

# Groups and Constraints: Symmetry Breaking During Search & Symmetry Breaking by Dominance Detection

Ian P. Gent<sup>1</sup>, Warwick Harvey<sup>2</sup>, Tom Kelsey<sup>1</sup>, Steve Linton<sup>1</sup>, and Karen Petrie<sup>3</sup>

<sup>1</sup> University of St Andrews, UK  
{ipg,tom,sal}@dcs.st-and.ac.uk,

<sup>2</sup> CrossCore Optimization, UK  
wh@crosscoreop.com,

<sup>3</sup> University of Oxford, UK  
karen.petrie@admin.ox.ac.uk

**Abstract.** In this paper we define, describe and analyse dynamic techniques for breaking symmetries in constraint satisfaction problems. The techniques, symmetry-breaking during search and symmetry-breaking by dominance detection, are sound – no solutions are lost – and complete – exactly one member of each symmetrically equivalent class of solutions will be returned, subject to the correct permutation group being identified before search. Our implementations use advanced computational group theory algorithms to answer symmetry-related questions during search. We demonstrate that the cost of computing answers to these questions is often considerably outweighed by the resulting pruning of search in an NP-complete class of problems. We compare our methods with each other, and with other related symmetry-breaking paradigms.

## 1 Introduction

Constraint programming is a powerful problem-solving technique. The *constraint satisfaction problem* (henceforth CSP) is to assign values to a set of variables such that all constraints among them are satisfied and, potentially, some objective is optimised. While still a very active research area, constraint programming is also applied to a wide range of commercially and scientifically important problems, such as airline crew rostering and statistical experiment design. Many important constraint systems possess high degrees of symmetry, arising either from the problem (where, for instance, an airline may have many interchangeable aircraft) or from the modelling of the problem in constraints (where a set of objects may be modelled by a list, but the order is not important).

Constraint systems are a generalization of the Boolean satisfiability problems that play a central role in theoretical computer science. Solving constraint satisfaction problems (CSPs) in general is thus NP-complete. Solution attempts can be conceptualised – and implemented – using search trees, with nodes representing problem variables and edges representing assignments of domain values. Sets

of assignments of values to variables define paths within the tree; a full assignment being a path to leaf, there being no remaining uninstantiated variables.

**Definition 1.** *A CSP instance  $P$  is a tuple  $\langle V, D, C \rangle$  where  $C$  is a set of constraints acting on a finite set of variables,  $V$ , each of which has finite domain of possible values  $D_i := \text{Dom}(V_i)$ . A solution to a CSP is an instantiation of all of the variables in  $V$  such that all of the constraints in  $C$  are satisfied.*

CSPs are often highly symmetric. Symmetries may be inherent in the problem, as in placing queens on a chess board that may be rotated and reflected. Additionally the modelling of a real problem as a CSP can introduce the extra symmetry: problem entities which are indistinguishable may in the CSP be represented by separate variables, leading to  $n!$  symmetries between  $n$  variables.

Two definitions of symmetries for constraint satisfaction problems were introduced by Cohen *et al.* [5]. These definitions are sufficiently general to encompass all the types of symmetry allowed by the definitions given in previous papers.

Note that the essential feature that allows any bijective mapping on a set of objects to be called a symmetry is that it leaves some property of those objects unchanged. The particular set of symmetries that is obtained depends on exactly what property it is that is chosen to be preserved. The first definition uses the property of being a solution.

**Definition 2.** *For any CSP instance  $P = \langle V, D, C \rangle$ , a solution symmetry of  $P$  is a permutation of the set  $V \times D$  that preserves the set of solutions to  $P$ .*

In other words, a solution symmetry is a bijective mapping defined on the set of possible variable-value pairs of a CSP, that maps solutions to solutions. Note that this general definition allows variable and value symmetries as special cases.

To state the definition of *constraint symmetries* it is necessary to first describe a mathematical structure associated with any CSP instance. For a binary CSP instance, the details of the constraints can be captured in a graph, the *microstructure* [9] of the instance.

**Definition 3.** *For any binary CSP instance  $P = \langle V, D, C \rangle$ , the microstructure of  $P$  is a graph with set of vertices  $V \times D$  where each edge corresponds either to an assignment allowed by a specific constraint, or to an assignment allowed because there is no constraint between the associated variables.*

Hence, the vertices of the microstructure correspond to variable-value pairs of the CSP. It is more convenient to deal with the complement of this graph. The *microstructure complement* has the same set of vertices as the microstructure, but with edges joining all pairs of vertices which are *disallowed* by some constraint, or else are incompatible assignments for the same variable. In other words, two tuples  $\langle v_1, a_1 \rangle$  and  $\langle v_2, a_2 \rangle$  are represented by two vertices in the microstructure complement, these vertices are connected by an edge if and only if:

- the vertices  $v_1$  and  $v_2$  are in the scope of some constraint, but the assignment of  $a_1$  to  $v_1$  and  $a_2$  to  $v_2$  is disallowed by that constraint; *or*

- $v_1 = v_2$  and  $a_1 \neq a_2$ .

The microstructure complement can be extended to non-binary CSPs by considering a hypergraph as opposed to a graph.

From this definition it is now possible to define a constraint symmetry. Recall that an *automorphism* of a graph is a bijective mapping of the vertices that preserves the edges (and hence also preserves the non-edges).

**Definition 4.** *For any CSP instance  $P = \langle V, D, C \rangle$ , a constraint symmetry is an automorphism of the microstructure complement of  $P$  (or, equivalently, of the microstructure).*

We focus on constraint symmetry throughout this paper.

Symmetries can give rise to redundant search since subtrees may be explored which are symmetric to subtrees already explored. To avoid this, constraint programmers try to exclude all but one in each equivalence class of solutions. Many methods have been developed for this purpose. Two of these methods for symmetry exclusion which operate during search are, symmetry breaking during search (SBDS) [2, 14], and symmetry detection by dominance detection (SBDD) [6, 7]. More recently computational group theoretic versions of these methods have been devised namely GAP-SBDS [12] and GAP-SBDD [13]. This paper provides an indepth look at both GAP-SBDS and GAP-SBDD, including a detailed comparison of the two methods.

## 2 Overview of Symmetry Breaking During Search

Symmetry-excluding search trees were introduced by Backofen and Will [1]. These operate by taking a set of symmetry breaking functions, which are provided by the user, and placing related constraints to eliminate this symmetry during search.

In a later publication [2] Backofen and Will give an example of how this may operate in practice. Consider the  $n$ -queens problem: The CSP has  $n$  variables corresponding to the rows of the chessboard, say  $Q[1], Q[2], \dots, Q[n]$ . The values correspond to the columns:  $D_i = 1, 2, \dots, n, 1 \leq i \leq n$ . The constraints can be expressed as:

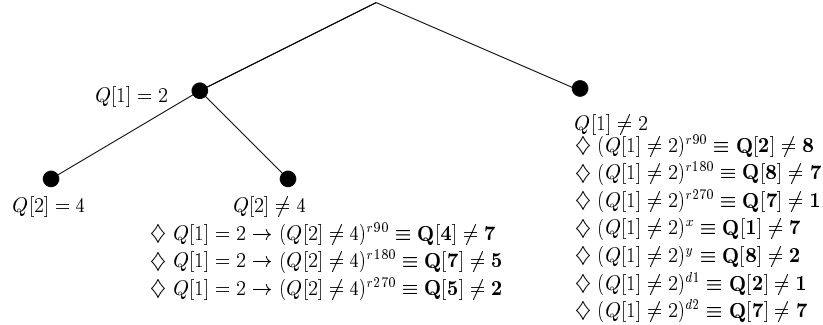
- $Q[1], Q[2], \dots, Q[n]$  are all different;
- $|Q[i] - Q[j]| \neq |i - j| \forall i, j, 1 \leq i < j \leq n$

Certainly, the chessboard symmetries (reflections in the horizontal and the vertical axes and the diagonals, and rotations  $90^\circ$ ,  $180^\circ$  and  $270^\circ$ , which can be labelled  $x, y, d1, d2, r90, r180, r270$  respectively) are constraint symmetries of  $n$ -queens for any  $n$ . An assignment  $Q[i] = j$  is transformed by the symmetries as follows:

$$\begin{aligned} x : Q[n + 1 - i] &= j; \\ y : Q[i] &= n + 1 - j; \end{aligned}$$

- $d1 : Q[j] = i$
- $d2 : Q[n + 1 - j] = n + 1 - i;$
- $r90 : Q[j] = n + 1 - i;$
- $r180 : Q[n + 1 - i] = n + 1 - j;$
- $r270 : Q[n + 1 - j] = i.$

$Q[i] = j$  means that there is a queen on row  $i$  and column  $j$ . The symmetry which reflects the board around the  $x$ -axis is represented by the function  $Q[i] = N - j$ . A search tree for the 8-Queens problem where the symmetry excluding constraints added by this method are indicated by  $\diamond$  can be found in Figure 1.

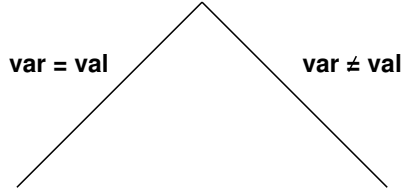


**Fig. 1.** Example of a symmetry excluding search tree on the N-queens problem

The search tree in Figure 1 intuitively means that having considered the first assignment ( $Q[1] = 2$ ) and found no solution, constraints (based on the symmetry functions) can be added on backtracking, to eliminate the symmetrical equivalents. If search continues down the left branch of the tree and backtracks from  $Q[2] = 3$  to take the alternative branch ( $Q[2] \neq 4$ ), then conditional constraints are placed. These conditional constraint state that if the symmetry is not broken (i.e.  $Q[1] \neq 2$ ) then constraints should be placed to break it (i.e.  $Q[4] \neq 7$ ).

Backofen and Will proved that this method is complete in that the full non-symmetric search space will be explored. They also showed that as long as the symmetry functions are correct and complete, all the symmetry will be eliminated. This method is very general as it applies to any branching constraints, so can be applied to most methods of constructing search trees. However, Backofen and Will failed to give more than a sketch as to what the actual algorithm entailed and no implementation details.

Gent and Smith [14] were the first to give an implementation of an algorithm which is closely related to symmetry-excluding search trees, a method which they named SBDS (Symmetry Breaking During Search). In SBDS the search tree is built from decision points. Throughout this paper a decision point can have two possible choices; either assign a value to a variable or do not assign that



**Fig. 2.** A decision point which provides the basis for an SBDS search tree

value to that variable. All search trees are built of decision points in a depth first manner. A diagram of a decision point can be found in Figure 2. Only allowing = and  $\neq$  branching decisions and depth first search, limits the generality of the approach, and is not crucial in order for SBDS to work. However, it does make it simpler to visualise exactly how SBDS operates and hence how different symmetry breaking methods compare; moreover, as far as we are aware, no-one has implemented anything else. SBDS takes a list of symmetry functions (provided by the user) and, in a similar manner to symmetry excluding search trees, places constraints to break this symmetry when backtracking to a decision point and taking the second branch.

To explain why this successfully breaks symmetry, it is necessary to examine why backtracking takes place. If using CP to find all solutions to a problem or to prove optimality, there are two possible reasons for a backtrack to occur at a decision point (if only one solution is required then only the first of these reasons holds):

1. The subtree under the decision point has been found not to yield any solutions;
2. A solution was found under the decision point, and backtracking has ensued to look for another solution.

Given the first of these cases, if a subtree has been found not to contain any solutions, than the symmetric equivalent to this subtree will also not yield any solutions, and hence a constraint can be placed to say that this subtree should not be considered. In the second of these cases, a solution has been found, so placing a constraint to state that the symmetrical equivalent to this decision point should not be considered will rule out finding a symmetrical solution. This means that by placing constraints on backtracking SBDS does not allow the symmetrical equivalent to any successful or unsuccessful subtrees to be explored; hence SBDS performs complete symmetry breaking.

A feature of SBDS is that it only breaks symmetries which are not already broken in the current partial assignment: this avoids placing unnecessary constraints. A symmetry is broken when the symmetric equivalent of the current partial assignment is not consistent with the problem constraints. The following expression provides an overview as to how SBDS works, where  $g$  is one of the problem symmetries:

$$A \ \& \ g(A) \ \& \ var \neq val \ \Rightarrow \ g(var \neq val) \tag{1}$$

where  $A$  is the partial assignment made so far during search,  $g(A)$  is the symmetric equivalent of  $A$ ,  $var \neq val$  is the failed assignment and  $g(var \neq val)$  is the symmetric equivalent to the failed assignment. If  $A$  is the current partial assignment, and it has been established that  $var \neq val$ , it must be ensured that only unbroken symmetries are dealt with, so it is checked that  $g(A)$  still holds. Then to ensure that the symmetrically equivalent subtree to the current subtree will not be explored, the constraint  $g(var \neq val)$  is placed.

Since  $g(A)$  involves all values set so far, it is potentially large, so checking that a symmetry is unbroken could be computationally expensive. However, it can be noted that if  $A$  is extended to the next partial assignment  $A_1$  then  $A_1 = A + (var = val)$  (where  $var = val$  is the next decision on the search tree). Then  $g(A_1) = g(A) + g(var = val)$ . So a Boolean variable can be constructed for each symmetry  $g$  representing whether  $g(A)$  is satisfied or not: its value for  $g(A_1)$  is the conjunction of its values for  $g(A)$  and  $g(var = val)$ . Hence, it can be decided incrementally whether  $g(A)$  holds. Further, when the value of one of these Boolean variables becomes false, it is known that the corresponding symmetry is permanently broken, and need no longer be considered, on this branch.

One problem with SBDS is that when the number of symmetries is large, a large number of symmetry functions has to be described. However, for problems in which the variables must take distinct values, Puget has shown that a low-degree polynomial number of binary inequality constraints can break all the variable symmetry, even though an exponential number of symmetries are present [23].

The above difficulty can also be dealt with by choosing a subset of the symmetry functions to use with SBDS. McDonald and Smith [19] give an algorithm for choosing a subset of symmetry functions which will break a large amount of the symmetry. It operates by choosing the symmetries which rule out no-goods near the root of the search tree, before those which only prune near the leaves of the search tree. Unfortunately, the complete algorithm needs to know ahead of time every partial assignment that will be considered during search e.g. from the variable and value ordering heuristic to be used. Every symmetry apparent in the problem is then applied to this partial assignment. This means it is infeasible to use the method in its entirety with all but the smallest problems with small symmetry groups.

It is a testament to the success of SBDS that it has been cited in many papers [3, 17, 25]. It does have a flaw though: in the majority of cases a symmetry function has to be written for each of the problem's symmetries, in order to ensure full symmetry exclusion. This may lead to more constraints being placed than the constraint solver can easily cope with, as we have already discussed, or may pose a problem for the user. If a problem has a large number of symmetries there may be too many for the user to identify and implement within their chosen constraint solver. SBDS has been used successfully, despite this difficulty, with problems containing a few thousand symmetries [15].

### 3 Overview of Symmetry Breaking by Dominance Detection

The original idea of an algorithm similar to what is now referred to as SBDD, was proposed by Brown, Finkelstein and Purdom in [4]. In fact, this paper describes a computational group theoretic version of this algorithm which is similar to GAP-SBDD, one of the algorithms that provides the focus of this paper. Unfortunately, the seminal paper by Brown *et. al.* has not received the attention it deserves from the constraints community. The method of Symmetry Breaking via Dominance Detection (SBDD) was developed independently by both Focacci & Milano [7], and Fahle, Schamberger & Sellmann [6]. The title of SBDD comes from the latter of these papers, and has been adopted by the CP community as the standard name for the method.

SBDD operates by performing a check at every node in the search tree to see if it is dominated by a symmetrically equivalent subtree already explored, and if so prunes this branch. The algorithm for SBDD requires:

- A state  $\tau$  that stores information on the search space already explored;
- A problem specific function  $\Phi : (A_1, A_2) \rightarrow \{false, true\}$  that yields true if a previous partial assignment ( $A_1$ ) is symmetrical to the current partial assignment ( $A_2$ ).

A schema for the method can then be outlined as follows:

1. Check the current partial assignment ( $A_2$ ) against all the partial assignments stored in  $\tau$ . If  $\exists A \in \tau : \Phi(A, A_2)$  then prune this branch.
2. (Normal constraint processing within the current search node.)
3. Update  $\tau$  to reflect the current state of search.

It could be a problem if all previous partial assignments were to be stored in  $\tau$ , as this could lead to a large number of checks at each node in the search tree. However, this is unnecessary as sibling nodes of the search tree which represent the same search variable can be merged, thus summarising and compressing the gathered information, in such a way that only a linear number of nodes should be stored. Puget [22] gives further efficiency improvements to the SBDD algorithm which minimises both the number of dominance checks and the size of store  $\tau$ .

One thing to note about the SBDD method is that the dominance check does not need to be done at every node of the search tree. As long as a check is undertaken at every leaf node then only the non-isomorphic solutions will be returned.

An example of how SBDD works in practice, based on the example outlined in [7], here follows. Consider a problem with three variables  $X_1$ ,  $X_2$  and  $X_3$  subject to an *all\_different* constraint. The domain of all the variables is  $\{1, 2, 3, 4\}$ , and all the values can be permuted. Search proceeds by setting  $X_1 = 1$ ,  $X_2 = 2$  and  $X_3 = 3$ , which corresponds to the first solution. At this point the store,  $\tau$ , is  $(\{1\}, \{2\}, \{3\})$ . Continuing with depth-first search the next state reached is  $X_1 = 1$ ,  $X_2 = 2$  and  $X_3 = 4$  at this point this solution is compared to the store,

where  $X_3 = 4$  is found to be symmetrical to the previously stored value for  $X_3$ . This node of the search tree is then pruned and  $\tau$  becomes  $(\{1\}, \{2\}, \{3, 4\})$ .

Both Focacci & Milano and Fahle *et. al.* show that SBDD can be efficiently used to break symmetry through empirical results. They also both discuss, as does Harvey [17], how SBDS and SBDD are related. The difference between the two algorithms is where symmetry breaking takes place. SBDS places constraints to stop nodes symmetrically equivalent nodes, to those previously explored in search, from ever being reached. On the other hand, SBDD prunes nodes having reached them and found them to be symmetrical to a previously explored part of search. Another key practical difference between SBDS and SBDD is how the user must describe the symmetry of a problem. SBDS requires an individual function describing how one variable/value assignment is mapped to another, for each of the problem symmetries. SBDD requires a single function describing a mapping of any given partial assignment to any other partial assignment that encapsulates all of the symmetry. SBDD can outperform SBDS, as it does not post constraints, so does not have the overhead of waiting for large numbers of symmetry breaking constraints to propagate. It can successfully be used with problems which have too much symmetry for SBDS to be an appropriate technique. SBDD is also completely independent of CP modelling techniques; like SBDS the model can be changed and as long as the search variables remain the same no changes have to be made to the SBDD implementation.

## 4 Overview of Group Theory

Group theory is the mathematical study of symmetry. The set of solution symmetries forms a group and the set of constraint symmetries is a subgroup of this group. Many recent methods to exclude symmetry use group theory to describe a CSP's symmetry and use the functionality of computational group theory (CGT). We now briefly summarise the key group theoretic notions we require for this paper. For a more detailed introduction to these concepts, we refer the reader to Gent, Petrie and Puget's survey [16].

### Definition 11 *Group*

*Formally, a group is a non-empty set  $G$  (of elements  $g$ ) with a composition operator  $\circ$  with the properties that:*

- $G$  is closed under  $\circ$  (that is, for all  $g, h \in G, g \circ h \in G$ );
- $\circ$  is associative (that is, for all  $g, h, k \in G, (g \circ h) \circ k = g \circ (h \circ k)$ );
- $G$  has an identity  $id$  (that is, for all  $g \in G, g \circ id = id \circ g = g$ ); and
- every element  $g$  of  $G$  has an inverse  $g^{-1}$  such that  $g \circ g^{-1} = g^{-1} \circ g = id$ .

The set (of size  $n!$ ) of distinct permutations of objects under function composition forms a group (called  $S_n$  the *symmetric group* over  $n$  elements). It can be shown that this is a group by observing:

- composing two permutations gives a permutation;
- function composition is associative  $((f \circ g) \circ h = f \circ (g \circ h))$ ;

- there is an identity permutation,  $()$ , which does nothing when composed with any other group element;
- every bijective function has an inverse,  $f^{-1} \circ f = ()$ .

#### 4.1 Group Theory in Constraint Satisfaction Programming

Permutation groups act on sets of numbers, usually an initial segment of the natural numbers. The numbers are often referred to as points, and, for CSPs each point represents a distinct variable-value assignment. If only variable or value symmetries are being considered, then we can reduce the number of points.

##### Example 11 *n*-queens

Variables - *The CSP has  $n$  variables corresponding to the rows of the chessboard, say  $r_1, r_2, \dots, r_n$ , to represent the variable requires  $n$  labels*

Values- *There are  $n$  possible values for each variable, so to represent the values requires  $n$  labels.*

Variable-Value - *There are  $n$  possible labels for the variables, and  $n$  possible labels for the variables, to represent variable-value pairs requires  $n^2$  labels.*

#### 4.2 Properties of Groups

The *order* of a group  $G$  is the number of elements in the set  $G$  and is denoted by  $|G|$ .

##### Example 12 *Symmetries of a Chessboard*

*The seven symmetries of a chessboard which were denoted  $x, y, d1, d2, r90, r180, r270$  in Section 2, along with the id element,  $()$ , form a group of order 8.*

Let  $S$  be a subset of a finite group  $G$ . The set  $S$  *generates*  $G$  if every element of  $G$  can be written as a product of elements in  $S$ . Such a set  $S$  is called a *set of generators* for  $G$  and is denoted by  $G = \langle S \rangle$ . A group will generally have many generating sets, and will always have one of size  $\log_2(|G|)$  or smaller. By maintaining a small number of generators, CGT algorithms can produce group elements as needed, rather than computing and storing each element.

##### Example 13 *Generators of Chessboard Symmetries*

*The 8 symmetries of a  $3 \times 3$  chessboard where a point is given to each possible variable-value pair (e.g. point 1 means a queen is placed in row 1, column 1) can be pictured as:*

*The chessboard symmetries are generated by  $\{r90, d1\}$  since:*

$$id = r90 \circ r90 \circ r90 \circ r90;$$

$$r90 = r90 ;$$

$$r180 = r90 \circ r90;$$

$$r270 = r90 \circ r90 \circ r90;$$

$$d1 = d1;$$

$$y = d1 \circ r90;$$

1	2	3	7	4	1	9	8	7	3	6	9
4	5	6	8	5	2	6	5	4	2	5	8
7	8	9	9	6	3	3	2	1	1	4	7
	<i>id</i>		<i>r90</i>			<i>r180</i>			<i>r270</i>		
3	2	1	7	8	9	1	4	7	9	6	3
6	5	4	4	5	6	2	5	8	8	5	2
9	8	7	1	2	3	3	6	9	7	4	1
	<i>x</i>		<i>y</i>			<i>d1</i>			<i>d2</i>		

**Fig. 3.** The 8 Symmetries of a  $3 \times 3$  chessboard

$$d2 = r90 \circ r90 \circ d1;$$

$$x = r90 \circ d1.$$

A *subgroup* of a group  $G$  is a subset of  $G$  that is itself a group, with the same composition operator as  $G$ . The simplest example of a subgroup is the identity permutation, as it is its own inverse so forms a group.

**Example 14** *Subgroup of Chessboard Symmetry:*

The rotations of a chessboard  $r90, r180$  &  $r270$  form a subgroup, with the *id* element, of order 4. As can be seen from Example 13, this can be generated by the elements *id* and  $r90$ .

Given a subgroup  $H$  of a group  $G$  and an element  $g$  of  $G$ , the (right) *coset*  $H \circ g$  is the set of elements  $\{h \circ g | h \in H\}$ . Two cosets of  $H$  in  $G$  given by different elements are either disjoint or the same. Thus the cosets of  $H$  partition the elements of  $G$ . Furthermore, all the cosets of  $H$  have size  $|H|$ . The number of cosets is called the *index* of  $H$  in  $G$  and is denoted by  $|G : H|$ . If one element is chosen from each coset of  $H$ , then a set of *coset representatives* is formed. The group  $G$  can be generated by composing the elements of  $H$  with these coset representatives.

**Example 15** *Cosets of Chessboard Symmetries*

The group  $G$  is the full chessboard symmetries, and  $H$  is the rotations of the chessboard. Then the two cosets of  $H$  are:  $\{id, r90, r180, r270\}$  and  $\{d1, x, d2, y\}$ , where  $H = H \circ id$  and  $H = H \circ d1$ . It is obvious from this that a set of coset representatives is  $\{id, d1\}$  as each element is uniquely expressible as a composition of an element of  $H$ , and a coset representative.

Let  $G$  be a permutation group acting on the set  $\Omega = \{1, 2, \dots, n\}$  of points. Let  $G$  be generated by the set  $S = \{s_1, \dots, s_m\}$ . The *orbit* of  $G$  on  $\Omega$  indicates how  $G$  acts on  $\Omega$ , it indicates whether there is an element mapping a give point to another given point. More formally, the orbit of  $G$  containing the point  $\delta$  is the set  $\delta^G = \{\delta^g | g \in G\}$ .

**Example 16** *Orbits of Points on Chessboard*

Looking back at the diagram of chessboard symmetries given in example 13, the

orbits of a given point can be seen.

The orbit of the point 1 is:

- point 7 by elements  $r90$  &  $y$ ;
- point 9 by elements  $r180$  &  $d2$ ;
- point 3 by elements  $r270$  &  $x$ ;
- back to itself, point 1, by elements  $id$  &  $d1$ .

So the orbit of point 1 is  $\{1, 3, 7, 9\}$ .

In constraint programming terms, where a point represents a variable-value pair, the orbit of a point shows which other points (i.e. variable-value pairs) are symmetrical to this point.

Let  $G$  be a permutation group acting on the points  $\Omega$ . Let  $\beta \in \Omega$  be any point. The *stabiliser* of  $\beta$  in  $G$  is defined by:  $G_\beta = \{g \in G | \beta^g = \beta\}$  the set of elements in  $G$  which fixes or *stabilises* the point  $\beta$ . This set is a subgroup of  $G$ .

**Example 17** *Stabilisers of chessboard symmetries*

From Diagram 13 of chessboard symmetries given in Example 13, the stabiliser of any given point can be identified, as the elements which do not change the position of a given point.

For instance the stabiliser of the point 1 is  $\{id \text{ \& } d1\}$  as these elements map point 1 back to itself.

In constraint programming terms, the stabiliser of a point (a variable/value pair) shows what symmetry is left unbroken once that value is assigned to the given variable, during search.

## 5 GAP-ECL<sup>i</sup>PS<sup>e</sup>

In order to explore the use of CGT methods in constraints, we set up an interface between the GAP computational group theory system and the ECL<sup>i</sup>PS<sup>e</sup> Constraint Logic Programming system.<sup>4</sup> The central feature is that GAP acts as a black box. While an ECL<sup>i</sup>PS<sup>e</sup> implementation is performing tree search to solve a CSP involving symmetries, GAP is asked to provide group theoretic results such as the symmetry group itself, stabilisers of points, and members of cosets of stabilisers. The ECL<sup>i</sup>PS<sup>e</sup> implementation uses these results to break any symmetries that arise during search.

GAP [11] (Groups, Algorithms and Programming) is an open-source system for computational discrete algebra with particular emphasis on, but not restricted to, computational group theory. GAP includes command line instructions for generating permutation groups, and for computing stabiliser chains and

---

<sup>4</sup> This choice of systems is not critical and reflected our expertise. One could use systems such as MAGMA on the algebraic side or Choco on the constraints side. However, the provision, within ECL<sup>i</sup>PS<sup>e</sup>, of attributed variables that allow the user to maintain across backtracks information such as depth in the tree, roots of previously explored trees, constraints posted in other branches, etc. is fundamental to the success of our implementations.

right transversals. Note that GAP does not explicitly create and store each element of a group. This would be impractical for, say, the symmetric group over 30 points, which has  $30!$  elements. Instead group elements are created and used as required by the computation involved, making use of results such as Lagrange's theorem and the orbit-stabiliser theorem to obtain results efficiently. GAP can also be programmed to perform specific calculations in a modular way.<sup>5</sup>

ECL<sup>i</sup>PS<sup>e</sup> is an open-source Constraint Logic Programming system which includes libraries for finite domain constraint solving.<sup>6</sup> In addition to its powerful modelling and search capabilities, ECL<sup>i</sup>PS<sup>e</sup> has three important features which we utilise to prune search trees using results from computational group theory.

The first, and most important, feature is efficient communication with subprocesses. It is straightforward to write ECL<sup>i</sup>PS<sup>e</sup> programs which start a GAP subprocess and send and receive information which can be used to prune search. We have implemented an ECL<sup>i</sup>PS<sup>e</sup> 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 ECL<sup>i</sup>PS<sup>e</sup>;
- loading GAP modules; and
- receiving information such as timings of GAP computations.

The second ECL<sup>i</sup>PS<sup>e</sup> feature is the provision of attributed variables. These allow us to attach extra information to a variable and retrieve it again later. We use this feature to avoid requiring the user to thread extra symmetry-related data through their code: during the search, any symmetry data one needs in relation to any variable (including the global symmetry-breaking state) can be retrieved directly from that variable.

The third feature is the provision of suspended goals. This allows us to evaluate and impose constraints in a lazy fashion, which is a crucial feature of our approach.

## 6 GAP-SBDS

In this section a GAP-ECL<sup>i</sup>PS<sup>e</sup> implementation of SBDS in which symmetric equivalents of assignments are determined by computation within GAP is explained.

It is assumed that the search algorithm has arrived at a value to variable assignment point. The idea, as it was with SBDS, is to try assigning *Val* to *Var*, and if that fails, to exclude all symmetrically equivalent assignments. The procedure takes as argument:

- *Stab*, the member of the stabiliser chain computed at the previous assignment node (i.e. the subgroup of  $G$  which stabilises each SBDS point assigned so far) ;

---

<sup>5</sup> GAP is available from <http://www.gap-system.org/>.

<sup>6</sup> ECL<sup>i</sup>PS<sup>e</sup> is available from <https://sourceforge.net/projects/eclipse-clp>.

- $A$ , the current partial assignment of values to variables;
- $RTchain$ , a sequence of the right transversals corresponding to the assignments made so far,  $\langle RT_1, \dots, RT_k \rangle$ , where  $RT_i$  is a set containing a representative group element for each coset of  $Stab_i$  in  $Stab_{i-1}$ . We define  $g \in RTchain$  as any group element  $g$  which can be expressed as  $g = p_k \circ p_{k-1} \circ \dots \circ p_1$  where  $p_i \in RT_i$ . Note that  $RTchain$  implicitly represents the partial assignments symmetric to  $A$ , since any such assignment can be obtained by applying some  $g \in RTchain$  to  $A$ .

The implementation is based on the pseudo-code given in Figure 4. The algorithm first choose a variable-value pair, and map this pair to a GAP point. The next stage is to use GAP to compute the next member of the stabiliser chain, and the right transversal of this in the previous member of the stabiliser chain. Once this is done, we update  $RTchain$ . Since  $RTchain$  is made up of (products of) group elements which appear in the orbits of the points fixed by the current partial assignment,  $RTchain$  can be thought of as the current set of potentially applicable symmetries. The method now tests assignments.

```

sbds_search := proc(Stab, A, RTchain)
local Point, NewStab, RT, NewA, BrokenSymms, g;
  choose(Var, Val);
  Point := var_value_to_point(Var, Val);
  NewStab := Stabilizer(Stab, Point);
  RT := RightTransversal(Stab, NewStab);
  NewRTchain = RTchain ~ RT;
  assert(Var = Val);
  if sbds_search(NewStab, A  $\wedge$  Var = Val, NewRTchain) = true
  then
    return TRUE
  else
    retract(Var = Val);
    BrokenSymms := lazy_check(NewRTChain, A);
    for g in BrokenSymms do
      assert(g(A)  $\Rightarrow$  g(Var  $\neq$  Val));
    end do;
    return sbds_search(Stab, A, RTchain)
  end if
end proc

```

**Fig. 4.** Pseudo-code for GAP-SBDS

If the assignment  $Var = Val$  leads to a successful search, the method stops searching and the final  $A$  is a solution to the CSP which is not symmetric to any other solution found using the same search method. If  $Var = Val$  leads to

a failure, the method (effectively) imposes the constraint  $g(A) \Rightarrow g(Var \neq Val)$  for each symmetry  $g$ .<sup>7</sup>

A symmetry is a member of the revised *RTchain*, which is, perforce, a member of the original symmetry group. However doing this by iterating over every symmetry would be very inefficient; indeed it would be completely impractical for large symmetry groups. Instead there are several observations which allow the number of considered symmetries to be cut drastically.

First, there are potentially many symmetries which map  $A \wedge Var \neq Val$  to the same constraint; only one of such need be considered  $g$ . This is where *RTchain* comes in: each member of *RTchain* is a representative of the set of symmetries which agree on what to map each of the variable/value pairs in  $A \wedge Var \neq Val$  to. Note that this can be generalised: any members of *RTchain* which select the same  $p_i \in RT_i$  for  $i = 1 \dots j$  agree on what to map the first  $j$  elements of  $A$  to. This also means that the truth value of the first  $j$  elements of  $A$  is the same for these members of *RTchain*, suggesting that they should be considered together.

The next observation is that constraints need not be posted for any symmetries for which the precondition  $g(A)$  is false. For some  $g$  this may be true for the entire search subtree under consideration; for some it may be true for some part of the subtree; for others it may never be true. This observation can be combined with the previous one to in effect evaluate  $g(A)$  lazily, only imposing the constraint  $g(Var \neq Val)$  when  $g(A)$  is known to be true, sharing as much of the evaluation as possible between different  $g$ , and deferring that evaluation until it is known to be needed.

Suppose a prefix of *RTchain* of length  $i \geq 0$  (call it  $RTchain_i$ ) and some  $p_i \in RTchain_i$  such that the prefix of  $A$  of length  $i$  is mapped to something which is true for the current point in the search tree. If  $RTchain_{i+1} = RTchain_i \hat{\ } \langle RT_{i+1} \rangle$  then  $p_{i+1} = rt_{i+1} \circ p_i$  should be considered for all  $rt_{i+1} \in RT_{i+1}$ . All such  $p_{i+1}$  map the first  $i$  elements of  $A$  to the same thing (indeed, the same thing as  $p_i$ ), but each maps the  $i + 1$ th element to something different. For each such  $p_{i+1}$  there are three cases:

1. The  $i + 1$ th element of  $A$  is mapped to something which is true. In this case, proceed to considering the  $i + 2$ th element.
2. The  $i + 1$ th element of  $A$  is mapped to something which is false. In this case, do not consider this  $p_{i+1}$  any further. (This excludes all symmetries which map the first  $i + 1$  elements of  $A$  to the same thing from further consideration.)
3. The  $i + 1$ th element of  $A$  is mapped to something which is neither true nor false at this point in the search. In this case we delay further computation based on this  $p_{i+1}$  until the truth value is known, and then apply the appropriate case above. This is delayed in order to avoid considering the next right transversal in *RTchain* until it is clear this is necessary. This is because each time a new right transversal  $RT$  is considered for a permutation under

---

<sup>7</sup> Note that we need not explicitly impose  $Var \neq Val$ , since the identity permutation will be one of the symmetries considered.

consideration, that permutation is expanded into  $|RT|$  candidate permutations for the next iteration, and to remain practical we need to minimise the number of such multiplicative expansions.

Whenever the computation determines that  $g(A)$  in its entirety is true,  $g(Var \neq Val)$  is asserted.

The check on classes of elements of  $RTchain$  is crucial to the efficiency of the search procedure, and is made possible by careful use of variable attributes and suspended goals in ECL<sup>i</sup>PS<sup>e</sup>. It also ensures that at most one constraint is posted for each set of symmetries which map  $A$  (as a tuple) to the same thing.

### 6.1 Comparison with McDonald [20]

In a prior implementation by McDonald [20] of SBDS using group theory, on backtracking the orbit of the current partial assignment under the action of the full symmetry group is computed from scratch, and then a constraint is imposed for each member of the orbit, excluding that particular partial assignment. The idea is to ensure that each constraint posted is different, since different symmetries applied to a partial assignment can yield the same constraint. The main drawbacks of this implementation are that the orbit is computed from scratch each time, and no account is taken of whether the constraints are already entailed. The latter is important since it may be that many constraints are posted excluding different partial assignments, but that many of these constraints are useless for the same reason; e.g. because they involve excluding configurations where  $X = 1$  when we already know that  $X = 2$ .

In contrast, the GAP-SBDS approach is incremental, this method tries to share computation between symmetries, and exploits knowledge of entailment and disentanglement to minimise the work done. On the other hand, this approach can result in the same constraint being imposed more than once because it treats assignments as (ordered) tuples rather than sets; e.g. it might post both  $X \neq 1 \wedge Y \neq 2$  and  $Y \neq 2 \wedge X \neq 1$ .

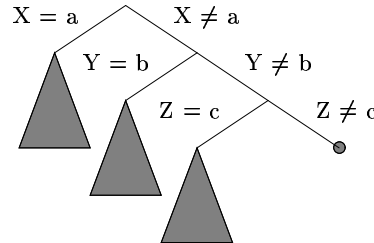
## 7 GAP-SBDD

In this section a GAP-ECL<sup>i</sup>PS<sup>e</sup> implementation of SBDD is outlined. In the next section we provide an empirical evaluation of GAP-SBDS in contrast to GAP-SBDD will be outlined.

SBDD works when given a symmetry group,  $G$ , and a list  $S$  of *fail sets*: sets corresponding to the roots of completed subtrees. Each fail set contains the points corresponding to the positive decisions<sup>8</sup> made during the search to reach the root of the subtree. Additionally, *Pointset* denotes the set 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

<sup>8</sup> As long as the positive decision ( $Var = Val$ ) are tried before the negative ( $Var \neq Val$ ) then it is possible to ignore the negative decisions [7].

situation is shown informally in Figure 5, 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*.



**Fig. 5.** A partial search tree

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

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

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

In practice, more information is passed to GAP about the current state than just the fixed variables, in order to facilitate domain reduction when dominance is not detected, this is outlined in more detail in the following section.

### 7.1 Generic SBDD

Pseudo-code for the GAP-SBDD implementation is given in Figure 6. The procedure assumes that the search state is at a node at a given depth in the search tree, and that a record of the fail set accumulated on the current branch of the search has been kept. The method first chooses a variable-value pair, posts the assignment  $Var = Val$ , and increments the depth counter. The  $Var = Val$  choice represents a point,  $pt$ , of the value-variable array; this is added to the fail set, as it represents the latest root node of a subtree. *Doms*, a list of the domains of all the variables at the current search node (after propagating the  $Var = Val$  assignment) is next obtained. 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, it is now possible to ask GAP for a dominance check, details of which are given in Section 7.2. If this check

```

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

```

**Fig. 6.** Pseudo-code for GAP-SBDD

succeeds (i.e. a dominating state was found), backtracking can take place in ECL<sup>i</sup>PS<sup>e</sup> as an equivalent state has already been explored. If this check fails (i.e. if no dominating state is found) benefit can still be gained from the call to GAP by domain reduction. GAP supplies a set of points that, if any one of the corresponding assignments is made, would result in a successful dominance check. Clearly a search should not be allowed which makes any of these assignments, so they can be removed 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, search can carry on 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 a final dominance check is performed to guarantee that all solutions returned are distinct.

When the dominance check succeeds, backtracking takes place and  $Var \neq Val$  is asserted. GAP is told that the current fail set is the most up to date, and removes  $pt$  from it. Now that a different node in the search tree is reached,  $Doms$  can be obtained again again and GAP can be asked to perform another dominance check. If this check fails, then, as before, value domain sizes can be reduced by removing from variable domains any elements that it is known would have led to dominance if they had been assigned to the variable, it can then be checked that any solution is not dominated by a previous one, and search can carry on below the  $Var \neq Val$  branch.

If, however, the  $Var \neq Val$  dominance check succeeds, then backtracking takes place to wherever the  $ECL^iPS^e$  search method is due to try next. This point becomes the root of a completed subtree, the fail set is updated accordingly, and search carries on

## 7.2 Dominance Check in GAP

In GAP a record of fail sets, and the depth of their roots is maintained. The symmetry group is computed from generators supplied from the  $ECL^iPS^e$  model of the problem. We reiterate that the whole group is never computed in its entirety: this is a central tenet of CGT, since 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. Such disjoint sets of points  $A_1, \dots, A_k$  and  $B_0, \dots, B_k$  are identified 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.

The dominance check is performed 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 a left-pointing leaf is reached, then dominance has been discovered. The implementation of the search uses the standard group theoretic machinery of stabilizer chains, Schreier vectors and transversals, described, for instance in [24].

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  $ECL^iPS^e$ , 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 [24] 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 one 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.

## 8 Empirical Analysis of GAP-SBDS versus GAP-SBDD

In this section some results of computations using GAP-SBDS and GAP-SBDD are provided. All the examples were run on a 2.6 GHz Pentium IV processor with 512 megabytes of memory, and times are reported in seconds. A major cost in dealing with larger symmetry groups in SBDS is the communication of information between GAP and ECL<sup>i</sup>PS<sup>e</sup> – the constraints posted during search are based on large algebraic structures which have to be returned to ECL<sup>i</sup>PS<sup>e</sup> from GAP. In SBDD, however, it is expected 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.

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

The first problem is that of finding a list  $[A, B, C, D, E, F, G]$  of distinct numbers in the range  $1 \dots 50$  such that  $A^3 + B^3 + C^3 + D^3 = E^3 + F^3 + G^3$ . Clearly, breaking symmetry in this problem is achievable by adding the constraints

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

This example is included to demonstrate that our implementation breaks all 144 symmetries, with comparable performance by both GAP-SBDS and GAP-SBDD. The results for all solutions with domains  $1 \dots 20$  are given in Table 1. It can be seen that GAP-SBDD and GAP-SBDS both eliminate all the symmetries in roughly the same time, whereas a search which ignores symmetry returns  $144 \times 265$  solutions.

### 8.2 Example: Balanced Incomplete Block Designs

In this section results for examples with much larger symmetry groups are presented. Consider the problem of finding  $v \times b$  binary matrices such that each row

	GAP-SBDD	GAP-SBDS	ECL <sup>i</sup> PS <sup>e</sup>
Solutions	265	265	38,160
Backtracks [BT]	38,703	38,483	$1.5 \times 10^6$
GAP cpu [Gcpu]	1,040	973	n/a
ECL <sup>i</sup> PS <sup>e</sup> cpu [Ecpu]	272	482	4,037
$\Sigma$ cpu	1,312	1,455	4,037

**Table 1.** Seven cubes problem – comparative results

has exactly  $r$  ones, each column has exactly  $k$  ones, and the scalar product of each pair of distinct rows is  $\lambda$ . This is a computational version of the  $(v, b, r, k, \lambda)$  BIBD problem. The  $v \times b$  matrix is labelled in column order, since  $v \leq b$  for all suitable parameters. Zeros are assigned before ones whenever  $k \geq b/2$ , otherwise ones are assigned before zeros; the heuristic is to use the minimum domain value whenever there are more ones than zeros in each column.

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. The largest BIBD that GAP-SBDS can deal with has parameters  $(6, 10, 5, 3, 2)$ , the results of this along with the  $(7, 7, 3, 3, 1)$  instance can be found in Table 2. The results for the GAP-SBDD implementation where it is possible to solve far larger instances are given in Table 3.

Parameters					GAP-SBDS					
$v$	$b$	$r$	$k$	$\lambda$	Symms.	Sols.	BT	Gcpu	Ecpu	$\Sigma$ cpu
7	7	3	3	1	$10^7$	1	2	0.71	0.68	1.39
6	10	5	3	2	$10^9$	1	2	0.89	5.57	6.46

**Table 2.** Balanced incomplete block designs – GAP-SBDS

The first point to note is that, as expected, it is possible to deal with much larger groups with GAP-SBDD than with GAP-SBDS. GAP-SBDS was up to about four times slower, while an interesting difference was that most cpu time was in ECL<sup>i</sup>PS<sup>e</sup> with GAP dominating time in GAP-SBDD.

It can be seen 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 is designed to find and exploit these sub-structures. As a general

Parameters					GAP-SBDD					Dominance checks			
$v$	$b$	$r$	$k$	$\lambda$	Symms.	Sols.	BT	Gcpu	Ecpu	$\Sigma$ cpu	Success	Fail	Delete
7	7	3	3	1	$10^7$	1	2	0.18	0.04	0.22	6	15	10
6	10	5	3	2	$10^9$	1	2	0.43	0.13	0.56	20	40	31
7	14	6	3	2	$10^{14}$	4	33	4.63	0.34	4.97	64	146	124
9	12	4	3	1	$10^{14}$	1	3	1.79	0.10	1.89	9	29	26
11	11	5	5	2	$10^{15}$	1	65	18.36	0.75	19.11	103	272	177
8	14	7	4	3	$10^{15}$	4	327	63.04	3.20	66.24	720	1344	727
13	13	4	4	1	$10^{19}$	1	2	41.92	0.26	42.18	11	38	29
6	20	10	3	4	$10^{21}$	4	171	53.40	2.19	55.59	381	665	648
7	21	6	2	1	$10^{23}$	1	2	10.42	0.15	10.57	12	36	31
16	20	5	4	1	$10^{31}$	1	10	6077.19	0.43	6077.62	22	64	65
13	26	6	3	1	$10^{36}$	2	425	59338.23	5.81	59344.04	576	1487	968

**Table 3.** Balanced incomplete block designs – GAP-SBDD

rule, though, it is harder to eliminate solution symmetries from a larger matrix model.

It is also 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 both GAP-SBDS and GAP-SBDD to break all the symmetries of the problem.

The timings obtained are comparable with those presented for the same problems in [8], 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. Moreover, while SBDD can work with any variable or value ordering heuristics, a heuristic can interact badly with lexicographic ordering constraints.

## 9 Case Study: Graceful Graphs

A labelling  $f$  of the nodes of a graph with  $q$  edges is *graceful* if  $f$  assigns each node a unique label from  $\{0, 1, \dots, q\}$  and when each edge  $xy$  is labelled with  $|f(x) - f(y)|$ , the edge labels are all different. (Hence, the edge labels are a permutation of  $1, 2, \dots, q$ .) Figure 7 shows an example. The study of graceful graphs is an active area of graph theory. Gallian [10] surveys graceful graphs, i.e. graphs which have a graceful labelling, and lists the graphs whose status is known. Indeed, symmetry breaking techniques have contributed new results on graceful graphs: Gallian credits Petrie and Smith with establishing ten cases of existence or non-existence of graceful labellings.

[10] is the latest version of a dynamic survey which first appeared in 1997 and has been regularly updated.

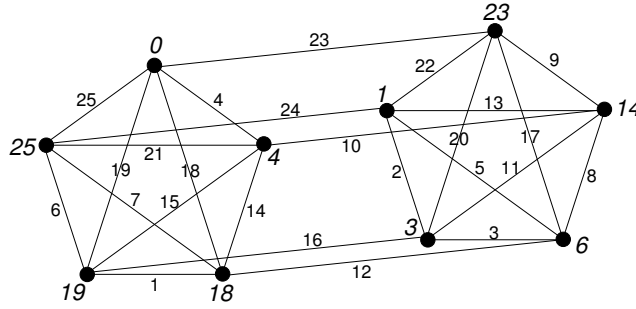


Fig. 7. The unique graceful labelling of  $K_5 \times P_2$ .

Finding a graceful labelling of a given graph, or proving that one does not exist, can be expressed as a CSP. Specific cases have previously been considered. For instance, the all-interval series problem (problem 007 in CSPLib, at <http://www.csplib.org>) is equivalent to finding a graceful labelling of a path, and Lustig & Puget [18] found a graceful labelling of one graph.

There are two kinds of symmetry in the problem of finding a graceful labelling of a graph: first, there may be symmetry in the graph. For instance, if the graph is a clique, any permutation of the vertex labels in a graceful labelling is also graceful. If the graph is a path,  $P_n$ , the labels  $x_1, x_2, \dots, x_n$  can be reversed to give an equivalent labelling  $x_n, \dots, x_2, x_1$ . This type of symmetry is referred to as geometric. The second type of symmetry is that it is possible to replace every vertex label  $x_i$  by its complement  $n - x_i$ . It is of course also possible to combine each geometric symmetry with the complement symmetry. The model used for this problem is the one outlined by Petrie and Smith in [21].

The graph  $K_5 \times P_2$ , shown in Figure 7, consists of two copies of  $K_5$ , with corresponding vertices in the two cliques forming the vertices of a path  $P_2$ . The symmetries of  $K_5 \times P_2$  are: first, any permutation of the 5-cliques which act on both in the same way. Second, inter-clique symmetry: all the node labels in the first clique can be interchanged with the labels of the adjacent nodes in the second. These symmetries can be combined with each other and with the complement symmetry. Hence, the size of the symmetry group is  $5! \times 2 \times 2$ . In general  $K_m \times P_2$  graphs have a symmetry group of size  $m! \times 2 \times 2$ . This study concentrates on symmetry breaking in 3 such graphs, namely  $K_3 \times P_2$ ,  $K_4 \times P_2$  and  $K_5 \times P_2$ . The results of finding all graceful labellings of these graphs using either GAP-SBDS or GAP-SBDD can be found in table 4.

It can be seen that GAP-SBDD is substantially slower than GAP-SBDS, and takes more backtracks for all instances. The results presented do seem to hold for larger instances,<sup>9</sup> and for other graphs such as  $K_3 \times K_3$  and  $DW_5$ . This contradicts the results reported on the BIBD problem. It is worth noting that both the GAP times and the ECL<sup>i</sup>PS<sup>e</sup> times are longer, in GAP-SBDD than in

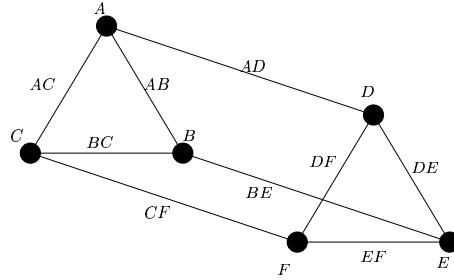
<sup>9</sup> It was proved that  $K_6 \times P_2$  has no solutions within 12 hours using GAP-SBDS, but no result was returned in 20% more time by GAP-SBDD

No. of Syms.	Graph	GAP-SBDD				GAP-SBDS			
		BT	ECL <sup>i</sup> PS <sup>e</sup> time	GAP time	Total time	BT	ECL <sup>i</sup> PS <sup>e</sup> time	GAP time	Total time
24	$K_3 \times P_2$	22	0.30	0.39	0.69	9	0.20	0.34	0.54
96	$K_4 \times P_2$	496	16.65	3.97	20.62	165	7.00	1.32	8.32
480	$K_5 \times P_2$	17997	1106	204	1311	4390	344.73	37.69	382.42

**Table 4.** Comparison of GAP-SBDS and GAP-SBDD for finding all graceful labellings

GAP-SBDS, for each graph. This shows that the delay is due not to just one of these processes, but to both.

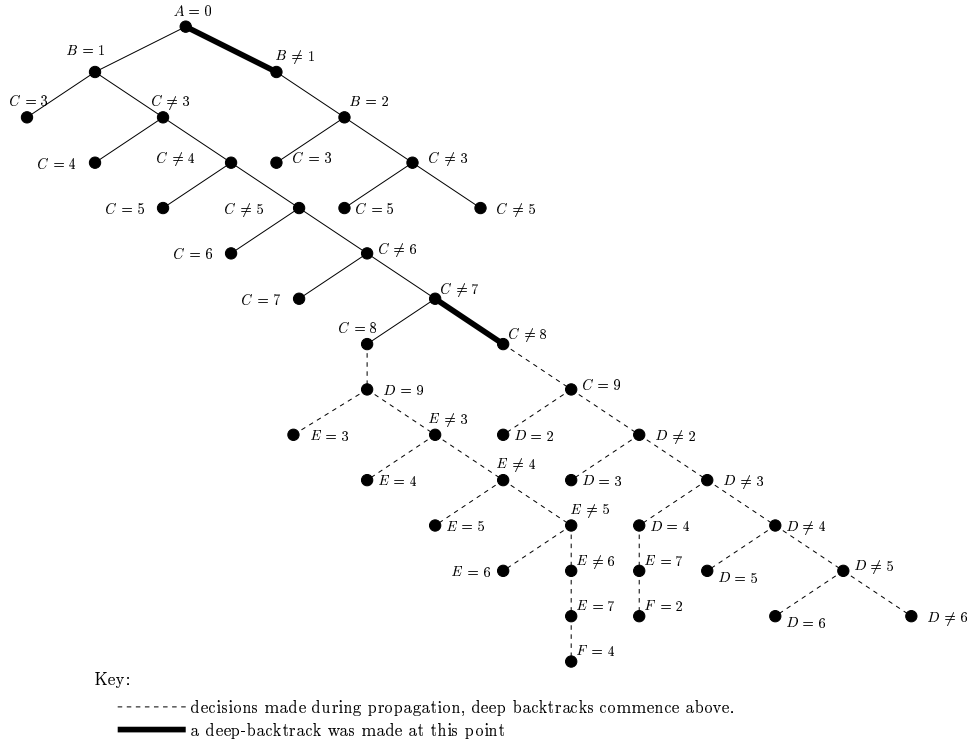
**Analysis** The difference between GAP-SBDS and GAP-SBDD has been analysed for the three graphs  $K_3 \times P_2$ ,  $K_4 \times P_2$  and  $K_5 \times P_2$ . The reason for the different times is the same in each case, but for reasons of simplicity only the results for  $K_3 \times P_2$  (Figure 8) are presented here. For ease of reference the node variables are named in capital letters, and the edge variables with a letter pairing corresponding to their attached nodes.



**Fig. 8.** Graph  $K_3 \times P_2$  showing node variable and edge variable naming

The first step in analysing where GAP-SBDS and GAP-SBDD differ was to find the first difference in the search tree. In finding all graceful labellings of  $K_3 \times P_2$  this occurs quite late in the search, after the first two solutions (from a possible four) have been found. A full diagram of the search tree until this point can be found in Figure 9. It should be noted that the backtrack count in table 4 refers to what is defined within this paper as a *deep-backtracks*. A deep-backtrack occurs when the search revisits a decision point it earlier moved beyond. In contrast a *shallow-backtrack* occurs when the *var = val* branch has been tried, and, due to propagation of that choice, immediately reversed in favour of the *var ≠ val* branch. The number of deep-backtracks is the standard backtrack count in most constraint programming environments, so eases the

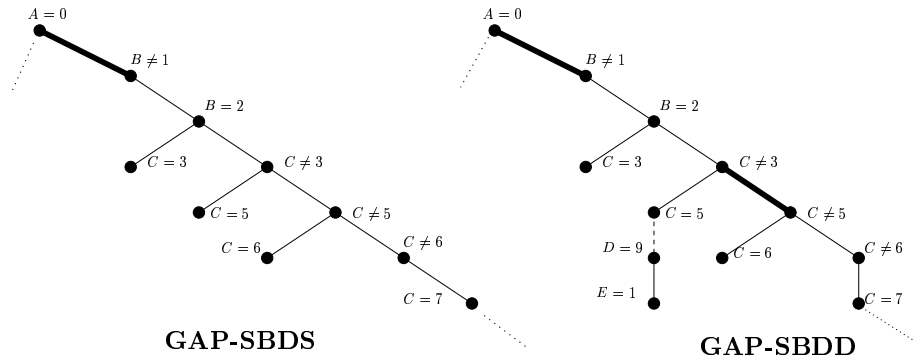
comparison of methods across environments, but in this case it does not give a sufficiently detailed picture of the search. In the fragment of the search tree shown in Figure 9 there are only 2 deep-backtracks, but the  $var \neq val$  branch is actually followed 18 times. This is important when studying GAP-SBDS, as every time the  $var \neq val$  branch is followed, symmetry-breaking constraints can be placed.



**Fig. 9.** Search tree for  $K_3 \times P_2$  to point where GAP-SBDS and GAP-SBDD differ

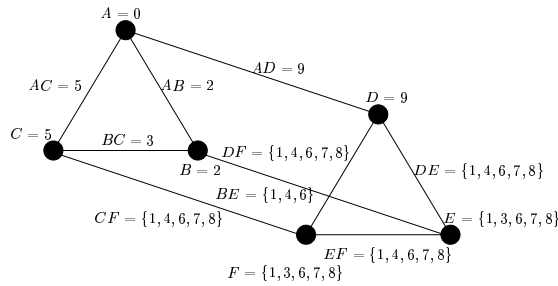
Looking more closely (Figure 10) at the branch of the search tree starting with  $A = 0$ , shows that GAP-SBDS enables earlier pruning than GAP-SBDD. This pruning happens after setting  $C = 5$ . GAP-SBDS immediately reverses from this decision to follow the  $C \neq 5$  branch, whereas GAP-SBDD carries on to set  $D = 9$  and  $E = 1$ , before returning to take the  $C \neq 5$  branch later in search (a deep backtrack).

The difference in the search trees is due to differences in constraint propagation. GAP-SBDD arrives at the search state shown in Figure 11. One of the edges must be labelled 9 (the number of edges in the graph) and the adjacent nodes must be labelled 0 and 9. At this stage  $A = 0$  and  $B$  and  $C$  are labelled with



Key:  
 — decision made due to propagation, deep-backtrack commences above  
 — a deep-backtrack was made at this point

**Fig. 10.** The search tree branch where GAP-SBDS and GAP-SBDD differ



**Fig. 11.** The domain of the node and the edge variables after propagating  $C = 5$ , using GAP-SBDD

values other than 9; hence  $D$ , the only other node adjacent to  $A$ , must take the value 9, and this inference is made by constraint propagation. Figure 11 shows the variable domains at this point. Because there are already edges labelled 2 ( $AB$ ) and 3 ( $BC$ ), the edges  $DE$  and  $DF$  cannot have those values, and hence  $E$  and  $F$  cannot have the values 6 or 7. Using GAP-SBDS, the domains of the variables are also reduced by symmetry-breaking constraints previously added on this branch. Those that are relevant in this case are symmetric equivalents of  $B \neq 1$ , namely  $E \neq 1$ ,  $F \neq 1$ ,  $E \neq 8$  and  $F \neq 8$ . (Because of the graph symmetry, nodes  $E$  and  $F$  are symmetric to node  $B$ , and the value 8 is symmetric to the value 1 because of the complement symmetry.) The only remaining value in the domains of both  $E$  and  $F$  is 3, and since these variables must have different values, this branch fails.

Most of this propagation cannot occur in GAP-SBDD. GAP just returns a boolean to indicate whether the current node is dominated or not, and possibly a list of values to prune from the domains of specific search variables. In the current implementation, a variable/value pair is returned for domain pruning if its assignment would cause dominance to be detected. In this case  $E/1$ ,  $F/1$ ,  $E/8$  and  $F/8$  are not returned. Although GAP-SBDD successfully breaks the symmetry (in this case by detecting dominance when the assignment  $E = 1$  is tried) posting SBDS constraints at an earlier stage can clearly lead to earlier pruning.

**Increasing the information returned by GAP-SBDD** The fact that GAP-SBDD returns limited information to  $ECL^iPS^e$  allows GAP-SBDD to solve much larger problems than GAP-SBDS, earlier in the BIBD experiments. However, the results of the above experiment have shown the disadvantage of this reduced communication. The constraints which GAP-SBDS places during search are adding information to the problem, and for some CP models this can cause an increase in propagation. It is not unusual for CP modellers to develop CSPs like the graceful graphs model, where only a subset of the variables are used for search, but constraint propagation over the full set of variables is crucial to solving the problem quickly. Hence the fact that GAP-SBDD may perform badly on such a simple model is an important finding.

In an attempt to counter this problem on such models, the amount of information returned by GAP-SBDD can be increased. Currently, GAP-SBDD only returns variable/value pairs that would cause dominance to be detected at the next level of the search tree. This information is used by  $ECL^iPS^e$  to delete the relevant value from the given variable's domain; this information can then be propagated through other variables. An alternative is to return variable/value pairs for domain deletion that would cause dominance to be detected both at the next level in the search tree *and* at the subsequent level. This extra information requires a more complicated group theoretic calculation, so there is a larger GAP overhead. The communication time will also be increased as GAP passes this more detailed information to  $ECL^iPS^e$ . However, if this extra information does increase the amount of constraint propagation, and hence lessens the amount of search needed, the overheads may be outweighed by these benefits.

	ECL <sup>i</sup> PS <sup>e</sup>			
	BT	time	GAP time	Total time
$K_3 \times P_2$	13	0.23	0.50	0.73
$K_4 \times P_2$	173	7.18	2.72	9.90
$K_5 \times P_2$	4402	337.69	88.20	426.89

**Table 5.** GAP-SBDD returning increased amount of information for finding all graceful labellings.

In Table 5 the results of this new version of GAP-SBDD, which returns more information, are shown for finding all graceful labellings of  $K_3 \times P_2$ ,  $K_4 \times P_2$  and  $K_5 \times P_2$ . Comparing these results for the new version of GAP-SBDD with the original version (shown in Table 4), it can be seen that the newer version has greatly reduced the search. This corresponds to a large reduction in ECL<sup>i</sup>PS<sup>e</sup> time, which in turn lowers the overall runtime. However, as expected, the percentage of GAP time in the total time has increased slightly, from 16% to 21% for  $K_5 \times P_2$ . This new version of GAP-SBDD is outperforming the original version in all cases.

However, even this new version of GAP-SBDD does not outperform GAP-SBDS (the results of which are in Table 4). GAP-SBDS still has both fewer backtracks and a lower runtime. There does not seem to be any way to cause GAP to pass all the information to ECL<sup>i</sup>PS<sup>e</sup> in GAP-SBDD that is passed in GAP-SBDS through the use of constraints. The reason for this is that GAP-SBDS can place constraints which are conditional on future search decisions, and this information cannot be passed through simple sets of variable/value pairs. It must be remembered that the fact that GAP-SBDD has a reduced communication overhead, in comparison to GAP-SBDS, can be an advantage as it allows GAP-SBDD to solve much larger problems than GAP-SBDS. GAP-SBDD may also outperform GAP-SBDS if there are many symmetries in the problem, as the constraint solver, may not propagate all the constraints placed by GAP-SBDS efficiently.

## 10 Conclusions

Before our generic approach to symmetry breaking using computational group theory, the CP practitioner had a limited range of options. A simplistic and static lex-ordering of variables and/or values will break many symmetries, but is not – in general – guaranteed to break them all. A bespoke SBDD implementation will identify and break all symmetries, but will not be generic: altering the model slightly, or identifying new symmetries, results in a major re-work of the existing SBDD implementation. SBDS without using computational group theory is effective when there are fewer than roughly  $10^5$  symmetries, but struggles to cope with larger symmetry groups.

Our approach is easily motivated, if non-trivial to implement. We let specialist systems such as ECL<sup>i</sup>PS<sup>e</sup> deal with depth-first backtrack search with propagation and search-ordering heuristics and other specialist systems such as GAP to deal with the intricacies of finding canonical representatives of permutation group elements that map our current search state to another. This combination of specialist systems allows us to deal with symmetry groups that are many orders of magnitude larger than is the case when CGT is not utilised.

Of course, we do not claim to have solved the symmetry-breaking problem for CSPs. It is not difficult – as we are dealing with an NP-complete class of decision problems – to generate instances for which either the number of SBDS constraints is prohibitively large, and/or for which the wait for a GAP dominance check

is too long. However, both constraint programming and computational group theory are exciting areas in which innovations are announced and implemented on a monthly basis: our approach cherry-picks the most useful and efficient methods from each area. The resulting systems are efficient and correct, and can be replicated by others, using their own choice of CSP and GGT systems.

## 11 Acknowledgements

We are extremely grateful to Barbara M. Smith of the Cork Constraint Computation Centre, University College Cork, Ireland, for her advice on the scope and contents on this paper.

## References

1. R. Backofen and S. Will. Excluding symmetries in concurrent constraint programming. In *Workshop on Modeling and Computing with Concurrent Constraint Programming held in conjunction with CP 98*, 1998.
2. R. Backofen and S. Will. Excluding symmetries in constraint-based search. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming - CP '99*, volume Lecture Notes in Computer Science 1713, pages 73–87. Springer, 1999.
3. N. Barnier and P. Brisset. Solving the Kirkman's Schoolgirl Problem in a Few Seconds. In P. Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP2002*, LNCS 2470, pages 477–491. Springer, 2002.
4. C. A. Brown, L. Finkelstein, and P. W. Purdom. Backtrack Searching in the Presence of Symmetry. In T. Mora, editor, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, LNCS 357, pages 99–110. Springer, 1988.
5. D. A. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B.M.Smith. Symmetry Definitions for Constraint Satisfaction Problems. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP2005*, LNCS. Springer, 2005.
6. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, volume Lecture Notes in Computer Science 2239, pages 93–107. Springer, 2001.
7. F.Focacci and M.Milano. Global Cut Framework for Removing Symmetries. In Toby Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, volume Lecture Notes in Computer Science 2239, pages 77–92. Springer, 2001.
8. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking Row and Column Symmetries in Matrix Models. In Pascal van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, LNCS 2470, pages 462–476. Springer, 2002.
9. Eugene C. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *In Proceedings AAAI'91*, pages 227–233, 1991.
10. J. A. Gallian. A Dynamic Survey of Graph Labeling. *The Electronic Journal of Combinatorics (DS6)*, 2004. (<http://www.combinatorics.org/Surveys>).
11. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.2*, 2000. (<http://www.gap-system.org>).
12. I. P. Gent, W. Harvey, and T. Kelsey. Groups and Constraints: Symmetry Breaking During Search. In Pascal van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, LNCS 2470, page 763. Springer, 2002.

13. I. P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD Using Computational Group Theory. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP2003*, LNCS 2833, pages 333–347. Springer, 2003.
14. I P. Gent and B. M. Smith. Symmetry Breaking in Constraint Programming. In *Proceedings of European Conference on Artificial Intelligence - ECAI 2002*, pages 599–603. IOS press, 2000.
15. I.P. Gent, S.A. Linton, and B.M. Smith. Symmetry breaking in the alien tiles puzzle. Technical Report APES-22-2000, APES Research Group, October 2000. Available from <http://www.dcs.st-and.ac.uk/~apes/reports>.
16. I.P. Gent, J.-F. Puget, and K.E. Petrie. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, pages 329–376. Elsevier, 2006.
17. W. Harvey. Symmetry Breaking and the Social Golfer Problem. In *Proceedings SymCon-01: Symmetry in Constraints*, pages 9–16, 2001.
18. I. J. Lustig and J. F. Puget. Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming. *INTERFACES*, 31(6):29–53, 2001.
19. I. McDonald and B. M. Smith. Partial Symmetry Breaking. In P. Van Hentenryck, editor, *In Proceedings of Principles and Practice of Constraint Programming - CP 2002*, pages 431–445, 2002.
20. Iain McDonald. Unique Symmetry Breaking in CSPs using group theory. In *Proceedings SymCon-01: Symmetry in Constraints*, pages 75–78, 2001.
21. Karen Petrie and Barbara Smith. Symmetry Breaking in Graceful Graphs. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003*, volume Lecture Notes in Computer Science 2833, pages 930–934. Springer, 2003.
22. Jean-François Puget. Symmetry Breaking Revisited. In P. Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP2002*, LNCS 2470, pages 446–461. Springer, 2002.
23. Jean-François Puget. Breaking symmetries in all different problems. In *Proceedings of the International Joint Conference on Artificial Intelligence - IJCAI05*, pages 272–277, 2005.
24. Akos Seress. *Permutation Group Algorithms*. 152. University Press, 2002.
25. B. M. Smith. Reducing Symmetry in a Combinatorial Design Problem. Technical report, School of Computer Studies, University of Leeds, January 2001.