Symmetry Breaking by Nonstationary Optimisation*

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. We describe a new partial symmetry breaking method that can be used to break arbitrary variable/value symmetries in combination with depth first search, static value ordering and dynamic variable ordering. The main novelty of the method is a new dominance detection technique based on local search in the symmetry group. It has very low time and memory requirements, yet in preliminary experiments on BIBD design it breaks most symmetries and is competitive with several other methods.

1 Introduction

A finite-domain constraint satisfaction problem (CSP) has variables $v_1 \ldots v_n$ each with a domain dom$(v_i) = \{a_1, \ldots, a_m\}$ of values, and constraints prescribing prohibited combinations of values (alternatively, constraints may prescribe *allowed* combinations of assignments). The problem is to find an assignment of values to all variables such that no constraint is violated.

Many CSPs contain *symmetries*: transformations of solutions that yield other solutions. For example the N-queens problem[1] has 8 (each solution may be rotated through 90 degrees and reflected to obtain other solutions) 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 more than once. By eliminating the symmetry (*symmetry breaking*) we may speed up the search significantly. Symmetries form groups, and there are close connections between symmetry breaking and computational group theory. Several distinct methods have been reported for symmetry breaking in CSPs and a summary is provided by [12].

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. An alternative approach, and probably the most commonly used, 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 [14], which was developed into the *lex-leader* method

* 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.

[1] Place N queens onto an N×N chessboard so that no two attack each other.

by [3]. But in practice too many constraints might be needed, as there might be an exponential number of 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* [5]) does not break all combined symmetries but is tractable. Another drawback with this method is that it does 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 [1] and further elucidated by [13]. 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 [10], 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 much more. A related method to SBDS called *Symmetry Breaking Using Stabilizers* (STAB) [15] 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 [4, 6] and combined with GAP to give GAP-SBDD [10]. 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 and can handle huge symmetries. 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 these can be ignored if the value ordering heuristic in the search algorithm is static, making SBDD a practicable method. Symmetry between these states 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 [16] that the dominance tests can be combined into a single auxiliary CSP then solved by standard constraint programming methods, with fast results.

In this paper we describe and test a new approach to partial symmetry breaking. It is related to SBDD but uses a different dominance detection technique, expressed as a nonstationary optimisation problem and solved by local search. It has lower time and memory requirements than SBDD 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, and Section 4 concludes the paper.

2 The SBNO method

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 branch-and-bound). Suppose also that the problem has a 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 A new dominance test

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 ordering* 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. We now show this informally, but a future version of this paper will contain a formal proof. If $A^g \prec_{lex} A$ then A^g and A must be of the form

$$\begin{aligned} A &= (v_1, a_1), \ldots, (v_{k-1}, a_{k-1}), (v_k, a_k), \ldots \\ A^g &= (v'_1, a_1), \ldots, (v'_{k-1}, a_{k-1}), (v'_k, a'_k), \ldots \end{aligned}$$

such that $a'_k < a_k$ under the relevant static value ordering on $\text{dom}(v_k) = \text{dom}(v'_k)$, where (v_i, a_i) denotes the assignment $v_i = a_i$. 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.

2.2 Dominance as optimisation

We now express the dominance test as an optimisation problem suitable for solution by local search. The problem at each search node A is to find a $g \in G$ such that $A^g \prec_{\text{lex}} A$. To solve this problem we can treat G as a local search space with each $g \in G$ being a 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. It is easy to show that if H is a generator set for G (denoted $\langle H \rangle = G$) then there exists a series of such local moves from any state to any other, which is usually a necessary condition for local search to solve a problem. Using a generator also has the advantage of yielding neighbourhoods of manageable

size, because any group G has a generator of size $\log_2(|G|)$ or smaller. However, we can also use a non-generator H and allow some random moves from $G \setminus H$ (see Section 3); the choice is a heuristic one. The objective function value of any state g is the lex-ranking of A^g (which can be considered as a number). To apply local search, from each state g we try to find a local move h such that the objective function is reduced: $A^{g \circ h} \prec_{\text{lex}} A^g$.[2] If a series of moves $(h_1, h_2, \ldots)$ reduces the lex-ranking sufficiently then we hope to find $A^{g \circ h_1 \circ h_2 \circ \cdots} \prec_{\text{lex}} A$, establishing dominance.

Of course, the auxiliary CSP for dominance detection defined in [16] could also be tackled by local search, but our optimisation problem has an even smaller memory requirement as we need store only the current partial assignment A and current group element g (plus whatever data structures are needed by the underlying constraint solver).

2.3 Dominance as nonstationary optimisation

How much effort should we devote to solving these dominance detection problems? If local 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 approach such as local search.

Our answer is to devote very little effort indeed at each search node: in fact we apply only *one* local move $h \in H$ per search tree node. Local 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). If a dominance is not detected by local search then it might detect it after a few extra local moves and search tree nodes, and therefore a small amount of wasted DFS. DFS can then backtrack, possibly jumping many levels in the search tree.

This scheme has the following nice feature. A symmetry that would only save a small amount of DFS effort is unlikely to be detected by SBNO, because DFS might backtrack past A before an appropriate g can be discovered. In contrast, one that would save a great deal of DFS effort has a great deal of time in which to be detected by local 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 for space reasons.

3 Case study

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.

[2] 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.

3.1 BIBD design

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 1.

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 [2] and some references and instances are given in CSPLib (problem 28).[3]

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

Fig. 1. A solution to the BIBD instance $(6, 10, 5, 3, 2)$

3.2 Constraint model and algorithm

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 trying 1 then 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 weak algorithm, and we believe that propagation is important when searching for BIBDs. But this algorithm is useful as a proof-of-concept for SDBO, and in future work we will use a constraint programming system to obtain better results.

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

3.3 SBNO implementation

We apply SBNO to this algorithm as follows. The local search states are the elements of the direct product $G = S_v \times S_b$. 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 makes the neighbourhood sizes either v or b depending on whether we swap a row or a column. The time to compare rows and columns takes $O(b)$ or $O(v)$ time respectively, so the time to find an improving move if one exists is $O(vb)$: linear in the number of problem variables. This heuristic is also inspired by conflict-directed heuristics used in many successful local search algorithms — it focuses search effort on the source of failure, and in experiments gave better performance than a more obvious use of a generator. A drawback is that, by restricting moves to a subset of the generator, we might fail to find an appropriate g. We compensate for this by randomising g at each local move with probability $1/vb$. From each search state the possible local moves h are tested in random order until finding one that satisfies $A^{g \circ h} \prec_{\text{lex}} A^g$. If no such move is possible then a random move could be made, but we use a move that gave better results: randomly exchange either *v*'s row or column with the next one. (We have no justification for this heuristic, and no doubt a better one can be found.) A TABU tenure of 10 is imposed: no improving move is allowed if it reverses a move made within the last 10 moves. These heuristics do not affect the correctness of symmetry breaking, only its efficiency.

3.4 Symmetry breaking overhead

Runtime profiling shows that SBNO consumes over 90% of the total execution time, which might seem to contradict our claim that it is a low-overhead method. However, recall that our algorithm currently performs no constraint propagation, so the time spent at each node is very small. In fact the time complexity of our constraint checking algorithm at each search node is only $O(v)$ whereas that of SBNO 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 overhead to be negligible when applied to a real constraint solver. We will test 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 test how much variation SBNO causes, Figure 2 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.

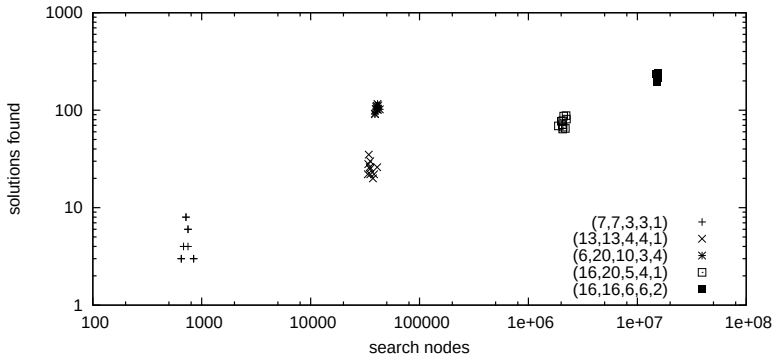

Fig. 2. Variation between SBNO runs

3.6 Comparison with other methods

Different researchers use different BIBD instances to test their algorithms, and we shall compare SBNO with several reported methods using the same instances. All our results are obtained on a 2.8 GHz Pentium (R). First a comparison with [7][4] who use a constraint programming system with global constraints for enforcing lexicographical orderings. Table 1 shows the time taken to find a single solution, and denotes the three methods in [7] by *GACLexLeq (adjacent pairs)*, *GACLexLeq (all pairs)* and *Decomposition*.[5] Runs taking more than 1 hour are denoted by "—". The results of [7] were obtained on a 750 MHz Pentium III. SBNO is not dominated by any of the other methods on these instances, and is roughly comparable in execution time to the Decomposition method.

v	b	r	k	λ	GACLexLeq adj pairs	GACLexLeq all pairs	Decomp	SBNO
6	50	25	3	10	1.7	1.8	11	1.6
6	60	30	3	12	4.6	4.9	45	6.0
10	90	27	3	6	111	120	742	104
9	108	36	3	9	8.4	7.6	73	248
15	70	14	3	2	6.2	8.4	21	0.02
12	88	22	3	4	249	317	1154	1333
9	120	40	3	10	8.0	7.2	82	648
10	120	36	3	8	1316	1132	—	1227
13	104	24	3	4	398	448	1667	328

Table 1. Comparison with a global constraint method [7]

[4] A more up-to-date citation is [9] but the results are in a graph instead of a table.
[5] We omit instance (6,70,35,3,10) whose parameters are incorrect.

Next a comparison with the double-lex results of [5] and the GAP-SBDD results of [11] in Table 2. The table shows results computing all (non-symmetric) solutions: the number of distinct solutions, the number of solutions found and the time taken. The machine used by [11] was a 2.6 GHz Pentium IV. That used by [5] 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 breaks more symmetries. SBNO also beats GAP-SBDD in search time because of the overhead of interfacing Eclipse with GAP: for example in the last instance it 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).

					distinct	double-lex		GAP-SBDD	SBNO	
v	b	r	k	λ	solns	solns	time	time	solns	time
7	7	3	3	1	1	1	1.1	0.2	6	0.004
6	10	5	3	2	1	1	1.0	0.6	4	0.008
7	14	6	3	2	4	24	11	5.0	55	0.05
9	12	4	3	1	1	8	28	1.9	10	0.02
8	14	7	4	3	4	92	171	66	162	0.3
6	20	10	3	4	4	21	10	56	107	0.2
11	11	5	5	2	1			19	12	0.08
13	13	4	4	1	1			42	25	0.2
7	21	6	2	1	1			11	32	0.05
16	20	5	4	1	1			6078	67	18
13	26	6	3	1	2			59344	5694	186

Table 2. Comparison with double-lex [5] and GAP-SBDD [11]

Finally Table 3 compares SBNO results with those of [15], which were obtained on a Pentium III 833 MHz. Again we compute all (non-symmetric) 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 Conclusion

SBNO is a new partial symmetry breaking method for Constraint Programming, related to the SBDD method but using a different dominance detection technique, and solving it by resource-bounded local search instead of by constraint programming or computational group theory. It has a smaller memory requirement than SBDD and is therefore suitable for large problems with many symmetries: for example BIBD instance (9,120,40,3,6) (solved in Figure 1) has more than 10^{200} symmetries. It also has a very low time complexity yet in experiments breaks most symmetries. A weak prototype without constraint propagation has already given promising results, and in future

					distinct	double-lex		STAB		SBDD	SBNO	
v	b	r	k	λ	solns	solns	time	solns	time	time	solns	time
6	10	5	3	2	1	1	0	1	0	0.01	4	0.008
7	7	3	3	1	1	1	0	1	0.01	0	6	0.004
6	20	10	3	4	4	21	0.02	4	0.01	0.3	107	0.2
9	12	4	3	1	1	2	0.01	1	0.02	0.01	10	0.02
7	14	6	3	2	4	12	0.01	7	0.02	0.1	55	0.05
8	14	7	4	3	4	92	0.04	6	0.03	0.5	162	0.3
6	30	15	3	6	6	134	0.1	7	0.04	2	653	4.0
11	11	5	5	2	1	2	0.01	1	0.05	0.06	12	0.08
10	15	6	4	2	3	38	0.05	4	0.05	0.8	137	1.3
7	21	9	3	3	10	220	0.07	24	0.05	2	905	1
13	13	4	4	1	1	2	0.03	1	0.07	0.03	25	0.2
6	40	20	3	8	13	494	0.7	15	0.1	11	3521	54
9	18	8	4	3	11	2600	2	41	0.1	14	3141	27
16	20	5	4	1	1	12	0.2	1	0.1	2	67	18
7	28	12	3	4	35	3209	1	116	0.2	19	9624	32
6	50	25	3	10	19	1366	3	26	0.2	45	10683	472
9	24	8	3	2	36	5987	1	344	0.5	28	16115	82
16	16	6	6	2	3	46	0.6	3	0.5	3	220	81
15	21	7	5	2	0	0	18	0	0.7	10	0	12843
13	26	6	3	1	2	12800	14	21	0.7	11	5694	186
7	35	15	3	5	109	33304	15	542	0.8	155	87769	578
15	15	7	7	3	5	118	1	19	1	13	802	183
21	21	5	5	1	1	12	0.5	1	2	0.5	153	160
25	30	6	5	1	1	864	78	1	2	156	$>$10 hours	
10	18	9	5	4	21	8031	25	302	2	131	6781	280
7	42	18	3	6	418	250878	136	2334	3	1258	729554	8601
22	22	7	7	2	0	0	35	0	8	10	$>$10 hours	
7	49	21	3	7	1508	1459585	966	8821	14	8062	$>$10 hours	
8	28	14	4	6	2310	2058523	1282	17890	17	11028	1498042	17538
19	19	9	9	4	6	6520	1511	71	23	6411	$>$10 hours	
10	30	9	3	2	960	724662	178	24563	26	2915	1242107	15943

Table 3. Comparison with various methods from [15]

work we will add propagation to obtain what we hope will be a very competitive BIBD algorithm. We will also apply it to other highly symmetrical problems such as the Social Golfer Problem, try to extend it to dynamic value ordering heuristics, and experiment with other nonstationary optimisation methods such as genetic algorithms.

References

1. R. Backofen, S. Will. Excluding Symmetries in Constraint-Based Search. *5th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 1713, Springer, 1999, pp. 73–87.
2. C. J. Colbourn, J. H. Dinitz (eds.). The CRC Handbook of Combinatorial Designs. CRC Press, 1996.
3. J. Crawford, M. L. Ginsberg, E. Luks, A. Roy. Symmetry-Breaking Predicates for Search Problems. *Principles of Knowledge Representation and Reasoning*, Morgan Kaufmann, 1996, pp. 148–159.
4. T. Fahle, S. Schamberger, M. Sellmann. Symmetry Breaking. *7th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2239, Springer, 2001, pp. 93–107.
5. P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh. Breaking Row and Column Symmetries in Matrix Models. *8th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2470, Springer, 2002, pp. 462–476.
6. F. Focacci, M. Milano. Global Cut Framework for Removing Symmetries. *7th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2239, Springer, 2001, pp. 77–92.
7. A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh. Global Constraints for Lexicographic Orderings. *8th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2470, Springer, 2002, pp. 93–108.
8. A. M. Frisch, C. Jefferson, I. Miguel. Symmetry Breaking as a Prelude to Implied Constraints: a Constraint Modelling Pattern. *16th European Conference on Artificial Intelligence*, IOS Press, 2004, pp. 171–175.
9. A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh. Propagation Algorithms for Lexicographic Ordering Constraints. *Artificial Intelligence* 170(10):803–834, Elsevier, 2006.
10. I. P. Gent, W. Harvey, T. Kelsey. Groups and Constraints: Symmetry Breaking During Search, 2002. *8th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2470, Springer, 2002, pp. 415–430.
11. I. P. Gent, W. Harvey, T. Kelsey, S. Linton. Generic SBDD Using Computational Group Theory. *9th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2833, Springer, 2003, pp. 333–347.
12. I. P. Gent, K. E. Petrie, J.-F. Puget. Symmetry in Constraint Programming. F. Rossi, P. van Beek, T. Walsh (eds.), Handbook of Constraint Programming, Elsevier, 2006, pp. 329–376.
13. I. P. Gent, B. M. Smith. Symmetry Breaking in Constraint Programming. *14th European Conference on Artificial Intelligence*, 2000, pp. 599–603.
14. J.-F. Puget. On the Satisfiability of Symmetrical Constraint Satisfaction Problems. *Methodologies for Intelligent Systems, Lecture Notes in Artificial Intelligence* vol. 689, Springer, 1993, pp. 350–361.
15. J.-F. Puget. Symmetry Breaking Using Stabilizers. *9th International Conference on Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* vol. 2833, Springer, 2003, pp. 585–599.
16. J.-F. Puget. Symmetry Breaking Revisited. *Constraints* 10(1):23–46, 2005.