix $I = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}$.

matrix within a meta-vertex falls into the one obtained by swapping those rows. Hence, $|C_I| = 36/(2!2!) = 9$.

5. Only 2 rows are the same: in this case, a matrix within a meta-vertex falls into the one obtained by swapping those rows. Hence, $|C_I| = 36/2! = 18$.
6. Only 2 columns are the same: in this case, a meta-vertex falls into the one obtained by swapping those columns. Hence, $|C_I| = 36/2! = 18$.
7. Every row permutation gives the same effect as a column permutation: in this case, all the meta-vertices fall into R. Hence, $|C_I| = 36/3! = 6$.
8. Swapping two columns gives the same effect as swapping two rows: in this case, a meta-vertex representing the swap of the columns falls into the one representing the corresponding row swaps. Hence, $|C_I|$ is $36/2$, which is 18.
9. None of the above: in this case, clearly $|C_I| = 36$.

5 Symmetry-breaking constraints

A way to break all row and column symmetries is to add to the model a complete set of symmetry-breaking constraints, i.e., one constraint for each symmetry [1]. Consider a 3×3 matrix I, which we represent as $[X_1, \ldots, X_9]$. Now, we suppose to have an ordering relation $\leq$ among matrices. The complete set of symmetry-breaking constraints is composed by imposing $I \leq A$, where A is any matrix obtained by permuting the rows and/or columns of I. Since the number of

symmetries is 36 (=3!3!), we need the same number of constraints. This complete set of constraints removes all symmetries but the identity.

As the matrix size enlarges, it becomes impractical to impose the complete set of symmetry-breaking constraints. In such a case, only a subset of these constraints could be used and thus not all symmetries are removed. In general, the effect of such symmetry-breaking constraints are evaluated experimentally, by for instance counting the number of symmetric solutions left unbroken. Here we provide some observations which could be generalized and used to evaluate the symmetry-breaking constraint methods.

One way of reducing much of row and column symmetries of a matrix model is to impose that the rows and the columns of the matrix are lexicographically ordered [2]. These constraints are a subset of the complete set of symmetry-breaking constraints, and prevent the row (resp. column) permutations. Hence, in an equivalence class, the symmetric configurations that are surely removed by these constraints reside on the leftmost meta-vertex R, as well as on the identity matrix of every other meta-vertex of the group-graph in Figure 2.

With the lexicographic ordering constraints, we observed that many other symmetric configurations in any equivalence class are also removed. One reason is that some matrices on the group-graph *collapse* into some others when there are repeating values in the matrix. If these matrices happen to fall into R and/or on the identity matrix of every other meta-vertex then lexicographic ordering constraints will remove these symmetric matrices. For instance, if a matrix is formed by only 0 and 1 then we know that there 9 possible scenarios of the group-graph as discussed in Section 4. In the 1st, 2nd, 3rd, and the 7th cases, all matrices fall into the ones that are removed by lexicographic ordering constraints. Hence, all symmetries are surely broken for such kind of equivalence classes. In all the other cases, matrices reside on the parts of the graph that are not reachable by the lexicographic ordering constraints. This hints that if there is any unbroken symmetric matrix, it must belong to one of the equivalence classes described by these cases.

As an example, consider the matrix $I = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$, where two rows as well as two columns are identical. The equivalence class of this matrix is described by case 4. The configuration $C_s^2 R_s^2 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}$ is obtained by applying two shifts on the columns and two shifts on the rows of I. This matrix resides on the topmost vertex of the graph and is not broken by the lexicographic ordering constraints.

In fact, for 0/1 values, 9 symmetric matrices remain to be unbroken, and they belong to the equivalence classes described by the 4th, 5th, 6th, and the 9th cases. It appears that all configurations at distance 1 and 2 from the identity are removed. Those 9 configurations left are all of distance 3 and 4.

Although we understand why some certain configurations are removed by this set of constraints, it is still an open question why all symmetries are broken for certain equivalence classes. For instance, for a 0/1 valued matrix I, all its symmetric configurations are removed if I belongs to an equivalence class described by case 8.

Note that if a matrix has few values then many symmetric configurations fall into the ones that are surely removed by the lexicographic ordering constraints. The more values a matrix can have the more it becomes difficult to remove the symmetric configurations using these symmetry-breaking constraints (i.e., lexicographically ordering the rows and columns).

6 Conclusions and Future Work

In this paper, we studied the structure of the group describing row and column symmetries of a matrix model, and showed how the associated graph can be constructed. We focused on a particular set of symmetry-breaking constraints that are posted on a matrix model so as to remove much of such symmetries. By studying the graph of the group describing these symmetries, we can see which symmetric configurations are removed by the constraints we considered. However, we still fully do not know how some symmetric configurations can be removed for some kind of equivalence classes.

Although this paper is a preliminary study on the relation between a group and its graph and symmetry-breaking constraints, we believe that it is the starting point of understanding the effect of certain symmetry-breaking constrains or algorithms from a formal perspective. We here considered only 3×3 matrices but our study could be generalised.

In the future, we will study this group further. An important question is whether there exist a characterisation that uniquely identifies a matrix in its equivalence class. The group-graph can help us to answer this question. If such a characterisation exists then all symmetric configurations of a matrix can be removed without having to consider each of them one by one. Also, such a graph can help to devise new symmetry-removal algorithms or constraints for matrix models. Least but not last, we plan to repeat this study for other symmetry-breaking constraints for matrix models. This will allow us to compare the relative strengths of the methods.

Acknowledgements

We would like to thank Andrea Roli, Marco Gavanelli, and Luca Benini for useful discussions, suggestions, and ideas.

References

1. J. Crawford, G. Luks, M. Ginsberg, and A. Roy. Symmetry breaking predicates for search problems. In *Proc. KR'96*, pp. 148–159. 1996.
2. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Proc. of CP'2002*. Springer, 2002.

3. P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Matrix modelling. Technical report APES-36-2001. Available from http://www.dcs.st-and.ac.uk/∼apes/reports/apes-36-2001.ps.gz.
4. I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: symmetry breaking during search. In *Proc. of CP'2002*. Springer, 2002.
5. J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proc. ISMIS'93*, pp. 350–361. Springer, 1993.

Breaking Row and Column Symmetries in Matrix Models

Pierre Flener[1], Alan M. Frisch[2], Brahim Hnich[3], Zeynep Kiziltan[3], Ian Miguel[2], Justin Pearson[1], and Toby Walsh[4]

[1] Dept of Information Tech, Uppsala University, Box 337, 751 05 Uppsala, Sweden
PierreF@csd.uu.se, justin@docs.uu.se
[2] Department of Computer Science, University of York, York YO10 5DD, England
Frisch@cs.york.ac.uk, IanM@cs.york.ac.uk
[3] Dept of Information Science, Uppsala University, Box 513, 751 20 Uppsala, Sweden
Zeynep.Kiziltan@dis.uu.se, Brahim.Hnich@dis.uu.se
[4] Cork Constraint Computation Centre, University College Cork, Cork, Ireland
TW@4c.ucc.ie

Abstract. We identify an important class of symmetries in constraint programming, arising from matrices of decision variables where rows and columns can be swapped. Whilst lexicographically ordering the rows (columns) breaks all the row (column) symmetries, lexicographically ordering both the rows and the columns fails to break all the compositions of the row and column symmetries. Nevertheless, our experimental results show that this is effective at dealing with these compositions of symmetries. We extend these results to cope with symmetries in any number of dimensions, with partial symmetries, and with symmetric values. Finally, we identify special cases where all compositions of the row and column symmetries can be eliminated by the addition of only a linear number of symmetry-breaking constraints.

The full text of this statement-of-interest appears in the Proceedings of CP'02, the Eighth International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science, volume 2470, Springer-Verlag, 2002.

Acknowledgements This work is partially supported by grant 221-99-369 of VR (the Swedish Research Council), by institutional grant IG2001-67 of STINT (the Swedish Foundation for International Cooperation in Research and Higher Education), and grant GR/N16129 of EPSRC (the UK Engineering and Physical Sciences Research Council). The last author was supported by an EPSRC advanced research fellowship. We thank our anonymous referees, Warwick Harvey and the members of the APES research group (www.dcs.st-and.ac.uk/~apes/), especially Barbara Smith, for their helpful discussions.

Groups and Constraints: Symmetry Breaking During Search*

Ian P. Gent[1], Warwick Harvey[2], and Tom Kelsey[1]

[1] School of Computer Science, University of St Andrews,
St Andrews, Fife, KY16 9SS, UK
{ipg,tom}@dcs.st-and.ac.uk
[2] IC-Parc, Imperial College
Exhibition Road, London SW7 2AZ, UK
wh@icparc.ic.ac.uk

Abstract. We present an interface between the ECL^{i}PS^e constraint logic programming system and the GAP computational abstract algebra system. The interface provides a method for efficiently dealing with large numbers of symmetries of constraint satisfaction problems for minimal programming effort. We also report an implementation of SBDS using the GAP-ECL^{i}PS^e interface which is capable of handling many more symmetries than previous implementations and provides improved search performance for symmetric constraint satisfaction problems.

Acknowledgements

The St Andrews' authors are assisted by EPSRC grant GR/R29666. We thank Steve Linton, Ursula Martin, Iain McDonald, Karen Petrie, Joachim Schimpf, Barbara Smith and Mark Wallace for their assistance.

The full version of this paper appears in the proceedings of CP 2002.

* This paper is dedicated to Alex Kelsey, 1991–2002.

Partial Symmetry Breaking

Iain McDonald[1] and Barbara Smith[2]

[1] University of St Andrews, Fife, Scotland,
iain@dcs.st-and.ac.uk,
[2] University of Huddersfield, West Yorkshire, England,
b.m.smith@hud.ac.uk

Abstract. In this paper we define *partial symmetry breaking*, a concept that has been used in many previous papers without being the main topic of any research.

1 Introduction and Motivation

We are now at a point in constraint programming research where there are many methods of both recognizing symmetries and breaking symmetries in CSPs. A symmetry breaking **method** for CSPs, can be broken into two parts, the symmetry breaking **technique** and the symmetry **representation**. The technique is how we apply the symmetry breaking. The symmetry representation is concerned with how the descriptions of symmetries are implemented and how we use this implementation to apply the symmetry breaking technique. Symmetry breaking can be improved by reducing the number of symmetries we need to consider. This affects the *representation* of the symmetries but not the *technique*.

In [1] it is shown that where there is a large number of symmetries, we can discard some of them and by doing so reduce run-time greatly. By only describing a subset of symmetries we are performing *partial symmetry breaking* (PSB) i.e. performing some redundant search because the symmetry breaking technique is too costly or even impossible to perform.

Acknowledgments

The first author is funded by an EPSRC studentship. He would also like to thank his supervisors Ian Gent and Steve Linton as well as Tom Kelsey. This research is supported by EPSRC grant GR/R29673. Both authors would like to thank all the members of the APES research group, especially Toby Walsh, for their support and helpful comments.

References

1. Iain McDonald and Barbara Smith. Partial Symmetry Breaking. To appear in Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP2002*. Springer-Verlag, 2002.

Symmetry Breaking Revisited

Jean-François Puget

ILOG, 9 avenue de Verdun, 94253 Gentilly, France,
puget@ilog.fr

Abstract. Symmetries in constraint satisfaction problems (CSPs) are one of the difficulties that practitioners have to deal with. We present in this paper a new method based on the symmetries of decisions taken from the root of the search tree. This method can be seen as an improvement of the nogood recording presented by Focacci and Milano[2] and Fahle, Schamberger and Sellmann[1]. We present a simple formalization of our method for which we prove correctness and completeness results. We also show that our method is theoretically more efficient as the number of dominance checks, the number of nogoods and the size of each nogood are smaller. This is confirmed by an experimental evaluation on the social golfer problem, a very difficult and highly symmetrical real world problem. We are able to break all symmetries for problems with more than 10^{36} symmetries. We report both new results, and a comparison with previous work.

The full version of this paper appears in the proceedings of CP 2002.

References

[1] Fahle, T., Shamberger, S., Sellmann, M.: Symmetry Breaking. Proceedings of CP01 (2001) 93–107.
[2] Focacci, F., Milano, M.: Global Cut Framework for Removing Symmetries. Proceedings of CP01 (2001) 75–92.