Handbook of Constraint Programming
Edited by F. Rossi, P. van Beek and T. Walsh
© 2006 Elsevier All rights reserved

Chapter 10

Symmetry in Constraint Programming

Ian P. Gent, Karen E. Petrie, Jean-François Puget

Symmetry in constraints has always been important but in recent years has become a major research area in its own right. A key problem in constraint programming has long been recognised: search can revisit equivalent states over and over again. In principle this problem has been solved, with a number of different techniques. As we write, research remains very active for two reasons. First, there are many difficulties in the practical application of the techniques that are known for symmetry exclusion, and overcoming these remain important research problems. Second, the successes achieved in the area so far have encouraged researchers to find new ways to exploit symmetry. In this chapter we cover both these issues, and the details of the symmetry exclusion methods that have been conceived.

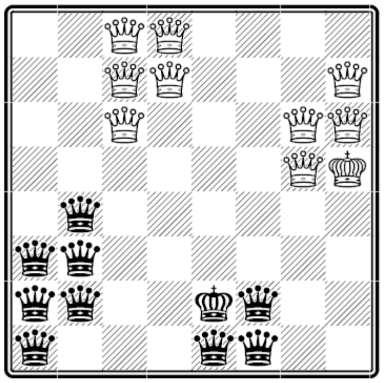

Figure 10.1: The solution to the puzzle of finding a chess position containing nine queens and a king of each colour, with the rule that no piece is on the same line (row, column or diagonal) as any queen of the opposite colour. Up to symmetry, the solution is unique.

To illustrate what we mean by symmetry, we consider the chess puzzle shown in Figure 10.1. The solution to this puzzle is unique "up to symmetry" [115], but what do we mean by symmetry in this context? By a "symmetry", we mean an operation which changes the positions of the pieces, but whose end-state obeys all the constraints if and only if the start-state does. Given a solution, which by definition satisfies all the constraints, we can find a new solution by applying any symmetry to the first solution we find. For example, given the pictured solution to the puzzle, we can swap the colours of each piece, so the black queens appear where the white queens are and vice versa. Similarly, we can rotate the chessboard by any multiple of 90 degrees to yield a new solution. Finally, we can reflect the chessboard about the horizontal axis, the vertical axis and both of the diagonal axes. Since these symmetries can be combined, there are 16 symmetries available, including the identity operation of leaving everything where it is.

Why is symmetry important? The main reason is that we can exploit symmetry to reduce the amount of search needed to solve the problem. This is of enormous potential benefit. For example, suppose we search for a solution to our chess puzzle, and the first assignment is to place a white queen in the top left hand corner. In fact, the search decision was not really to try a white queen in the top left corner, but instead the decision was to try all potential solutions with a queen of either colour in any corner of the board. Since there are 16 symmetries, we have the potential to reduce search by a factor of 16. A second reason for symmetry's importance is that many constraint problems have symmetry in them. Moreover, the act of modelling can introduce symmetries. For example, if we modelled the chess puzzle above with a variable for each queen ranging from 1 to 64 expressing its placement, there would be $2(9!)^2$ symmetric versions of each solution, as each set of queens can be permuted and the two sets swapped. Yet, this model might be desirable for effective propagation and heuristics, and so we would like to be able to deal with its symmetries effectively.

By far the most important application of symmetry in constraint programming is "symmetry breaking" in order to reduce search. The goal of symmetry breaking is never to explore two search states which are symmetric to each other, since we know the result in both cases must be the same.[1] It is common to identify three main approaches to symmetry breaking in constraint programming. The first approach is to reformulate the problem so that it has a reduced amount of symmetry, or even none at all. The second is to add symmetry breaking constraints before search starts, thereby making some symmetric solutions unacceptable while leaving at least one solution in each symmetric equivalence class. The final approach is to break symmetry dynamically during search, adapting the search procedure appropriately. This breakdown is simplistic, in that there is enormous variation within each approach, and great commonalities between approaches. However, it is a very useful informal categorisation and we will structure our discussion around it.

In the rest of this chapter we hope to answer the following questions: How do we go about achieving the search reductions that are possible? What general methods are there, and what tradeoffs are involved? How can we make it as easy as possible for the day-to-day constraint programmer to use? What research directions remain?

[1]The phrase "symmetry breaking" might be misleading, because not all methods actually break symmetry in the sense of creating a problem without symmetry. However, the usage is entrenched in the community and it would be even more confusing to try to change it.

10.1 Symmetries and Group Theory

The study of symmetry in mathematics is called *group theory*. We assume no background in group theory for reading this chapter, so we introduce all the concepts we need. We make no apologies for emphasising the role of group theory at this early stage, as it essential to understanding the role of symmetry in constraint programming. We can only introduce very briefly the key concepts, so this section should be taken only as the lightest introduction to what is one of the largest research areas in mathematics. Fortunately for most constraint programmers, a little knowledge of group theory is in fact enough to understand most of the work done to date. Sadly, introductions to group theory for mathematicians often take for granted the link with symmetry, so we will spend a little time explaining this. We will explain group theory through *permutations*. First, nothing is lost since every group can be expressed as a set of permutations. Second, it makes it very easy to understand the link between a symmetry as an element of a group and a symmetry as an *action*. We will emphasise the notion of a group action, since this expresses how symmetries transform search states, and this is our main interest.

Example 10.1. Chessboard Symmetries
Consider a 3×3 chessboard. We label the nine squares with the numbers 1 to 9. These numbers are the *points* that will be moved by symmetries. There are eight natural symmetries of a chessboard. We always include the *identity* symmetry, which leaves every point where it is. The identity is shown at the top left of Figure 10.2. Then, we can rotate the chessboard by 90, 180, and 270 degrees in a clockwise direction. The resulting locations of the points are shown in the rest of the top row of Figure 10.2. Finally, there are reflections in the vertical axis, in the horizontal axis, and in the two main diagonal axes, and these are shown in the bottom row.

$$\begin{array}{ccc} 1&2&3\\ 4&5&6\\ 7&8&9\\ &id& \end{array} \qquad \begin{array}{ccc} 7&4&1\\ 8&5&2\\ 9&6&3\\ &r90& \end{array} \qquad \begin{array}{ccc} 9&8&7\\ 6&5&4\\ 3&2&1\\ &r180& \end{array} \qquad \begin{array}{ccc} 3&6&9\\ 2&5&8\\ 1&4&7\\ &r270& \end{array}$$

$$\begin{array}{ccc} 3&2&1\\ 6&5&4\\ 9&8&7\\ &x& \end{array} \qquad \begin{array}{ccc} 7&8&9\\ 4&5&6\\ 1&2&3\\ &y& \end{array} \qquad \begin{array}{ccc} 1&4&7\\ 2&5&8\\ 3&6&9\\ &d1& \end{array} \qquad \begin{array}{ccc} 9&6&3\\ 8&5&2\\ 7&4&1\\ &d2& \end{array}$$

Figure 10.2: The 8 Symmetries of a 3×3 chessboard

The link between symmetries and permutations can be seen very simply. A permutation is a one-to-one correspondence between a set and itself. Each symmetry defines a permutation of the set of points. An easy way to write down permutations is in *Cauchy form*. Cauchy form is two rows of numbers. The top row is the complete set of elements, in ascending order, that the permutation acts over. The second row shows which number each element of the top rows maps to. For example, the identity symmetry and permutation maps each point to itself, and is shown in Cauchy form on the top left of Figure 10.3. The rotational symmetry by 90 degrees is shown in Cauchy form on the top right. We see that the point 1 is replaced by 7 after $r90$, 7 in turn is replaced by 9, 9 by 3, and 3 by 1. This gives a *cycle* $(1\ 7\ 9\ 3)$. Another cycle is $(2\ 4\ 8\ 6)$ and there is another trivial cycle just containing (5). In the main, group theorists do not use Cauchy form, preferring a notation based on the set of cycles a permutation defines.

$$id : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix} \qquad r90 : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 4 & 1 & 8 & 5 & 2 & 9 & 6 & 3 \end{pmatrix}$$

$$r180 : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix} \qquad r270 : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 6 & 9 & 2 & 5 & 8 & 1 & 4 & 7 \end{pmatrix}$$

$$x : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 2 & 1 & 6 & 5 & 4 & 9 & 8 & 7 \end{pmatrix} \qquad y : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 8 & 9 & 4 & 5 & 6 & 1 & 2 & 3 \end{pmatrix}$$

$$d1 : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 4 & 7 & 2 & 5 & 8 & 3 & 6 & 9 \end{pmatrix} \qquad d2 : \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 9 & 6 & 3 & 8 & 5 & 2 & 7 & 4 & 1 \end{pmatrix}$$

Figure 10.3: Permutations representing the symmetries of a chessboard, written in Cauchy form

Example 10.2. Cyclic form
The symmetry $r90$ above contains cycles $(1\ 3\ 9\ 7)$, $(2\ 4\ 6\ 8)$, and (5). The *cyclic form* of $r90$ is $(1\ 3\ 9\ 7)(2\ 4\ 6\ 8)(5)$ although cycles of length one can be omitted. The permutation maps each point to the succeeding element of the cycle it is in, except that the last element is mapped to the first and a point in no cycle is mapped to itself. In cyclic form, we can write the symmetries of the 3×3 chessboard as shown in Figure 10.4.

id :	$()$	$r90$:	$(1\ 3\ 9\ 7)(2\ 4\ 6\ 8)$
$r180$:	$(1\ 9)(2\ 8)(3\ 7)(4\ 8)$	$r270$:	$(1\ 7\ 9\ 3)(2\ 4\ 8\ 6)$
x :	$(1\ 3)(4\ 6)(7\ 9)$	y :	$(1\ 7)(2\ 8)(3\ 9)$
$d1$:	$(2\ 4)(3\ 7)(6\ 8)$	ha $d2$:	$(1\ 9)(2\ 6)(4\ 8)$

Figure 10.4: Permutations representing the symmetries of a chessboard, written in cyclic form

Comparing Figures 10.3 and 10.4 shows that the cyclic form is far more concise, especially when many points are unmoved by a permutation. One disadvantage is that it does not define exactly the set of points that the permutation is acting on: for example none of the permutations above move the point 5, so the number 5 does not appear in Figure 10.4. Also, the same permutation can be written down in many different ways, since cycles can appear in any order and each cycle can start with any element in it. However, the cyclic form is so natural for people to use that the computational group theory system GAP uses it as its input language for permutations, even though it then converts them internally into a more computationally efficient form.

Both forms of writing down permutations make it easy to see how a permutation *acts* on a point. In general, if p is a point and g a permutation, then we will write p^g to write down the point that p is moved to under g. For example, $1^{r90} = 7$, and $1^{r270} = 3$. We often extend this notation in the natural way to sets of other data structures containing points. For example, we have $\{1,3,8\}^{r90} = \{1^{r90}, 3^{r90}, 8^{r90}\} = \{7,1,6\} = \{1,6,7\}$: the equivalence between the last two terms is simply because sets are unordered.

There are certain key facts about permutations which provide the link between them and groups. We will explain these, then provide the fundamental definition of a group. First, it is easy to work out the composition of two permutations, which for permutations f and g we will write as $f \circ g$. The result of $f \circ g$ is calculated by taking, for each point, the result of moving that point under f and then by g. That is, for any point p, $p^{f \circ g} = (p^f)^g$. It is important to notice the order of action, i.e. we do f and then g when we write $f \circ g$, which is the other way round compared to function composition such as $\sin(\cos(x))$. Since both f and g are one-to-one correspondences, so is their composition, so the composition of two permutations is another permutation. We can calculate the composition pointwise: we simply work out what 1 moves to under f, then what the result moves to under g, and repeat for each other point.

Example 10.3. Composition of Permutations

$$r90 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 4 & 1 & 8 & 5 & 2 & 9 & 6 & 3 \end{pmatrix}$$

$$x = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 2 & 1 & 6 & 5 & 4 & 9 & 8 & 7 \end{pmatrix}$$

$$r90 \circ x = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 9 & 6 & 3 & 8 & 5 & 2 & 7 & 4 & 1 \end{pmatrix} \quad = d2$$

We have already described the existence of the identity permutation, which we call id. This can be defined as the empty set of cycles for any set of points. For any permutation f there is an *inverse* permutation, such that $f \circ f^{-1} = id$. This is easily calculated: in the cyclic form we just reverse the order of each cycle; and in the Cauchy form we swap the two rows and then reorder the columns so that the first row is in numerical order.

Example 10.4. Inverse of a permutation

$$r90 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 4 & 1 & 8 & 5 & 2 & 9 & 6 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 7 & 4 & 1 & 8 & 5 & 2 & 9 & 6 & 3 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix} \quad \text{by swapping rows}$$

$$r90^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 6 & 9 & 2 & 5 & 8 & 1 & 4 & 7 \end{pmatrix} \quad \text{by reordering} \qquad = r270$$

Finally, we note that composition of permutations is associative. That is, $f \circ (g \circ h) = (f \circ g) \circ h$. The truth of this relies on the definition of permutation composition, i.e. that $g \circ h$ gives, for each point, the same result as applying g and then h. So, for example, $7^{(f \circ g) \circ h)}$ is the result of applying f to 7, g to the result, and h to the result of that: but this is exactly the same as $7^{f \circ (g \circ h)}$, which is also found by applying f, to 7, g to the result, and h to the result of that. We now, finally, present the axioms defining a group.

Definition 10.5. ***Group Axioms***
A group is a non-empty set G with a composition operator $\circ$ such that:
- G is closed under $\circ$. That is, for all $g, h \in G, g \circ h \in G$; and
- there is an identity $id \in G$. 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 $g \circ g^{-1} = g^{-1} \circ g = id$; and
- $\circ$ is associative. That is, for all $f, g, h \in G, (f \circ g) \circ h = f \circ (g \circ h)$.

Definition 10.6. ***Order of a Group***
The order *of a group G is the number of elements in the set G. It is denoted by $|G|$.*

Example 10.7. The set of symmetries of a chessboard $\{id, x, y, d1, d2, r90, r180, r270\}$ form a group of order 8. We have that $r90^{-1} = r270$, $r270^{-1} = r90$, and all other elements g are self-inverse, i.e. $g^{-1} = g$. The group of a chessboard is non-commutative, since $d1 \circ r90 = x$ but $r90 \circ d1 = y$. In most applications in constraint programming, the group is not commutative. Note that we omitted mention of the operation associated with the group, i.e. permutation composition: this is often done where it will not cause confusion.

Note that our (entirely standard) definition of a group nowhere mentions the action done by the group element. It is vital to understand that group elements can operate in two distinct ways. First, there is the action that a group element (i.e. symmetry) has on the points that it acts on. In the chessboard example, the points were $1 \ldots 9$, and we wrote p^g for the result of the action of g on point p. This is what we have emphasised up to now. Second, a group element g operates by the composition operator to permute the values of other elements in the group. That is, $f \circ g$ gives another group element. The latter kind of operation is the focus of most study in group theory. In contrast, it is the group action which is of far more importance to us, since it is this action which represents the function of a symmetry on the variables and values in a constraint problem.

For permutations, the operation $\circ$ is composition as described above. We have already shown that there is an identity permutation, all permutations have inverses, and that composition of permutations is associative. The final condition is *closure*. We have shown that the composition of two permutations is another permutation. However, for a set of permutations to form a group, we have to have that the composition of any two permutations is in the set. This depends on the set of permutations we have chosen. There is an easy way to guarantee closure, which is to take a set of permutations and *generate* all permutations which result from composing them arbitrarily.

Definition 10.8. ***The Generators of a Group***
Let S be any set of elements (for example, permutations) that can be composed by the group operation $\circ$ (for example, permutation composition). The set S generates *G if every element of G can be written as a product of elements in S and every product of any sequence of elements of S is in G. The set S is called a* set of generators *for G and we write $G = \langle S \rangle$.*

Example 10.9. Generators of Chessboard Symmetries
The chessboard symmetries are generated by $\{r90, d1\}$ since:

$$id = r90 \circ r90 \circ r90 \circ r90 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

$$r90 = r90 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 4 & 1 & 8 & 5 & 2 & 9 & 6 & 3 \end{pmatrix}$$

$$r180 = r90 \circ r90 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix}$$

$$r270 = r90 \circ r90 \circ r90 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 6 & 9 & 2 & 5 & 8 & 1 & 4 & 7 \end{pmatrix}$$

$$d1 = d1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 1 & 4 & 7 & 2 & 5 & 8 & 3 & 6 & 9 \end{pmatrix}$$

$$y = d1 \circ r90 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 7 & 8 & 9 & 4 & 5 & 6 & 1 & 2 & \end{pmatrix}$$

$$d2 = r90 \circ r90 \circ d1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 9 & 6 & 3 & 8 & 5 & 2 & 7 & 4 & 1 \end{pmatrix}$$

$$x = r90 \circ d1 = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 3 & 2 & 1 & 6 & 5 & 4 & 9 & 8 & 7 \end{pmatrix}$$

Given any set of permutations, we can always work with the group generated by that set, since it is by definition closed. Finding and working with generators can be a very important means of representing groups. If there are $|G|$ elements in a group, there is always a generating set of size $\log_2(|G|)$ or smaller.

Definition 10.10. ***Subgroup***
A subgroup *H of a group G is a subset of G that is itself a group, with the same composition operator as G. Two simple and universal examples of subgroups are that G is always a subgroup of G, as is $\{id\}$.*

Example 10.11. Subgroup of Chessboard Symmetry:
The set $\{id, r90, r180, r270\}$ form a subgroup of order 4. As can be seen from Example 10.1, this can be generated by the element $r90$.

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 constructed with different elements are either disjoint or the same, i.e. if $H \circ f \cap H \circ g \neq \emptyset$ then $H \circ f = H \circ g$. 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, this set is the *right transversal*. The group G is the union of the cosets formed by composing the elements of H with these coset representatives.

Example 10.12. 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 $H = H \circ d1$. One set of coset representatives is $\{id, d1\}$, but there are a total of 16 possible sets of coset representatives, comprising one element from each coset.

The *orbit* of a point in G is a set of the different points that the point can be mapped to by elements of G.

Definition 10.13. ***Orbit***
The orbit of a point δ in G is the set $\delta^G = \{\delta^g \mid g \in G\}$.

Example 10.14. Orbits of Points on Chessboard
Looking back at the diagram of chessboard symmetries given in Example 10.1, the orbits of a given point can be calculated. For example, the orbit of 1 is $\{1, 3, 7, 9\}$, because $1^{id} = 1$, $1^{r90} = 7$, $1^{r180} = 9$, $1^{r270} = 3$, and because all other group elements map 1 to one of these points.

The *stabiliser* of a point is the set of elements which fixes or stabilises the point. It indicates which elements can be applied to a point, which do not cause the value of the point to move.

Definition 10.15. ***Stabiliser***
Let G be a permutation group acting on (amongst others) a point β. The stabiliser *of β in G is defined by: $G_\beta = \{g \in G | \beta^g = \beta\}$. The stabiliser G_β is a subgroup of the group G.*

Example 10.16. Stabilisers of chessboard symmetries
From Figure 10.2 of chessboard symmetries, the stabiliser of any given point can be identified. For instance, the stabiliser of point 1 is $G_1 = \{id, d1\}$ as these elements map point 1 back to itself. The stabiliser of point 5 is the whole group G, i.e. $G_5 = G$, since none of the symmetries move point 5.

10.1.1 Group Theory in Constraint Programming

Whenever a constraint problem has symmetry, we can construct a group to represent the symmetry in the problem. The elements of the group permute points, dependent on the symmetries in the particular problem. The points that the elements of the group act on will typically be variable-value pairs.

Example 10.17. Representing the Symmetry of the Chess problem in Figure 10.1
Variables - The CSP has n^2 variables corresponding to the squares of the chessboard, to represent the variables requires n^2 labels
Values- There are 5 possible values for each square of the chessboard: a white queen, a black queen, a white king, a black queen, or it is empty.
Variable-Value - There are n^2 possible labels for the variables, and 5 possible labels for the variables, to represent variable-value pairs requires $5n^2$ points.

There is one particularly important group of permutations. The set S_n of *all* permutations of n objects forms a group and is called the *symmetric group* over n elements. The group S_n is of size $n!$. This group comes up frequently in constraints because we often have n objects which are indistinguishable between each other, and which are thus acted on by S_n. It also arises in combinations. For example, we will see that a commonly occurring situation is a two-dimensional matrix of variables, where we can freely permute m columns (preserving the rows) and n rows (preserving the columns). Since we can first permute the columns and then the rows, the group we have is a combination known as the *direct product* $S_m \times S_n$.

10.1.2 Computational Group Theory

Computational Group Theory is a large interdisciplinary research area in mathematics and computer science. Butler has written a text book on the algorithms used in this area [17] and Holt *et al.* [65] have recently written a handbook of computational group theory. There are two major packages for computational group theory called GAP [46] and Magma [14]. Constraint programmers using symmetry can often employ GAP or Magma as an external function. However, if that is not available or appropriate, understanding of computational group theory algorithms is important. The most important algorithm of all is the Schier Sims algorithm, for which we give an extremely brief outline.

The Schreier Sims algorithm [108] is used to construct a stabiliser chain G_0, G_1, ..., G_n as follows:

$$\begin{array}{rcl} G_0 & = & G \\ \forall i \in I^n,\ G_i & = & G_{i-1} \end{array}$$

By definition,

$$G_i = \{\sigma \in G : 0^\sigma = 0 \wedge \ldots \wedge (i-1)^\sigma = i-1\}$$
$$G_n \subseteq G_{n-1} \subseteq \ldots G_1 \subseteq G_0$$

The Schreier Sims algorithm also computes set of coset representatives U_i. Those are orbits of i in G_i: $U_i = i^{G_i}$.

By definition, U_i is the set of values which i is mapped to by all symmetries in G that leave at least $0, \ldots, (i-1)$ unchanged.

In constraint programming terms the stabiliser is perhaps the most useful concept of those outlined above. The stabiliser of a variable/value pair shows what symmetry is left unbroken once that value is assigned to the given variable, during search. This is explained in more detail, with regards to GAP-SBDS, in Section 10.5.4. The stabiliser chain represents the symmetry which remains, after a collection of variables have been assigned during search.

10.2 Definitions

It might seem self-evident that in order to deal with symmetry in constraint satisfaction problems (CSPs), practitioners must first understand what is meant by symmetry. This appears not to be true: many papers on the topic do not offer a precise definition of what a symmetry is. The papers which do offer definitions often give fundamentally different ones to each other, while still identifying the same symmetries in a given problem and dealing with them correctly. There are two broad types of definition: those that define symmetry as a property of the solution set and those that define symmetry as a property that can be identified in the statement of the problem, without solving it. These will be referred to as solution symmetry and problem symmetry. In this section we give a brief survey of the definitions contained in the literature, before concentrating in more depth on recent definitions proposed by Cohen *et al* [21].

An example of a definition of solution symmetry in CSPs is given by Brown, Finkelstein & Purdom [16]: “A symmetry is a permutation that leaves invariant the set of solutions sequences to a problem.” Backofen and Will [5] allow for a much broader class of problem transformations: “A symmetry S for a constraint program C_{Pr}, where a set of solutions for a given problem is denoted $\| C_{Pr} \|$, is a bijective function such that S: $\| C_{Pr} \| \rightarrow \| C_{Pr} \|$.” Although not explicitly stated, Backofen and Will allow a symmetry to be specified by its effect on each individual assignment of a value to a variable, this allows them to consider symmetries with regard to partial assignments.

Many definitions define restricted forms of symmetry that affect only the variables or only the values. Interchangeability, as outlined in Definition 10.18 by Freuder [39], is a limited form of solution symmetry, which only operates over values.

Definition 10.18. *Two values a, b for a variable v are fully interchangeable iff every solution to the CSP containing the assignment $\langle v, a\rangle$ remains a solution when b is substituted for a, and vice versa.*

As Freuder notes, in general identifying fully interchangeable values requires finding all solutions to the CSP. He defines local forms of interchangeability that can be identified by inspecting the problem. Definition 10.19 outlines neighbourhood interchangeability, which is a form of constraint symmetry.

Definition 10.19. *Two values a, b for a variable v are neighbourhood interchangeable iff for every constraint C on the variable v, the set of variable-value pairs that satisfies the constraint with the pair $\langle v, a\rangle$ is the same as the set of variable-value pairs that satisfies the constraints with the pair $\langle v, b\rangle$.*

Choueiry and Noubir extend the idea of interchangeability to compute another form of local interchangeability, and showed how to exploit these results in practice [19].

Benhamou [8] extends the ideas of value interchangeability slightly and distinguishes between *semantic* and *syntactic* symmetry in CSPs, corresponding to solution symmetry and problem symmetry respectively. He defines two kinds of semantic symmetry. Two values a_i and b_i for a CSP are symmetric for satisfiability if: there is a solution which contains the value a_i iff there is a solution which contains the value b_i. Two values a_i and b_i are symmetric for all solutions if: each solution containing the value a_i can be mapped to a solution containing the value b_i, and vice versa. If two values are symmetric for all solutions they are also symmetric for satisfiability. Identifying semantic symmetries requires solving the CSP to find all solutions, and then examining them. Benhamou defines syntactic symmetry to mean that the permutation does not change any constraint relation, defined as a set of tuples.

The notion of interchangeable values has been and is still widely used and studied, e.g. [72, 55]. However, for the purposes of this overview, we regard interchangeability as a kind of *value symmetry*. Thus, we often discuss methods below which can be applied to interchangeable values, but do not point this out explicitly.

In contrast to this value centric approach, variable centric definitions have also been proposed. In CSPs, permuting the variables in a constraint defined intensionally will in general change the constraints, e.g. the constraint $x+y = z$ is not the same as the constraint $x+z = y$. Puget [97] defines the notion of a symmetrical constraint, i.e. a constraint which is not affected by the order of the variables. For instance, the $\neq$ constraint is symmetrical. Puget’s definition means that a symmetry of a CSP is a permutation of the variables which maps the set of constraints into a symmetrically equivalent set: any constraint is either unchanged by the permutation or is an instance of a symmetrical constraint and is mapped onto a constraint on the same set of variables.

A similar idea was introduced by Roy and Pachet [105]. They define the notion of *intensional permutability*. For two variables to be intensionally permutable they must have the same domain; any constraints affecting either of the two variables must affect both; and the two variables must be interchangeable in these constraints. The constraint is assumed to be defined intensionally, i.e. in terms of a formula, hence the name. An example of intensional permutability can be given by considering a linear constraint: in this case any two variables with the same coefficient and the same domain are intensionally permutable with respect to that constraint.

The definitions of symmetry given by Puget [97] and Roy & Pachet are restricted to permuting variables of the problem. Meseguer and Torras [84] give a definition of symmetry which acts on both the variables and the values of a CSP. Their definition allows both variable symmetries (that permute only the variables) and value symmetries (that permute only the values) as special cases. However, it does not fit every transformation of a CSP that we would want to recognise as a symmetry. Meseguer and Torras use the chessboard symmetries as an example, in a commonly used CSP formulation where only one piece is placed per row, the variables correspond to the rows of the chessboard and the values correspond to the columns. They show that reflection through 180° is a symmetry of the CSP by their definition, but four symmetries are not: reflection in the diagonals, rotation through 90°, and 270°.

McDonald and Smith [82] state that "a symmetry of P is a bijective function $\sigma : A \rightarrow A$ where A is some representation of a state in search e.g. a list of assigned variables, a set of current domains etc., such that the following holds: 1. Given A a partial or full assignment of P, if A satisfies the constraints C, then so does $\sigma(A)$; and 2. Similarly, if A is a nogood, then so too is $\sigma(A)$." This allows symmetries operating on both the variables and values; it gives a good intuitive view of problem symmetry space. However, due to the undefined nature of A it does not provide a rigorous definition, that could be used to identify the symmetry of a problem.

The above survey of symmetry definitions shows that symmetry definitions differ both on what aspect of the CSP they act on (only the variables, only the values or variable-value pairs) and in what they preserve (the constraints or the set of solutions). All definitions agree that symmetries map solutions to solutions; they disagree over whether this is a defining property, so that any bijective mapping of the right kind that preserves the solutions must be a symmetry, or a consequence of leaving the constraints unchanged.

Defining symmetry as preserving the set of solutions does not seem to offer a practical route to identifying symmetry in CSPs. Detecting semantic symmetries is, unsurprisingly, intractable [85, 111]: to find the full symmetry group we might need all the solutions to the CSP. On the other hand, the solution symmetry group is well-defined, whereas equivalent CSPs differing only slightly in the way that constraints are expressed, may have different problem symmetries. It may be possible, either deliberately or inadvertently, to write the constraints of a CSP in such a way that the symmetry of the problem being modelled is not apparent.

For the purpose of this chapter a definition of both solution symmetry, Definition 10.20, and a definition of problem symmetry, Definition 10.21 are given. These are in the spirit of [21], but are less formally defined.

Definition 10.20. ***Solution Symmetry***
A solution symmetry is a permutation of the set of $\langle variable, value \rangle$ pairs which preserves

the set of solutions.

Definition 10.21. ***Problem Symmetry***
A problem symmetry is a permutation of the set of $\langle variable, value\rangle$ pairs which preserves the set of constraints.

Both problem and solution symmetry allow variable and value symmetries as special cases.

In order for Definition 10.21 to be complete, a suitable interpretation of what it means to preserve the sets of constraints is needed. Any constraint c_i with scope $V_i \subseteq V$ can be defined by a set of satisfying $\langle variable, value\rangle$ tuples. A symmetry whose action on the set of possible $\langle variable, value\rangle$ tuples has been specified can be applied to the set of $\langle variable, value\rangle$ tuples satisfying a constraint, yielding a new set of $\langle variable, value\rangle$ tuples. The resulting $\langle variable, value\rangle$ tuples may not all relate to the same set of variables as the original constraint or each other. However, if the results of applying the symmetry to all the $\langle variable, value\rangle$ tuples, defining all the constraints is the same set of $\langle variable, value\rangle$ tuples, it can be said that the constraints are unchanged by the action of the symmetry. This definition does not require a symmetry to leave each individual constraint unchanged, but rather the set of constraints.

A recent paper by Cohen *et al.*, looks more closely at the differences between Solution Symmetry and Problem Symmetry (which they call Constraint Symmetry) [21]. Cohen *et al.* both give more rigorous definitions of the two concepts, and show the difference between the two definitions in practice.

10.3 Reformulation

Modelling has a substantial effect on how efficiently a problem can be solved. An appropriate reformulation of a model can turn an infeasible problem in practical terms into a feasible one. Modelling and reformulation are equally important for symmetry breaking. Different models of the same problem can have different symmetries; one formulation can have symmetries which are easier to deal with than another. In extreme cases, one formulation can have no symmetry at all. In other cases, the amount of symmetry can be greatly reduced from one model to another. Moreover, once a problem has been reformulated the remaining symmetries can still be dealt with before or during search, while other symmetry breaking methods can lead to great difficulties in combination with each other. Thus, reformulation of a problem can be critical in dealing with symmetries.

For a first example we mention the well known “social golfers problem”, problem 10 in CSPLib, although we will only sketch the issues here since Smith goes into some detail in her chapter.[2] In this problem, 32 golfers want to play in 8 groups of 4 each week, so that any two golfers play in the same group at most once, for as many weeks as possible, the difficult case being 10 weeks. We can construct an otherwise sensible model with $32!10!8!^{10}4!^{80}$ symmetries: we can permute the 32 players; we can permute the 10 weeks; within each week we can (separately) permute the groups; and within each group we can (separately) permute the four players. In this model there are more than 10^{198} symmetric versions of each essentially different solution, and there is a very good chance that search

[2]“Modelling”, by Barbara Smith, this Handbook.

will thrash impossibly. By remodelling, Smith reduces this number to $32!10!$ [113]. For each pair of players we have a variable indicating which week they play together in (or an extra variable if they never meet): the only symmetries left are the week and the player symmetries. There are still huge numbers of symmetries left, but they are of a much simpler form.

One important technique is the use of *set variables* where we have a number of indistinguishable variables. Set variables are available in most constraint solvers, and allow us to express constraints on sets' size, intersection, union, etc. In the golfers' problem, for example, one might encode the groups playing within each week as being eight set variables. Each one is constrained to be of size 4 and they are pairwise constrained to have no intersection. Because set variables are implemented with no implicit ordering between elements, we have lost the $4!$ symmetries in each group, reducing the total symmetries by a factor of 24^{80} in the problem. Furthermore, in the golfers' problem we can represent the constraint that two players play together no more than once, by saying that the intersection of any two groups in different weeks is of size 1 or 0. There are some theoretical and practical difficulties associated with set variables. One is that different representations of set variables by the solver have dramatically different behaviours in propagation [67]: if a solver happens to use the representation least suitable for the constraints being used, search can be dramatically increased. Another difficulty occurs when we wish to use a mixture of set and integer variables. We may have constraints on elements of a set that are most natural to express on integer variables. "Channelling" between set and integer variables can be difficult and can again lead to a failure to propagate until late in search. Alternatively, we might re-introduce integer variables to represent the elements of the set, thereby bringing back many of the symmetry problems that the set variables avoided in the first place. Despite these potential disadvantages, set variables remain a very important modelling technique to consider in any problem where a number of variables have the symmetry group S_n. Because of this, representation of set variables and propagation techniques for them are an important area of study [56, 67, 75, 106]. Another chapter in this Handbook describes set variables in detail.[3]

Another example of reformulation again illustrates the dramatic improvements that can be achieved, while even more dramatically illustrating the extent to which reformulating is an art more than a science. The all-interval series problem (problem 7 in CSPLib) is to find a permutation of the n integers from 0 to $n-1$ so that the differences between adjacent numbers are also a permutation of the numbers from 1 to $n-1$. There are 4 obvious symmetries in the problem: the identity, reversing the series, negating each element by subtracting it from $n-1$, and doing both. Gent et al [54] report on a reformulation of the problem based on the observation that we can cycle a solution to the problem about a pivot to generate another solution. The location of this pivot is dependent on the assignments made. As an example, here are two solutions for $n = 11$. Differences are written underneath the numbers:

```
0 10 1 9 2 8 3 7 4 6 5            3 7 4 6 5 0 10 1 9 2 8
 10 9 8 7 6 5 4 3 2 1              4 3 2 1 5 10 9 8 7 6
```

The difference between the first number (0) on the left and last number (5) is 5. This means we can split the sequence between the 8 and 3, losing the difference 5. We can join the

[3]"Constraints over Structured Domains", by Carmen Gervet, this Handbook.

rest of the sequence on to the start, because the $5 - 0$ will now replace $8 - 3$. This yields exactly the solution shown on the right. In this case the pivot is between the values 8 and 3. The difference between first and last terms must always duplicate a difference in the sequence, so this operation can be applied to any solution. Because of this, we do not merely reformulate the constraint model, but actually move to solve a different problem, whose solutions lead to solutions of the original. In the reformulated problem, we find a permutation of the sequence $0, 1, \ldots n - 1$, but we now include the difference between first and last numbers, giving n differences instead of $n - 1$. The sequence has to obey two constraints: that the permutation starts $0, n - 1, 1$; and that the n differences between consecutive numbers contain all of $1, \ldots n-1$ with one difference occurring exactly twice. [50] show that (for $n > 4$) each solution gives 8 distinct solutions to the all-interval series problem, but the reformulation has no symmetry at all. Search in this model is about 50 times faster than any competing technique.

It is possible to take advantage of different 'viewpoints' [73] of a constraint problem. To take a simple example, suppose we insist that n variables, each with the same n values, must all take different values. We can look at this problem from two points of view: we can find values for each variable, or we can find variables for each value. If there is symmetry, then value symmetry in the first viewpoint is interchanged with variable symmetry in the second viewpoint, and vice versa. This is useful if we have at hand a technique which is good at one kind of symmetry: for example Roney-Dougal *et al* [103] used this idea to break a group of variable symmetries using an efficient technique for value symmetries. Flener et al [36]. showed how value symmetries in matrix models can be transformed to variable symmetries by adding a dimension of 0/1 variables to the matrix, the new symmetries being broken using techniques described in Section 10.4.5 below. Law and Lee studied this idea theoretically and generalised it to cases where the translation is not to 0/1 variables [74].

Recently, there has been one significant advance in understanding how reformulation can be applied mechanically. Prestwich has shown that value symmetries can be eliminated automatically by a new encoding from constraints into SAT, the 'maximality encoding' [93]. This breaks all value symmetries of a special kind Prestwich calls 'Dynamic Substitutability', a variant of Freuder's value interchangeability [39]. A particular important aspect of the contribution is that Prestwich's encoding eliminates all dynamic substitutability without any detection being necessary. This means that no detection program needs to be run, nor does the constraint programmer need to specify the symmetry in any form. A disadvantage of this technique is its limitation to certain types of value symmetry. However, as mentioned above, any remaining symmetry in the translated SAT problem can be detected and broken using standard SAT techniques.

Sadly, for a method with so many advantages, there is very little we can say about reformulation because there is no fully general technique known. Not only that, but the illustrative examples above show considerable insight about their respective problems. Also, there was no guarantee (before running the relevant constraint programs) that they would lead to improved search. About the only truly general comment we can make is that the very great importance of formulating problems to reduce symmetry is not fairly reflected in the short space we devote to it in this chapter. There is a wonderful research opening for the discovery of general techniques akin to the lex-leader method for adding constraints, moving the area of reformulation from a black art to a science where questions were on tradeoffs and implementation issues, rather than the need for magical insights.

10.4 Adding Constraints Before Search

Without doubt, the method of symmetry breaking that has been most used historically involves adding constraints to the basic model. In this context, the term “symmetry breaking” is entirely appropriate. We move from a problem with a lot of symmetry to a new problem with greatly reduced symmetry – ideally with none at all. The constraints we add to achieve this are called “symmetry breaking constraints”.

Constraint programmers have always added symmetry breaking constraints in an ad hoc fashion when they have recognised symmetry in a constraint problem. Often it is easy to think of constraints that break all or a large part of symmetry. For example, suppose that we have 100 variables in an array X which are indistinguishable (so that they can be freely permuted). It is straightforward, and correct, to insist that the variables are in nondecreasing order, $X[1] \leq X[2] \ldots \leq X[100]$. If we further know that all variables must be different, we can make this strictly increasing order: $X[1] < X[2] \ldots < X[100]$. If it happens that the variables take the values 1..100, then simple constraint propagation will deduce that $X[1] = 1, X[2] = 2, \ldots, X[100] = 100$. If the programmer notices this beforehand, then we can reformulate the problem to replace each variable $X[i]$ with the value i throughout our program. There are many examples where constraint programmers have added more complicated constraints to break symmetries. A typical example from the literature is adding constraints to break symmetry in the template design problem [96]. This is fine if done correctly, but can obviously lose solutions if done incorrectly. Standard methods have been developed which can make the process easier and more likely to be correct, in the situations where they apply. Even where these are not directly applicable, a knowledge of them will serve constraint programmers well, as it should simplify the derivation of correct constraints which can be added by hand.

10.4.1 The Lex-Leader Method

Puget [97] proved that whenever a CSP has symmetry, it is possible to find a ‘reduced form’, with the symmetries eliminated, by adding constraints to the original problem. Puget found such a reduction for three simple constraint problems, and showed that this reduced CSP could be solved more efficiently than in its original form. Following this, the key advance was to show a method whereby such a set of constraints could be generated. Crawford, Ginsberg, Luks and Roy outlined a technique, called “lex-leader” for constructing symmetry-breaking ordering constraints for variable symmetries [22]. In later work, Aloul et al also showed how the lex-leader constraints for symmetry breaking can be expressed more efficiently [2]. This method was developed in the context of Propositional Satisfiability (SAT), but the results can also be applied to CP.

The idea of lex-leader is essentially simple. For each equivalence class of solutions under our symmetry group, we will predefine one to be the canonical solution. We will achieve this by adding constraints before search starts which are satisfied by canonical solutions and not by any others.

The technique requires first choosing a static variable ordering. From this, we induce an ordering on full assignments. The ordering on full assignments is straightforward. The tuple is simply the values assigned to variables, listed in the order defined by our static ordering. Since the method is defined for variable symmetries, any permutation g converts this tuple into another tuple, and we prefer the lexicographically least of these. This method

$$\begin{array}{|c|c|c|}\hline \text{A} & \text{B} & \text{C} \\ \hline \text{D} & \text{E} & \text{F} \\ \hline \end{array} \quad \rightarrow \quad \begin{array}{|c|c|c|}\hline \text{F} & \text{E} & \text{D} \\ \hline \text{C} & \text{B} & \text{A} \\ \hline \end{array}$$

1.	ABCDEF $\preceq_{\text{lex}}$ ABCDEF			7.	ABCDEF $\preceq_{\text{lex}}$ DEFABC		
2.	ABCDEF $\preceq_{\text{lex}}$ ACBDFE			8.	ABCDEF $\preceq_{\text{lex}}$ DFEACB		
3.	ABCDEF $\preceq_{\text{lex}}$ BACEDF			9.	ABCDEF $\preceq_{\text{lex}}$ EDFBAC		
4.	ABCDEF $\preceq_{\text{lex}}$ CBAFED			10.	ABCDEF $\preceq_{\text{lex}}$ FEDCBA		
5.	ABCDEF $\preceq_{\text{lex}}$ BCAEFD			11.	ABCDEF $\preceq_{\text{lex}}$ EFDBCA		
6.	ABCDEF $\preceq_{\text{lex}}$ CABFDE			12.	ABCDEF $\preceq_{\text{lex}}$ FDECAB		

Figure 10.5: A 3×2 matrix containing 6 variables; and the result of swapping the two rows and reversing the columns, giving the permutation mapping ABCDEF to FEDCBA. Also shown are the 12 lex-leader constraints, including the trivial one, arising from its 12 symmetries. Note how each constraint corresponds to a permutation of the variables, as the method is defined to work for variable symmetries: the illustrated matrix transformation gives constraint 10.

is, in principle, simple to implement. Each permutation in the group gives us one $\preceq_{\text{lex}}$ constraint. So the set of constraints defined by the lex-leader method is

$$\forall g \in G, \; V \preceq_{\text{lex}} V^g \tag{10.1}$$

where V is the vector of the variables of the CSP, and $\preceq_{\text{lex}}$ is the lexicographic ordering relation. The lexicographic ordering is exactly as is standard in computer science, e.g. $AD \preceq_{\text{lex}} BC$ iff either $A < B$ or $A = B$ and $D \leq C$.

A small example illustrates the method. Consider a 3×2 matrix depicted in Figure 10.5, in a context where the rows and columns may be freely permuted. The symmetries form the group $S_3 \times S_2$, with order $3!2! = 12$. We pick the variables in alphabetical order, so the vector of the variables of the problem is ABCDEF. The 12 symmetries lead to the 12 lex-leader constraints shown in Figure 10.5, including the vacuous symmetry from the identity.

An important practical issue with the lex-leader constraints is that they do not "respect" the variable and value ordering heuristics used in search. That is, it may well be that the leftmost solution in the search tree, which would otherwise be found first, is not canonical and so is disallowed, leading to increased search. This is in contrast to techniques such as SBDS and SBDD (Sections 10.5.1 and 10.5.2), which do respect the heuristic. Simple examples have been reported where the "wrong" heuristic can lead to dramatic increases in runtime [52]. This problem is inherent in the method, but in many cases it is easy to work out what is the "right" heuristic. In particular, if the same static variable ordering is used in search as was used to construct the lex-leader ordering, and values are tried from smallest to largest, this conflict should not occur. However, this does limit the power of the constraint programmer to use dynamic variable ordering heuristics.

A less easily solved problem with lex-leader is that many groups contain an exponential number of symmetries. Lex-leader requires one constraint for each element of the group. In the case of a matrix with m rows and n columns, this is $m!n!$, which is impractical in general. Therefore there are many cases where lex-leader is applicable but impractical.

1.	*true*	7.	ABC $\preceq_{\text{lex}}$ DEF
2.	BE $\preceq_{\text{lex}}$ CF	8.	ABC $\preceq_{\text{lex}}$ DFE
3.	AD $\preceq_{\text{lex}}$ BE	9.	ABC $\preceq_{\text{lex}}$ EDF
4.	AD $\preceq_{\text{lex}}$ CF	10.	ABC $\preceq_{\text{lex}}$ FED
5.	ABDE $\preceq_{\text{lex}}$ BCEF	11.	ABCDE $\preceq_{\text{lex}}$ EFDBC
6.	ABDE $\preceq_{\text{lex}}$ CAFD	12.	ABCDE $\preceq_{\text{lex}}$ FDECA

Figure 10.6: The lex-leader constraints for the 3×2 matrix reduced on an individual basis.

2.	BE $\preceq_{\text{lex}}$ CF	9.	ABC $\preceq_{\text{lex}}$ EDF
3.	AD $\preceq_{\text{lex}}$ BE	10.	ABC $\preceq_{\text{lex}}$ FED
7.	ABC $\preceq_{\text{lex}}$ DEF	11.	ABCD $\preceq_{\text{lex}}$ EFDB
8.	ABC $\preceq_{\text{lex}}$ DFE	12.	ABC $\preceq_{\text{lex}}$ FDE

Figure 10.7: The lex-leader constraints for the 3×2 matrix reduced as a set.

However, lex-leader remains of the highest importance, because there are a number of ways it is used to derive new symmetry breaking methods. We discuss these in the following sections.

Finally we reiterate that the lex-leader method is defined only for variable symmetries: i.e. those which permute the variables but always leave the value unchanged. Thus the same restriction applies to the methods below based on lex-leader. It is not an issue in the technique's original domain, SAT, since there are only 2 values [22]. If necessary, we can add a new variable to represent the negation of each variable, and so symmetries which change values can be made into variable symmetries. Unfortunately, if we have d values, we need $d!$ versions of each variable to apply this simple idea. Therefore, a proper generalisation of lex-leader to deal with value symmetries would be valuable, even if restricted to some special cases.

10.4.2 Simplifying Lex-Leader Constraints

Lex-leader constraints can be simplified, or 'pruned' to remove redundancies [22, 77]. Following Frisch and Harvey [41], we can illustrate this using the example from Figure 10.5. The first idea is to look at each constraint individually. For example, consider constraint 2 above, ABCDEF $\preceq_{\text{lex}}$ ACBDFE. We can remove the first and fourth variables from each tuple, since clearly $A = A$ and $D = D$, giving BCEF $\preceq_{\text{lex}}$ CBFE. But if B<C the constraint is satisfied whatever the other values, and otherwise we have B=C to satisfy the constraint. In other words, if the second variables in the tuples are relevant, they must be equal. Similarly for E and F, so in fact the constraint is equivalent to BE $\preceq_{\text{lex}}$ CF. Applying this reasoning everywhere we get the constraints shown in Figure 10.6. It is interesting to note that constraints 2 and 3 show that the columns must be lexicographically ordered, and that constraint 7 forces the rows to be lexicographically ordered. We return to this observation in Section 10.4.5.

We can go further, treating the constraints as a set and not just individually. For example, $\preceq_{\text{lex}}$ is transitive, so constraints 2 and 3 imply constraint 4. A more complicated example is in constraint 11. The last elements of each tuple are E and C. But if they are relevant, we have A=E and B=F=D=C. But constraint 3 implies A≤B, from which it follows that E≤C, so the last elements of the tuple are irrelevant and may be deleted. This reasoning leads to a set of 8 constraints shown in Figure 10.7, equivalent to the original 12 [41].

Unfortunately, the approach outlined here does not get round the fundamental problem of the exponential number of symmetries. In general the number of symmetries even in the reduced set will still be exponential [77]. However, the approach does illustrate how the set of constraints can be simplified, and we will see in the next section a special case where the results are quite dramatic.

10.4.3 Symmetry with All-Different

The 'all-different' constraint occurs very commonly in constraint programming. It perhaps occurs even more often in problems with symmetry. Puget has shown that [101] if we have only variable symmetry (the only case where lex-leader is defined) on a set of n variables constrained by an all-different constraint, symmetry can be broken completely by only $n-1$ binary constraints. This result applies to *any* group G acting on the set of variables. This remarkable result could hardly be bettered, but is in fact relatively simple.

We begin with an example, which contains only variable symmetries on a set of all-different variables, but in which the group is not a straightforward group such as S_n.

Example 10.22. Graceful Graph
We say that a graph with m edges is *graceful* if there exists a labeling f of its vertices such that:

- $0 \leq f(i) \leq m$ for each vertex i,
- the set of values $f(i)$ are all-different,
- the set of values $abs(f(i), f(j))$ for every edge (i, j) are all-different.

A straightforward translation into a CSP exists where there is a variable v_i for each vertex v_i, see [78]. The variable symmetries of the problem are induced by the automorphism of the graph. There is one value symmetry, which maps v to $m-v$, but we ignore that symmetry to leave only value symmetries. More information on symmetries in graceful graphs is available in [89]. Petrie and Smith have considered various forms of both dynamic and static symmetry breaking methods in graceful graphs [89], using these techniques they found instances of graceful graphs that were not previously known. As an example, let us consider the the graph $K_3 \times P_2$, which is shown in Figure 10.8. The group allows any of the 3! permutations of K_3, as long as the same permutation is applied to both copies of K_3 at the same time, as well as swapping the two triangles. There are thus 12 symmetries. In fact, the group is isomorphic to that of the matrix in Figure 10.5, so the constraints are the same as we showed there.

Using the fact that the variable are subject to an all-different constraint, we can significantly reduce the number of symmetry breaking constraints. For example, consider the symmetry breaking constraint

$$ABCDEF \preceq_{\text{lex}} ACBDFE$$

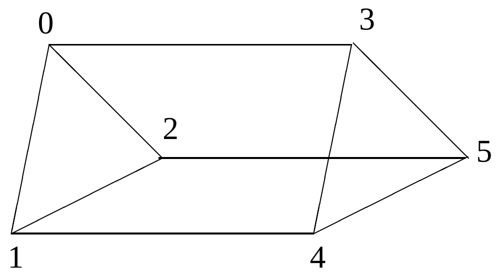

Figure 10.8: The graph $K_3 \times P_2$.

Since $A = A$ is trivially true, and since $B = C$ cannot be true because of the all-different constraint, this constraint can be simplified to just be:

$$B < C$$

This simplification is true in general and can be formalized as follows. Given a permutation g, let $s(g)$ be the smallest i such that $i^g \neq i$, and let $t(g)$ be equal to $(s(g))^g$.

Lemma 10.23. *[101]*
Given a CSP where the variables $\mathcal{V}$ are subject to an all-different constraint, and a variable symmetry group G for this CSP, then all variable symmetries can be broken by adding the following constraints:

$$\forall \sigma \in G, v_{s(\sigma)} < v_{t(\sigma)}$$

Note that if two permutations g and h are such that $s(g) = s(h)$ and $t(g) = t(h)$, then the corresponding symmetry breaking constraints are identical. Therefore, it is sufficient to state only one symmetry breaking constraints for each pair i, j such that there exists a permutation g with $i = s(g)$ and $j = t(h)$. The set of these pairs can be computed using the Schreier Sims algorithm [108]. In our example, these constraints are :

$$A < B, A < C, A < D, A < E, A < F, B < C$$

Note that these constraints are redundant. The constraint $A < C$ is entailed by the first and the last constraints. This remark can be used to reduce the number of constraints further by taking into account the transitivity of the $<$ constraints. The Schreier Sims algorithm gives us a stabiliser chain and sets of coset representatives U_i, as defined in Section 10.1.2. Puget uses this to prove:

Theorem 10.24. *[101]*
Given a CSP with n variables $\mathcal{V}$, such that there exists an all-different constraint on these variables, then all variable symmetries can be broken by at most $n - 1$ binary constraints.

For our example, we get exactly 5 constraints: notice this is $n - 1$ in this case.

$$A < B, A < D, A < E, A < F, B < C$$

While this is only a reduction of a single constraint, that is simply because of the small size of the example. In general, Puget has reduced the number of symmetries required from a possibly $n!$ to as little as $n-1$, for the commonly occurring case of variable symmetries in the presence of an all-different constraint. As well as its value in its own right, this shows the power of combining symmetry breaking constraints with constraints from a problem, and this remains an area ripe for exploitation.

10.4.4 Subsets of Lex Leader

The previous section showed how, in the right circumstances, a polynomial number of constraints can lead to equivalent reasoning to the full set of lex-leader constraints. Unfortunately, such a subset is not always available. Several researchers have proposed ways to state only a polynomial number of constraints without preserving complete symmetry breaking. Since the full set of lex-leader constraints leaves exactly one solution in each equivalence class, using a subset must leave *at least* one in each class, but may leave more than one. Thus, the symmetry breaking constraints do not guarantee to break all symmetries. In general the goal is to reach an acceptable tradeoff, with the greatly reduced number of constraints leading to more efficient search. In some cases, sets of symmetry breaking constraints have been proposed, and only later has it been realised that they represent a subset of the lex-leader constraints. This is a testament to the generality and naturalness of lex-leader.

Aloul et al have shown, in SAT, that very successful results can be obtained from very small subsets of lex-leader constraints, on examples such as FPGA routing problems [1]. Unlike the constraint-based work described so far, the symmetry group of the SAT problem was found using a graph-automorphism procedure on the instance. Surprisingly, the subset of symmetry breaking predicates used was simply the generators of the group found by the graph-automorphism check. It is remarkable that this gave good performance since, for example, a set of 21 generators was used in a group with 10^{16} elements. Only very special sets of generators are as effective as this, and it is not well understood what makes the generators found by graph-automorphism programs so good. While this is in a SAT context, similar results should apply to constraint programming.

Shlyakhter showed that good (though generally incomplete) subsets of lex-leader constraints could be found for acyclic digraphs, permutations, relations, and functions [109]. Apart from the individual contributions, this establishes the methodology of using the lex-leader constraints as a means of finding incomplete sets of symmetry breaking constraints, but subsets which are effective in practice. A particular case where this has proven to be of great interest is that of matrix models, to which we turn next.

10.4.5 Specialised Ordering Constraints for Matrix Models

A number of authors have taken a rather different approach to choosing appropriate constraints to break symmetry. This is based on the kind of symmetries that seem to arise very often in constraint programming. Given some class of symmetries we decide are important, we can analyse in general a subset of the lex-leader constraints which typically break a substantial number of symmetries and which can be reasoned with efficiently. The advantage of such a focus is that one can build special purpose methods for dealing with the symmetry breaking constraints, for example specialised algorithms for propagating Generalised Arc Consistency for the given set of constraints. Most work has concentrated on symmetry-breaking constraints for *matrix models*; where ‘a matrix model is a constraint program that contains one or more matrices of decision variables’ [36]. Matrix models are indeed very commonly occurring. For example the golfers problem can be modelled as a 3-d boolean matrix whose dimensions correspond to weeks, players and groups. A variable $x_{ijk} = 1$ iff in week i, player j plays in group k [113]. Other problems one could use as illustration are balanced incomplete block designs, steel meel slab design, progressive party problem, rack configuration, template design, and the warehouse location problem [35].

The prime example of this body of work is that on lexicographically ordering rows and columns in matrix models [36]. We deal with matrices where *rows* and *columns* are independently fully permutable. By this we mean that we can swap, at the same time, all the variables in any two rows, preserving the order of the variables within each row. Alternatively, we can swap any two columns, preserving the order of the variables within a column. This kind of symmetry occurs very commonly because we often introduce symmetry through modelling. For example, in the 3-d model of the golfers problem, the order of players, groups and weeks are arbitrary and each can be permuted freely. An $n \times m$ matrix with row and column symmetry is acted on by a group of order $n!m!$. These symmetries change the orders of rows and columns, but important relationships are preserved: specifically two elements in the same row are always in the same row, while two elements in the same column remain in the same column.

The rows in a 2-d matrix are *lexicographically ordered* if each row is lexicographically smaller (denoted $\preceq_{\text{lex}}$) than the next (if any). Adding lexicographic ordering on the rows breaks all row symmetries. Similarly, we can break all column symmetry by ordering the columns. But the interesting case is where we insist that both rows and columns should be simultaneously lexicographically ordered [36]. It is not at all obvious that this is consistent, i.e. that there is always a symmetry which will permute the rows and columns so that both sets are lexicographically ordered. In fact, we can remove solutions if we insist on the rows being in increasing order and the columns in decreasing order. However, it is always possible if we insist that all dimensions are in increasing order, and the best way to understand this is that the set of constraints are equivalent to a subset of the lex-leader constraints [109]. In general a lexicographic ordering on both the rows and the columns does *not* break all the compositions of the row and column symmetries. Nevertheless, in practice it often breaks a useful amount of symmetry. This is important, because in general it is NP-hard to find the lexicographical least representative of a matrix under row and column symmetry [22, 10].

Because of the usefulness of lex-ordering rows and columns, and the simplicity of the constraints, Frisch *et al* introduced an optimal algorithm to establish generalised arc-consistency for the $\preceq_{\text{lex}}$ constraint [40] between two vectors. Time complexity is $O(nb)$, where n is the length of the vectors and b is the time taken to adjust the bounds of an integer variable (dependent on the implementation of variables being used). This therefore gives an extremely attractive point on the tradeoff: a linear time to establish a high level of consistency on constraints which often break a lot of the symmetry in matrix models. The algorithm can be used to establish consistency in any use of $\preceq_{\text{lex}}$, so in particular is useful for any use of lex-leader constraints. Carlsson and Beldiceanu [18] showed that the propagation algorithm can be extended to generalised arc-consistency for a *chain* of vectors $V_1 \preceq_{\text{lex}} V_2 \preceq_{\text{lex}} \ldots \preceq_{\text{lex}} V_m$. This can deduce information not available if vectors are compared pairwise, and is done in linear time $O(nbm)$ if there are m vectors.

Provided care is taken, lex ordering matrices can be combined with additional constraints to break more symmetry while still taking advantage of the efficient algorithms for the lex constraints. Normally, this is done by choosing an appropriate subset of lex-leader constraints. This is an easy way to guarantee correctness provided that the same ordering of variables in the matrix is used as for the lex constraints: this means starting at the top left and going either in row order or column order. A very common technique is to insist that the top left hand corner is occupied by the (possibly equal) smallest element in the matrix: this is not guaranteed by the lex constraints themselves. If all values in the entire matrix are different, this additional constraint guarantees that all symmetry is broken [36]. There are other special cases where all symmetries are broken [36].

Other constraints have been developed in a similar way to the double-lex constraint, and applied to breaking symmetries in matrices. Kiziltan and Smith investigated the *multiset* ordering [70] following a suggestion of Frisch. The ordering is lexicographic ordering of the multiset of elements of vectors written in increasing order. More formally, we have $V_1 \preceq_{ms} V_2$ if either the smallest element of V_1 is less than the smallest element of V_2, or if the smallest elements are the same, and $V_1^- \preceq_{ms} V_2^-$, where the new vectors result from the deletion of a single occurrence of the smallest element. An advantage of the multiset ordering is that it can be placed on (say) the rows without affecting the column symmetries of a matrix. So it can for example be used even if the column symmetries do not form the group S_n. Frisch *et al* proposed a linear propagation algorithm for it, giving it similar efficiency to their algorithm for double-lex [43].

Another example of work in the same style was the introduction of the "allperm" constraint by Frisch, Jefferson and Miguel [42]. We have $V_1 \preceq_{perm} V_2$ if array V_1 is lexicographically less than any permutation of V_2. Frisch *et al* study the ways in which this may (and may not) be combined with other constraints such as double-lex and multiset. For example, it is consistent to insist that rows and columns are lex ordered, and simultaneously that the first row is $\preceq_{perm}$ all others, but we may not demand that the second row is $\preceq_{perm}$ later rows. Again, a specialised propagation algorithm was given for allperm.

We can construct 2-dimensional $2n \times 2n$ matrix model in which we insist that rows and columns are lex ordered, yet still contains full matrix symmetry in an $n \times n$ submatrix. An example for $n = 3$ is shown in Figure 10.9. The values of variables A to I are unconstrained, but if we insist that they form a latin square, the example is valid where rows are columns are ordered by the multiset ordering. If we also insist that all of A to I are ≥ 0, the example works if we insist that the first row is $\preceq_{perm}$ all other rows.

10.5 Dynamic Symmetry Breaking Methods

Dynamic symmetry breaking methods are those that operate to break symmetry during the search process. SBDD and SBDS are two such methods described in this section. In both these methods symmetry acts on variable/value pairs. Symmetry breaking by heuristic is included in this category, as although these variable and value ordering heuristics are fully defined before search commences, they are used during search. These methods are outlined in the subsequent sections.

$$\begin{array}{|c|c|c||c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ \hline \hline 0 & 0 & 1 & A & B & C \\ \hline 0 & 1 & 0 & D & E & F \\ \hline 1 & 0 & 0 & G & H & I \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c||c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ \hline \hline 1 & 0 & 0 & G & H & I \\ \hline 0 & 1 & 0 & D & E & F \\ \hline 0 & 0 & 1 & A & B & C \\ \hline \end{array}$$

$$\Downarrow$$

$$\begin{array}{|c|c|c||c|c|c|} \hline 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ \hline \hline 0 & 0 & 1 & G & H & I \\ \hline 0 & 1 & 0 & D & E & F \\ \hline 1 & 0 & 0 & A & B & C \\ \hline \end{array}$$

Figure 10.9: An example to show that full matrix symmetry in a bottom right submatrix can remain even when all rows and columns are lex ordered. The initial matrix (divided into quarters for clarity) is first transformed by swapping the fourth and sixth rows, and then by swapping the first and third columns. The result is still double-lex ordered but we have swapped the first and third rows of the submatrix. We can swap two columns similarly.

10.5.1 Symmetry Breaking During Search (SBDS)

Symmetry-excluding search trees were introduced by Backofen and Will [4, 5]. Gent and Smith [51] described in more detail the implementation of this technique, using the name "Symmetry Breaking During Search", but it is this latter name and its acronym "SBDS" which seems to have stuck as the general name for this method. This is perhaps slightly unfortunate since there are many other ways to break symmetry during search, most notably SBDD to be discussed in Section 10.5.2.

The basic idea of SBDS is to add constraints to a problem so that, after backtracking from a search decision, the SBDS constraints ensure that no symmetric equivalent of that decision is ever allowed. This is a *dynamic* technique, since we cannot add the constraints until we know what search decision is being made. In general terms, SBDS can work on any kind of search decision. However, for simplicity of discussion we will assume that all search decisions are of the form $var = val$, and a number of implementations of SBDS make the same assumption.

We will first illustrate with an example the kind of constraints added by SBDS. A search tree for the 8-Queens problem where the symmetry constraints added by SBDS are indicated by $\diamondsuit$ can be found in Figure 10.10. It is easiest to explain SBDS by going over the tree breadth-first, instead of the depth-first search the actual search algorithm would explore.

- Starting from the root, the first search decision in the search tree in Figure 10.10 is $Q[1] = 2$, i.e. the queen in row 1 goes in position 2. SBDS adds no constraints to the positive decision $Q[1] = 2$. If we backtrack at the root, we can assert $Q[1] \neq 2$. From that point on, we never want to try any state which contains any symmetric equivalent to $Q[1] = 2$. We can achieve this by adding symmetric versions of $Q[1] \neq$

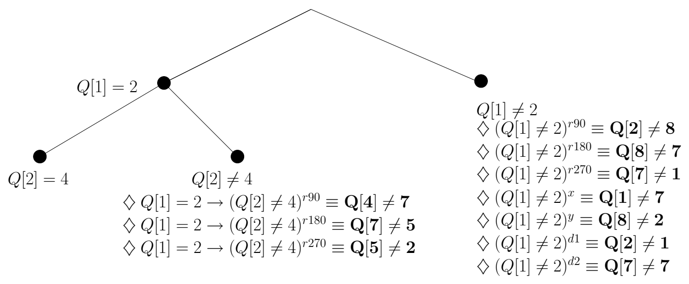
$Q[1] = 2$
$Q[1] \neq 2$
$\diamondsuit\ (Q[1] \neq 2)^{r90} \equiv \mathbf{Q[2] \neq 8}$
$\diamondsuit\ (Q[1] \neq 2)^{r180} \equiv \mathbf{Q[8] \neq 7}$
$\diamondsuit\ (Q[1] \neq 2)^{r270} \equiv \mathbf{Q[7] \neq 1}$
$\diamondsuit\ (Q[1] \neq 2)^{x} \equiv \mathbf{Q[1] \neq 7}$
$\diamondsuit\ (Q[1] \neq 2)^{y} \equiv \mathbf{Q[8] \neq 2}$
$\diamondsuit\ (Q[1] \neq 2)^{d1} \equiv \mathbf{Q[2] \neq 1}$
$\diamondsuit\ (Q[1] \neq 2)^{d2} \equiv \mathbf{Q[7] \neq 7}$
$Q[2] = 4$
$Q[2] \neq 4$
$\diamondsuit\ Q[1] = 2 \rightarrow (Q[2] \neq 4)^{r90} \equiv \mathbf{Q[4] \neq 7}$
$\diamondsuit\ Q[1] = 2 \rightarrow (Q[2] \neq 4)^{r180} \equiv \mathbf{Q[7] \neq 5}$
$\diamondsuit\ Q[1] = 2 \rightarrow (Q[2] \neq 4)^{r270} \equiv \mathbf{Q[5] \neq 2}$

Figure 10.10: Example of SBDS on a search tree with the 8-queens problem.

2, i.e. by adding $(Q[1] \neq 2)^g$ for each g in the group, although we can omit id if the search algorithm itself asserts $Q[1] \neq 2$. The result of this in the case of the chessboard symmetries is given on the right hand branch of Figure 10.10.

- At the next level of search, on the left hand side, the next search decision is $Q[2] = 4$. Again we add no constraints to this positive decision. But if we backtrack from this, we assert $Q[2] = 4$. It would no longer be correct to ban each symmetric version of this decision, because the initial search decision $Q[1] = 2$ may have broken some of the symmetry already. In this example, $Q[1] = 2$ rules out $Q[1] = 7$, $Q[7] = 7$, $Q[2] = 1$ and $Q[8] = 2$ since those squares are on the same row, column, diagonal, and diagonal respectively. This means that no solution containing $Q[1] = 2$ can possibly have the symmetries x, y, $d1$ or $d2$. There is no need to assert any constraints for these symmetries below $Q[1] = 2$, and they are omitted from Figure 10.10.

- Still considering the same node, the more complicated case is that of the symmetries $r90$, $r180$ and $r270$. At this point in search we do not know whether or not these symmetries apply. They may hold in some future states and not in others. We cannot rule out, for example, $(Q[2] = 4)^{r90}$ for all future states, as we might lose solutions in states where $r90$ does not hold, but equally we will do redundant search if we do not rule it out in states where $r90$ does hold. SBDS solves this dilemma by adding constraints which rule out $(Q[2] = 4)^{r90}$ conditionally. This conditional constraint states that if the $r90$ is not broken (i.e. $Q[2] \neq 8$) then we have $(Q[2] \neq 4)^{r90}$, i.e. $Q[4] \neq 7$.

We can now state in general the constraints that SBDS adds, consider a node in search where the partial assignment A is to be extended by the decision $var = val$. For any problem symmetry g, we can add the constraint:

$$A \;\&\; A^g \;\&\; var \neq val \;\Rightarrow\; (var \neq val)^g \tag{10.2}$$

To understand this constraint's soundness, note that it is equivalent to

$$(A \Rightarrow\; var \neq val) \Rightarrow (A \Rightarrow\; var \neq val)^g$$

which is almost a triviality. More significantly this equation is true before search begins, since it holds for any A, var, val and g. From this point of view we can view SBDS as making heuristic choices as to which of the (in practical terms) infinite variety of such constraints to add. Practical implementations of SBDS do not normally add the full form of (10.2). If we add the constraint at the point in search where we backtrack from the choice of $var = val$, then we know A is true and we also know that $var \neq val$, leading to the much simpler, but in context equivalent, form:

$$A^g \Rightarrow (var \neq val)^g \tag{10.3}$$

We can explain this simpler form by pointing out that it must be ensured that only unbroken symmetries are dealt with, so it is checked that A^g still holds. Then to ensure that the symmetrically equivalent subtree to the current subtree will not be explored, the $(var \neq val)^g$ is placed.

Backofen and Will proved that this method is sound in that the full non-symmetric search space will be explored [5], i.e. no solutions can be completely missed. Backofen and Will also showed that as long as all symmetries are correctly supplied, all the symmetry will be eliminated, i.e. no two solutions returned by SBDS can be equivalent.

A number of implementations of SBDS have been provided. The most publicly available to date is that in ECL^{i}PS^e [88]. These implementations always demand from the constraint programmer a separate function to implement the action of each symmetry g, in the programming language the system uses. If a problem has a large number of symmetries there may be too many for the user to identify and implement by hand. SBDS has been used successfully, despite this difficulty, with problems containing a few thousand symmetries [49]. We will see in Section 10.5.4 how computational group theory can be used to ease the burden on the programmer.

There are some important implementation issues for SBDS. 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. Since A^g involves all values set so far, it is potentially large, so checking that a symmetry is unbroken could be 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 $A_1^g = A^g + (var = val)^g$. So a Boolean variable can be constructed for each symmetry g representing whether A^g is satisfied or not: its value for A_1^g is the conjunction of its values for A^g and $var = val^g$. Hence, it can be decided incrementally whether A^g 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, than a large number of symmetry functions has to be described. In the worst case, there could be too many to be successfully compiled. This difficulty can be addressed by choosing a subset of the symmetry functions to use with SBDS. McDonald and Smith [82] explore this idea of ‘partial symmetry breaking’ with random subsets of symmetries, and also give an algorithm for choosing a subset of symmetry functions which should, heuristically, break a large amount of the symmetry. Unfortunately, it is infeasible to use the method in its entirety with all but the smallest problems with small symmetry groups.

SBDS has some major advantages over adding symmetry breaking constraints before search. First, the symmetries do not need to be variable symmetries, in contrast to the lex-leader method. Second, the solution found in each class of equivalent solutions is always the leftmost one in the search tree being traversed. Again in contrast to lex-leader, this means that arbitrary variable and value ordering heuristics may be used without changing either SBDS or the set of symmetry functions supplied. We say that SBDS "respects" the variable and value ordering heuristics used by the search.

10.5.2 Symmetry Breaking via Dominance Detection (SBDD)

The method of Symmetry Breaking via Dominance Detection (SBDD) was developed independently by Focacci & Milano [34], and by Fahle, Schamberger & Sellmann [33]. 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. A similar algorithm was proposed by Brown, Finkelstein and Purdom in 1988 [16]. In fact, this paper describes a computational group theoretic version of this algorithm which is now familiar to the CP community as GAP-SBDD (described below in Section 10.5.5). Unfortunately, the paper by Brown *et. al.*, while reasonably well known, seemed to have little influence on the constraints community, perhaps being too far ahead of its time.

```
                 0:root
                  /
              1:v1=1
            /        \
      2:v2=2          5:v2!=2
     /      \            /
3:v3=3   4:v3!=3   6:v2=3
                   /      \
              7:v3=2    8:v3!=2
```

node	decisions	v_1	v_2	v_3	solution
0	-	1,2,3,4	1,2,3,4	1,2,3,4	
1	$v_1 = 1$	1	2,3,4	2,3,4	
2	$v_1 = 1,\ v_2 = 2$	1	2	3,4	
3	$v_1 = 1,\ v_2 = 2,\ v_3 = 3$	1	2	3	yes
4	$v_1 = 1,\ v_2 = 2,\ v_3 \neq 3$	1	2	4	yes
5	$v_1 = 1,\ v2 \neq 2$	1	3,4	2,3,4	
6	$v_1 = 1,\ v2 \neq 2,\ v_2 = 3$	1	3	2,4	
7	$v_1 = 1,\ v2 \neq 2,\ v_2 = 3,\ v_3 = 2$	1	3	2	yes
8	$v_1 = 1,\ v2 \neq 2,\ v_2 = 3,\ v_3 \neq 2$	1	3	4	yes

Figure 10.11: A partial search tree to illustrate SBDD, and a table listing the search states corresponding to each node.

SBDD operates by performing a check at every node in the search tree to see if the node about to be explored is symmetrically equivalent to one already explored, and if so prunes this branch. While a simple idea, this has the apparent problem that we will need to store the whole, exponentially sized, tree already explored. A single key idea transforms SBDD into a space-efficient method. This is that we need only store nodes at the roots of fully explored subtrees. We do not check if a node is equivalent in full to one of our stored nodes. Instead, we determine if a node is equivalent to any node which is an extension of one of the stored nodes, i.e. a node which is in a fully explored subtree. Since search has backtracked, we must have either visited the equivalent node before, or deduced for some other reason that there was no need to visit it: in either case there is no need to visit a symmetric equivalent.

Like SBDS, SBDD is based on binary branching between setting a variable and removing the value from a variable's domain. An example of how SBDD works in practice, based on the example outlined in [34], here follows. Consider a problem with three variables v_1, v_2 and v_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. There are 24 solutions of the problem. Figure 10.11 shows a part of the tree search that will be explored by a depth first search procedure to enumerate all solutions, assuming only a basic all-different propagation procedure. Decisions are selected in a lexicographic ordering. Nodes are represented by $n : \delta$ where n means the node is the nth node to be expanded by the search procedure, and δ is the decision or the negation of a decision for the arc between a node n and its parent. The Figure also gives for each node n the set of decisions taken on the path from the root node to n, as well as the domains of the variables corresponding to its state. Four solutions have been obtained in the illustrated search. However the solution found at node 7 is symmetrical with the one found at node 3. Those solutions are $\{v_1 = 1,\ v_2 = 3,\ v_3 = 2\}$, $\{v_1 = 1,\ v_2 = 2,\ v_3 = 3\}$ The first solution can be mapped into the second one by the symmetry swapping variables v_2 and v_3. More generally, any variable permutation is a symmetry of the problem.

SBDD is based on the notion of no-goods. No-goods are the roots of maximal sub trees that are completely traversed by a depth first search before n. Those no-goods can be found by traversing the path from the root node to n as follows. Each time the path goes from a node to its right child, then the left child of that node is a no-good: before traversing the right child of a given node a depth first search completely traverses the left sub tree of that node. Note that we use the name "no-good" although where we are searching for all solutions, fully explored trees may contain solutions. In such a case, we still wish to avoid searching any symmetric equivalent to any node in the subtree, including the solutions. In Figure 10.11, node 3 is a no-good w.r.t. node 4, node 2 is a no-good for node 5 and all the nodes in its sub tree. The no-goods w.r.t. node 8 are nodes 2 and 7.

Definition 10.25. ***No-good.***
Node ν is a no-good *w.r.t. n if there exists an ancestor n_a of n s.t. ν is the left hand child of n_a and ν is not an ancestor of n.*

For each no-good, SBDD stores information to be compared against the current state. We use the set of decisions labeling the path from the root of the tree to the no-good [99]. We write $\delta(n)$ for this. The column labeled "decisions" in the table in Figure 10.11 gives the decision information corresponding to each node. We also use the state information at the node being searched. Specifically, we write $\Delta(S)$ for set of pairs $v_i = a_i$ for all variables v_i whose domains are reduced to a singleton. In node 8 in our example, the decisions made from the root node are $\delta(8) = \{v_1 = 1, v_2 = 3\}$, while $\Delta(8) = \{v_1 = 1, v_2 = 3, v_3 = 4\}$.

Definition 10.26. ***Dominance***
We say that a node n is dominated *if there exists a no-good ν w.r.t. n and a symmetry g s.t. $(\delta(\nu))^g \subseteq \Delta(n)$. We say that ν* dominates *n.*

SBDD is then quite simple, conceptually: it never generates the children of dominated nodes, and it excludes dominated solutions. Therefore, a node n is a leaf iff it is either a solution, a failure, or a dominated node. In our example, no-good 2 dominates node 7. We have $\delta(2) = \{v_1 = 1,\ v_2 = 2\}$. Using the symmetry g which swaps variables v_2 and v_3, we obtain $(\delta(2))^g = \{v_1 = 1, v_3 = 2\}$ which is a subset of $\Delta(7) = \{v_1 = 1,\ v_3 = 2, v_2 = 3\}$.

Other definitions of dominance are possible: Definition 10.26 is from [99]. However, if we know that search always chooses $(var = val)$ before its negative $(var \neq val)$ we are free to ignore the negative decisions in $\delta(n)$ in Definition 10.26 [34, 59, 102]. For example, consider the negative decision $v_2 \neq 2 \in \delta(7)$ in Figure 10.11, and now suppose that some future node is dominated by $\delta(7)$ with $v_2 \neq 2$ removed. We can still terminate search, because every leaf node in the future subtree either has the symmetric image of $v_2 \neq 2$ or $v_2 = 2$. If the former, the node is dominated by $\delta(7)$. But if the latter, the node is dominated by $\delta(2)$ since the corresponding subtree fully explored $v_2 = 2$ and we have $\delta 7 \setminus \{v_2 \neq 2\} \cup \{v_2 = 2\} \supset \delta(2) = \{v_1 = 1, v_2 = 2\}$. The structure of search trees makes this observation general, so if positive differences are explored first, we can omit negative decisions from dominance checks.

The original definition of dominance from [33] is rather more different and based on *state inclusion*. A node n is dominated if there exists a no-good ν for n and a symmetry g such that the domains of the variables in ν^g contains the domains of the variables in n. This has the disadvantage that more space is required to store no-goods. Also, it goes against the following intuition. Since we wish to establish that a set at this node is a superset of a past no-good, we would like the set at this node to be big, and the set at the no-good small: this should make it as easy as possible to pass the dominance test. On the other hand, we also want the dominance check to be as easy as possible to implement and as fast as possible to run. We might therefore be best to use a definition of dominance which fails as quickly as possible, which may be an argument in favour of state inclusion dominance. No definitive study of this issue has ever been performed, and perhaps is not possible any more than deciding what the single best heuristic is for backtrack search.

The critical issue we have glossed over so far is just *how* the dominance check is performed. The algorithm for SBDD requires a problem specific function $\Phi : (\nu, n) \rightarrow \{false, true\}$ that yields true if the previous no-good ν dominates the current partial assignment n. For problems with small symmetry groups, provision of Φ can be a greater load on the programmer than of the few symmetry functions required by SBDS. For larger groups, SBDD has the enormous advantage that it has very limited space needs. However, this does not solve the time complexity problems. For each pair ν and n, the search for g amounts to solving a sub graph isomorphism problem, which is known to be NP-complete. Although SBDD requires the solution of several NP-complete problems at each node, good results have been obtained using the technique.

There seem to be three broad techniques for implementing dominance checks. First, a programmer can implement a dominance checker for a particular class of problems, for example instances of the social golfers' problem [33]. The need for this is a major problem with SBDD. Encoding a function that will recognise when a node in the search tree is symmetrically dominated by another one can be difficult, and it does not generalise between different problems with different types of symmetry. However, if such a function can be found than SBDD is a very efficient method at breaking large amounts of symmetry. Sellmann and Van Hentenryck have created a more general dominance detection function

[107]. This can lead to very efficient solutions tuned to a particular application, but this approach relies on the skill and insight of the programmer and imposes a considerable burden on them. Second, since it is an NP-complete problem, one can construct a constraint encoding of the dominance problem [99]. This is particularly interesting since it amounts to using constraint programming for computational group theory. However, it is still necessary to construct a special purpose constraint problem for each new class of problems to be solved. The third approach is to use computational group theory directly [16, 53], and we describe this further in Section 10.5.5.

An important refinement of SBDD is to notice that sometimes *failed* dominance checks can result in propagation [34, 33]. Suppose our dominance check can report that a certain variable-value pair would, if set in the current node, lead to the current node being dominated. Then we can remove that value from the domain of the variable and propagate on the result. This can perform extremely useful propagation. As usual, we have to take care that the benefits do not outweigh the costs of performing the necessary calculations. Another point to note 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. Deciding where to apply the check is a tradeoff between the cost of checks and the search savings that result: unfortunately little is known about the best place on this tradeoff.

Like SBDS, SBDD can be shown to be a sound and complete symmetry breaking method provided that the dominance check is implemented correctly. That is, exactly one solution from each equivalence class is returned. Like SBDS, the solution found is the leftmost one in the search tree with respect to the variable and value ordering heuristics. SBDD therefore respects the variable ordering heuristic, and dynamic variable ordering may be used without any change to SBDD [33]. SBDD has also been applied to soft CSP's [11].

Harvey [59] discusses 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. In fact, it is entirely reasonable to see the difference between it and SBDS as merely one of implementation. As the set of acceptable solutions is the same in each case, an implementation of SBDS is, in a sense, an implementation of SBDD, and vice versa. This view can be a useful way of understanding the techniques in principle, but there are enormous practical differences. SBDD can outperform SBDS on many problems, 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.

10.5.3 Symmetry Excluding Heuristics

Meseguer and Torras [84] studied how symmetry can be used to guide search. Specifically, they direct search towards subspaces with a high degree of non-symmetric assignments, by breaking as many symmetries as possible with each variable assignment. The symmetry breaking heuristic proposed breaks as many symmetries as possible. Meseguer and Torras go on to propose the 'variety-maximization' heuristic which combines the smallest domain first heuristic, which can perform better than the symmetry breaking heuristic under certain conditions, with the symmetry breaking heuristic. On a simple problem the variety-maximization heuristic builds a slightly smaller search tree, than the smallest domain first heuristic, to find all solutions. Variety-maximization does notably better to find the first solution. Meseguer and Torras go on to combine the variety-maximization heuristic with symmetric value pruning by no-good recording. The size of the set of no-goods is potentially exponential, so only a subset is stored and used. The results of this combination are rather disappointing; the inclusion of symmetric value pruning does not provide a major advantage over variety-maximization alone, in any of the problems undertaken.

Using variable ordering heuristics (and indeed value ordering heuristics) to break symmetry is an attractive proposition, as it does not have the computational cost associated with the other dynamic symmetry breaking methods. Despite the work of Meseguer and Torras, there seems to be much more scope for using symmetry in heuristics than the community has exploited to date. There is also considerable scope for other ways to use symmetry heuristically. For example, if we are solving a problem with a large amount of symmetry, we might look only for solutions with some symmetry. This excludes enormous parts of the search space. On the one hand, this means that search will be incomplete, in that a negative answer does not mean there are no solutions with less symmetry. On the other hand, we might get lucky, and there may be such solutions: if there are we have all the advantage of the reduced search. While this may seem incredibly optimistic, it has been successful in practice [119].

10.5.4 SBDS with Computational Group Theory

To allow SBDS to be used in situations where there are too many symmetries to allow a function to be created for each, Gent *et al.* [52] linked SBDS in ECL^{i}PS^e with GAP (Groups, Algorithms and Programming) [46]. GAP is a system for computational algebra and in particular Computational Group Theory (CGT). There is nothing fundamental about the use of GAP or ECL^{i}PS^e, the point being that this is a co-operation between subsystems to provide constraint algorithms and to provide computational group theory algorithms. A library for GAP-SBDS is distributed with ECL^{i}PS^e.

GAP-SBDS allows the symmetry group, rather than its individual elements, to be described. GAP-SBDS operates over a set of points, where each point corresponds to a variable-value pair. One way to think of a point is in terms of a member of a $n \times m$ array, where n is the number of variables, and m is the size of the domain of each variable. The i, j-th element in the array denotes variable i and value j. The symmetry group, G, acts on these $n \times m$ points, each point being represented by a single integer. The group generators are given in ECL^{i}PS^e and passed to GAP in terms of points. This means that functions are required in ECL^{i}PS^e which convert from variable/value pairs to integers representing points and vice-versa. This is a possible source of error when using the GAP-SBDS system. Even without error, to effectively use the system, the user must have some knowledge of CGT. While it is enough to write down a set of generators, even this makes the system inaccessible for many users. However, some progress has been made in this area, as we describe in Section 10.8.

The GAP-SBDS algorithm can be described in terms of Equation 10.2 from Section 10.5.1: $A \;\&\; A^g \;\&\; var \neq val \;\; \Rightarrow \;\; (var \neq val)^g$, In GAP-SBDS, the only part of this process that is controlled by GAP is $g(A)$, the other components are controlled by ECL^{i}PS^e as in the standard SBDS algorithm. $g(A)$ is calculated with the use of a right-transversal chain, a set calculated iteratively at every node in the search-tree, as the Cartesian product of every right-transversal obtained so far. More formally *RTchain* $= RT_k \times RT_{k-1} \times \ldots \times RT_1$ is defined as $p_k \circ p_{k-1} \circ \ldots \circ p_1$ where $p_i \in RT_i$. 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. This means the symmetry breaking constraint can be placed on backtracking, by transforming $var \neq val$ according to the elements in *RTchain*. However, doing this by iterating over every symmetry would be infeasible for all but relatively small groups. To counter this difficulty Gent *et al.* use lazy evaluation. The constraint $g(var \neq val)$ is only imposed when $g(A)$ is known to be true, rather than placing conditional constraints as in the original SBDS. This means that although GAP-SBDS is guaranteed to break all the symmetry, it may not break the symmetry as early in search as SBDS. This can lead to GAP-SBDS having a larger backtrack count than SBDS.

In GAP-SBDS Gent *et al.* have created an efficient implementation of SBDS, which can handle relatively large amounts of symmetry effectively. This moves the order of groups that SBDS can be be used with from the thousands to the billions. The number of symmetries is limited to this kind of scale because the number of constraints being added during search can cause space problems. Even billions is a small number when groups can grow exponentially.

10.5.5 SBDD and Computational Group Theory

We mentioned, in Section 10.5.2, that in practice SBDD can be difficult to implement. The design of the dominance detection function may be complicated, and there are no general rules for designing the function for problems with similar types of symmetry. In reality two ‘similar’ problems may have completely different dominance functions; there may be more than one dominance function for a problem, some of which prune more effectively than others. Gent *et al.* [53], developed GAP-SBDD to address this problem by providing a generic dominance checker, which is now available as an ECL^{i}PS^e library.

GAP-SBDD is a generic version of SBDD that uses the symmetric group of the problem, rather than an individual dominance detection function. Like GAP-SBDS, GAP-SBDD works by way of an interface between GAP and ECL^{i}PS^e and performs calculations over points, with the symmetry groups defined by a generating set of permutations. GAP-SBDD operates by maintaining both a *failSet* and a *pointSet*. The *failSet* corresponds to the set of points attributed to the positive decisions made during search to reach the root of completed subtrees. The *pointSet* denotes the set of points corresponding to variables which have been set to a fixed value on the current branch of search (both through assignments and propagation). The current node on the search tree is dominated by a complete subtree if there exists a g in the symmetry group G and a s in the *failSet* S such that $s^g \subseteq$ *pointSet*.

The dominance check is implemented using a tree data structure which encodes all the *failSets* currently applicable. Disjoint sets of points $A_1, \ldots, A_k$ and $B_0, \ldots, B_k$ can be identified, the *failSets* associated with these points are $A_1 \cup \ldots \cup A_i \cup B_i$ for each i. The right branching elements of the tree are labelled with elements of A, the left ones elements of B. Each node of the tree is associated with the sequence of labels on the path to it from the root. The dominance check is performed using a recursive search, which traverses the tree, entering each node once for every way of mapping the associated sequence of points into the current point. This is implemented in GAP with the use of stabiliser chains. While inherently just a backtracking search for an appropriate group element, the search in GAP contains a number of optimisations which cause several orders of magnitude speedup from a naive implementation [53]. As in regular SBDD, it is possible to identify cases where all but the final element of a *failSet* can be mapped into a *pointSet*, and report them back to ECL^{i}PS^e, so the value can be removed from the associated variables domain. However, like GAP-SBDS, not all possible values are identified, as this would cause too much increase in runtime.

GAP-SBDD is an efficient method for dealing with problems with large symmetry groups. In principle, the size of groups that can be used is unlimited, and certainly it has been used in practice on groups with 10^{36} elements. However, it provides no magic bullet to the inherent hardness of the dominance check. This hardness can manifest itself with individual dominance checks that take inordinate amounts of time to run. Also, it has been found that subtle differences can make enormous differences to run time. For example, while GAP-SBDD does respect the variable ordering heuristic, different heuristics on the same instance can lead to dramatic changes in the time dominance checks take. This means that the method can suffer from a lack of robustness. Nevertheless, from a very small input with no algorithmic content, GAP-SBDD constructs a complete symmetry breaking tool automatically.

10.5.6 GE-Trees

The idea of constructing GE-trees is the most recent idea to join the suite of dynamic symmetry breaking methods [103]. (GE stands for "Group Equivalence" but the abbreviation is used universally.) It differs from the others in this class, as it is more a way of considering symmetry breaking methods, than a symmetry breaking method in its own right. A GE-tree is defined as a tree in which no two nodes are symmetrically equivalent, and in which, for every solution to the problem, a symmetrically equivalent node is in the tree.[4] GE-trees are defined analogously to search trees in general: in particular, algorithms are free to stop before they construct a complete tree, for example after finding a first solution. GE-trees are intended in part to be viewed as a conceptual paradigm, to classify and compare symmetry breaking techniques. Any method of GE-tree construction will (by definition) break all problem symmetries as the search for solutions proceed. SBDS and SBDS can both be viewed as methods for constructing GE-trees, but so can lex-leader provided that all constraints it requires are used. Careful analysis of properties of GE-trees (constructed by different methods such as SBDS and SBDD, applied to the same instances) may allow the refinement and extension of existing techniques and even the development of new ones.

In some cases GE-trees are almost in the folklore. For example, consider graph colouring where each colour is indistinguishable. Many programmers have realised that the first node can be coloured arbitrarily. For the second node, we only have to consider giving it the same colour or an arbitrary different one. The third node need only be given the colours given to the first two nodes, and an arbitrary different one. The process continues until we have used all colours. This intuition can be generalised and formalised. Roney-Dougal *et al.* used the GE-tree paradigm to create a polynomial time algorithm for breaking arbitrary

[4]The latter condition is easy to forget but otherwise legal GE-trees could exclude some or even all solutions.

value symmetries. This can be viewed as a computational group theory-based generalisation of an algorithm presented in [64]. It is a special purpose algorithm which only creates new nodes which are guaranteed to be unique in the tree. In general this is hard, but Roney-Dougal *et al.* showed that the special nature of groups of value symmetries can be used to construct GE-trees very efficiently [103]. They report on an experimental comparison between this method and GAP-SBDD for problems which only have value symmetry. GE-trees are found to be the best method in all cases. This is not surprising: GAP-SBDD performs a potentially exponential search, at each node in the search tree, in order to break symmetry, compared to the low-order polynomial algorithm to break all value symmetry.

10.5.7 The STAB Method

Symmetry Breaking Using Stabilizers (STAB), like SBDS, adds symmetry breaking constraints during search [98]. Unlike SBDS, which places constraints to break all the symmetry of the problem, STAB places symmetry breaking constraints only for symmetries that leave the partial assignment A at the current node unchanged. That is, instead of breaking symmetry in the whole group, STAB breaks symmetry in the stabiliser G_A. These constraints take the form of lexicographic ordering constraints. Stabilisers were introduced in Section 10.1. In practice the size of stabilisers is often much smaller than the size of G. The STAB method, amounts to adding the following set of constraints at each node A.

$$V \preceq_{\text{lex}} V^g, \text{ for all } g \in \ G_A$$

These constraints remove all the solutions that are not lexicographically minimum with respect to the stabiliser G_A in the sub tree rooted at A.

Example 10.27. Consider a 4×5 matrix model. For simplicity we will refer to the matrix of variable by V, i.e. we identify the vector of variables with its matrix representation:

$$\begin{array}{|ccccc|}\hline x_1 & x_2 & x_3 & x_4 & x_5 \\ x_6 & x_7 & x_8 & x_9 & x_{10} \\ x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\ x_{16} & x_{17} & x_{18} & x_{19} & x_{20} \\ \hline\end{array}$$

Consider a partial assignment A where the first 10 variables are assigned as follows:

$$\begin{array}{|ccccc|}\hline 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ & & & & \\ & & & & \\ \hline\end{array}$$

Every symmetry for A is defined by a row permutation and a column permutation.We can state constraints for each of these symmetries, except for the identity permutation. For instance, let us state the constraint for the symmetry σ made up of the permutation $(1\ 2)$ of rows and $(2\ 4\ 3\ 5)$ on columns. The matrix $W = V^\sigma$ is

$$\begin{array}{|ccccc|}\hline x_6 & x_9 & x_{10} & x_8 & x_7 \\ x_1 & x_4 & x_5 & x_3 & x_2 \\ x_{11} & x_{14} & x_{15} & x_{13} & x_{12} \\ x_{16} & x_{19} & x_{20} & x_{18} & x_{17} \\ \hline\end{array}$$

The symmetry breaking constraint is then, allowing for the assignment A, is then

$$\begin{aligned} &[0,0,0,1,1,0,1,1,0,0,x_{11},x_{12},x_{13},x_{14},x_{15},x_{16},x_{17},x_{18},x_{19},x_{20}] \\ \preceq_{\text{lex}}\ &[0,0,0,1,1,0,1,1,0,0,x_{11},x_{14},x_{15},x_{13},x_{12},x_{16},x_{19},x_{20},x_{18},x_{17}] \end{aligned}$$

As the two vectors have the same first 10 elements, the constraint can be simplified into:

$$\begin{aligned} &[x_{11},x_{12},x_{13},x_{14},x_{15},x_{16},x_{17},x_{18},x_{19},x_{20}] \\ \preceq_{\text{lex}}\ &[x_{11},x_{14},x_{15},x_{13},x_{12},x_{16},x_{19},x_{20},x_{18},x_{17}] \end{aligned}$$

Going back to the general case, the simplification based on identical prefixes is valid in general. Given the n-vector V, let $tail(V$,n-d$)$ be the vector obtained by removing the first d elements of V. If A is a d-vector then constraints can be simplified into

$$tail(V,n-d) \preceq_{\text{lex}} (tail(V,n-d))^g, \text{ for all } g \in G_A$$

It should be noted that STAB is an incomplete method with regard to symmetry breaking, meaning that it does not return only the non-isomorphic solutions.

10.6 Combinations of Symmetry Breaking Methods

We have described a variety of symmetry breaking methods, each with advantages and disadvantages. It would naturally seem a good idea to combine two or more methods in order to reap all of the advantages. Unfortunately, correctly combining symmetry breaking methods has proven to be very difficult. Symmetry breaking methods try to preserve one solution from each equivalence class. How this solution is chosen depends on the method of symmetry breaking. Combining any two methods arbitrarily may mean that different solutions are ruled out by each symmetry breaking method, and so solutions are lost.

For a while it was thought by many that if the same variable and value ordering are used, then it was valid to apply several symmetry breaking methods at the same time, since none of these methods remove canonical solutions. Therefore, any combination will keep canonical solutions. For instance, one could state the Lex^2 constraints and use SBDD at the same time for matrix models. However, Smith [112] has shown this not to be the case. Harvey [60] later documented an example of this result, then showed how SBDS and SBDD can be modified so that they can be correctly combined with lexicographic ordering constraints. Unfortunately, the empirical results for this combination were disappointing. Another area which has received some study is combining a method to break variable symmetry, with a method to break value symmetry. As both of these methods are acting on different symmetry groups, this combination is intrinsically safe. Puget [101], showed how all the variable symmetry can be broken by simple ordering constraints, when there is an all-different constraint across all of the variables. He successfully combined this method with GE-trees to also break the value symmetry. Another combination which includes GE-trees has been tried [69]. This successfully combined GE-tree to break value symmetry and GAP-SBDD to break variable symmetry. This method is provably complete and sound with regard to breaking all symmetry, but is less efficient than GAP-SBDD alone on all problems tried.

In Section 10.5.7 it was noted that STAB is an incomplete method with regard to symmetry breaking, meaning that it does not return only the non-isomorphic solutions. In order to counter this drawback, Puget combined SBDD with STAB. This combined method produces good results, outperforming SBDD alone.

Another successful combination of symmetry breaking methods was completed by Petrie [90]. Petrie showed that neither of GAP-SBDS or GAP-SBDD is universally better, so combined the two methods. This was done by having one operate at the top of the search tree, than switching to the other method at a lower level. The empirical analysis of this method showed it to sometime outperform either of the methods alone. However, perhaps more importantly it was shown to be a more robust method, in that the combined method was never less efficient, than the least efficient of the two contributing systems.

10.7 Successful Applications

Many of the symmetry breaking methods outlined in the previous sections have been applied successfully to a variety of problems. We have already mentioned two successful applications: the chess puzzle at the start of the chapter, and graceful graphs. In both cases new results were obtained using symmetry breaking in constraint programming. We briefly outline some more applications, and indicate which symmetry breaking methods have been successfully applied to them. CSPLib (www.csplib.org), is an online directory of constraint problems: whenever a problem is contained within this resource, the number of the problem is given in this text to help the interested party find a detailed problem specification. The references provided are the papers which discuss symmetry breaking within the applications.

Balanced Incomplete Block Design (BIBD) generation is a standard combinatorial problem from design theory, originally used in the design of statistical experiments but since finding other applications such as cryptography. It is a special case of Block Design, which also includes Latin Square problems. BIBD's are problem 28 in CSPLib, Lam's problem, which is problem number 25 in CSPLib is that of finding a specific BIBD instance. BIBD's can be easily modelled with the use of matrices, and so many of the methods that place symmetry breaking constraints on matrices have been applied to them [44]. This problem was also used as a test bed for STAB [98] and GAP-SBDD [53].

Steel Mill Slab Design is a simplification of a real industry problem, which is to schedule the production of steel in a factory. It is problem number 38 in CSPLib. Gent *et al.* have considered conditional symmetry breaking in this problem [54]

Maximum Density Still Life problem arises from the Game of Life, invented by John Horton Conway in the 1960s and popularized by Martin Gardner in his Scientific American columns. A stable pattern, or still-life, is not changed by the rules which cause the Game of Life to iterate. The problem is to find the densest possible still-life pattern, i.e. the pattern with the largest number of live cells, that can be fitted into an $n \times n$ section of the board, with all the rest of the board dead. Maximum density still life is problem 38 in CSPLib. Smith [114] and both Bosch and Trick [12] have considered modelling and symmetry breaking in Still Life. Petrie *et al.* have studied dynamic symmetry breaking in a CP-LP hybrid, then applied this idea to this problem [90].

Social Golfers Problem is where the coordinator of a local golf club has the following problem. In her club, there are 32 social golfers, each of whom play golf once a week, and always in groups of 4. She would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion. The problem can easily be generalized to that of scheduling m groups of n golfers over p weeks, such that no golfer plays in the same group as any other golfer twice (i.e. maximum socialisation is achieved). Social Golfers is problem number 10 in CSPLib. Harvey and Winterer at the time of writing have the most efficient algorithm for this problem, which includes symmetry breaking [62].

Peaceable Coexisting Armies of Queens was introduced by Robert Bosch in his column in Optima in 1999 [13]. It is a variant of a class of problems requiring pieces to be placed on a chessboard, with requirements on the number of squares that they attack: Martin Gardner [47] discusses more examples of this class. In the "Armies of Queens" problem, we are required to place two equal-sized armies of black and white queens on a chessboard so that the white queens do not attack the black queens (and necessarily v.v.) and to find the maximum size of two such armies. Bosch asked for an integer programming formulation of the problem and how many optimal solutions there would be for a standard 8 $\times$ 8 chessboard. However, this problem can obviously be generalised to a $n \times n$ chessboard. Various models for this problem in conjunction with dynamic symmetry breaking, are considered by Smith *et al.*, the puzzle at the start of this chapter being a spinoff of this research [115].

Fixed Length Error Correcting Codes are defined as follows: A fixed length error correcting code C of length n over an alphabet F is a set of strings from F^n. Given two strings from F^n we can define the distance between them. The most commonly used distance is the Hamming distance, defined as the number of positions where the strings differ. Using this we define the minimum distance of C as the minimum of the distances between distinct pairs of strings from C. Fixed Length Error Correcting codes are problem 36 in CSPLib. This problem is studied in conjunction with symmetry breaking constraints by Frisch *et al.* [42].

Peg Solitare (also known as Hi-Q) is played on a board with a number of holes. In the English version of the game, the board is in the shape of a cross with 33 holes. Pegs are arranged on the board so that at least one hole remains. A number of different problems arise from Solitaire, e.g. finding a path from the initial to a goal state, or finding the shortest number of moves to a state where no more moves are possible. Peg solitare is problem 37 in CSPLib. Jefferson *et al.* have studied on how to solve this problem using various AI paradigms, including a discussion of symmetry breaking [42].

Alien Tiles is available for play over the internet at www.alientiles.com. Alien Tiles is problem 27 in CSPLib. Gent, Smith and Linton have studied how dynamic symmetry breaking can be successfully applied to the problem of finding the hardest instance. [49].

We can see that symmetry methods are flexible enough to be applied to a great variety of problems, and we have by no means mentioned all the successful applications to date. The most successful seem to be on combinatorial puzzles such as graceful graphs. It therefore remains to be proved by the community that symmetry breaking has significant applications on industrial scale problems.

10.8 Symmetry Expression and Detection

Most research on symmetry constraints has assumed that the symmetries of a problem can be provided, in some form, by the programmer. For example, SBDS relies on programmers providing a list of functions to implement the actions of symmetries; SBDD needs a dominance check to be written; and methods such as lex-leader and GAP-SBDS need a group to be provided. Even for methods where this is not necessary, such as the use of double-lex constraints in a matrix model, a programmer has to recognise that the matrix symmetries occur and that the double-lex constraints are an appropriate technique to break them with. In recent years, increasing amounts of work have been based on the premise that this is an untenable position. Two ways of overcoming this flaw have been considered. First, it could be made easier for the programmers to write down the symmetries. Second, the symmetries could be detected automatically so there is no work at all for the programmer. This first method has been independently studied by both McDonald [48] and Harvey *et al.* [63], with a view to creating a system which produces the group needed for methods using computational group theory, without the user having to understand exactly how groups are generated. There are three main aims to these systems. The first is to get rid of the need for functions which map variable/value pairs to points (this concept is outlined in Section 10.1.1). The second is to allow the expression of the symmetry in a simple yet powerful way. The third is to create a system which requires the user to have no prior knowledge of group theory. These objectives are achieved, by providing a set of functions, which map expressions of the symmetry with regard to the variables and values (e.g. all the rows of a matrix can be interchanged) to group generators. These techniques to date are limited to the most commonly occurring kinds of symmetry, and do not allow users to express arbitrary groups.

The automatic identification of the symmetry group of some CSPs is possible through determining the automorphism group of the graph associated with the constraint problem. Puget [100] showed that this could be done in connection with the microstructure graph, which closely relates to the existential representation of the constraints. The concern with this approach is that it may not scale well (it is in the complexity class of graph isomorphism in most cases). When the problem is large, so is the graph, and the automorphism cannot be calculated in a reasonable time. Small problems can also have big graphs due to non-binary constraints, which again would mean the automorphism could not be calculated in reasonable time. To counter this potential problem Puget also introduced a method where he considered a graph related to intensional representation of each constraint. Puget found that the symmetry could be detected very efficiently, using both methods, on a variety of problems. Ramani and Markov have also has recently had some success [23], with a method closely linked to Puget's. This method undertakes graph isomorphism of the parser trees associated with the intensional representation of the constraints. Although this method is not guaranteed complete, in that it does not guarantee to find the full symmetry group, it has had some successful results in practice. It should be mentioned that in the SAT community outstanding results have been obtained using automatic symmetry detection, even on large problems [1]. This is particularly encouraging because natural structures, such as the microstructure of constraint problems, correspond to encodings into SAT. In order to know when this attractively simple method is feasible with regard to CP empirical analysis is required.

Many symmetries are also added in the modelling process, as pointed out by Frisch *et al.* [45]. If we can understand when this symmetry is added, it should be easy to pass those symmetries to symmetry-breaking methods.

10.9 Further Research Themes

We now discuss a diverse range of research using symmetry, which is unified only by not fitting easily elsewhere in this chapter. We do not to suggest that the work in this section is less important than in the rest of this chapter. Indeed, the same reasons that make this research hard to categorise makes some of it particularly innovative, and so it has significant potential for future exploration.

Symmetry and inference: Almost all research on symmetry in constraints has focussed on breaking symmetries to reduce search. When symmetry is present in a search problem, its presence is information that we can exploit in other ways. A particular example is to enhance representation and inference of constraint problems. First, we can greatly compress the size of the representation and thus enhance the representational power. Second, we can exploit knowledge of the group structure to change our algorithms for inference and propagation. This is necessary if we use a compressed representation based on the group, but also offers advantages. If we perform some work to deduce that a value can be removed, we need perform no more work to remove all symmetric equivalents of it. Similarly, if we know that a given value cannot be removed, then we need perform no more work on trying to propagate symmetric equivalents. A simple example of this idea was used to exploit the bidirectionality of constraints in arc consistency to improve AC-6 [9]. More generally, we may have arbitrary groups acting on the set of constraints. Dealing with groups correctly and efficiently raises many difficult problems. These difficulties are now being seriously attacked. By far the most significant body of work has been applied in the context of SAT, by Dixon *et al.*, in a three part journal paper spreading over almost 200 pages [27, 26, 28]. They have produced ZAP, a powerful SAT solver designed to allow concise expression of problems using group theory, while exploiting the great efficiency advances made in modern SAT solvers. At a much smaller scale, new propagation algorithms have been introduced into Constraint Programming. Gent *et al.* introduced symmetric variants of (i,j)-consistency and singleton consistency, together with algorithms for their enforcement. While implementations have been provided, it has not yet been proved that symmetry-based inference techniques will be powerful in practice. Nevertheless, the research area is of high promise, as suggested by Dixon *et al.*'s conclusion to their mammoth effort: “it is important to realize that our results only scratch the surface of what ZAP's representational shift allows” [28].

Symmetry and implied constraints: When constraints are added before search, it is possible to use them to derive ‘implied’ constraints. These implied (or ‘redundant’) constraints may greatly reduce search, in ways which are not possible using the original problem constraints [44]. We saw, in Section 10.4.3, another example where symmetry breaking and problem constraints can interact very fruitfully. Dynamic symmetry breaking techniques do not allow implied constraints to be added – at least not in the simple way that static symmetry breaking methods do. Adding implied constraints suffers from similar problems to reformulation. It can give very dramatic search reductions, but no automatic technique for adding effective implied constraints is known.

Symmetry and local search: When we know about symmetry in a constraint problem, there is no moral imperative to remove it: we can exploit the symmetry in any way we choose. Where local search is used instead of backtracking search, Prestwich pointed out that it is disadvantageous to add symmetry breaking constraints [94, 91]. While more solutions are good for *any* search method, local search suffers particularly when solutions are excluded. Since stochastic methods are designed to explore search spaces randomly, it is hard to guide them away from parts of the search space where solutions have been excluded, while each excluded solution becomes a new local optimum. In a detailed analysis, Prestwich and Roli identified two pathologies that symmetry breaking constraints caused for local search methods: global solutions have relatively smaller basins of attractions while local optima become relatively larger [95]. Since excluding symmetries seems bad for local search, can we improve it by *introducing* symmetries? Prestwich proposed this lovely idea with some success [92, 91]. Exploiting symmetry within local search algorithms remains an interesting but underdeveloped research area: Petcu and Faltings used interchangeability to guide local search away from conflicts [87], but otherwise little has been done in this direction.

Dominance and almost symmetries: Researchers have investigated cases where standard symmetry breaking methods are not applicable. For example, Beck and Prestwich studied ‘dominances’ in constraint problems [7]. A dominance is a transition between assignments which is guaranteed to improve (or at least make no worse) some notion of a cost function. We can see symmetries as special cases of dominances, where the cost is kept the same, and hence the transitions are invertible. Beck and Prestwich argue that dominance “should rank alongside symmetry breaking as a generic CP technique, and that it can be profitable to treat both in a uniform way.” Another issue receiving recent attention is that of “almost symmetries” [37], the subject of a recent workshop [29]. The general idea is that these are symmetries which are almost, but not quite, there in the original problem. They can arise by either adding or removing constraints on the problem [58]. Examples of the former case would be symmetries which arise during search: dealing with these can be difficult, because they cause significant problems for techniques such as SBDS or SBDD [54]. The idea underlying relaxing constraints on a problem to give new symmetries is as follows [61, 81]. If the relaxed problem is highly symmetric, its reduced search space should help search. If the relaxed problem has no solutions, then neither does the original. If we do find solutions to the relaxed problem, we need additional steps to ensure that they correspond to solutions of the original.

Symmetry in other problems: Given the focus of this handbook to constraints, we have naturally restricted our discussion to constraints and SAT. However, researchers interested in symmetry in constraints should also be interested in how symmetry is tackled in other domains. For example, can we apply ideas from symmetry in constraints to other problems? Can ideas developed in other domains be applied to constraints? While we now point to some key literature in other areas, we do not even claim to provide an exhaustive list of other search problems with symmetry, much less an exhaustive or analytical study of relevant literature in those areas. In *Integer Programming*, Margot has shown how algorithms can be adapted to exploit large symmetry groups [79, 80]. New cuts are generated based on the symmetries, and large groups are handled using Schreier-Sims to represent them. In *Planning*, Fox and Long have integrated symmetry reasoning into a state-of-the-art planner, SymmetricStan [38, 37]. Fox and Long point out that symmetry detection during search, and the presence of almost-symmetry, is very important due to the nature of planning problems. It is perhaps surprising that there does not seem to be a more significant body of work in these two research areas. This is less the case in *Automated Theorem Proving*, where there is a substantial literature in adapting proof systems to deal with symmetry, e.g. [3, 71, 86, 24, 117]. There have also been substantial efforts in *Model Checking* over some years, e.g. [15, 20, 66, 110]. As in constraints, this work has tended to assume that users recognise the symmetry, and they have also been limited to simple cases of symmetry rather than using the power of computational algebra. Recently, Donaldson *et al.* have shown that more general symmetries in model checking using GAP and graph isomorphism software [30, 31, 32]. Finally, we touch only in the slightest way on the extensive mathematical literature on search with symmetry. We particularly mention McKay's *nauty* software for *Graph Isomorphism* [83] and Soicher's GRAPE package for dealing with graphs relating to groups [116]. The key algorithm is "partition backtrack" [83, 76], a very subtle intermingling of backtrack search and group-theoretic computations. The mathematical applications of such techniques are numerous: one is *Design Theory*, which deals with combinatorial designs such as balanced incomplete block designs, and for which an online repository is available at www.designtheory.org [6].

10.10 Conclusions

We have presented an overview of symmetry in constraint programming. We have been unashamed in emphasising the links with group theory. The study of symmetry *is* group theory, so anyone who has ever considered symmetry in constraints has been thinking about group theory – though perhaps without realising it. We have also emphasised the ability of computational group theory to contribute methods to the efficient exploitation of symmetry in constraints. Researchers and users of constraint programming can access these techniques either through linking to the existing computational algebra packages, or by implementing their own algorithms. Again, anyone who has written code for exploiting symmetry in constraints has been, whether unconsciously or not, doing computational algebra.

The main part of this chapter is taken up by a study of symmetry breaking methods, since this is by far the largest body of research that has been undertaken on symmetry in constraints. We considered methods in three broad categories: reformulation of problems, adding symmetry breaking constraints before search, and dynamic symmetry breaking methods that operate during the search procedure. It is worth summarising in one place the very broad advantages and disadvantages of each method. It will be seen that there is no 'one-size-fits-all' method to recommend, but it should be noted that researchers have proposed solutions which at least ameliorate many of the disadvantages we mention:

- **Reformulation** is when the problem is remodelled to eliminate some or all symmetry. It can be an astonishingly efficient method of breaking symmetry, but unfortunately there is no known systematic procedure for performing the remodelling process in general. Where reformulation is possible, it can be combined easily with other methods.

- **Symmetry breaking constraints** are perhaps the most natural technique and the easiest to understand. Ideally, the new constraints should be satisfied by only one assignment in any equivalence class, but it can be difficult to find simple constraints that eliminate all the symmetry. A systematic method called ‘lex-leader’ is known for generating symmetry breaking constraints, when the problem only contains variable symmetry. The efficiency with which some kinds of symmetry breaking constraints can be propagated, for example in matrix models, means that adding symmetry breaking constraints can be very cost-effective even where they do not break all the symmetry. Symmetry breaking constraints do have a disadvantage in that they can interact badly with the search heuristics being used.

- **Dynamic symmetry breaking** includes many methods which adapt the search process in some way. This includes SBDS, SBDD (and the computational group theoretic versions of these methods), and STAB. Both SBDD and SBDS are complete, in that only one solution will be returned from each symmetric equivalence class. Group-theoretic versions add the advantage that only a small number of symmetries in a problem need be specified, but this is related to a disadvantage, that they often require symmetries to be expressed in a mathematical language that is unnatural for constraint programmers. A significant advantage of these methods is that they do not conflict with search heuristics. As relatively new methods, a final drawback is that they can only be used if a library package is available, or the user spends the time to adapt the search program themselves.

Quite apart from its inherent importance, we conclude by commending the study of symmetry in constraints as an enjoyable and rewarding research topic.

Acknowledgments

We thank all those who have helped us with this paper, most especially Barbara Smith, but also Gene Freuder, Carmen Gervet, Chris Jefferson, Steve Linton, Ian Miguel, and Toby Walsh. Parts of the writing of this paper were supported by EPSRC grants GR/R29673, EP/C523229/1 and GR/S30580/01 and Science Foundation Ireland grant 00/PI.1/C075.

Bibliography

[1] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult sat instances in the presence of symmetry. In *DAC*, pages 731–736. ACM, 2002. ISBN 1-58113-461-4.

[2] F. A. Aloul, K. A. Sakallah, and I. L. Markov. Efficient symmetry breaking for boolean satisfiability. In Gottlob and Walsh [57], pages 271–276.

[3] N. H. Arai and A. Urquhart. Local symmetries in propositional logic. In R. Dyckhoff, editor, *TABLEAUX*, volume 1847 of *Lecture Notes in Computer Science*, pages 40–51. Springer, 2000. ISBN 3-540-67697-X.

[4] 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.

[5] R. Backofen and S. Will. Excluding symmetries in constraint-based search. In J. Jaffar, editor, *Principles and Practice of Constraint Programming - CP '99*, volume Lecture Notes in Computer Science 1713, pages 73–87. Springer, 1999.

[6] R. Bailey, P. Cameron, P. Dobcsányi, J. Morgan, and L. Soicher. Designs on the web. *Discrete Math.* Forthcoming.

[7] J. C. Beck and S. D. Prestwich. Exploiting dominance in three symmetric problems. In *Fourth International Workshop on Symmetry and Constraint Satisfaction Problems*, 2004.

[8] B. Benhamou. Study of symmetry in constraint satisfaction problems. In *Proceedings of the 2nd workshop on Principles and Practices of Constraint Programming - PPCP'94*, pages 246–254, 2003.

[9] C. Bessière and J.-C. Régin. Using bidirectionality to speed up arc-consistency processing. In M. Meyer, editor, *Constraint Processing, Selected Papers*, volume 923 of *Lecture Notes in Computer Science*, pages 157–169. Springer, 1995. ISBN 3-540-59479-5.

[10] C. Bessière, E. Hebrard, B. Hnich, and T. Walsh. The complexity of global constraints. In D. L. McGuinness and G. Ferguson, editors, *AAAI*, pages 112–117. AAAI Press / The MIT Press, 2004. ISBN 0-262-51183-5.

[11] S. Bistarelli and B. O'Sullivan. Combining Branch & Bound and SBDD to solve Soft CSPs. In *Proceedings of CSCLP 2004: Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming*, June 2004.

[12] R. Bosch and M. Trick. Constraint programming and hybrid formulations for three Life designs. In *Proc. of CP-AI-OR'02*, pages 1396–1407, 2002.

[13] R. A. Bosch. Peaceably coexisting armies of queens. *Optima (Newsletter of the Mathematical Programming Society)*, 62:6–9, 1999.

[14] W. Bosma and J. Cannon. *Handbook of MAGMA functions*. sydneypm, Sydney University, 1993.

[15] D. Bosnacki, D. Dams, and L. Holenderski. Symmetric spin. *STTT*, 4(1):92–106, 2002.

[16] 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.

[17] G. Butler. *Fundamental Algorithms for Permutation Groups*. LNCS 559. Springer-Verlag, 1991.

[18] M. Carlsson and N. Beldiceanu. Arc-consistency for a chain of lexicographic ordering constraints, 2002.

[19] B. Y. Choueiry and G. Noubir. On the computation of local interchangeability in discrete constraint satisfaction problems. In *AAAI/IAAI*, pages 326–333, 1998.

[20] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[21] D. A. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith. Symmetry definitions for constraint satisfaction problems. In van Beek [118], pages 17–31. ISBN 3-540-29238-1.

[22] J. Crawford, M. L. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In L. C. Aiello, J. Doyle, and S. Shapiro, editors, *KR'96: Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan

Kaufmann, San Francisco, California, 1996.

[23] P. Darga, M. Liffiton, K. Sakallah, and I.L.Markov. Exploiting Structure in Symmetry Generation for CNF. In *ACM/IEEE Design Automation Conference - DAC*, pages 530–534, 2004.

[24] T. B. de la Tour. Ground resolution with group computations on semantic symmetries. In M. A. McRobbie and J. K. Slaney, editors, *CADE*, volume 1104 of *Lecture Notes in Computer Science*, pages 478–492. Springer, 1996. ISBN 3-540-61511-3.

[25] R. L. de Mántaras and L. Saitta, editors. *Proceedings of the 16th European Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, 2004. IOS Press. ISBN 1-58603-452-9.

[26] H. E. Dixon, M. L. Ginsberg, E. M. Luks, and A. J. Parkes. Generalizing boolean satisfiability ii: Theory. *J. Artif. Intell. Res. (JAIR)*, 22:481–534, 2004.

[27] H. E. Dixon, M. L. Ginsberg, and A. J. Parkes. Generalizing boolean satisfiability i: Background and survey of existing work. *J. Artif. Intell. Res. (JAIR)*, 21:193–243, 2004.

[28] H. E. Dixon, M. L. Ginsberg, D. Hofer, E. M. Luks, and A. J. Parkes. Generalizing boolean satisfiability iii: Implementation. *J. Artif. Intell. Res. (JAIR)*, 23:441–531, 2005.

[29] A. F. Donaldson and P. Gregory. Almost-symmetry in search. Technical Report TR-2005-201, University of Glasgow, 2005.

[30] A. F. Donaldson and A. Miller. Automatic symmetry detection for model checking using computational group theory. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *Lecture Notes in Computer Science*, pages 481–496. Springer, 2005. ISBN 3-540-27882-6.

[31] A. F. Donaldson, A. Miller, and M. Calder. Finding symmetry in models of concurrent systems by static channel diagram analysis. *Electr. Notes Theor. Comput. Sci.*, 128(6):161–177, 2005.

[32] A. F. Donaldson, A. Miller, and M. Calder. Spin-to-grape: A tool for analysing symmetry in promela models. *Electr. Notes Theor. Comput. Sci.*, 139(1):3–23, 2005.

[33] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, volume Lecture Notes in Computer Science 2239, pages 93–107. Springer, 2001.

[34] F.Focacci and M.Milano. Global cut framework for removing symmetries. In T. Walsh, editor, *Principles and Practice of Constraint Programming - CP 2001*, volume Lecture Notes in Computer Science 2239, pages 77–92. Springer, 2001.

[35] P. Flener, A. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, and T. Walsh. Matrix modelling. In *Proceedings of the CP'01 Workshop on Modelling and Problem Formulation*, pages 1–7, 2001.

[36] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In P. van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, LNCS 2470, pages 462–476. Springer, 2002.

[37] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *AIPS*, pages 83–91. AAAI, 2002. ISBN 1-57735-142-8.

[38] M. Fox and D. Long. The detection and exploitation of symmetry in planning prob-

lems. In T. Dean, editor, *IJCAI*, pages 956–961. Morgan Kaufmann, 1999. ISBN 1-55860-613-0.

[39] E. Freuder. Eliminating Interchangeable Values in Constraint Satisfaction Problems. In *AAAI-91*, pages 227–233, 1991.

[40] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In P. van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 93–108, 2002.

[41] A. M. Frisch and W. Harvey. Constraints for breaking all row and column symmetries in a three-by-two matrix. In *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems"*,, September 2003.

[42] A. M. Frisch, C. Jefferson, and I. Miguel. Constraints for breaking more row and column symmetries. In Rossi [104], pages 318–332. ISBN 3-540-20202-1.

[43] A. M. Frisch, I. Miguel, Z. Kiziltan, B. Hnich, and T. Walsh. Multiset ordering constraints. In Gottlob and Walsh [57], pages 221–226.

[44] A. M. Frisch, C. Jefferson, and I. Miguel. Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern. In de Mántaras and Saitta [25], pages 171–175. ISBN 1-58603-452-9.

[45] A. M. Frisch, C. Jefferson, B. Martinez-Hernández, and I. Miguel. The rules of constraint modelling. In *IJCAI*, pages 109–116, 2005.

[46] *GAP – Groups, Algorithms, and Programming, Version 4.2*. The GAP Group, 2000. http://www.gap-system.org.

[47] M. Gardner. Chess queens and maximum unattacked cells. *Math Horizon*, pages 12–16, November 1999.

[48] I. Gent and I. McDonald. NuSBDS: Symmetry Breaking made Easy. In B. Smith, I. Gent, and W. Harvey, editors, *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems*, pages 153–160, 2003. URL `http://scom.hud.ac.uk/scombms/SymCon03/notes.html`.

[49] I. Gent, S. Linton, and B. 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/apesreports.html.

[50] I. Gent, I. McDonald, and B. Smith. Conditional symmetry in the all-interval series problem. In B. Smith, I. Gent, and W. Harvey, editors, *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems*, pages 55–65, 2003.

[51] I. P. Gent and B. M. Smith. Symmetry breaking in constraint programming. In *Proceedings of European Conference on Artificial Intelligence - ECAI 2000*, pages 599–603. IOS press, 2000.

[52] I. P. Gent, W. Harvey, and T. Kelsey. Groups and Constraints: Symmetry Breaking During Search, 2002.

[53] I. P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD Using Computational Group Theory. In F. Rossi, editor, *Principles and Practice of Constraint Programming - CP2003*, LNCS 2833, pages 333–347. Springer, 2003.

[54] I. P. Gent, T. Kelsey, S. Linton, I. McDonald, I. Miguel, and B. M. Smith. Conditional symmetry breaking. In van Beek [118], pages 256–270. ISBN 3-540-29238-1.

[55] I. P. Gent, P. Nightingale, and K. Stergiou. Qcsp-solve: A solver for quantified constraint satisfaction problems. In Kaelbling and Saffiotti [68], pages 138–143.

ISBN 0938075934.

[56] C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[57] G. Gottlob and T. Walsh, editors. *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, 2003. Morgan Kaufmann.

[58] P. Gregory and A. F. Donaldson. Conclusions of the SymNET workshop on almost-symmetry in search. In *Almost-Symmetry in Search* Donaldson and Gregory [29], pages 60–61.

[59] W. Harvey. Symmetry Breaking and the Social Golfer Problem. In *Proceedings SymCon-01: Symmetry in Constraints*, pages 9–16, 2001.

[60] W. Harvey. A note on the compatibility of static symmetry breaking constraints and dynamic symmetry breaking methods. In *Proceedings SymCon-04: Symmetry and Constraint Satisfaction Problems*, pages 42–47, 2004.

[61] W. Harvey. Symmetric relaxation techniques for constraint programming. In *Almost-Symmetry in Search* Donaldson and Gregory [29], pages 50–59.

[62] W. Harvey and T. Winterer. Solving the molr and social golfers problems. In van Beek [118], pages 286–300. ISBN 3-540-29238-1.

[63] W. Harvey, T. Kelsey, and K. Petrie. Symmetry Group Expressions for CSPs. In *Proceedings SymCon-03: Third International workshop on Symmetry in Constraint Satisfaction Problems*, pages 86–96, 2003.

[64] P. V. Hentenryck, P. Flener, J. Pearson, and M. Agren. Tractable Symmetry Breaking for CSPs with Interchangeable Values. In *International Joint Conference on Artificial Intelligence - IJCAI 2003*, pages 277–282, 2003.

[65] D. F. Holt, B. Eick, and E. A. O'Brien. *Handbook of Computational Group Theory*. Chapman and Hall/CRC, 2005. ISBN 1 58488 372 3.

[66] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[67] C. Jefferson and A. M. Frisch. Representations of sets and multisets in constraint programming. In B. Hnich, P. Prosser, and B. Smith, editors, *The Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems*, pages 102–116, 2005.

[68] L. P. Kaelbling and A. Saffiotti, editors. *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005*, 2005. Professional Book Center. ISBN 0938075934.

[69] T. Kelsey, S. Linton, and C. M. Roney-Dougal. New developments in symmetry breaking in search using computational group theory. In B. Buchberger and J. A. Campbell, editors, *AISC*, volume 3249 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2004. ISBN 3-540-23212-5.

[70] Z. Kiziltan and B. M. Smith. Symmetry-Breaking Constraints for Matrix Models. In *Proceedings SymCon-02: Symmetry in Constraints*, pages 1–8, 2002.

[71] B. Krishnamurthy. Short proofs for tricky formulas. *Acta Inf.*, 22(3):253–275, 1985.

[72] A. Lal, B. Y. Choueiry, and E. C. Freuder. Neighborhood interchangeability and dynamic bundling for non-binary finite CSPs. In M. M. Veloso and S. Kambhampati, editors, *AAAI*, pages 397–404. AAAI Press AAAI Press / The MIT Press, 2005. ISBN 1-57735-236-X.

[73] Y. C. Law and J. H.-M. Lee. Model induction: A new source of CSP model redun-

dancy. In *AAAI/IAAI*, pages 54–60, 2002.

[74] Y. C. Law and J. H.-M. Lee. Symmetry breaking constraints for value symmetries in constraint satisfaction. *Constraints*. Forthcoming.

[75] Y. C. Law and J. H.-M. Lee. Global constraints for integer and set value precedence. In Wallace [120], pages 362–376. ISBN 3-540-23241-9.

[76] J. S. Leon. Permutation group algorithms based on partitions, i: Theory and algorithms. *J. Symb. Comput.*, 12(4/5):533–583, 1991.

[77] E. Luks and A. Roy. The complexity of symmetry-breaking formulas. *Ann. Math. Artif. Intell.*, 41(1):19–45, 2004.

[78] 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.

[79] F. Margot. Pruning by isomorphism in branch-and-cut. *Mathematical Programming*, 94:71–90, 2002.

[80] F. Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming Ser B.*, 98:3–21, 2003.

[81] R. Martin. Approaches to symmetry breaking for weak symmetries. In *Almost-Symmetry in Search* Donaldson and Gregory [29], pages 37–49.

[82] I. McDonald and B. Smith. Partial symmetry breaking. In P. V. Hentenryck, editor, *Principles and Practice of Constraint Programming*, pages 431–445, 2002.

[83] B. D. McKay. Practical graph isomorphism. *Congressium Numerantium*, 30:45–87, 1981.

[84] P. Meseguer and C. Torras. Exploiting Symmetries within Constraint Satisfaction Search. *Artificial Intelligence*, 129:133–163, 2001.

[85] J. Molony. *Symmetry and Complexity in Propositional Reasoning*. PhD thesis, University of Edinburgh, 1999.

[86] N. Peltier. A new method for automated finite model building exploiting failures and symmetries. *J. Log. Comput.*, 8(4):511–543, 1998.

[87] A. Petcu and B. Faltings. Applying interchangeability techniques to the distributed breakout algorithm. In Gottlob and Walsh [57], pages 1381–1382.

[88] K. E. Petrie. *Constraint Programming, Search and Symmetry*. PhD thesis, University of Huddersfield, 2005.

[89] K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. In Rossi [104], pages 930–934. ISBN 3-540-20202-1.

[90] K. E. Petrie, B. M. Smith, and N. Yorke-Smith. Dynamic symmetry breaking in constraint programming and linear programming hybrids. In *In the proceedings of the Second Starting AI Researchers' Symposium - STAIRS 2004*, volume Frontiers in Artificial Intelligence and Applications 109, pages 96–106. IOS Press, 2004.

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

[92] S. Prestwich. Supersymmetric modeling for local search. In *Second International Workshop on Symmetry in Constraint Satisfaction Problems*, 2002.

[93] S. D. Prestwich. Full dynamic substitutability by sat encoding. In Wallace [120], pages 512–526. ISBN 3-540-23241-9.

[94] S. D. Prestwich. First-solution search with symmetry breaking and implied constraints. In *CP-01 Workshop on Modelling and Problem Formulation*, 2001.

[95] S. D. Prestwich and A. Roli. Symmetry breaking and local search spaces. In

R. Barták and M. Milano, editors, *CPAIOR*, volume 3524 of *Lecture Notes in Computer Science*, pages 273–287. Springer, 2005. ISBN 3-540-26152-4.

[96] L. G. Proll and B. M. Smith. ILP and Constraint Programming Approaches to a Template Design Problem. *INFORMS Journal on Computing*, 10:265–275, 1998.

[97] J.-F. Puget. On the satisfiability of symmetrical constraint satisfaction problems. In *Methodologies for Intelligent Systems (Proceedings of ISMIS'93)*, LNAI 689, pages 350–361. Springer, 1993.

[98] J.-F. Puget. Symmetry breaking using stabilizers. In Rossi [104], pages 585–599. ISBN 3-540-20202-1.

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

[100] J.-F. Puget. Automatic detection of variable and value symmetries. In van Beek [118], pages 475–489. ISBN 3-540-29238-1.

[101] J.-F. Puget. Breaking symmetries in all-different problems. In *IJCAI*, pages 272–277, 2005.

[102] J.-F. Puget. Symmetry breaking revisited. In P. V. Hentenryck, editor, *CP*, volume 2470 of *Lecture Notes in Computer Science*, pages 446–461. Springer, 2002. ISBN 3-540-44120-4.

[103] C. M. Roney-Dougal, I. P. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In de Mántaras and Saitta [25], pages 211–215. ISBN 1-58603-452-9.

[104] F. Rossi, editor. *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, 2003. Springer. ISBN 3-540-20202-1.

[105] P. Roy and F. Pachet. Using Symmetry of Global Constraints to Speed up the Resolution of Constraint Satisfaction Problems. In *Workshop on Non Binary Constraints, ECAI-98*, August 1998.

[106] A. Sadler and C. Gervet. Hybrid set domains to strengthen constraint propagation and reduce symmetries. In Wallace [120], pages 604–618. ISBN 3-540-23241-9.

[107] M. Sellmann and P. V. Hentenryck. Structural symmetry breaking. In Kaelbling and Saffiotti [68], pages 298–303. ISBN 0938075934.

[108] A. Seress. *Permutation group algorithms*. Number 152 in Cambridge tracts in mathematics. Cambridge University Press, 2002.

[109] I. Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, To appear. Earlier version presented at the SAT 01 Workshop.

[110] A. P. Sistla, V. Gyuris, and E. A. Emerson. Smc: a symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9(2):133–166, 2000.

[111] A. Smaill. Symmetry in boolean constraints: Some complexity issues. In *Tenth Workshop on Automated Reasoning*, 2003.

[112] B. Smith. Personal communication to Warwick Harvey, 2004.

[113] B. M. Smith. Reducing Symmetry in a Combinatorial Design Problem. Technical report, School of Computer Studies, University of Leeds, Jan. 2001.

[114] B. M. Smith. A dual graph translation of a problem in 'Life'. In *Principles and Practice of Constraint Programming - CP2002*, LNCS 2470, pages 402–414. Springer, 2002.

[115] B. M. Smith, K. E. Petrie, and I. P. Gent. Models and symmetry breaking for 'peaceable armies of queens'. In J.-C. Régin and M. Rueher, editors, *CPAIOR*, volume 3011 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2004. ISBN 3-540-21836-X.

[116] L. H. Soicher. Computing with graphs and groups. In L. Beineke and R. Wilson, editors, *Topics in Algebraic Graph Theory*, pages 250–266. Cambridge University Press, 2004.

[117] A. Urquhart. The symmetry rule in propositional logic. *Discrete Applied Mathematics*, 96-97:177–193, 1999.

[118] P. van Beek, editor. *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*, 2005. Springer. ISBN 3-540-29238-1.

[119] M. Vasquez and D. Habet. Complete and incomplete algorithms for the queen graph coloring problem. In de Mántaras and Saitta [25], pages 226–230. ISBN 1-58603-452-9.

[120] M. Wallace, editor. *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *Lecture Notes in Computer Science*, 2004. Springer. ISBN 3-540-23241-9.