

Generic SBDD using GAP and ECLiPSe

Content Areas: constraint programming, search, constraint satisfaction

Abstract

We introduce a generic implementation of symmetry breaking by dominance detection. The implementation uses ECLⁱPS^e to model and constrain a constraint satisfaction problem. Binary backtrack search with propagation is also performed in ECLⁱPS^e, together with a check for a dominating group element at nodes in the search tree. These checks are performed in the GAP computational group theory system, which runs as a sub-process of ECLⁱPS^e. The results of the dominance checks are used to restrict search to parts of the tree which are not symmetrically equivalent to a previously searched section. A constraint logic programming practitioner using the interface does not need to implement a dominance check for the problem; given a small number (typically 2–6) of generating symmetries, the GAP sub-process constructs a symmetry group, performs search for dominating elements when required, and provides the ECLⁱPS^e super-process with the information needed to restrict search. Our implementation is deterministic: all non-dominated nodes are visited during search, and each dominance check either succeeds or fails in finite time, so that only non-symmetric solutions are returned. The implementation easily handles problems involving 10^{23} symmetries, with only four permutations needed to direct the dominance checks during search.

1 Introduction

Dealing with symmetries in constraint satisfaction problems has become a popular topic for research in recent years. Main areas of recent study include

1. the modification of backtracking search procedures so that they only return unique solutions, and
2. the use of computational group theory (henceforth CGT) methods to effectively utilise the algebraic structure of symmetries.

The modified search techniques currently broadly fall into two main categories. The first involves adding constraints whenever backtracking occurs, so that symmetric versions

of the failed part of the search tree will not be considered in future [Backofen and Will, 1999; Gent and Smith, 2000]; these techniques are collectively known as SBDS (Symmetry Breaking During Search). The second category involves performing checks at nodes in the search tree to see whether they are dominated by the symmetric equivalent of some state already considered [Fahle *et al.*, 2001; Focacci and Milano, 2001]; we will collectively refer to these techniques as SBDD (Symmetry Breaking by Dominance Detection). The SBDD approach as implemented to date (with one exception) involves the coding of a dominance check. This dominance check is bespoke, in the sense that it uses properties of the structure of the problem under consideration to detect dominating search nodes. Note that SBDS and SBDD are closely related; the main difference is when and how the symmetry-breaking conditions are enforced. A comparison of SBDS and SBDD, together with a dominance check for a highly symmetric problem, is given in [Harvey, 2001].

The use of CGT methods is motivated by the fact that the symmetries of a problem form a group: a tuple $\langle S, \circ \rangle$ where S is a set and \circ is a closed binary operation over S such that:

1. \circ acts associatively: $(a \circ b) \circ c = a \circ (b \circ c)$ for every $a, b, c \in S$;
2. there is a neutral element, e , such that $a \circ e = e \circ a = a$ for every $a \in S$;
3. each element has an inverse: $a \circ a^{-1} = a^{-1} \circ a = e$.

Modern CGT systems such as GAP [GAP, 2000] are designed to exploit this algebraic structure, and are very efficient: they allow rapid calculations to be done on large groups without the need to iterate over or explicitly represent more than a tiny fraction of the group elements. As well as offering a clear benefit in both time and space, using a CGT approach can make the expression of the symmetries by the programmer much easier: typically only a handful of example symmetries are required to generate the full symmetry group, even for very large groups; we provide examples of this in Section 5.

An interface between the ECLⁱPS^e [Wallace *et al.*, 1997] constraint logic programming system and the GAP CGT system was presented in [Gent *et al.*, 2002]. This interface provided an implementation of SBDS (also in [Gent *et al.*, 2002]) which is capable of handling many more symmetries

than previous implementations and provides improved search performance for symmetric constraint satisfaction problems. In this paper we present an implementation of SBDD in GAP-ECLⁱPS^e. The implementation has the novel feature that it is generic – the constraint logic programmer does not need to implement a bespoke dominance checker. The dominance checks either succeed (resulting in a backtrack), or fail supplying a set of assignments, any of which, if added to the current partial assignment of values to variables, would result in the dominance check succeeding – this set is used to reduce the size of domains, thus improving search efficiency, as described in [Fahle *et al.*, 2001; Focacci and Milano, 2001].

Although not set in constraint satisfaction terms, *Backtrack Searching in the Presence of Symmetry* [Brown *et al.*, 1988; 1996] contains many SBDD ideas, and has influenced our approach.

We describe the GAP-ECLⁱPS^e interface in more detail in Section 2. We outline SBDD in Section 3, and provide a detailed exposition of our generic implementation in Section 4. Section 5 consisted of some experimental results. We discuss our results and highlight future avenues of research in Section 6.

2 GAP-ECLⁱPS^e

GAP and ECLⁱPS^e are large, mature and widely-used systems for computational algebra and constraint logic programming respectively. Both systems incorporate libraries and packages for computation in specific areas, together with tools and resources for software development. For our purposes, the GAP permutation group libraries and the ECLⁱPS^e finite domain libraries are of interest.

2.1 Finite Domains in ECLⁱPS^e

ECLⁱPS^e supports standard finite domain constraints, and, during search, applies constraint propagation techniques developed by the AI community [Hentenryck, 1989]. The standard search method is depth-first, and assigns values to variables at choice points. A complete assignment that satisfies the constraints is a solution. If no symmetry breaking constraints have been posted before search, then ECLⁱPS^e will search for and return all solutions, irrespective of any symmetries involved. For example, consider the illustrative problem of finding a list $[A, B, C, D, E, F, G]$ of distinct numbers in $1 \dots 50$ such that $A^3 + B^3 + C^3 + D^3 = E^3 + F^3 + G^3$. A solution is $[1, 2, 3, 39, 18, 22, 35]$, but this is symmetrically equivalent to those lists with 1, 2, 3, and 39 permuted in any way, and/or 18, 22 and 35 permuted. Our aim is to restrict search to choices which do not lead to one of these other lists. Restricting search to avoid symmetrically equivalent solutions has a larger benefit than avoiding duplicate solutions. Symmetry breaking methods such as SBDD or SBDS also avoid duplicating search from failed nodes. This can have a dramatic effect on time taken, as the same failed search state can reoccur in many symmetric guises, only one of which need be explored.

2.2 Permutation Groups in GAP

A permutation is a rearrangement of elements in an ordered list S into a one-to-one correspondence with S itself. The number of permutations on a set of n elements is $n!$. Two permutations can be composed by composing their respective correspondences with S ; since all such correspondences are bijective, the composition has an inverse. If we take the identity mapping on S as the required neutral element, composition of permutations forms a group. GAP contains libraries for defining, composing, and manipulating individual permutations, and for computation within permutation groups.

Taking S to be a position indexing of the list $[A, B, C, D, E, F, G]$ described in the previous section, we have $S = [1, 2, 3, 4, 5, 6, 7]$. Permutations in GAP are entered and displayed in cycle notation, such as $(1, 2, 3)(5, 7)$ which denotes the correspondence which has as image the list $[3, 1, 2, 4, 7, 6, 5]$. (A human might describe this permutation as ‘1 goes to 2, 2 goes to 3, 3 goes to 1, 5 and 7 are swapped, 4 and 6 are unchanged.’.)

Given that there are $7!$ possible permutations on S , which are the permutations which preserve solutions to the $A^3 + B^3 + C^3 + D^3 = E^3 + F^3 + G^3$ problem? This is straightforward in GAP – we simply supply some example permutations, and let GAP compute the resulting permutation group. Swapping A and D and cycling A, B, C and D are solution preserving symmetries, represented by the permutations $(1, 4)$ and $(1, 2, 3, 4)$ respectively. Similarly $(5, 7)$ and $(5, 6, 7)$ denote the swapping of E and G and a cycling of E, F and G . Supplying these 4 permutations to GAP results in permutation group containing 144 elements ($3!$ permutations of E, F and G for each of the $4!$ permutations of A, B, C and D). In group theoretic terminology this is known as the direct product of the symmetric group on 4 points and the symmetric group on 3 points. In order to use GAP-ECLⁱPS^e with either the SBDS version given in [Gent *et al.*, 2002] or the SBDD implementation described in Section 3 of this paper, these 4 permutations are the only information that the constraint logic programmer has to provide to GAP for this problem. GAP can now answer questions such as

- What is the composition of $(1, 4)$ with $(1, 3)(6, 5, 7)$?
 - $(1, 4, 3)(5, 7, 6)$
- Which of our 144 permutations do not move A, C or E ?
 - $()$, $(6, 7)$, $(2, 4)$, $(2, 4)(6, 7)$
- To which points is G mapped to by our group elements?
 - $7, 6, 5$ (i.e. G is mapped to E, F and itself)

Many of the questions passed to GAP by ECLⁱPS^e during search are answered by (rather more complicated) calculations similar to those given above.

2.3 The Interface

In GAP-ECLⁱPS^e all constraint satisfaction modeling and constraint handling is done in ECLⁱPS^e, as is the choosing of value to variable assignments during search, and any resulting elimination of values from domains by propagation. GAP runs as a sub-process, and is called as and when symmetry breaking information is needed. In effect, ECLⁱPS^e is the master, and GAP the slave.

It is straightforward to write ECLⁱPS^e programs which start a GAP subprocess and send and receive information which can be used to prune search. We have implemented an ECLⁱPS^e module which exports predicates for

- starting and ending GAP processes;
- sending commands to a GAP process;
- obtaining GAP results in a format which is usable by ECLⁱPS^e;
- loading GAP modules; and
- receiving information such as timings of GAP computations.

The key concept that motivates the interface is that the symmetries of a constraint satisfaction problem are permutations on a suitable initial segment of the natural numbers. Since sets of permutations have a well known algebraic structure, and since GAP uses the algebraic structure to enhance and extend computational capability, we use GAP to provide symmetry information to ECLⁱPS^e during search.

3 Symmetry Breaking by Dominance Detection

In order to deal with symmetry-related questions arising at nodes in the search tree, we define an $M \times N$ array where N is the number of variables in our constraint satisfaction problem, and M is the number of values that the variables can take. The i, j -th element in the array denotes assigning the value i to the variable j , and each element is associated with a unique number (point) from 1 to MN . We then define symmetries in terms of permutations on these $M \times N$ points. It should be noted that

- this allows us to define symmetries on both variables (as in the example above) and values (as in, for example, a graph colouring problem where values representing colours can be interchanged); and
- the algebraic structure is preserved: for the above example we have 144 permutations on 50×7 points, with each permutation corresponding to a unique member of the original group acting on 7 points.

This structure allows us to ask GAP questions regarding the image of (sets of) variable–value assignments under permutations. We write this as p^g , where p is an assignment point and g is an element of a permutation group. We have, for example, $17^{(2,5,17,9,8)} = 9$.

Suppose that we have identified a symmetry group, G , and that we maintain a record in a list S of *fail sets*: sets corresponding to the roots of completed subtrees. Each fail set contains the points from the $M \times N$ array corresponding to the positive decisions¹ made during the search to reach the root of the subtree. Suppose also that *Pointset* denotes the set

¹As long as we try a positive decision ($Var = Val$) before its negative ($Var \neq Val$) we are free to ignore the negative decisions [Focacci and Milano, 2001]. E.g. looking at the $Y = b$ subtree in Figure 1, $Y = b$ has already been fully explored regardless of whether or not $X \neq a$, since the $X = a$ case was covered by the $X = a$ subtree.

of points corresponding to variables which have been set to a fixed value in the current search node (either through direct assignment or through propagation). This situation is shown informally in Figure 1, where the circle indicates the current search node and the shaded triangles denote completed subtrees: S contains three single-element sets containing the points corresponding to the assignments $X = a$, $Y = b$ and $Z = c$, and if any variables have been given fixed values as a result of propagating the decisions $X \neq a$, $Y \neq b$ and $Z \neq c$, the corresponding points will appear in *Pointset*.

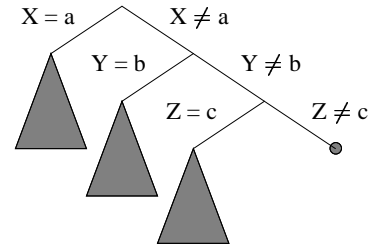


Figure 1: A partial search tree

We say that our current node is dominated by a completed subtree if there exists a g in G and an s in S such that

$$s^g \subseteq \text{Pointset} .$$

If dominance is detected, then it is safe to backtrack, since the current search state is symmetrically equivalent to one considered previously.

In practice, we pass to GAP more information about the current state than just the fixed variables, in order to facilitate domain reduction when dominance is not detected (see Section 4).

4 Generic SBDD

Pseudo-code for our generic SBDD implementation is given in Figure 2. The procedure assumes that the search state is at a node at a given depth in the search tree, and that we have a record of the fail set accumulated on the current branch of the search. We first choose a variable–value pair, try the assignment $Var = Val$, and increment the depth counter. The $Var = Val$ choice represents a point, pt , of our value–variable array; we add this to our fail set, as it represents the latest root node of a subtree. We next obtain *Doms*, a list of the domains of all the variables at the current search node (after propagating the $Var = Val$ assignment). Note that *Doms* implicitly contains *Pointset*, as well as information about which values cannot be assigned to particular variables (either from propagation or from explicit $Var \neq Val$ assertions made on the current branch).

Provided that the CSP is still consistent, we are now ready to ask GAP for a dominance check, details of which are given in Section 4.1. If this check succeeds (i.e. a dominating state was found), we can backtrack in ECLⁱPS^e as we have already explored an equivalent state. If this check fails (i.e. if no dominating state is found) then we can still benefit from the call to GAP by domain reduction. GAP supplies a set of points that,

```

generic_sbdd(Failset, depth) : –
  choose(Var, Val)
  assert(Var = Val)
  depth := depth + 1
  pt := Point(Var = Val)
  Failset := [pt, Failset]
  Doms := [current_domains]
  if consistent(CSP)
  and askGAP(Doms) = [false, Q] then
    reduce_domains(Q)
    solution_check
    generic_sbdd(Failset, depth)
  else
    tellGAP(Failset, depth)
    Failset := tail(Failset)
    assert(not (Var = val))
    Doms := [current_domains]
    if consistent(CSP)
    and askGap(Doms) = [false, Q] then
      reduce_domains(Q)
      solution_check
      generic_sbdd(Failset, depth)
    end if
    else
      backtrack(newdepth)
      generic_sbdd(Failset, newdepth)
    end if
  end if

```

Figure 2: Pseudo-code for generic SBDD

if any one of the corresponding assignments is made, would result in a successful dominance check. Clearly we should not allow a search which makes any of these assignments, so we remove them from the domains of the variables involved. This not only reduces the sizes of the value domains, but also allows further propagation based on the removals. This is a significant benefit over obtaining a mere yes/no answer to the dominance check.

In this situation, since there was no dominance, we carry on searching by choosing the next variable–value pair, using the updated fail set and depth value. A small, but important, point arises in this situation. The domain reduction by GAP after a failed dominance check can lead through propagation to setting all variables and obtaining a complete solution. This solution might turn out to be equivalent to a previously obtained solution. Therefore in this situation we perform a final dominance check to guarantee that all solutions returned are distinct.

When the dominance check succeeds, we backtrack and assert $Var \neq Val$. We tell GAP that our current fail set is the most up to date, and remove pt from it. Now that we are at

a different node in the search tree, we can obtain *Doms* again and ask GAP to perform another dominance check. If this check fails, then, as before, we reduce value domain sizes by removing from variable domains any elements we know would have led to dominance if they had been assigned to the variable, check that any solution is not dominated by a previous one, and carry on searching below the $Var \neq Val$ branch.

If, however, the $Var \neq Val$ dominance check succeeds, then we backtrack to wherever the ECLⁱPS^e search method is due to try next. This point becomes the root of a completed subtree, we update the fail set accordingly, and carry on searching.

4.1 Dominance Check in GAP

We maintain in GAP a record of fail sets, and the depth of their roots. The symmetry group is computed from generators supplied from the ECLⁱPS^e model of the problem. In fact, the whole group is not usually computed explicitly – a permutation group on n points can have $n!$ elements, leading to a large space overhead unless techniques are used for computing group elements as and when required.

The dominance check in GAP is implemented using a tree-like data structure which encodes all of the fail sets currently applicable, while taking maximum advantage of their overlaps. We can identify disjoint sets of points A_1, \dots, A_k and B_0, \dots, B_k such that the fail sets are $A_1 \cup \dots \cup A_i \cup B_i$ for each i . The right-pointing edges of the tree are labelled with elements of an A_i , the left-pointing ones with elements of a B_i . Each node of the tree can be associated with the sequence of labels on the path to it from the root.

We perform the dominance check using a recursive search, which descends this tree, entering each node once for every way of mapping the associated sequence of points into the current point list. If we reach a left-pointing leaf, then we have discovered dominance. The implementation of the search uses the standard group theoretic machinery of stabilizer chains, schreier vectors and transversals, described, for instance in [Seress, 2002].

We can detect relatively easily cases where all but the final element of a fail set can be mapped into *Pointset*, and report them, eventually, back to ECLⁱPS^e, so that domain deletion can occur. A few other cases can also be detected quickly. It is possible to enhance the search to detect all cases where all-but-one element of a fail set can be mapped, but the benefit of the extra propagation never seems to outweigh the cost of the extra search.

Since fail sets and point lists are not, in general, the same size, the more powerful machinery of partition backtrack searching also described in [Seress, 2002] does not appear to be helpful.

This implementation produces good performance on moderate-sized examples (up to about 10^{22} symmetries), but the internal search can become bogged down when a subset of some F has a large stabilizer, so that we can find elements of G mapping f_1, \dots, f_k to p_1, \dots, p_k in any order, but none of these allow us to map f_{k+1} to anything in *Pointset*. Actually computing the set stabilizers of initial segments of F , while possible, seems to be prohibitively expensive in many

cases, but such situations usually arise when the group G preserves a system of imprimitivity (for example the rows and columns of a matrix-structured problem) and this condition can be recognized cheaply. Exploiting this information will be an important part of future work.

5 Examples

In this section we provide some results of computations using generic SBDD in GAP–ECLⁱPS^e. All the examples were run on a 1 GHz pentium III processor with 512 Megabytes of memory, and times are reported in seconds. We compare the performance of our SBDD implementation with that of the GAP–ECLⁱPS^e SBDS implementation given in [Gent *et al.*, 2002], which provided full symmetry breaking in a few seconds for problems having up to 10^9 symmetries. A major cost in dealing with larger symmetry groups in SBDS is the communication of information between GAP and ECLⁱPS^e – the constraints posted during search are based on large scale group theoretic structures which have to be returned to ECLⁱPS^e from GAP. In SBDD, however, we expect to be able to deal with much larger groups, since inter-process communication consists of the word *true*, the word *false*, or lists of points of length at most $M \times N$.

5.1 Example: $A^3 + B^3 + C^3 + D^3 = E^3 + F^3 + G^3$

We first consider the illustrative problem given in Section 2. Clearly, breaking symmetry in this problem is achievable by adding the constraints

$$A \leq B \leq C \leq D \quad \text{and} \quad E \leq F \leq G .$$

We include this example to demonstrate that our implementation breaks all 144 symmetries, with performance comparable to that of SBDS in GAP–ECLⁱPS^e. The results for all solutions with domains $1 \dots 20$ are given in Table 1.

	SBDD	SBDS	ECL ⁱ PS ^e
Solutions	265	265	38,160
Backtracks	38,703	38,483	1.5×10^6
GAP cpu	1,040	973	n/a
ECL ⁱ PS ^e cpu	272	482	4,037
Σ cpu	1,312	1,455	4,037

Table 1: Seven cubes problem – comparative results

We see that SBDD and SBDS both eliminate all the symmetries in roughly the same time.

5.2 Example: Balanced Incomplete Block Designs

We now present results for examples with much larger symmetry groups. Consider the problem of finding $v \times b$ binary matrices such that each row has exactly r ones, each column has exactly k ones, and the scalar product of each pair of distinct rows is λ . This is a computational version of the (v, b, r, k, λ) BIBD problem [Colbourn and Dinitz, 1996].

Solutions do not exist for all parameters, and results are useful in areas such as cryptography and coding theory. A solution has $v! \times b!$ symmetric equivalents: one for each permutation of the rows and/or columns of the matrix. Gent

et al. [2002] reported the results given in Table 2, with the largest symmetry group having $6! \times 10! \approx 3 \times 10^9$ elements.

Parameters					GAP–ECL ⁱ PS ^e (SBDS)			
v	b	r	k	λ	Sols.	GCPU	ECPU	Σ CPU
7	7	3	3	1	1	0.71	0.68	1.39
6	10	5	3	2	1	0.89	5.57	6.46

Table 2: Balanced incomplete block designs – SBDS

The results for our generic SBDD implementation are given in Table 3. We see that the implementation is faster than SBDS in both the problems given in Table 2, and can deal with much larger symmetry groups: the $(7, 21, 6, 2, 1)$ BIBD problem has $7! \times 21! \approx 2.5 \times 10^{23}$ symmetries.

Parameters					GAP–ECL ⁱ PS ^e (SBDD)			
v	b	r	k	λ	Sols.	GCPU	ECPU	Σ CPU
7	7	3	3	1	1	0.58	0.10	0.68
6	10	5	3	2	1	1.31	0.50	1.81
7	14	6	3	2	4	13.84	2.46	16.30
9	12	4	3	1	1	4.75	0.47	5.22
13	13	4	4	1	1	44.12	1.63	45.75
8	14	7	4	3	4	200.18	29.11	229.29
6	20	10	3	4	4	164.82	18.00	182.82
7	21	6	2	1	1	23.50	1.50	25.00

Table 3: Balanced incomplete block designs – SBDD

We can see from these results that the absolute number of symmetries of a problem is not necessarily a guide to the difficulty in eliminating them from solutions. The $(8, 14, 7, 3, 4)$ BIBD problem has “only” $\approx 3.5 \times 10^{15}$ symmetries, but is harder to solve than ones with $O(10^{21})$ and $O(10^{23})$ symmetries. As well as the inherent difficulty of the original constraint problem, much depends on the size and nature of structures within the algebraic structure of each symmetry group, which is another reason for utilising a specialised CGT system such as GAP, which are designed to find and exploit these sub-structures. As a general rule, though, it is harder to eliminate solution symmetries from a larger matrix model.

It also is worth noting that the entire symmetry group for any BIBD can be generated from just four permutations: cycling the rows and columns, and swapping the first and last row and the first and last column. These permutations are trivially implemented, and comprise the only information needed by GAP–ECLⁱPS^e to break all the symmetries of the problem.

The timings obtained are comparable with those presented for the same problems in [Flener *et al.*, 2002], where lexicographic ordering constraints were used to break the row and column symmetries. The advantage of using SBDD is that all symmetries are broken, whereas a lexicographic solution for the $(6, 20, 10, 3, 4)$ BIBD problem returns 21 solutions.

6 Conclusions

We have presented an implementation of SBDD which

- uses specialist CGT techniques to detect dominance.

- guarantees to return only symmetrically distinct solutions.
- does not require a new dominance checker to be implemented for each new problem – the user only has to supply a small sample of symmetries.
- allows value domains to be reduced at search nodes where no dominance occurs. We do this by removing values of variables which, if set, we know would lead to a successful dominance check.
- eliminates all symmetries in large scale combinatoric problems.

We believe that the use of CGT techniques in SBDD solvers is an important contribution. In comparison with other recent symmetry breaking implementations, it is true that we are not currently reporting the best run times. While there is scope for further optimisation of our techniques, we already have significant advantages over related work. Compared to other implementations of SBDD, we have the key advantage of avoiding the need for a separate dominance check to be implemented, either directly [Fahle *et al.*, 2001] or as a separate constraint satisfaction problem [Puget, 2002]. This is an extremely important step forward in the application of SBDD. Compared to the use of GAP for SBDS [Gent *et al.*, 2002], we avoid the large space overhead, meaning that – as we reported here – we are able to solve completely problems with groups many orders of magnitude larger. Some techniques, such as [Flener *et al.*, 2002], do not guarantee to eliminate all symmetries, while we do.

Our implementation is robust: both the ECLiPSe and GAP searches are deterministic, and will break all the supplied symmetries, since a dominance check is performed at each node visited during search. This robustness may have a negative effect on efficiency. There is evidence that performance can be improved by making full dominance checks at a subset of the visited nodes [Fahle *et al.*, 2001; Puget, 2002], or by using a subset of the full symmetry group of the problem [McDonald and Smith, 2002]. Both of these approaches depend on the size and structure of the problem being addressed, and we will investigate their applicability to our implementation in the future.

References

- [Backofen and Will, 1999] R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *Proceedings, CP-99*, pages 73–87. Springer, 1999. LNCS 1713.
- [Brown *et al.*, 1988] C.A. Brown, L. Finkelstein, and P.W. Purdom, Jr. Backtrack searching in the presence of symmetry. In T. Mora, editor, *Proc. AAECC-6*, number 357, pages 99–110. Springer-Verlag, 1988.
- [Brown *et al.*, 1996] C.A. Brown, L. Finkelstein, and P.W. Purdom Jr. Backtrack searching in the presence of symmetry. *Nordic Journal of Computing*, 3(3):203–219, 1996.
- [Colbourn and Dinitz, 1996] C.H. Colbourn and J.H. Dinitz, editors. *The CRC Handbook of Combinatorial Designs*. CRC Press, Rockville, Maryland, USA, 1996.
- [Fahle *et al.*, 2001] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In T. Walsh, editor, *Proc. CP 2001*, pages 93–107, 2001.
- [Flener *et al.*, 2002] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kızıltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In Pascal Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming — CP’2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 462–476. Springer-Verlag, 2002.
- [Focacci and Milano, 2001] Filippo Focacci and Michaela Milano. Global cut framework for removing symmetries. In T. Walsh, editor, *Proc. CP 2001*, pages 77–92, 2001.
- [GAP, 2000] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.2*, 2000. (<http://www.gap-system.org>).
- [Gent and Smith, 2000] I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In W. Horn, editor, *Proceedings of ECAI-2000*, pages 599–603. IOS Press, 2000.
- [Gent *et al.*, 2002] Ian P. Gent, Warwick Harvey, and Tom Kelsey. Groups and constraints: Symmetry breaking during search. In Pascal Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming — CP’2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 415–430. Springer-Verlag, 2002.
- [Harvey, 2001] Warwick Harvey. Symmetry breaking and the social golfer problem. In Pierre Flener and Justin Pearson, editors, *Proceedings, SymCon-01: Symmetry in Constraints*, pages 9–16, 2001.
- [Hentenryck, 1989] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, MA, 1989.
- [McDonald and Smith, 2002] Iain McDonald and Barbara Smith. Partial symmetry breaking. In Pascal Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming — CP’2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 431–445. Springer-Verlag, 2002.
- [Puget, 2002] J-F. Puget. Symmetry breaking revisited. In Pascal Van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming — CP’2002*, volume 2470 of *Lecture Notes in Computer Science*, pages 446–461. Springer-Verlag, 2002.
- [Seress, 2002] Akos Seress. *Permutation group algorithms*. Number 152 in Cambridge tracts in mathematics. Cambridge University Press, 2002.
- [Wallace *et al.*, 1997] M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe : A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, May 1997.