Partial Symmetry Breaking by Local Search in the Group*

S. D. Prestwich[1], B. Hnich[2], H. Simonis[1], R. Rossi[3], and S. A. Tarim[4]
[1]Cork Constraint Computation Centre, Department of Computer Science, University College Cork, Ireland
[2]Faculty of Computer Science, Izmir University of Economics, Turkey
[3]Logistics, Decision & Information Sciences Group, Wageningen UR, the Netherlands
[4]Department of Management, Hacettepe University, Ankara, Turkey
s.prestwich@cs.ucc.ie, brahim.hnich@ieu.edu.tr, h.simonis@4c.ucc.ie, roberto.rossi@wur.nl, armtar@yahoo.com

Abstract. The presence of symmetry in constraint satisfaction problems can cause a great deal of wasted search effort, and several methods for breaking symmetries have been reported. In this paper we describe a new method called Symmetry Breaking by Nonstationary Optimisation, which interleaves local search in the symmetry group with backtrack search on the constraint problem. It can be tuned to break arbitrarily many symmetries with high runtime overhead, or as a lightweight but powerful method with low runtime overhead. It has negligible memory requirement, it combines well with lex-leader constraints, and its benefit increases with problem hardness.

1 Introduction

Many constraint satisfaction problems (CSPs) contain symmetries. For example the N-queens problem has 8 (each solution may be rotated through 90, 180 or 270 degrees, and reflected) while other problems may have exponentially many symmetries. The presence of symmetry implies that search effort is being wasted by exploring symmetrically equivalent regions of the search space. By eliminating the symmetry (*symmetry breaking*) we may speed up the search significantly. Several distinct methods have been reported for symmetry breaking in CSPs.

In principle all symmetries can be broken, but this becomes problematic when there are very many symmetries. A common case is that of *matrix symmetry* which often occurs in *matrix models*: constraint problems containing one or more matrices of variables. Given a solution, some or all of the rows (or columns) can often be exchanged to obtain another solution: this is called *row (or column) symmetry*. When this form of symmetry occurs in matrices with more than two

* This material is based in part upon works supported by the Science Foundation Ireland under Grant No. 05/IN/I886. B. Hnich is supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No. SOBAG-108K027.

dimensions it is often called *index symmetry*. Some problems have more complex forms of symmetry on matrices, for example it may be possible to permute elements in one dimension independently for each value in another dimension. All these cases give rise to vast numbers of symmetries that are hard to break completely.

Symmetry Breaking by Nonstationary Optimisation (SBNO) is a new approach to partial symmetry breaking that interleaves local search [29, 30], or evolutionary search [28], with standard backtrack search in order to detect and break symmetry. The local search is performed in the symmetry group associated with the constraint problem, and only limited time is devoted to it. In this paper we define SBNO, show how to use it to break matrix symmetries, and evaluate it on standard benchmarks alone and in combination with another method. The paper extends previously published work [28–30] **[say how]** and is organised as follows. Section 2 provides background material and surveys related work. Section 3 describes SBNO and its application to matrix symmetries. Section 4 evaluates it on standard benchmarks. Section 5 concludes the paper and discusses future work.

2 Background and related work

First we provide some background on symmetry and group theory, and their application to Constraint Programming. For a more complete introduction see [14] from which we draw much of our material.

2.1 Groups and symmetry

For this work we need only the most basic ideas of group theory, so we omit many concepts that are normally mentioned in connection with symmetry breaking.

Group theory is essentially the study of symmetry in mathematics. A group is a non-empty set G of elements with a composition operator $\circ$ and properties called the *group axioms*:

- G is closed under $\circ$: for all $g, h \in G$, $g \circ h \in G$.
- There is an identity element $id \in G$: for all $g \in G$, $id \circ g = g \circ id = g$.
- Every $g \in G$ has an inverse $g^{-1} \in G$ such that $g \circ g^{-1} = g^{-1} \circ g = id$.
- $\circ$ is associative: for all $f, g, h \in G$, $(f \circ g) \circ h = f \circ (g \circ h)$.

An important example is the *symmetric group* S_n which is the group of permutations of n objects.

The *order* of a group G is the cardinality of the set, denoted by $|G|$. A *generating set* for a group is a subset H of the group G that can be used to generate all elements of G, denoted $\langle H \rangle = G$. The elements of H are generators for G. Given two (or more) groups G_1, G_2 we can form their *direct product* $G_1 \times G_2$ which is also a group: the set $\{(x, y) \mid x \in G_1,\ y \in G_2\}$ with composition operator defined by $(x, y) \circ (x', y') = (x \circ x', y \circ y')$.

2.2 Symmetry in Constraint Programming

An important concept for symmetry in Constraint Programming is that of a *group action*. A group element g can operate on the elements of G via the composition operator, but it can also operate on another set S set by permuting its elements. We refer to the elements of S as *points* and denote the *action* of $g \in G$ on point $p \in S$ by p^g. So p^g is the new position of p after S has been permuted by g. We also refer to the *image* S^g of S under G.

For example consider a small 3×3 chessboard. It has 9 squares which are the points, and a symmetry group of 8 elements that permute them. The symmetries include an element $r90$ that rotates the board through 90 degrees, and an element x that reflects the board about a vertical axis; the other elements can be viewed as compositions of these (for example $r180 = r90 \times r90$ performs rotation by 180 degrees) including the identity element id that leaves the board unchanged. Now suppose we have a CSP whose variables correspond to squares on the chess board, and values correspond to pieces placed on the squares, with constraints such as those in the well-known N-queens problem. Then some of the solutions to our constraint satisfaction problem are symmetric to others: applying a group element transforms one solution into another.

The precise meaning of symmetry in Constraint Programming has only recently been formalised satisfactorily [5]. *Solution symmetry* is a permutation of variable-value pairs which preserves the set of solutions, while *problem symmetry* is a permutation of variable-value pairs which preserves the set of constraints. Special cases are *variable (or value) symmetry* in which a set of variables (or values) can be permuted. A special case of variable symmetry is *matrix symmetry* in which the variables of a matrix can be permuted row-wise and column-wise.

A set of variable-value pairs in which each variable appears at most once is a partial assignment during search. If the current partial assignment is symmetric to a previous partial assignment that has already been encountered, then there is no need to search below the current one. This fact might be detected, or prevented by constraints, in various ways.

2.3 Symmetry breaking methods

Reformulation is the ideal approach to handling symmetry in a constraint problem: if we can reformulate the problem so that the model contains no symmetry, then there is no need to break symmetry at all. A case study in [38] uses several reformulations of a combinatorial problem to eliminate various symmetries, and shows that this can pay off in terms of runtime. But it is rarely possible to remove all symmetries by reformulation so we require other methods.

A popular approach to symmetry breaking is to add constraints to the model. It has been shown that all symmetries can in principle be broken by this method [31], which was developed into the *lex-leader* method for Boolean variables and variable symmetries by [6], extended to non-Boolean variables and independent variable and value symmetries by [27, 35], and to arbitrary symmetries by [39]. But in practice too many constraints might be needed if there are exponentially

many symmetries. Instead of explicitly adding lex-leader constraints to a model, a Computational Group Theory system such as GAP [11] can be used during search to find relevant (unposted) constraints, as in the GAPLex method [23].

Symmetry Breaking During Search (SBDS) was invented by [3] and developed by [15]. In SBDS constraints are added during search so that, after backtracking from a decision, future symmetrically equivalent decisions are disallowed. SBDS has been implemented by combining a constraint solver with the GAP system, giving GAP-SBDS [12], which allows symmetries to be specified more compactly via group generators. SBDS can still suffer from the problem that too many constraints might need to be added: GAP-SBDS, for example, can handle billions of symmetries but some problems require many more. A related method to SBDS called *Symmetry Breaking Using Stabilizers* (STAB) [32] only adds constraints that do not conflict with the current partial variable assignment, and has other optimisations to reduce the arity and number of constraints. It does not break all symmetries but has given very good results on problems with up to 10^{91} symmetries.

Symmetry Breaking by Dominance Detection (SBDD) was independently invented by [7, 9] (a similar algorithm was also described by [4]) and combined with GAP to give GAP-SBDD [**?**]. SBDD breaks all symmetries but does not add constraints before or during search, so it does not suffer from the space problem of some methods: GAP-SBDD, for example, has been applied to groups of size 10^{36}, while another SBDD implementation has handled groups of size 10^{78} [33]. Instead it detects when the current search state is symmetrical to a previously-explored "dominating" state. A potential drawback with SBDD is that dominance detection is itself an NP-hard problem (equivalent to subgraph isomorphism) and solving several such problems at each search node can be expensive. However, it was shown by [33] that the dominance tests can be combined into a single auxiliary CSP then solved by standard Constraint Programming methods. Dominance tests can also be written by the programmer for specific problems [7] or more general classes of problem [37], or solved by Computational Group Theory software such as GAP.

For the particular case of matrix symmetry, special symmetry breaking methods have been devised. For an $n \times m$ matrix with full row and column symmetry the symmetry group is $S_n \times S_m$ so there are $n!\,m!$ symmetries. Breaking all such symmetries is NP-hard [6] and requires an exponential number of lex-leaders. However, all row symmetries and all column symmetries can be broken by the lex^2 (or *double-lex*) method [8], which adds constraints to the model to lexically order rows and (separately) columns. Because of its respectable power, low memory and runtime overhead and ease of use, lex^2 is a popular way of eliminating many matrix symmetries, though it can leave an exponential number of symmetries unbroken [22]. A variant of lex^2 is *snake-lex* [16] which uses a different variable ordering. Other methods can be used to break more or all matrix symmetries, though sometimes with high computational cost or problem-specific implementation effort.

Good results have been obtained by *partial* symmetry breaking. If the aim is to minimise runtime then breaking only some of the symmetry can be the best trade-off, for example lex^2 and STAB are good trade-offs for matrix symmetry. The study of partial symmetry breaking methods in general has been proposed by [26].

2.4 More on lex-leaders

We shall make use of lex-leaders so here we provide more background. [31] shows that any form of symmetry can be broken by adding *lex-leader constraints* $X \preceq_{\text{lex}} X^g$ for all $g \in G$, where X is a total assignment on a fixed ordering of the problem variables, and $\preceq_{\text{lex}}$ is the standard lexicographical ordering relation. These constraints prune all solutions except the canonical (lex-least) ones. But in general exponentially many constraints are needed, making the method impractical for problems with large symmetry groups.

Lex-leaders can also be used to derive some other symmetry breaking methods. In particular, lex^n constraints can be derived by posting a lex-leader for each possible adjacent row exchange (or exchange in another dimension), then applying simplification rules as in [6, 10, 25]. For lex^n we need only one rule: a lex-leader of the form $\alpha^n X \beta \preceq_{\text{lex}} \gamma^n Y \delta$, where $\alpha = \gamma$ logically implies $X = Y$, can be replaced by $\alpha\beta \preceq_{\text{lex}} \gamma\delta$. For example the lex-leader $ABCDEF \preceq_{\text{lex}} ACBDFE$ is transformed to the simpler lex-constraint $BE \preceq_{\text{lex}} CF$ where A, B, C, D, E, F are variables.

3 Detecting violated lex-leaders by local search

We now describe the SBNO method. Suppose that we wish to solve a CSP using a standard constraint solver with depth-first search (DFS) and constraint processing. Suppose also that the CSP has symmetry defined by a group G.

3.1 The detection problem

We would like to work with the full set of lex-leaders for G in order to break all symmetries of the CSP. But for some problems this set is too large to work with, so instead of posting them as constraints we try to detect violated lex-leaders indirectly. At a search tree node with partial assignment A, if we can find a group element $g \in G$ such that $A^g \prec_{\text{lex}} A$ then we can backtrack, because A violates the lex-leader $A \preceq_{\text{lex}} A^g$. We shall call the problem of finding such a g the *detection problem*. This is not quite the same as dominance detection in SBDD, which detects states that are symmetric to the current one that have already been visited.

As an example, consider the 4-queens problem with the usual 8 symmetries including reflection about the vertical axis: the group element denoted by x. Suppose that we solve this problem using a constraint model in which each square on the board corresponds to a binary variable, 1 denotes a queen and 0 no

queen at that position. Suppose also that we apply DFS and assign variables in a static row-by-row then column-by-column order. Consider the partial assignment $A = (1, 0, 0, 0, ?, \ldots)$ corresponding to the board configuration in Figure 1(i), where a space denotes no queen, "•" denotes a queen, and "?" denotes no decision. Now A^x is the partial assignment $(0, 0, 0, 1, ?, \ldots)$ corresponding to the board configuration in Figure 1(ii). But $A^x \prec_{\text{lex}} A$ whatever values are chosen for the unassigned variables, so A is symmetric to the lex-smaller node A^x and backtracking can occur from A. It is also possible to reason on unassigned variables, for example if variables x, y are unassigned but the current partial assignment reduces the domain of y to $\{1\}$ then $(1, 0, x, 1, 0) \prec_{\text{lex}} (1, 0, 1, y, 1)$ holds.

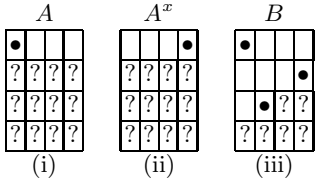


Fig. 1. Search states in 4-queens

Detecting violated lex-leaders does not depend on the details of the constraint solver (for example its value and variable ordering heuristics, or its filtering algorithms) and applies to all variable symmetries. If we fail to detect a violation then we waste some search effort but do not lose correctness, so we can spend a limited time on detection.

3.2 Detection as nonstationary optimisation

We can treat the detection problem as an optimisation problem with G as the search space, so that each $g \in G$ is a search state. The objective function on G to be minimised is the lex ranking of A^g. On finding an element g with sufficiently small objective value such that $A^g \prec_{\text{lex}} A$ (if such a g exists) we have solved the detection problem. This opens up the field of symmetry breaking to a wide range of metaheuristic algorithms.

A practical question here is: how much effort should we devote to detection at each DFS node? If an incomplete search algorithm fails to find an appropriate g, this might be because there is no such element or because the algorithm has not searched hard enough. Too little search might miss important symmetries, while too much will slow down DFS. Our solution is to expend limited effort at each search node to ensure reasonable computational overhead. For example if we apply local search then we might apply one or a few local moves per search tree node, or only at some nodes. The optimisation problem now has an objective function that changes in time: as DFS changes variable assignments A, the objective value of any given g changes because it depends on A^g. This is

called *nonstationary optimisation* in the optimisation literature, so we call our framework *Symmetry Breaking by Nonstationary Optimisation* (SBNO).

Note that even if detection fails at a node, it might succeed a few nodes later. DFS can then backtrack, possibly jumping many levels in the search tree. For example consider the 4-queens problem again. Suppose we did not manage to find group element x at search state A, but instead continued with DFS and only discovered x on reaching search state B shown in Figure 1(iii). Now $B^x \prec_{\text{lex}} B$ so we can backtrack from B. On successful detection we backtrack until we reach a partial assignment A such that $A \preceq_{\text{lex}} A^x$ is no longer violated. Apart from some wasted DFS effort (during which we might find additional non-canonical solutions) the effect on the solutions found is the same as if we had detected the symmetry immediately. Thus SBNO effectively continues to try to break symmetry at a node until DFS backtracks past that node. This gives it an interesting property: a symmetry that would only save a small amount of DFS effort is unlikely to be detected, because DFS might backtrack past A before an appropriate g is discovered; in contrast, one that would save a great deal of DFS effort has a long time in which to be detected by local search. So SBNO should tend to detect and break the *important* symmetries, which we define to be those that make a significant difference to the total execution time. Whether it detects them, and how long it takes to do so, depends on the heuristics we use to solve the detection problem.

3.3 Detection by local search

To make SBNO more concrete we now show how to use local search for detection, though in principle any metaheuristic algorithm can be used. We have already defined the search space (G) and objective function (the lex ranking of A^g). Local search also requires a neighbourhood structure defining the possible local moves from each search state. To impose a neighbourhood structure on G we choose some subset $H \subset G$: from any search state g the possible local moves are the elements of H leading to neighbouring states $g \circ H$ (the set $\{g \circ h \mid h \in H\}$). Thus all G elements are potentially local search states, and some of them (H) are also local moves. To apply hill climbing, from each state g we try to find a local move h such that the objective function is reduced ($A^{g \circ h} \prec_{\text{lex}} A^g$). If a series of moves $(h_1, h_2, \ldots)$ reduces the lex ranking sufficiently then we will find $A^{g \circ h_1 \circ h_2 \circ \cdots} \prec_{\text{lex}} A$ and can backtrack from A.

There is a relationship between group generators and local search in a group. A local search space is *connected* if there exists a series of local moves from any state to any other state. Connectedness is an important property for local search, because a disconnected space may prevent it from finding an optimal solution. It is easy to show that the search space induced by H is connected if and only if H is a generator set for G, as follows. Suppose that H is a generator set for G. We can move from any g to any g' via element $g^{-1} \circ g'$ because $g \circ (g^{-1} \circ g') = (g \circ g^{-1}) \circ g' = g'$. H is a generator set so we can always find a series of elements $h_1, h_2, \ldots$ such that $h_1 \circ h_2 \circ \ldots = g^{-1} \circ g'$. Therefore $g \circ h_1 \circ h_2 \circ \ldots = g'$ and the space is connected. Conversely, suppose that H

is not a generator set for G. Then there exists a $g^* \in G$ such that no series of elements satisfies $h_1, h_2, \ldots = g^*$. But for any g it holds that $g^* = g^{-1} \circ g'$ for some unique g'. Therefore there exists an unreachable state g' from any state g.

If a non-generator set H is used then the local search can become trapped in a subspace that does not contain an appropriate g, so random moves from $G - H$ must be used to counteract this. Random restarts are a well-known technique for both local and backtrack search, but if H is not a generator set then they are necessary not only for heuristic reasons but because the space is disconnected. In our initial experiments we used a generator set H. This is a natural approach which can yield neighbourhoods of manageable size, because any group G has a generator set of size $\log_2(|G|)$ or smaller [19]. However, we found better results using a non-generator set H (which varies dynamically) and restoring connectedness by allowing occasional random moves, as described below.

We use a version of Iterated Local Search [20]. Initialise g to be any group element (we use the identity element). At each search tree node A call the DETECT procedure shown in Figure 2 which returns another group element g' and a truth value: if the truth value is T then detection has occurred with g' and backtracking occurs; if it is F then tree search proceeds as usual, but with the new group element g'. DETECT performs a hill-climbing move on g where possible (via the IMPROVE function), and if $A^g \prec_{\text{lex}} A$ then a violated lex-leader has been detected. On detection, g is kept fixed and backtracking is enforced until reaching a node at which this no longer holds. The IMPROVE function applies an improving local move to g, that is a move h such that $A^{g \circ h} \prec_{\text{lex}} A^g$. The neighbourhood is explored in random order to find these moves. If no such move exists then the state is a local minimum and we call the INITIAL function. INITIAL starts from the identity group element and applies a random move with probability 0.5, a second random move with probability 0.25, a third with probability 0.125, and so on. In this way it is biased toward the identity element but may in principle return any group element. Because we use an unbounded number of random moves at each local minimum, the local search algorithm is *probabilistically approximately complete* [20]: it is guaranteed to find a solution given sufficient time. We will return to this property in Section 3.5.

```
procedure DETECT(g,A)
    if A^g ≺_lex A
        return (g, T)
    else if A is a local minimum
        g' ← INITIAL
        return (g', F)
    else
        g' ← IMPROVE(g)
        return (g', F)
```

Fig. 2. A detection algorithm based on Iterated Local Search

3.4 Application to symmetry in matrix models

The SBNO scheme can be applied to the particular case of matrix symmetry. The current group element g is represented by two lists, one representing a row permutation and the other a column permutation. Choosing a random move g in INITIAL (see Figure 2) might not be practicable for all problems as it is not always possible to efficiently generate a random group element [19]. But in the case of matrix symmetry it is easy: we simply exchange a randomly selected pair of values in the row or column permutation. The local move neighbourhood explored in IMPROVE is the set of row or column exchanges involving the matrix entry corresponding to the variable at which the last $\prec_{\text{lex}}$ test failed. This heuristic is inspired by the conflict-directed heuristics used in many local search algorithms, which focus search effort on the source of failure. We extend SBNO to other forms of matrix symmetry in Sections 4.4 and 4.5.

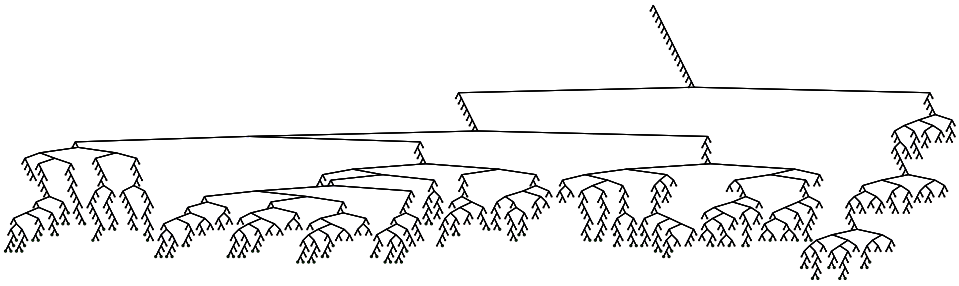
Figure 3 illustrates the effect of SBNO on an all-solution search tree for a Balanced Incomplete Block Design (see Section 4.3). The first search tree uses lex^2 alone while the second also uses SBNO. The triangles in the latter tree indicate where SBNO caused backtracking. The search tree for this problem shows two main branches after an initial fixed assignment. On the left SBNO dramatically reduces the size of the tree, while on the right only a few nodes are removed, reflecting SBNO's nondeterministic nature. Most of the removed nodes on the right are solutions, which are cut off only when all variables have been assigned. In contrast, on the left large subtrees are cut off, containing the majority of the removed solutions. We can also observe some chains of failure, in which a useful symmetry group element discovered at a lower level in the tree is immediately applied to prune higher nodes.

3.5 Symmetry breaking power

The runtime overhead of SBNO depends on how much local search effort we permit at each search tree node. We now investigate the trade-off between local search effort, broken symmetries and runtime. Using the first 10 instances of a benchmark set for the Balanced Incomplete Block Design problem (see Section 4.3) we compare the number of solutions found in an all-solution search with symmetry breaking, as a function of the number of local moves at each search tree node. Table 1 shows the results, along with the actual number of non-symmetrical solutions ("asym") and the number of solutions found by lex^2 and the leading partial symmetry breaking method STAB+lex^2.

The probabilistic approximate completeness property of our local search algorithm (see Section 3.3) implies that SBNO has a nonzero probability of breaking any given symmetry, so if it spends enough time at a search tree node then it will almost certainly detect and break any symmetry. This theoretical result is supported by the experiments, which show that SBNO's symmetry breaking power can be arbitrarily increased simply by performing more local search at each node, to the point that it breaks more symmetry than the best known partial symmetry breaking method for this problem; in fact almost all symmetries.

Search tree under lex^2:



Search tree under SBNO+lex^2:

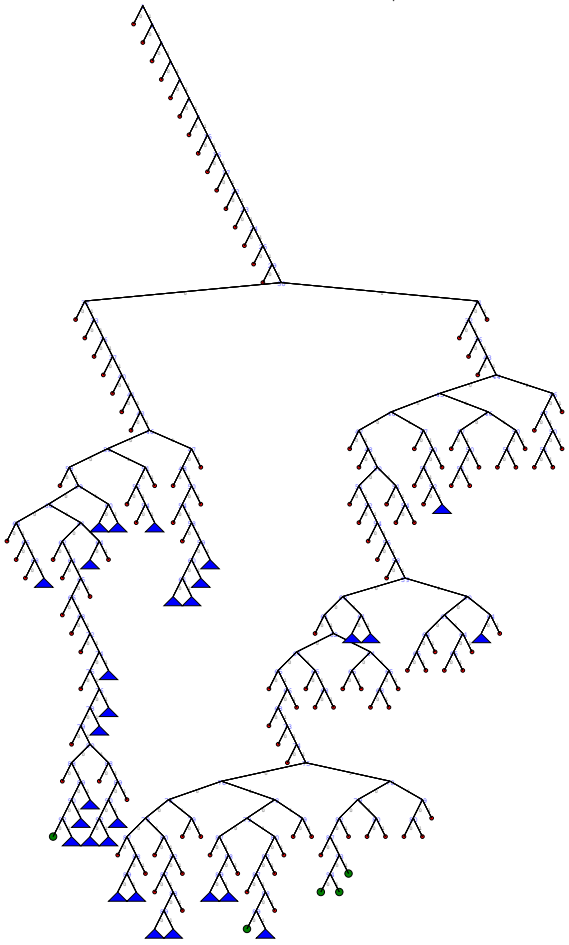


Fig. 3. Effect of SBNO on a BIBD search tree

					SBNO						
instance	asym	lex^2	STAB	1	3	10	30	100	300	1000
6 3 2	1	1	1	11	4	1	1	1	1	1
7 3 1	1	1	1	10	3	1	1	1	1	1
6 3 4	4	21	4	265	109	20	8	4	4	4
9 3 1	1	2	1	29	5	3	1	1	1	1
7 3 2	4	12	7	116	32	17	10	5	4	4
8 4 3	4	92	6	103	33	8	5	4	5	4
6 3 6	6	134	7	1306	601	140	51	20	9	6
11 5 2	1	2	1	74	27	3	2	1	1	1
10 4 2	3	38	4	229	56	9	5	3	4	3
7 3 3	10	220	24	1237	344	111	38	20	12	11

Table 1. Symmetry breaking as a function of local moves (number of solutions found)

3.6 Runtime overhead

We use profiling to estimate the proportion of time spent on SBNO processing at each search tree node. Table 2 shows the results expressed as a percentage (we omit the first two instances because their lex^2 runtimes are too small to measure). They show that with one local move per search tree node SBNO takes between $\frac{1}{3}$ and $\frac{2}{3}$ of the execution time. Thus even if SBNO detects and breaks no symmetries, it will only approximately double the runtime. This shows that SBNO is a lightweight symmetry breaking method compared to complete methods such as SBDS and SBDD, which can spend an arbitrarily high proportion of runtime on symmetry detection.

	SBNO		
instance	1	3	10
6 3 4	65%	85%	94%
9 3 1	67%	86%	89%
7 3 2	60%	82%	92%
8 4 3	53%	76%	91%
6 3 6	73%	88%	96%
11 5 2	33%	75%	87%
10 4 2	53%	74%	90%
7 3 3	65%	85%	94%

Table 2. Time spent on SBNO processing as a function of local moves

The runtime overhead of SBNO is more than compensated for by the gains achieved by breaking symmetry, as shown in Table 3 which shows the runtime for each case. It is clear that performing many local search moves at each node is not worthwhile if our goal is simply to reduce execution time, but that performing

a few moves is worthwhile (and we shall show that the advantage of SBNO increases with problem hardness). In the rest of this paper we shall use just one local move per node, as making more moves often increases runtime when SBNO is combined with static symmetry breaking methods such as lex^2. Table 3 also shows lex^2 runtimes, which are uniformly smaller than those for SBNO. This indicates that lex^2 is much more efficient than SBNO on easy problems, but we shall show that SBNO pays off on some harder problems and that their combination is even better.

			SBNO						
instance		lex^2	1	3	10	30	100	300	1000
6	3 2	0.01	0.08	0.06	0.07	0.2	0.4	1.1	3.5
7	3 1	0.01	0.03	0.02	0.04	0.08	0.2	0.6	1.9
6	3 4	0.06	1.1	1.0	1.1	1.6	3.7	9.2	28
9	3 1	0.01	0.3	0.2	0.3	0.4	1.0	2.7	8.3
7	3 2	0.01	0.4	0.4	0.5	0.8	2.0	5.0	16
8	4 3	0.2	1.1	0.9	1.1	1.8	4.2	11	35
6	3 6	0.5	7.5	6.5	6.7	9.5	19	44	129
11	5 2	0.03	0.7	0.5	0.6	1.1	2.3	5.8	17
10	4 2	0.2	2.7	2.1	2.7	4.6	10	24	71
7	3 3	0.2	3.4	3.1	3.6	5.4	11	28	84

Table 3. Runtime as a function of local moves (sec)

3.7 Memory requirement

SBNO has a negligible memory requirement: it maintains just one dynamically changing group element g representing the current local search state. For matrix symmetry with an $n \times m$ matrix, g is simply a pair of lists using $O(n + m)$ memory.

3.8 Non-determinism

An unusual property of SBNO as a symmetry breaking method is its non-determinism. This explains its behaviour on instance (10,4,2) in Table 1: with 100 and 1000 local moves per node SBNO finds 3 solutions, but with 300 moves it finds 4 solutions. More local search effort *usually* breaks more symmetry but this is not guaranteed. We believe that non-determinism in backtrack search is an undesirable feature that would merely annoy users. We therefore use a built-in pseudo-random number generator to make SBNO deterministic, and use single runs in all our experiments. This does not cure the behaviour noted above, but it means that users need not average results over multiple runs.

4 Experiments

We now test SBNO on problems with different forms of variable symmetry, starting with row and column symmetry. SBNO is implemented in the ECLiPSe Constraint Logic Programming system [2]. For each problem we use a static variable ordering (ordering by rows then columns for matrix symmetry) and a static 0/1 value ordering. With SBNO and lex^2 we constrain each problem so that the rows and columns are both in decreasing lexical order, starting from the top and left of the matrix. All our experiments are performed on a Dell Optiplex 980 mini tower containing an Intel Core i5-650 3.20 GHz processor with 4M cache running Ubuntu Linux.

4.1 Error correcting codes

A *Hamming code* with distance d and length l is a set of l-bit codewords such that each pair of codewords has at least d different bits. The variation considered here has w bits set in each word and we must find a maximal code, that is n words with greatest n. We use a simple constraint model with a matrix of binary variables m_{ij} $(i = 1 \dots n,\ j = 1 \dots l)$ and constraints

$$\begin{array}{ll} \sum_{j=1}^{l} m_{ij} = w & (1 \le i \le n) \\ \sum_{j=1}^{l} \text{reify}(m_{ij} = m_{i'j}) \ge d & (1 \le i < i' \le n) \end{array}$$

To obtain a set of benchmarks we increase n until the problem is only just satisfiable, so a benchmark is characterised by the 4 numbers n, l, d, w. There are 11 hard instances taken from [36]; 4 of these were too hard for our approach but we show results for the remaining 7 in Table 4.

The results clearly show that SBNO scales better than lex^2 and the combination scales better still: the harder the problem the greater the advantage of using SBNO+lex^2, with a speedup of 120 for the hardest instance. Our results are also very competitive with published results. The best results we know of are those of [40] who use set variable methods with a special representation, and were faster than the ROBDD-based methods of other researchers. Though we cannot directly compare execution times with [40] they are generally of similar magnitude and we are able to solve the same instances, with some exceptions: they solve instance (14,10,4,7) very quickly (2 sec) while we found it hard; we appear to be faster on instance (14,10,4,3) (they took 359 sec); and we solve (19,9,4,4) which they did not (though they do not report their cutoff time). Ours is the only method we know of that has solved 7 of the 11 hard benchmarks.

4.2 Steiner systems

A *Steiner system* $S(t, k, n)$ is a set X of n *points*, and a collection of subsets of X of size k called *blocks*, such that any t points of X are in exactly one block. A Steiner system must have exactly $m = \binom{n}{t}/\binom{k}{t}$ blocks. The special case $(t = 2, k = 3)$ is a *Steiner triple system* and the case $(t = 3, k = 4)$ is a *Steiner*

n	l	d	w	lex^2	SBNO	SBNO+lex^2
proof of optimality						
15	8	4	4	6.0	3.8	1.6
13	9	4	3	2.9	3.5	1.3
19	9	4	4	24,243	4,449	798
13	9	4	6	17	5.4	1.2
14	10	4	3	129	64	15
14	10	4	7	3,002	141	25
7	10	6	5	0.03	0.11	0.05
first optimal solution						
14	8	4	4	5.2	2.9	1.5
12	9	4	3	2.2	2.5	1.0
12	9	4	6	2.8	3.1	0.9
13	10	4	3	28	22	9.1
13	10	4	7	104	14	11
6	10	6	5	0.02	0.09	0.04

Table 4. Error correcting code results (sec)

quadruple system. A Steiner triple (or quadruple) system has a solution if and only if $n \bmod 6$ is 1 or 3 (or 2 or 4).

We use a constraint model with a binary matrix x_{ij} $(i = 1 \dots n,\, j = 1 \dots m)$ and constraints

$$\begin{array}{ll} \sum_{j=1}^{n} x_{ij} = k & (1 \le i \le m) \\ \sum_{j=1}^{n} x_{ij} x_{i'j} \le t - 1 & (1 \le i < i' \le m) \end{array}$$

This model has row and column symmetry. Table 5 shows the results. SBNO scales better than lex^2, while SBNO+lex^2 scales better than both: again, the harder the problem the greater the advantage of SBNO+lex^2 over lex^2 alone, with a speedup of 52 for the hardest instance. On these problems we are not competitive with the results of [40], who solve several instances that we cannot. However, they only give results for solvable problems, and they use special labeling strategies whereas we use a simple static strategy.

4.3 Balanced incomplete block designs

Balanced Incomplete Block Designs (BIBDs) have been used to test several symmetry breaking methods. They were originally used in the statistical design of experiments but find other applications such as cryptography. A BIBD is defined as an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. Another way of defining a BIBD is in terms of its *incidence matrix*, which is a binary matrix with v rows, b columns, r ones per row, k ones per column, and scalar product λ between any pair of distinct rows. A BIBD is therefore characterised by its parameters (v, b, r, k, λ).

t	k	n	lex^2	SBNO	SBNO+lex^2
all-solution					
2	3	6	0.01	0.02	0.02
2	3	7	0.02	0.07	0.05
2	3	8	0.1	0.3	0.1
2	3	9	5.0	4.3	1.5
2	3	10	283	86	21
2	4	13	36	9.8	4.4
3	4	7	0.04	0.07	0.05
3	4	8	9.3	5.8	1.7
3	4	9	52327	8510	1007
first-solution					
2	3	7	0.03	0.07	0.04
2	3	9	4.5	3.3	1.5
2	4	13	36	8.1	4.4
3	4	8	9.3	4.5	1.7

Table 5. Steiner system results (sec)

For a BIBD to exist its parameters must satisfy the conditions $rv = bk$, $\lambda(v-1) = r(k-1)$ and $b \geq v$, so we can also characterise a BIBD by (v, k, λ), but these are not sufficient conditions. Constructive methods can be used to design BIBDs of special forms, but the general case is very challenging and there are surprisingly small open problems, the smallest being (22,33,12,8,4).

We use the most direct CSP model for BIBD generation, which represents each matrix element by a binary variable m_{ij} and has constraints:

$$\begin{array}{ll} \sum_{j=1}^{b} m_{ij} = r & (1 \leq i \leq v\} \\ \sum_{i=1}^{v} m_{ij} = k & (1 \leq j \leq b\} \\ \sum_{j=1}^{b} m_{ij} m_{i'j} = \lambda & (1 \leq i < i' \leq v) \end{array}$$

This model also has row and column symmetry. Different researchers use different BIBD instances to test their algorithms. We use the instances of [32] which are the hardest used for all-solution search in the literature, and contain most other problem sets. Table 6 compares lex^2 alone, SBNO alone and SBNO+lex^2, in terms of the number of solutions found and the time for all-solution search. We also show the results for SBDD+lex^2 and STAB+lex^2 from [32]. These times are normalised to our machine: we compared our lex^2 times with those of [32] on 8 instances of different hardness and took the geometric mean of their ratios. However, because of the use of different computers, operating systems, constraint solvers and lex^2 algorithms these estimated times should be treated with caution. Results unreported in [32] (and therefore here) are denoted "—" while results taking longer than 200,000 seconds on our machine are denoted "?".

On BIBD instances SBNO alone turns out to be weaker than lex^2, usually breaking fewer symmetries and taking longer. However, SBNO+lex^2 beats both lex^2 and SBNO alone, with a speedup of up to 64 with respect to lex^2 alone, and

			SBDD+lex^2		STAB+lex^2		lex^2		SBNO		SBNO+lex^2	
v	k	λ	solns	sec*	solns	sec*	solns	sec	solns	sec	solns	sec
6	3	2	1	0.04	1	0.0	1	0.01	11	0.08	1	0.02
7	3	1	1	0.0	1	0.04	1	0.0	10	0.03	1	0.01
6	3	4	4	1.3	4	0.04	21	0.06	265	1.1	8	0.1
9	3	1	1	0.04	1	0.08	2	0.01	29	0.3	2	0.03
7	3	2	4	0.4	7	0.08	12	0.02	116	0.4	11	0.05
8	4	3	4	2.2	6	0.1	92	0.2	103	1.1	17	0.09
6	3	6	6	9.1	7	0.2	134	0.5	1,306	7.5	16	0.6
11	5	2	1	0.2	1	0.2	2	0.03	74	0.7	2	0.07
10	4	2	3	3.4	4	0.2	38	0.2	229	2.7	14	0.2
7	3	3	10	6.2	24	0.2	220	0.3	1,237	3.4	83	0.4
13	4	1	1	0.1	1	0.3	2	0.03	143	1.7	1	0.09
6	3	8	13	47	15	0.4	494	2.6	6,254	33	36	2.1
9	4	3	11	56	41	0.5	2,600	9.4	839	15	97	1.1
16	4	1	1	8.2	1	0.6	12	0.6	1,858	34	2	0.4
7	3	4	35	80	116	0.7	3,209	4.1	9,868	24	412	2.4
6	3	10	19	186	26	1.0	1,366	10	20,546	129	73	7.0
9	3	2	36	115	344	2.1	5,987	5.7	20,266	50	1,499	5.0
16	6	2	3	12	3	2.1	46	1.9	4,753	103	11	1.0
15	5	2	0	40	0	2.8	0	76	0	1,867	0	5.0
13	3	1	2	47	21	2.8	12,800	44	15,572	104	403	5.1
7	3	5	109	639	542	3.1	33,304	52	63,331	160	1,482	13
15	7	3	5	53	19	4.1	118	3.2	3,157	152	18	1.5
21	5	1	1	1.9	1	7.0	12	1.4	11,803	217	2	0.7
25	5	1	1	643	1	7.8	864	220	718,637	41,425	15	15
10	5	4	21	540	302	7.8	8,031	104	4,105	114	301	7.7
7	3	6	418	5,183	2,334	14	250,878	490	365,435	964	6,057	72
22	7	2	0	39	0	34	0	122	0	10,654	0	9.6
7	3	7	1,508	33,215	8,821	57	1,460,332	3,604	1,741,472	4897	20,753	330
8	4	6	2,310	45,435	17,890	70	2,058,523	4,399	255,445	1,767	33,649	233
19	9	4	6	26,413	71	95	6,520	5,092	27,386	100,368	38	97
10	3	2	960	12,010	24,563	105	724,662	689	763,852	1,994	45,083	154
31	6	1	1	1,965	1	80	864	522	?	?	4	17
7	3	8	5,413	—	32,038	103	6,941,124	21,136	8,284,396	24,634	66,136	1,438
9	3	3	22,521	—	315,531	313	14,843,772	14,639	6,301,776	18,987	382,891	1,636
7	3	9	—	—	105,955	404	28,079,394	105,737	33,806,558	103,471	192,446	5,472
15	3	1	80	17,765	6,782	527	32,127,296	138,230	2,876,638	26,608	84,161	1,296
21	6	2	0	—	0	1,388	0	?	?	?	0	4,774
13	4	2	2,461	—	83,337	1,648	3,664,243	?	?	?	72,133	2,719
11	5	4	4,393	—	106,522	1,949	6,143,408	?	?	?	67,494	3,455
12	6	5	—	—	228,146	13,168	—	?	?	?	155,638	26,360
25	9	3	—	—	17,016	38,209	—	?	?	?	1,428	16,156
16	6	3	—	—	769,482	60,749	—	?	?	?	265,792	91,478

Table 6. BIBD results (*estimated times)

on the hardest problems it overtakes STAB+lex^2 in symmetry breaking and has similar (estimated) runtime. As shown by [32] SBDD+lex^2 is much slower than the partial methods, which is the price paid for complete symmetry breaking.

4.4 Equidistant frequency permutation arrays

The problem of finding *equidistant frequency permutation arrays* (EFPAs) was recently attacked with Constraint Programming by Gent *et al.* [21]. An instance with parameters (d, λ, q, v) is the problem of finding v codewords of length $q\lambda$ with an alphabet $\{1, \ldots, q\}$, each symbol occurring λ times in each codeword, and a Hamming distance of d between each pair of codewords.

We use the Boolean model of Gent *et al.* (though they obtained better results using a more complex model). The variables form a 3-dimensional Boolean matrix m_{ijk} ($i = 1 \ldots v$, $j = 1 \ldots q$, $i = 1 \ldots q\lambda$) and $m_{ijk} = 1$ means that codeword i has symbol j at position k. The constraints are as follows. Each position contains one symbol:

$$\sum_{j=1}^{q} m_{ijk} = 1$$

for all i, k. Each symbol occurs λ times per codeword:

$$\sum_{k=1}^{q\lambda} m_{ijk} = \lambda$$

for all i, j. Hamming distances:

$$\sum_{j=1}^{q} \sum_{k=1}^{q\lambda} \text{reify}(m_{ijk} \neq m_{i'jk}) = 2d$$

for all i, i' such that $i < i'$. This model has 3-dimensional index symmetry, which is a straightforward generalisation of row and column symmetry to 3-dimensional matrices, so we can use lex^3 instead of lex^2 as do Gent *et al.* We generated these constraints as follows: for each pair of adjacent positions in each dimension we generate a lex-leader with those positions exchanged, then apply the simplification rule described in Section 2.4.

It is easy to extend SBNO to index symmetry in n dimensions by maintaining n permutation lists. To apply a random local move we exchange two randomly-chosen positions in a randomly-chosen permutation. The local move neighbourhood explored in IMPROVE (see Figure 2) is again the set of exchanges involving the matrix entry corresponding to the variable at which the last $\prec_{\text{lex}}$ test failed.

Gent *et al.* choose 10 sets of parameters d, λ, q and for each set choose v to be just small enough for a satisfiable problem. They then take a pair of instances (d, λ, q, v) and $(d, \lambda, q, v + 1)$, the latter problem being unsatisfiable. They use the Minion constraint solver [13] on one processor of an Intel Core 2 Duo P8400 2.26GHz. We consider only the 10 unsatisfiable instances.

Table 7 shows our lex^3 and SBNO+lex^3 results. We use the canonical variable ordering i, j, k on the matrix m_{ijk}. The results show that adding SBNO to lex^3 always improves runtimes, with the greatest (known) improvement of 10 times on the hardest problem solved by both (4,5,4,11). This agrees with our earlier results showing that the improvement due to SBNO tends to increase with problem hardness.

d	λ	q	v	lex^3	SBNO+lex^3
3	7	7	7	3048	404
3	8	8	8	8321	1377
4	3	4	7	541	129
4	4	3	8	19	16
4	4	4	9	3011	519
4	4	5	11	?	10585
4	5	4	11	28813	1937
5	4	3	8	119	35
5	4	4	9	?	27406
6	4	3	13	122623	11697

Table 7. EFPA results (sec)

Better results were reported by Gent et al. on the same Boolean model, but we believe that this is because Minion is faster than ECLiPSe, especially on matrix models. The authors report that on BIBDs Minion is up to 128 times faster than ILOG Solver [1], and Solver is often more efficient than ECLiPSe. Whether adding SBNO to Minion would produce similar speedups is an interesting open question, but we see no reason for pessimism.

4.5 The social golfer problem

The Social Golfer Problem (SGP) is a commonly used benchmark for symmetry breaking techniques. A group of n golfers wish to play golf each week, arranged into g groups of s golfers, where $n = gs$. The problem is to find the maximum number of weeks w such that no two golfers play in the same group more than once.

We use a pure Boolean model with a $w \times g \times n$ Boolean matrix m_{ijk}, where $m_{ijk} = 1$ means that golfer k plays in group j in week i. The constraints are as follows. Each group contains s golfers:

$$\sum_{k=1}^{n} m_{ijk} = s$$

for all i, j. Each golfer plays in one group per week:

$$\sum_{j=1}^{g} m_{ijk} = 1$$

for all i, k. No two golfers plays in the same group more than once:

$$\sum_{i=1}^{w} \sum_{j=1}^{g} m_{ijk} m_{ijk'} \leq 1$$

for all k, k' such that $k < k'$. There are three forms of symmetry in this model:

- weeks can be permuted;
- players can be permuted;
- groups can be *independently* permuted for each week.

The latter symmetry means that we can perform permutations in the group dimension for every value in the week dimension, so there is more symmetry than 3-dimensional index symmetry. Yet we would like to add some form of lex-leader to the model. Recall from Section 2.4 that lex^n constraints can be derived by posting a lex-leader for each possible adjacent row exchange (or exchange in another dimension) then applying a simplification rule. We apply the same strategy to the SGP: for each exchange of adjacent weeks or players, and for each exchange of adjacent groups within each week, we generate a lex-leader then apply the simplification rule.

We extend SBNO to handle this form of symmetry by maintaining as many permutations as required: one for the weeks, one for the players, and a group permutation for each week. As for lex^n, to apply a random local move we exchange two randomly-chosen values in a randomly-chosen permutation, and the local move neighbourhood explored in IMPROVE (see Figure 2) is the set of exchanges involving the matrix entry corresponding to the variable at which the last $\prec_{\text{lex}}$ test failed.

We label m_{ijk} with dimensions ordered weeks-groups-golfers. Results on a set of benchmarks are shown in Table 8. The simplified lex-leaders are denoted by "lex". WH is the model of Harvey [17], JFP1 is Puget's first model in [33] using set variables and SBDD (so the number of solutions is optimal), JFP2 is Puget's improved model called "top" in [33] with only partial symmetry breaking to reduce overhead, LL is the "int-set" model of Law and Lee [24] which was the best of several integer and set variable models. We do not list the number of solutions for WH as they are the same as for JFP1. The number of solutions for LL were not reported in [24]. Empty entries denote unreported results, "—" means timed out, and "?" indicates a probable typo (Puget wrote "0" here but (5,5,6) is solvable). Even taking into account different machine speeds our results are better than all others except JFP2, again showing the effectiveness of SBNO combined with (simplified) lex-leaders: either we break more symmetries or have smaller overheads than most other approaches.

Why is JFP2 better? It uses a different model with a set variable for each week denoting which of the possible groups play that week. This requires all possible groups to be precomputed, which is a large number (for instance (8,4,10) has 35960 groups), breaking all group symmetries and leaving only player and week symmetries. This is a very significant reduction in symmetry but we do

			WH	JFP1		JFP2		LL	lex		SBNO+lex	
g	s	w	sec	solns	sec	solns	sec	sec	solns	sec	solns	sec
4	3	2	0.7	1	0.03	1	0.02		2	0.0	1	0.01
4	3	3	15	4	0.2	14	0.05		112	0.07	48	0.1
4	3	4	49	3	0.4	15	0.06	0.8	82	0.2	34	0.3
4	3	5	29	0	0.5	0	0.03		0	0.1	0	0.2
4	4	2	2.2	1	0.04	1	0.07		1	0.0	1	0.0
4	4	3	8.5	2	0.07	8	0.1		24	0.03	24	0.08
4	4	4	8.5	1	0.1	5	0.1	8.1	6	0.04	4	0.1
4	4	5	12	1	0.1	4	0.2	5.6	2	0.04	2	0.1
4	4	6	21	0	0.2				0	0.03	0	0.1
5	3	2	17	2	0.2	2	0.1		24	0.08	7	0.1
5	3	3	—	251	199	1493	15	140	197440	116	9868	18
5	3	4	—			353812	105				659406	2216
5	3	5	—			528980	298				596765	10468
5	3	6	—			3765	100				2479	14869
5	3	7	—			102	7.8				52	18520
5	4	2	17	1	0.3	1	0.6		6	0.1	1	0.2
5	4	3	2502	40	33	182	5	98	56448	86	5768	24
5	4	4	—			524	8.1	4771			5953	512
5	4	5	—			147	7.5				684	495
5	4	6	—			0	3.6				0	495
5	5	2	52	1	0.4	1	6.5		1	0.02	1	0.04
5	5	3	249	2	0.7	18	12	0.5	1344	2.7	707	4.5
5	5	4	1304	1	1.5	5	22	2.1	216	12	47	11
5	5	5	4027	1	3.5	4	23		144	12	28	13
5	5	6	—	?	6.2	12	38		36	8	5	9
6	4	2	605			4	7	1.0	351	8.2	42	3.1
6	5	6	—			30	42811				—	—
6	5	7	—			0	5657				—	—

Table 8. SGP results

not think that it fully explains the results. A more compact model by Smith [38] breaks the same symmetries by reformulation but does not give results as good as those of JFP2. We conjecture that JFP2 also benefits from its use of higher-level variables.

4.6 Covering arrays

Finally, a problem that appears unrelated to the SGP but with similar symmetry. A *covering array* $CA(t, k, g)$ of size b is an $b \times k$ array consisting of b vectors of length k with entries from $\{0, 1, \ldots, g-1\}$ (g is the size of the alphabet) such that every one of the g^t possible vectors of size t occurs at least once in every possible selection of t elements from the vectors. The parameter t is referred to as the *covering strength*. The objective is to find the minimum b for which a $CA(t, k, g)$ of size k exists.

Several constraint models were described and compared by Hnich *et al.* [18] and we now review their best model. First, the obvious "naïve" model has a $b \times k$ matrix of integer variables x_{ri} for $1 \leq r \leq b$ and $1 \leq i \leq k$, such that $x_{ri} = m$ if the value of parameter i in test vector r is m. However, it is hard to express the *coverage constraints* (every subset of t parameters must be combined in all possible g^t ways). A different viewpoint of the problem can concisely express the covering constraints: an alternative matrix of integer variables, each of whose b rows represents a possible setting of the parameters, as before. But there are now $\binom{k}{t}$ columns, each representing one of the possible t-combinations. So each variable represents a tuple of t variables in the naïve model. Now the coverage constraints can easily be expressed via global cardinality constraints: every number in the range 0 to $2^t - 1$ should be present *at least* once and *at most* $b - 2^t + 1$ times in the b test vectors in the column corresponding to the t-tuple. But the values assigned to two compound variables must be consistent in terms of the values they imply for the covering array, and these *intersection constraints* are harder to express in the new matrix. Best results were found using by using both the original and alternative matrices. The coverage constraints can be expressed on the alternative matrix. Linear *channelling constraints* associate each compound variable in the alternative matrix with the t corresponding variables in the original matrix. The intersection constraints are now redundant. The alternative matrix is used for variable labelling, by column then by row. There is a great deal of symmetry in the original matrix: rows and columns can both be permuted and there is also a value symmetry: the values in each column can be permuted. To break some of this symmetry, lex^2 was applied to the original matrix, and the number of occurrences of each value in each column was ordered. The values in the first row of the matrix were set to 0.

We use the same model but without lex^2 or constraints to order value occurrences, instead channelling the original matrix into a new 3-dimensional binary matrix to break symmetries. As in the SGP, one dimension has a permutation for every value of another dimension. We generate simplified lex-leaders and extend SBNO in the same way as for the SGP. We use the binary matrix for variable labelling with the canonical variable ordering with dimensions ordered

column-row-value. Because our version of ECLiPSe does not have an efficient implementation of the global cardinality constraint, we use linear constraints on the alternative matrix instead.

Results are shown in Table 9, where "Solver" denotes the ILOG Solver results of Hnich *et al.*, "lex^2+ord" an ECLiPSe recreation of the model used by Hnich *et al.* but with linear cardinality constraints, "lex" our new model with lex-leaders on the binary matrix, and "SBNO+lex" the same but with SBNO added. Only unsatisfiable problems are used. Comparing the Solver results against the lex^2+ord results shows that ECLiPSe is less efficient than Solver on this problem. Experiments with a newer version of ECLiPSe indicate that this is only partly due to the lack of a global cardinality constraint, so Solver must have some other advantage. But the lex results are better than the lex^2+ord results, showing that our use of lex-leaders is more efficient than the symmetry breaking method of Hnich *et al.* The SBNO+lex results are better still, and similar to the (unnormalised) times for Solver. We conjecture that implementing SBNO+lex in Solver will give even better results.

t	k	g	b	Solver	lex^2+ord	lex	SBNO+lex
3	5	2	8	0.01	0.0	0.0	0.02
3	5	2	9	0.01	0.05	0.02	0.06
3	6	2	10	0.02	0.4	0.1	0.2
3	6	2	11	0.09	0.8	0.5	0.5
3	12	2	12	5.8	177	41	10
3	12	2	13	270	922	1145	69
4	6	2	16	0.01	0.3	0.5	0.8
4	6	2	17	0.02	34	1.5	2.4
4	6	2	18	0.06	325	2.6	3.6
4	6	2	19	0.4	1126	6.0	6.4
4	6	2	20	20	2448	27	15
4	7	2	21	35	10792	374	62
4	7	2	22	247	19537	2663	261
4	7	2	23	1505	39008	17053	1138

Table 9. CAN results (sec)

5 Conclusion

This paper described SBNO, a framework for applying metaheuristic search to symmetry breaking during backtrack search, and an implementation using local search for various symmetries in matrix models. On six classes of highly symmetric problem SBNO was shown to be a powerful technique, especially in combination with lex^2 and other lex-leaders. Interestingly, the benefit of SBNO is greatest on the hardest problems. Though methods such as lex^2 are fast and

use good filtering algorithms, they break only a limited (though significant) set of symmetries. In contrast, SBNO breaks arbitrarily many of the symmetries, and the harder the problem the longer it has to detect more symmetries. Adding lex-leaders to the model and applying SBNO are complementary techniques that work well together.

The negligible memory requirement and modest computational overhead of SBNO make it suitable for problems with arbitrarily large symmetry groups. Other symmetry breaking methods have used Constraint Programming or Computational Group Theory algorithms to solve auxiliary problems arising in symmetry breaking, but as far as we know SBNO is the first use of metaheuristics for this purpose. This connection between symmetry breaking and metaheuristics is likely to be fruitful for Constraint Programming. An obvious way to try to improve SBNO is to use more sophisticated metaheuristics than the simple Iterated Local Search algorithm used in this paper.

An issue unexplored in this paper is that of combining SBNO with dynamic variable ordering heuristics. It breaks most symmetry when used with the canonical (static) variable ordering but this might not always give the best runtimes. Another way in which SBNO might be improved is to make it dynamic, possibly using techniques from [34]: though SBNO dynamically detects violated lex-leaders, the set of lex-leaders it uses is statically determined by the chosen canonical variable ordering, so it is classed as a static symmetry breaking method. SBNO can also be generalised to arbitrary variable symmetry in a straightforward way, and it may be possible to generalise it to value symmetry (directly instead of via binary matrices) and conditional symmetry by using the results of [39] on *generalised lex-leaders*. We also hope to combine it with other partial symmetry breaking methods such as STAB and Snake-lex. We conjecture that SBNO will boost the performance of *any* partial symmetry breaking method for variable symmetry, as it may discover any violated lex-leader.

Acknowledgement Thanks to Kish Shen for help with ECLiPSe, including performing experiments with a new version of the global cardinality constraint.

References

1. ILOG S. A. *ILOG Solver 6.0: Reference Manual*, 2003. Gentilly, France.
2. K. R. Apt and M. G. Wallace. *Constraint Logic Programming Using Eclipse*. Cambridge University Press, 2007.
3. R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 1999.
4. C. A. Brown, Finkelstein, and P. W. Purdom. Backtrack searching in the presence of symmetry. In *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 357 of *Lecture Notes in Computer Science*, pages 99–110. Springer, 1988.
5. D. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11:115–137, 2006.

6. J. Crawford, M. L. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, 1996.
7. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2001.
8. P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *8th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 2002.
9. F. Focacci and M. Milano. Global cut framework for removing symmetries. In *7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 77–92. Springer, 2001.
10. A. M. Frisch and W. Harvey. Constraints for breaking all row and column symmetries in a three-by-two matrix. In *3rd International Workshop on Symmetry in Constraint Satisfaction Problems*, 2003.
11. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.12*, 2008.
12. I. P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: Symmetry breaking during search. In *8th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 415–430. Springer, 2002.
13. I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast, scalable, constraint solver. In *17th European Conference on Artificial Intelligence*, pages 98–102, 2006.
14. I. P. Gent, K. E. Petrie, and J.-F. Puget. *Handbook of Constraint Programming*, chapter Symmetry in Constraint Programming, pages 329–376. Elsevier, 2006.
15. I. P. Gent and B. M. Smith. Symmetry breaking in constraint programming. In *14th European Conference on Artificial Intelligence*, pages 599–603, 2000.
16. A. Grayland, I. Miguel, and C. M. Roney-Dougal. Snake lex: An alternative to double lex. In *15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 391–399. Springer, 2009.
17. W. Harvey. Symmetry breaking and the social golfer problem. In *Workshop on Symmetry in Constraints*, 2001.
18. B. Hnich, S. D. Prestwich, E. Selensky, and B. M. Smith. Constraint models for the covering test problem. *Constraints*, 11(3):199–219, 2006.
19. D. F. Holt, B. Eick, and E. A. O'Brien. *Handbook of Computational Group Theory.* Chapman & Hall/CRC, 2005.
20. H.H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications.* Morgan Kaufmann, 2004.
21. P. Gent I, P. McKay, I. Miguel, P. Nightingale, and S. Huczynska. Modelling equidistant frequency permutation arrays in constraints. In *8th Symposium on Abstraction, Reformulation, and Approximation*, 2009.
22. G. Katsirelos, N. Narodytska, and T. Walsh. On the complexity and completeness of static constraints for breaking row and column symmetry. In *16th International Conference on Principles and Practice of Constraint Programming*, volume 6308 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2009.
23. T. Kelsey, C. Jefferson, K. Petrie, and S. Linton. Gaplex: Generalised static symmetry breaking. In *6th International Workshop on Symmetry Breaking*, pages 17–23, 2006. Nantes, France.

24. Y. C. Law and J. H. M. Lee. Symmetry breaking constraints for value symmetries in constraint satisfaction. *Constraints*, 11:221–267, 2006.
25. E. Luks and A. Roy. The complexity of symmetry-breaking formulas. *Annals of Mathematics and Artificial Intelligence*, 41(1):19–45, 2004.
26. I. McDonald and B. Smith. Partial symmetry breaking. In *8th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 207–213. Springer, 2002.
27. K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. Technical Report APES-56a-2003, APES Research Group, 2003.
28. S. D. Prestwich, B. Hnich, R. Rossi, and S. A. Tarim. Symmetry breaking by metaheuristic search. In *8th International Workshop on Symmetry and Constraint Satisfaction Problems*, 2008.
29. S. D. Prestwich, B. Hnich, R. Rossi, and S. A. Tarim. Symmetry breaking by nonstationary optimisation. In *19th Irish Conference on Artificial Intelligence and Cognitive Science*, 2008.
30. S. D. Prestwich, B. Hnich, R. Rossi, and S. A. Tarim. Boosting partial symmetry breaking by local search. In *9th International Workshop on Symmetry and Constraint Satisfaction Problems*, 2009.
31. J.-F. Puget. On the satisfiability of symmetrical constraint satisfaction problems. In *Methodologies for Intelligent Systems*, volume 689 of *Lecture Notes in Artificial Intelligence*, pages 350–361. Springer, 1993.
32. J.-F. Puget. Symmetry breaking using stabilizers. In *9th International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, pages 585–599. Springer, 2003.
33. J.-F. Puget. Symmetry breaking revisited. *Constraints*, 10(1):23–46, 2005.
34. J.-F. Puget. Dynamic lex constraints. In *12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 453–467. Springer, 2006.
35. J.-F. Puget. An efficient way of breaking value symmetries. In *21st National Conference on Artificial Intelligence*, pages 117–122. AAAI Press / The MIT Press, 2006. Stanford, California, USA.
36. A. Sadler and C. Gervet. Enhancing set constraint solvers with lexicographic bounds. *Journal of Heuristics*, 14(1):23–67, 2008.
37. M. Sellmann and P. van Hentenryck. Structural symmetry breaking. In *19th International Joint Conference on Artificial Intelligence*, pages 298–303, 2005. Edinburgh, Scotland, UK.
38. B. M. Smith. Reducing symmetry in a combinatorial design problem. In *International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2001.
39. T. Walsh. General symmetry breaking constraints. In *12th International Conference on Principles and Practice of Constraint Programming*, volume 4204 of *Lecture Notes in Computer Science*, pages 650–664. Springer, 2006.
40. J. Yip and P. van Hentenryck. The evaluation of length-lex set variables. In *15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 817–832. Springer, 2009.