Symmetry Breaking by Metaheuristic Search*

S. D. Prestwich[1], B. Hnich[2], R. Rossi[1], and S. A. Tarim[3]
[1]Cork Constraint Computation Centre, Ireland
[2]Faculty of Computer Science, Izmir University of Economics, Turkey
[3]Department of Management, Hacettepe University, Turkey
s.prestwich@cs.ucc.ie, brahim.hnich@ieu.edu.tr,
r.rossi@4c.ucc.ie, armagan.tarim@hacettepe.edu.tr

Abstract

Several methods exist for breaking symmetry in constraint problems, but most potentially suffer from high memory requirements, high computational overhead, or both. We describe a new partial symmetry breaking method that can be applied to arbitrary variable/value symmetries. It models dominance detection as a nonstationary optimisation problem, and solves it by resource-bounded metaheuristic search in the symmetry group. It has low memory requirement and computational overhead, yet in preliminary experiments on BIBD design it breaks most symmetries.

1 Introduction

Many constraint satisfaction problems (CSPs) contain symmetries. For example the N-queens problem has 8 (each solution may be rotated through 0, 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 and a summary is provided by [Gent *et al.*, 2006].

Reformulating a problem to eliminate its symmetries is an excellent approach when possible, but in many problems it is difficult or impossible to eliminate all symmetries. Probably the most common approach is to break symmetries by *adding constraints* to the model. It has been shown that all symmetries can in principle be broken by this method [Puget, 1993], which was developed into the *lex-leader* method by [Crawford *et al.*, 1996]. But in practice too many constraints might be needed if there are exponentially many symmetries. Good results have been obtained by adding subsets of the constraints to obtain *partial* symmetry breaking: for example in matrix models it is common to have permutation symmetry on both rows and columns, but breaking all such symmetries is NP-hard and requires an exponential number of symmetry breaking constraints. Breaking row and column symmetries separately (*double-lex* [Flener *et al.*, 2002]) does not break all combined symmetries but is tractable. Another drawback with symmetry breaking constraints is that they do not respect the search heuristics: the excluded symmetrical solutions might have been found quickly by the search algorithm, and those remaining might take much longer to find.

Dynamic symmetry breaking methods have been devised that do respect search heuristics. *Symmetry Breaking During Search* (SBDS) was invented by [Backofen and Will, 1999] and further elucidated by [Gent and Smith, 2000]. 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 computational group theory system, giving GAP-SBDS [Gent *et al.*, 2002], 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: it can handle billions of symmetries but some problems require many more. A related method to SBDS called *Symmetry Breaking Using Stabilizers* (STAB) [Puget, 2003] only adds constraints that do not affect the current partial variable assignment. STAB does not break all symmetries but has given good results on problems with up to 10^{91} symmetries.

Symmetry Breaking by Dominance Detection (SBDD) was independently invented by [Fahle *et al.*, 2001; Focacci and Milano, 2001] (though a similar algorithm was described earlier by [Brown *et al.*, 1988]) and combined with GAP to give GAP-SBDD [Gent *et al.*, 2002]. SBDD breaks all symmetries but does not add constraints before or during search, so it does not suffer from the space problem of other methods. Instead it detects when the current search state is symmetrical to a previously-explored "dominating" state, thus respecting search heuristics. It does not need to compare the current search state with *all* previous states, only those corresponding to fully-explored subtrees (*nogoods*). The number of these states is at worst linear in the number of problem variables, and some of them can be ignored if the value ordering heuristic in the search algorithm is static, making SBDD a practicable method. Symmetry between these states

*S. A. Tarim and B. Hnich are supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No. SOBAG-108K027. R. Rossi is supported by Science Foundation Ireland under Grant No. 03/CE3/I405 as part of the Centre for Telecommunications Value-Chain-Driven Research (CTVR) and Grant No. 05/IN/I886.

and the current state is established by *dominance detection* which checks whether a previous nogood can be transformed by a symmetry then extended to the current state. A 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 [Puget, 2005] that the dominance tests can be combined into a single auxiliary CSP then solved by standard constraint programming methods.

In this paper we describe and test a new approach to partial symmetry breaking. It is related to SBDD but models dominance detection as a nonstationary optimisation problem, and solves it by resource-bounded metaheuristic search. It has low time and memory requirements and, unlike other partial symmetry breaking methods, the symmetries it fails to break are likely to be those with little effect on runtime. Section 2 describes the new method, Section 3 presents a case study using local search for dominance detection, Section 4 applies a memetic algorithm, and Section 5 concludes the paper.

2 Dominance detection by metaheuristics

Suppose that we are solving a problem using depth-first search (DFS) with static value orderings, static or dynamic variable ordering, and constraint processing (or relaxation in branch-and-bound). Suppose also that the problem has variable and/or value symmetry defined by a group G. We further assume that any two variables that are symmetric under G have the same domain and static value ordering.

2.1 Dominance as optimisation

We use a different dominance test than that used in SBDD, based on the following idea. If we can apply a group element $g \in G$ to the current partial assignment A such that $A^g \prec_{\text{lex}} A$, where $\prec_{\text{lex}}$ means *strictly less than under the lexicographical ordering induced by domain value orderings* and A^g denotes the action of g on A, then under the above assumptions A^g dominates A in the SBDD sense and we can backtrack from A.

Proposition 1 *Given a CSP with variables $v_1, \ldots, v_n$ to be solved by DFS with static value orderings. If from a partial assignment A of variables $v_{i_1}, \ldots, v_{i_m}$ $(m < n)$ we can find some $g \in G$ such that $A^g \prec_{\text{lex}} A$ then A is a nogood.*

Proof If $A^g \prec_{\text{lex}} A$ then A^g and A must be of the form

$$A = [(v_{i_1}, a_1), \ldots, (v_{i_{k-1}}, a_{k-1}), (v_{i_k}, a_k), \ldots]$$
$$A^g = [(v_{j_1}, a_1), \ldots, (v_{j_{k-1}}, a_{k-1}), (v_{j_k}, a'_k), \ldots]$$

where $k \geq 1$ and $a'_k < a_k$ under the relevant static value ordering on $\text{dom}(v_{i_k}) = \text{dom}(v_{j_k})$. Because $a'_k < a_k$, under the DFS assumption the search tree below partial assignment

$$[(v_{i_1}, a_1), \ldots, (v_{i_{k-1}}, a_{k-1}), (v_{i_k}, a'_k), \ldots]$$

has already been explored. Therefore under the assumptions on value ordering a subtree symmetric to that under A^g has also been explored. But A is symmetric to A^g so we can backtrack from A. □

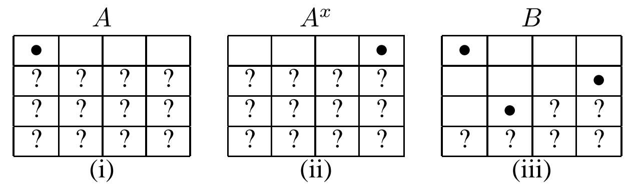

Figure 1: Search states in 4-queens

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 matrix 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 with static variable ordering 0<1 for all variables, 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 we have already explored the subtree under A^x and found no solutions, and A^x is a nogood. A is symmetric to A^x, so A is also a nogood and we can backtrack from it.

We can model this dominance test as an optimisation problem with G as the search space, so that each $g \in G$ is a search state. The objective function to be minimised is the lex ranking of A^g, which can be considered as a number. On finding a state with sufficiently small objective value (less than the lex ranking of A) we have solved the problem, establishing that A is dominated. This opens up the field of symmetry breaking to a wide range of metaheuristic algorithms.

2.2 Dominance as nonstationary optimisation

A practical problem is: how much effort should we devote to solving these dominance detection problems at each DFS node? If metaheuristic search fails to find a dominating state, this might be because there is no such state — but it could also be because the algorithm has not searched hard enough. Too little search might miss important symmetries, while too much will slow down DFS. This is a drawback of using an incomplete search algorithm.

Our answer is to devote very little effort indeed at each search node: the metaheuristic search is *resource-bounded* to ensure low computational overhead. For example if we apply local search then we might apply one local move per search tree node, while if we apply an evolutionary algorithm then we might breed one new offspring per node.

Metaheuristic search is now being used to solve an optimisation problem whose objective function 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 method *Symmetry Breaking by Nonstationary Optimisation* (SBNO).

Note that if dominance is not detected at a node then it might be detected 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. But as we backtrack we can check each search state C, and if $C^x \prec_{\text{lex}} C$ then we can backtrack again. This process continues until we backtrack past A, and apart from some wasted DFS effort the effect is the same as if we had detected the symmetry at A by finding x. To take advantage of this phenomenon, we freeze metaheuristic search on discovering dominance, and restart it as soon as the current partial assignment is undominated under the current element g.

SBNO has the following nice feature. 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 metaheuristic search. Thus we hope that SBNO will detect and break all *important* symmetries: those that make a significant difference to the size of the search tree and hence the execution time. This distinguishes it from partial symmetry breaking methods such as double-lex and STAB, which choose symmetries to break using other criteria.

In principle we could apply a complete algorithm to the nonstationary optimisation problem, but metaheuristic search often outperforms complete search given limited time so we believe that it is a more suitable approach. We could also apply metaheuristics to the auxiliary CSP for dominance detection defined in [Puget, 2005], but our optimisation problem has an even smaller memory requirement because we do not need to store a set of search states. However, this might be the best way of generalising SBNO to non-DFS search or dynamic value orderings.

3 Dominance detection by local search

To make SBNO more concrete we now show how to use local search for dominance detection, and apply it to a specific problem.

3.1 Neighbourhood structure

We have already defined the search space (G) and objective function (the lex ranking of A^g) in Section 2. 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 $g \circ H$. Thus all G elements are local search states, and some of them (H) are also local moves. To apply local search, from each state g we try to find a local move h such that the objective function is reduced: $A^{g \circ h} \prec_{\text{lex}} A^g$.[1] If a series of moves $(h_1, h_2, \ldots)$ reduces the lex ranking sufficiently then we will find $A^{g \circ h_1 \circ h_2 \circ \cdots} \prec_{\text{lex}} A$, establishing dominance of A.

[1] If an unassigned variable is encountered before establishing this property then the $\prec_{\text{lex}}$ test fails, but we could also reason on unassigned variables.

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 (denoted $\langle H \rangle = G$).

Proposition 2 *H is a generator set for G if and only if the search space induced by H is connected.*

Proof Suppose that H is a generator 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 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 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. □

Using a generator H can yield neighbourhoods of manageable size, because any group G has a generator of size $\log_2(|G|)$ or smaller. However, in Section 3.3 we use a non-generator H for heuristic reasons, and restore connectedness by allowing occasional moves from $G \setminus H$.

3.2 An application: BIBD design

We test SBNO on a problem with very large symmetry groups, which has been used to test several symmetry breaking methods. Balanced Incomplete Block Design (BIBD) generation is a standard combinatorial problem, originally used in the statistical design of experiments but since finding 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 specified by its parameters (v, b, r, k, λ). An example is shown in Figure 2.

For a BIBD to exist its parameters must satisfy the conditions $rv = bk$, $\lambda(v-1) = r(k-1)$ and $b \geq v$, 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). One source of intractability is the very large number of symmetries: given any solution, any two rows or columns may be exchanged to obtain another solution. The symmetry group is the direct product $S_v \times S_b$ so there are $v!\,b!$ symmetries. A survey of known results is given in [Colbourn and Dinitz, 1996] and some references and instances are given in CSPLib (problem 28).[2]

[2] http://www.csplib.org

$$\begin{bmatrix} 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2: A solution to the BIBD instance $(6, 10, 5, 3, 2)$

3.3 SBNO for BIBD design

The most direct CSP model for BIBD generation represents each matrix element by a binary variable. There are three types of constraint: (i) v b-ary constraints for the r ones per row, (ii) b v-ary constraints for the k ones per column, and (iii) $v(v-1)/2$ $2b$-ary constraints for the λ matching ones in each pair of rows. This is the constraint model we use. We implemented a simple BIBD solver: DFS with static variable ordering, ordered by rows then columns, and a static value ordering 1<0. No constraint propagation at all is used in this prototype: at each search node we simply check that no constraint has been violated. No constraint programmer would use such a feeble algorithm but it is useful for a proof-of-concept test of SBNO, and in future work we will use a constraint programming system to obtain better results.

The most obvious way to apply SBNO to this algorithm, and to other algorithms solving problems with row and column symmetry, is as follows. The local search states are the elements of the direct product $G = S_v \times S_b$. The local moves are the group generator consisting of adjacent and first-last row and column swaps. If a pair of such rows or columns can be found that are out of lex-order then swapping them is an improving move. There are v possible row swaps and b possible column swaps, and the time to compare rows and columns takes $O(b)$ or $O(v)$ time respectively, so the time to find an improving move is $O(vb)$: linear in the number of problem variables.

However, we obtained much better results by using a different neighbourhood structure. The local moves are the elements h of the group generator H consisting of *arbitrary* row or column swaps, but restricted to the subset of swaps involving the matrix entry corresponding to the binary variable v at which the last $\prec_{\text{lex}}$ test failed. This restriction keeps the time complexity of finding an improving move linear. It is also inspired by conflict-directed heuristics used in many successful local search algorithms — it focuses search effort on the source of failure.

A drawback of the restriction is that, by limiting moves to a subset of the generator, we might fail to find an appropriate g (recall Proposition 2). We compensate for this by randomising g at each local move with probability $1/vb$. The heuristic of choosing a random g might not be practicable for all problems, as it is not always possible to efficiently generate a random group element [Holt *et al.*, 2005]. But in this case it is easy: we simply build random permutations of the rows and columns. Even for groups in which it is hard to find a truly random element, a random factor in the search might be sufficient.

We make some further heuristic choices based on experimentation, which do not affect the correctness of symmetry breaking, only its effectiveness. We impose a TABU tenure of 10: an improving move is disallowed if it reverses a move made within the last 10 moves. The candidate improving moves are tested in random order, and the first successful one is used. If none is successful then we randomly exchange either *v*'s row or column with the next row or column in cyclic order. While searching for a pair of rows [columns] that are out of order, with probability 0.5 we ignore the current column [row] permutation: that is, we search for a pair of rows [columns] that are out of order under the *identity* column [row] permutation. This represents a compromise between searching for combined row/column symmetries and pure row or column symmetries. Finally, when randomising g in the above heuristic, with probability 0.5 we instead reset it to the group identity element.

We will refer to this SBNO implementation as SBNO-TABU. Its heuristics are tentative and will probably be changed in future work, but they have given reasonable results.

3.4 Symmetry breaking overhead

Runtime profiling shows that SBNO-TABU consumes approximately 90% of the total execution time (varying between BIBD instances), which might seem to contradict our claim that it is a low-overhead method. However, recall that our DFS algorithm performs no constraint propagation, so the time it spends at each tree node is very small. In fact the time complexity of our DFS at each search node is only $O(v)$ whereas that of SBNO-TABU is $O(vb)$. But constraint propagation algorithms are typically at least linear in the number of problem variables, which is vb in this application, so we expect the SBNO-TABU overhead to be almost negligible when it is applied to a real constraint solver. We will verify this in future work.

3.5 Performance variation

The use of local search for symmetry breaking makes the DFS runtime and number of solutions found nondeterministic. To examine how much variation SBNO-TABU causes, Figure 3 plots 10 runs of five different instances. The scatter plot shows that there is little variation in the number of search nodes needed for a complete tree search with symmetry breaking. There is more variation in the number of solutions found, but this reduces as the problem hardness increases. Harder problems are most interesting so we are justified in using a single run per instance in our experiments below.

3.6 Comparison with other methods

Different researchers use different BIBD instances to test their algorithms, and we shall compare SBNO-TABU with several reported methods using the same instances. All our results are obtained on a 2.8 GHz Pentium (R). First a comparison with [Frisch *et al.*, 2002] who use a constraint programming system with global constraints for enforcing lexi-

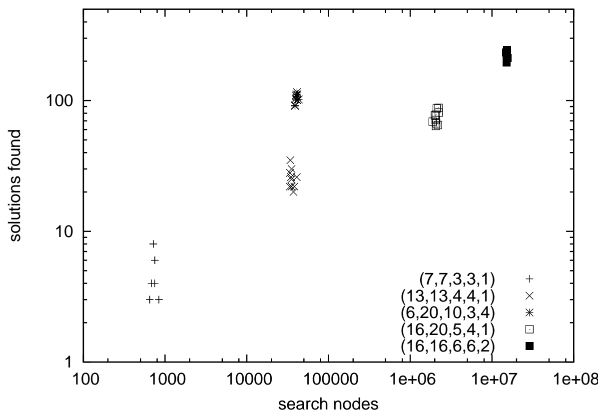

Figure 3: Variation between SBNO-TABU runs

v	b	r	k	λ	adj	all	dec	SBNO-TABU
6	50	25	3	10	1.7	1.8	11	2.1
6	60	30	3	12	4.6	4.9	45	7.3
10	90	27	3	6	111	120	742	158
9	108	36	3	9	8.4	7.6	73	416
15	70	14	3	2	6.2	8.4	21	0.02
12	88	22	3	4	249	317	1154	1781
9	120	40	3	10	8.0	7.2	82	1007
10	120	36	3	8	1316	1132	—	1722
13	104	24	3	4	398	448	1667	510

Table 1: Comparison with a global constraint method

cographical orderings.[3] Table 1 shows the time taken to find a single solution for the three methods in [Frisch *et al.*, 2002] called *GACLexLeq with adjacent pairs* ("adj"), *GACLexLeq with all pairs* ("all") and *Decomposition* ("dec").[4] Runs taking more than 1 hour are denoted by "—". The results of [Frisch *et al.*, 2002] were obtained on a 750 MHz Pentium III. SBNO-TABU is not dominated by any of the other methods on these instances, and is roughly comparable in execution time to the Decomposition method.

Next a comparison with the double-lex results of [Flener *et al.*, 2002]. Table 2 shows results computing all solutions: the number of distinct solutions, the number of solutions found and the time taken. The machine used by [Flener *et al.*, 2002] was unspecified, but given the 6-year gap is probably a few times slower than ours. It is clear that our execution times are much smaller than those of double-lex, while double-lex usually breaks more symmetries.

Next a comparison with the GAP-SBDD results of [Gent *et al.*, 2003] in Table 3. The table shows results computing all solutions: the number of distinct solutions, the number of solutions found and the time taken. The machine used by [Gent *et al.*, 2003] was a 2.6 GHz Pentium IV. SBNO-TABU beats GAP-SBDD in search time because of the overhead of interfacing Eclipse with GAP: for example in the last instance this consumed all but a fraction of 1% of the total execution time. But GAP-SBDD has the compensating advantage that it requires less work from the user to implement symmetry breaking (and of course it breaks all symmetries).

[3]A more up-to-date citation is [Frisch *et al.*, 2006] but its results are in a graph instead of a table.

[4]We omit the incorrectly written instance (6,70,35,3,10).

v	b	r	k	λ	distinct solns	double-lex solns	double-lex time	SBNO-TABU solns	SBNO-TABU time
7	7	3	3	1	1	1	1.1	5	0.0
6	10	5	3	2	1	1	1.0	4	0.0
7	14	6	3	2	4	24	11	66	0.03
9	12	4	3	1	1	8	28	12	0.02
8	14	7	4	3	4	92	171	81	0.1
6	20	10	3	4	4	21	10	111	0.1

Table 2: Comparison with double-lex

v	b	r	k	λ	distinct solns	GAP-SBDD time	SBNO-TABU solns	SBNO-TABU time
7	7	3	3	1	1	0.2	5	0.0
6	10	5	3	2	1	0.6	4	0.0
7	14	6	3	2	4	5.0	66	0.03
9	12	4	3	1	1	1.9	12	0.02
8	14	7	4	3	4	66	81	0.1
6	20	10	3	4	4	56	111	0.1
11	11	5	5	2	1	19	23	0.04
13	13	4	4	1	1	42	35	0.1
7	21	6	2	1	1	11	32	0.04
16	20	5	4	1	1	6078	151	4.5
13	26	6	3	1	2	59344	5893	43

Table 3: Comparison with GAP-SBDD

Finally Table 4 compares SBNO-TABU results with those for several algorithms on all but the hardest instances of [Puget, 2003], which were obtained on a Pentium III 833 MHz. The methods are STAB, and different implementations of double-lex and SBDD than those in [Flener *et al.*, 2002; Gent *et al.*, 2003]. Again we compute all solutions. These results are much faster than those cited above, which might be the result of superior constraint handling. Here at last our non-propagating algorithm is uncompetitive, but for such a trivial algorithm it does surprisingly well.

4 Dominance detection by memetic search

Local search is just one way of solving nonstationary optimisation problems, and there are other metaheuristic methods: for example [Stroud, 2001; Trojanowski and Michalewicz, 1999] use evolutionary computation. We now design a memetic algorithm for SBNO: a hybrid of genetic and local search. We use a steady-state genetic algorithm with three separate populations of group elements, each population having 1000 organisms and each organism using two chromosomes to represent a row and a column permutation. In the first population the row permutation is fixed to be the identity permutation, in the second population the column permutation is the identity, and in the third population neither is

v	b	r	k	λ	solns	double-lex solns	double-lex time	STAB solns	STAB time	SBDD time	SBNO-TABU solns	SBNO-TABU time
6	10	5	3	2	1	1	0	1	0	0.01	4	0.0
7	7	3	3	1	1	1	0	1	0.01	0	5	0.0
6	20	10	3	4	4	21	0.02	4	0.01	0.3	111	0.1
9	12	4	3	1	1	2	0.01	1	0.02	0.01	12	0.02
7	14	6	3	2	4	12	0.01	7	0.02	0.1	66	0.03
8	14	7	4	3	4	92	0.04	6	0.03	0.5	81	0.1
6	30	15	3	6	6	134	0.1	7	0.04	2	1112	3.3
11	11	5	5	2	1	2	0.01	1	0.05	0.06	23	0.04
10	15	6	4	2	3	38	0.05	4	0.05	0.8	81	0.3
7	21	9	3	3	10	220	0.07	24	0.05	2	845	0.7
13	13	4	4	1	1	2	0.03	1	0.07	0.03	35	0.1
6	40	20	3	8	13	494	0.7	15	0.1	11	5374	57
9	18	8	4	3	11	2600	2	41	0.1	14	849	5.4
16	20	5	4	1	1	12	0.2	1	0.1	2	151	4.5
7	28	12	3	4	35	3209	1	116	0.2	19	9842	15
6	50	25	3	10	19	1366	3	26	0.2	45	18694	545
9	24	8	3	2	36	5987	1	344	0.5	28	7513	14
16	16	6	6	2	3	46	0.6	3	0.5	3	333	16
15	21	7	5	2	0	0	18	0	0.7	10	0	1416
13	26	6	3	1	2	12800	14	21	0.7	11	5893	43
7	35	15	3	5	109	33304	15	542	0.8	155	96088	317
15	15	7	7	3	5	118	1	19	1	13	674	43
21	21	5	5	1	1	12	0.5	1	2	0.5	1191	36

Table 4: Comparison with double-lex, STAB and SBDD

fixed. We use three populations in order to treat row, column and row/column symmetries separately. In principle we could use just the third population, because row/column symmetries subsume row and column symmetries, but better results were obtained by treating them separately.

Instead of making a local move at each DFS node, as in SBNO-TABU, the new algorithm generates a single offspring g_o^p from two randomly chosen parents in a randomly chosen population p. We cannot use group-theoretic methods for generating new organisms as we did in SBNO-TABU, so we must choose genetic operators suited to the symmetry group. Here we are dealing with permutations, so we apply *cycle crossover* [Oliver *et al.*, 1987] separately to the row and column chromosomes, followed by a single *exchange mutation* [Banzhaf, 1990] in each chromosome. Cycle crossover and exchange mutation are standard operators used to apply genetic algorithms to permutation problems. The new offspring g_o^p is then compared to a random population member g_r^p, and if it is currently fitter ($A^{g_o^p} \prec_{\text{lex}} A^{g_r^p}$) then g_o^p replaces g_r^p in population p. The organism used for dominance detection at each node is the fitter of the two most recent g_o^p and g_r^p chromosomes. To avoid stagnation, if the parents are identical then we randomise one of them before applying the genetic operators.

Genetic algorithms can often be enhanced by applying local search to chromosomes, giving a *memetic algorithm* [Moscato, 1989]. We use a form of local search specifically designed for nonstationary optimisation, similar to that used in SBNO-TABU. Before using a chromosome for dominance testing we improve it as follows. First select the row r and column c corresponding to the variable at which the dominance test using that chromosome failed, as in SBNO-TABU. (If the chromosome is a new offspring then select r and c from one of its parents, chosen randomly.) Check each possible row and column swap with r and c respectively, in random order, swapping any that are out of order. The improved chromosome is placed back into the population, regardless of whether dominance was shown. In this way the chromosomes are continually modified to keep up with DFS. This improvement can be done in linear time, as in SBNO-TABU. We will refer to this SBNO implementation as SBNO-MEME.

v	b	r	k	λ	SBNO-TABU	SBNO-MEME
6	50	25	3	10	2.1	1.0
6	60	30	3	12	7.3	1.9
10	90	27	3	6	158	16
9	108	36	3	9	416	16
15	70	14	3	2	0.02	0.02
12	88	22	3	4	1781	129
9	120	40	3	10	1007	29
10	120	36	3	8	1722	118
13	104	24	3	4	510	25

Table 5: TABU vs memetic search (1)

Table 5 compares SBNO-MEME with SBNO-TABU on the instances from Table 1: recall that for these instances we halt on finding the first solution, and report the CPU time. SBNO-MEME is clearly superior to SBNO-TABU, and is sometimes more than an order of magnitude faster. Table 6 compares the two algorithms on the instances from Table 4: recall that for these instances we compute all solutions. The harder the problem the better SBNO-MEME performs with respect to SBNO-TABU, both in terms of broken symmetries and execution time, whowing that the memetic approach is more scalable than the local search approach.

5 Conclusion

We described SBNO, a framework for applying metaheuristic search to the problem of symmetry breaking in backtrack search. SBNO has a small memory requirement and is suitable for large problems with many symmetries: for example BIBD instance (9,120,40,3,10) in Figure 1 has more than 10^{204} symmetries. It also has low computational overhead yet in experiments breaks most symmetries, and a prototype without constraint propagation has already given promising results. In future work we will add propagation to obtain what we hope will be a powerful BIBD algorithm. We will also apply SBNO to other highly symmetrical problems such as the Social Golfer Problem, and try to extend it to dynamic value ordering heuristics.

There are few connections between metaheuristics and symmetry. [Petcu and Faltings, 2003] used a form of symmetry (*interchangeability*) to guide a distributed local search algorithm. Symmetry breaking is often applied to genetic algorithms, but here it has a different meaning, and refers to clustering of the population within a symmetric region of the search space [Pelikan and Goldberg, 2000]. An alternative is to design more complex genetic operators and problem mod-

v	b	r	k	λ	SBNO-TABU solns	SBNO-TABU time	SBNO-MEME solns	SBNO-MEME time
6	10	5	3	2	4	0.0	59	0.02
7	7	3	3	1	5	0.0	18	0.008
6	20	10	3	4	111	0.1	396	0.2
9	12	4	3	1	12	0.02	66	0.06
7	14	6	3	2	66	0.03	188	0.08
8	14	7	4	3	81	0.1	101	0.2
6	30	15	3	6	1112	3.3	291	1.0
11	11	5	5	2	23	0.04	60	0.1
10	15	6	4	2	81	0.03	23	0.6
7	21	9	3	3	845	0.7	660	0.6
13	13	4	4	1	35	0.1	111	0.3
6	40	20	3	8	5374	57	256	7.0
9	18	8	4	3	849	5.4	125	3.4
16	20	5	4	1	151	4.5	36	3.0
7	28	12	3	4	9842	15	1521	4.6
6	50	25	3	10	18694	545	510	59
9	24	8	3	2	7513	14	1565	8.5
16	16	6	6	2	333	16	34	9.7
15	21	7	5	2	0	1416	0	2419
13	26	6	3	1	5893	43	193	11
7	35	15	3	5	96088	317	5174	93
15	15	7	7	3	674	43	83	31
21	21	5	5	1	1191	36	113	20

Table 6: TABU vs memetic search (2)

els [Galinier and Hao, 1999], akin to the reformulation approach to symmetry breaking mentioned in Section 1. A negative result of [Prestwich, 2003; Prestwich and Roli, 2005] is that some forms of symmetry breaking have a detrimental effect on local search performance. As far as we know, SBNO is the first use of metaheuristic search to break symmetry.

With respect to the area of hybrid search algorithms, SBNO-TABU is an example of an integration of local and tree search — see [Focacci *et al.*, 2003] for a survey of such hybrids. There is often a trade-off in tree search between (i) performing expensive reasoning at each node to potentially eliminate large subtrees, and (ii) processing nodes cheaply to reduce overheads. When the reasoning is used to solve another NP-hard problem, incomplete reasoning can be applied in the hope of finding something useful in a short time. For example [Sellmann and Harvey, 2002] use local search within backtrack search to generate tight redundant constraints, an approach they call *heuristic propagation*. SBNO-TABU is another example of this type of integration. We do not know of any similar use of evolutionary methods other than SBNO-MEME.

References

[Backofen and Will, 1999] 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.

[Banzhaf, 1990] W. Banzhaf. The “molecular” traveling salesman. *Biological Cybernetics*, 64:7–14, 1990.

[Brown *et al.*, 1988] 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.

[Colbourn and Dinitz, 1996] C. J. Colbourn and J. H. Dinitz, editors. *The CRC Handbook of Combinatorial Designs*. CRC Press, 1996.

[Crawford *et al.*, 1996] 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.

[Fahle *et al.*, 2001] 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.

[Flener *et al.*, 2002] 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.

[Focacci and Milano, 2001] 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.

[Focacci *et al.*, 2003] F. Focacci, F. Laburthe, and A. Lodi. *Handbook of Metaheuristics*, chapter Local Search and Constraint Programming, pages 369–403. Kluwer Academic Publishers, 2003.

[Frisch *et al.*, 2002] A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *8th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 93–108. Springer, 2002.

[Frisch *et al.*, 2006] A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Propagation algorithms for lexicographic ordering constraints. *Artificial Intelligence*, 170(10):803–834, 2006.

[Galinier and Hao, 1999] P. Galinier and J. K. Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4):379–397, 1999.

[Gent and Smith, 2000] I. P. Gent and B. M. Smith. Symmetry breaking in constraint programming. In *14th European Conference on Artificial Intelligence*, pages 599–603, 2000.

[Gent *et al.*, 2002] 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.

[Gent *et al.*, 2003] I. P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic sbdd using computational group theory. In *9th International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2003.

[Gent *et al.*, 2006] I. P. Gent, K. E. Petrie, and J.-F. Puget. *Handbook of Constraint Programming*, chapter Symmetry in Constraint Programming, pages 329–376. Elsevier, 2006.

[Holt *et al.*, 2005] D. F. Holt, B. Eick, and E. A. O'Brien. *Handbook of Computational Group Theory*. Chapman & Hall/CRC, 2005.

[Moscato, 1989] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Technical Report 826, Caltech Concurrent Computation Program, 1989.

[Oliver *et al.*, 1987] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the travelling salesman problem. In *2nd International Conference On Genetic Algorithms*, pages 224–230. Lawrence Erlbaum Associates, 1987.

[Pelikan and Goldberg, 2000] M. Pelikan and D. E. Goldberg. Genetic algorithms, clustering, and the breaking of symmetry. In *6th International Conference on Parallel Problem Solving from Nature*, 2000.

[Petcu and Faltings, 2003] A. Petcu and B. Faltings. Applying interchangeability techniques to the distributed breakout algorithm. In *18th International Joint Conference on Artificial Intelligence*, pages 1381–1382, 2003.

[Prestwich and Roli, 2005] S. D. Prestwich and A. Roli. Symmetry breaking and local search spaces. In *Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 3524 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2005.

[Prestwich, 2003] S. D. Prestwich. Negative effects of modeling techniques on search performance. *Annals of Operations Research*, 118:137–150, 2003.

[Puget, 1993] 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.

[Puget, 2003] 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.

[Puget, 2005] J.-F. Puget. Symmetry breaking revisited. *Constraints*, 10(1):23–46, 2005.

[Sellmann and Harvey, 2002] M. Sellmann and W. Harvey. Heuristic constraint propagation. In *4th International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 191–204, 2002. Le Croisic, France.

[Stroud, 2001] P. D. Stroud. Kalman-extended genetic algorithm for search in nonstationary environments with noisy fitness functions. *IEEE Transactions on Evolutionary Computation*, 5(1):66–77, 2001.

[Trojanowski and Michalewicz, 1999] K. Trojanowski and Z. Michalewicz. Evolutionary approach to non-stationary optimisation tasks. In *Foundations of Intelligent Systems*, volume 1609 of *Lecture Notes in Computer Science*, pages 538–546. Springer, 1999.