Symmetry Group Expression for CSPs

Warwick Harvey[1], Tom Kelsey[2], and Karen Petrie[3]

[1] IC-Parc, Imperial College London
Exhibition Road, London, SW7 2AZ, UK
wh@icparc.ic.ac.uk

[2] School of Computer Science, University of St Andrews
St Andrews, Fife, KY16 9SS, UK
tom@dcs.st-and.ac.uk

[3] School of Computing and Engineering, University of Huddersfield
Queensgate, Huddersfield, HD1 3DH, UK
k.e.petrie@hud.ac.uk

Abstract. We describe methods for the expression of symmetry groups for constraint satisfaction problems. The methods provide groups which can be used in search-based symmetry breaking techniques. In particular, we focus on groups used by the GAP–ECL^{i}PS^e interface, in which group theoretic results are supplied to the ECL^{i}PS^e constraints system by the GAP computational algebra system. Our main contribution is a way for constraint programmers to express symmetries in terms of the constraint model, avoiding the need for detailed group theory knowledge, and relieving them of most of the problem-specific implementation burden previously required when using these techniques.

1 Introduction

The aim of this paper is to describe methods for expressing the symmetry associated with a Constraint Satisfaction Problem (henceforth CSP). *Group Theory* is the mathematical study of symmetries, and so groups are a natural way to express the symmetry of a CSP. Thus it is no surprise that constraint practitioners wishing to deal with symmetry are increasingly turning to computational group theory (CGT) methods. In particular, the two approaches Symmetry Breaking During Search (SBDS) [3, 9] and Symmetry Breaking via Dominance Detection (SBDD) [4, 5] have recently had variants developed which use CGT methods [6, 7]. In both of these implementations, the modelling, constraining and search is performed in ECL^{i}PS^e (a system for constraint logic programming), while the symmetry breaking performed at choice points is guided by answers to group theoretic questions asked of GAP (a system for symbolic computation specialising in group theory).

A key issue in expressing these groups is that from a CSP point of view, a symmetry is quite naturally expressed as a permutation of value-to-variable assignment pairs, either across the variables or values or a combination, which preserves solutions. GAP, on the other hand, treats symmetries as permutations of a set $\{1,\ldots,N\}$ of points, with N taking the least value necessary to preserve the correct algebraic structure. This means care must be taken when identifying the symmetry group of a CSP and then generating it in GAP. We must either extend a GAP group to the correct number of points (typically one for each value-to-variable assignment pair), or supply generators involving permutations of the correct number of points. It should be stressed that the different number of points involved has no effect on the resulting algebraic structure: there will be one permutation for each valid symmetry, and the permutations will combine in exactly the same manner.

It will also be obvious from the above that there needs to be an agreed-upon mapping between CSP variable-value assignments and GAP points; in prior GAP–ECL^{i}PS^e implementations this mapping had to be provided by the user, in the form of a pair of functions (one for mapping in each direction). The main result of this paper is the description of an interface which allows the CSP user to generate the correct group — acting on the correct set of points, and with the correct mappings — in a concise and efficient manner. The resulting generators are supplied to GAP, with search and symmetry breaking proceeding in the usual way.

Note that an alternative to requiring the user to provide the symmetries of a problem is to have them discovered automatically. This has been done for SAT problems [1] and pseudo-boolean problems [2], but to the best of our knowledge there is as yet no method that works for general constraint problems. Such techniques, when developed, are also likely to be remain restricted to detecting structural symmetries; that is symmetries that always map objects of a given type in the model to objects of the same type. While this likely accounts for the vast majority of symmetries encountered in practice, it for instance misses half the symmetries of the N Queens problem when using the standard model (one integer variable for each row indicating the location of the queen). This is due to the fact that some symmetries interchange the roles of variables and values in this model. Thus we see a present need, and potential ongoing scope for, making the expression of symmetries for constrint problems as easy as possible.

In the next Section we describe, using examples, the basic ideas of symmetry group generation. Section 3 details more advanced topics, such as wreath products and dihedral groups. In Section 4 we present our simplified framework for expressing symmetries.

2 Basic group generation using GAP

We start with a simple example. The fractions puzzle involves labelling each of 9 variables, $\{A,B,\ldots,I\}$, with a different value from $\{1,2,\ldots,9\}$, such that the equality

$$\frac{A}{BC}+\frac{D}{EF}+\frac{G}{HI}=1$$

holds. There are six permutations of the summands:

1. the identity permutation — permute nothing;

2. swap $\frac{A}{BC}$ with $\frac{D}{EF}$;
3. swap $\frac{A}{BC}$ with $\frac{G}{HI}$;
4. swap $\frac{G}{HI}$ with $\frac{D}{EF}$;
5. cycle $\frac{A}{BC}$ to $\frac{D}{EF}$, $\frac{D}{EF}$ to $\frac{G}{HI}$, and $\frac{G}{HI}$ to $\frac{A}{BC}$;
6. cycle $\frac{A}{BC}$ to $\frac{G}{HI}$, $\frac{G}{HI}$ to $\frac{D}{EF}$, and $\frac{D}{EF}$ to $\frac{A}{BC}$.

If we have a solution (or non-solution), then any permutation of the summands is also a solution (respectively non-solution). These permutations form a group — known as the symmetric group acting on three points, or S_3. In order for a set of permutations to form a group, they must satisfy the following three axioms: (i) any combination of them results in one of them, (ii) combining any of them with the identity gives the original permutation, and (iii) each of them has a unique inverse permutation which gives the identity after combination. It is easy to verify that the above set of permutations satisfy these axioms; for example, applying permutation 6 followed by permutation 5 gives the identity.

The generators of a group are a set of permutations which when combined form the complete group; this is a standard way for expressing groups in a concise fashion. S_3 can be generated from any one of permutations 2, 3 or 4, plus either of permutations 5 and 6. If we choose permutations 3 and 5 as generators, we see that 3 followed by 5 is permutation 4, 5 followed by 3 is permutation 2, and 2 followed by 4 is permutation 6 which is equivalent to 3 by 5 then 5 by 3. The identity is any combination of a permutation with its inverse.

Generating this group in GAP is easy:

```
gap> g1 := SymmetricGroup(3);;
```

We can also generate a list of all the elements of the group using the GAP command *AsList*:

```
gap> AsList(g1);
[ (), (2,3), (1,2), (1,2,3), (1,3,2), (1,3) ]
```

It is easily checked that the GAP group has the same structure as the symmetries of our fractions puzzle if we supply an injective map from the summands to the points $\{1, 2, 3\}$, and note that the identity is typeset as empty parentheses. GAP uses cycle notation by default; the element (1,2) is interpreted as "point 1 goes to point 2, point 2 goes to point 1; point 3 is not moved". Sometimes it can be easier to interpret permutations in list notation; the GAP command *ListPerm* can be used for this purpose:

```
gap> l := List([1..6], i -> ListPerm(AsList(g1)[i]));
[ [ ], [1,3,2], [2,1], [2,3,1], [3,1,2], [3,2,1] ]
```

Here, the ith element of each list is the position of point i after the permutation.

To solve the puzzle with symmetry breaking using one of the GAP–ECL^{i}PS^e methods we construct a 9×9 symmetry array. Each element of the array corresponds to an assignment of a value to a variable. Suppose we choose that the columns relate to the variables. We need to generate S_3 acting on 81 points, with solution symmetries being preserved. A straightforward approach is to denote each variable by its column index, giving $A = 1, B = 2, \ldots, I = 9$. We can then write down generators for S_3 acting on 9 points and generate the group:

```
gap> p1 := (1,7)(2,8)(3,9);;
gap> p2 := (1,4,7)(2,5,8)(3,6,9);;
gap> g2 := Group(p1,p2);
Group([ (1,7)(2,8)(3,9), (1,4,7)(2,5,8)(3,6,9) ])
gap> LargestMovedPoint(g2);
9
```

Since there is no value symmetry in the problem, it only remains to extend the group so that it acts the same way in each row of our symmetry array. Suppose we assign points to the elements of the symmetry array row-wise, so that the first row comprises the points $1 \ldots 9$, the second row $10 \ldots 18$, etc. Thus, since rows correspond to values and columns to variables, the point corresponding to assigning the value i to the variable j is $j + 9 * (i - 1)$. We can take the generator `p1` above (which swaps the first three variables with the last three) and extend it to all rows using a simple GAP function:

```
gap> l1 := List([1..9], i -> List([1..9], j ->
                                        ListPerm(p1)[j]+9*(i-1)));;
gap> p3 := PermList(Concatenation(l1));
( 1, 7)( 2, 8)( 3, 9)(10,16)(11,17)(12,18)(19,25)(20,26)(21,27)
(28,34)(29,35)(30,36)(37,43)(38,44)(39,45)(46,52)(47,53)(48,54)
(55,61)(56,62)(57,63)(64,70)(65,71)(66,72)(73,79)(74,80)(75,81)
```

Hence our final symmetry group is generated by two extended generators of S_3.

The above method is easily extended to deal with full symmetries of values, and to multi-dimensional models (such as matrix models), where our symmetry array consists of one copy of the model array for each value. Moreover, routines that map between array indices and variable-value pairs are straightforward; in the above example, the *i,j*th element of the array corresponds to value i and variable j, so arithmetic modulo 9 is all that is needed.

3 Advanced group generation using GAP

In many problems the symmetry group is not the set of all permutations of variable and/or value points. Two common examples are geometric symmetries which arise in problems such as N Queens, and repeated block symmetries which arise in problems with several copies of the same structure, where each copy has the same solution symmetries. Again, using GAP, the user has a choice of two approaches: either (i) identify the group, give GAP its name, and extend to the correct number of points, or (ii) write down an example of each distinct symmetry of the solution array, and use these to generate the group. We provide an example of each method.

3.1 Wreath products

Consider the problem of getting 8 groupings of golfers from 32 players for 10 weeks, such that no player is in the same grouping as another in more than one week. If we ignore player symmetries, there are 10 blocks — one per week — of groupings. The 8 groupings in each block can be permuted in any way, giving S_8. In group theoretic terms, we form 10 copies of S_8 by Cartesian product, define the action of a permutation of the weeks on a fixed permutation of the blocks, form the set of pairs from the group of 10 blocks and the symmetry group of the weeks, and specify the semi-direct product on these pairs. This is known as forming the wreath product of S_8 with S_{10}, and is an inbuilt GAP routine:

```
gap> WreathProduct(SymmetricGroup(8),SymmetricGroup(10));
<permutation group of size
 41206740932217923537836077156872289779712000000000000
with
22 generators>
```

This group acts on 80 points — one for each week-grouping pair — and can easily be extended to act on a suitable number of points for whichever player model is under consideration, as long as one knows which week-grouping pair corresponds to each point.

3.2 Geometric symmetries

There are 8 symmetries in a standard model of the N Queens problem. If we label the corners of the chessboard as $1 \mapsto$ top left, $2 \mapsto$ top right, $3 \mapsto$ bottom right, and $4 \mapsto$ bottom left, then it is simple to generate the symmetry group. We write down one non-trivial rotation and one non-trivial reflection, and then we can generate the group and check that it is of size 8:

```
gap> rot := (1,2,3,4);;
gap> ref := (1,2)(3,4);;
gap> g4 := AsList(Group(rot,ref));
[(),(2,4),(1,2)(3,4),(1,2,3,4),(1,3),
             (1,3)(2,4),(1,4,3,2),(1,4)(2,3)]
gap> Size(g4);
8
```

Again extending the group to the desired number of points is usually straightforward.

3.3 Problems with the GAP approach

So far we have seen that generating symmetry groups in GAP is a relatively simple process. There are, however, two drawbacks that often arise in practice. The first is the provision of a map between points and variable-value pairs.

Usually one chooses a fairly straightforward mapping and then takes care to generate/extend the group in the right way to match it. We have written a number of functions in GAP for generating common symmetry groups to match standard mappings, and for extending permutations on, say, variables to take into account values, and this helps. But it becomes harder with more involved symmetric structures, and the user has to know at least some GAP if they wish to use symmetries for which functions have not been provided. Mismatches between the mapping used and the group generated are also an unwanted source of bugs.

The second problem involves the combination of symmetry groups. If our CSP has several types of structures, each with its own symmetry, and with each type possibly having a wreath product with other types, then it can be hard to form a group in GAP that both supplies the correct symmetries and acts on a number of points that can be easily extended to the points of a symmetry array. GAP will always form a group acting on the fewest points needed; this is an advantage to the computational group theorist, but an inconvenience to the CSP practitioner using computational group theory methods.

4 Easier symmetry group generation

We would like to have a way of expressing symmetries for CSPs that relieves the user of much of the burden currently imposed. In particular, we would like to:

- avoid the necessity of coding mapping functions;
- allow the expression of the symmetries in terms of the CSP model in a simple yet powerful way;
- require only minimal (if any) knowledge of group theory.

As well as reducing much of the drudge work required to construct the symmetry group, this would make it much easier for a constraint practitioner without specialist group theory knowledge to apply advanced symmetry-breaking techniques to their problems.

We believe the interface presented in this section achieves these aims for symmetries in (multi-dimensional) matrix models. We have implemented it for the GAP–ECL^{i}PS^e interface and used it in conjunction with the SBDD approach presented in [7] and the SBDS approach presented in [6]. The implementation automatically determines a suitable mapping between assignments in the constraint model and GAP points, and constructs an appropriate set of generators operating on the right number of points in the right way.

Note that we are not the first to try to provide easy access to symmetry groups for constraint programmers; Iain McDonald's Nu-SBDS system [11] provided a simple way to specify common symmetry groups. Our approach is deliberately more powerful and flexible; for example it can be used to express the symmetries of the social golfer problem, something which currently cannot be done with Nu-SBDS due to the wreath product involved.

4.1 Expressing symmetries

Before expressing the symmetries of the problem, we require that the user provide some basic information about the model. Specifically, we require information about:

- the number of dimensions in the array containing the search variables;
- the extent (range) of each dimension; and
- the range of values the variables may take.

This information specifies how the symmetries should act on the variables and values, and allows us to construct a suitable mapping between assignments and GAP points. In addition, we require that the user provides:

- a name for each of the variable dimensions plus the value "dimension"

(as far as the symmetry expression is concerned, the values are just another dimension, indistinguishable from the variable dimensions). As well as providing a convenient way for the user to refer to the dimensions, this actually makes the symmetry expression more robust with respect to changes to the model (adding/removing/rearranging dimensions). Indeed, it is not uncommon for different models of the same problem to have exactly the same symmetry expression as a result.

The expression of a symmetry comes in three parts. The first part specifies the nature of the symmetry (e.g. full permutation, or the symmetries of a square). Each kind of symmetry usually applies to a particular number of dimensions; the second part of the symmetry expression specifies which dimension(s) the symmetry applies to (e.g. the full permutation should be applied to the rows, or the symmetries of a square should be applied to the rows and columns taken together). Note that sometimes a symmetry should not apply to a whole dimension, but just to a subset of the indices of a dimension or dimensions (e.g. permuting a subset of the columns, or applying the symmetries of a square to just the top left quarter of a matrix), so we allow the dimensions specified to be annotated to indicate this. This annotation can also be used to rearrange the indices in a dimension; this is useful if position is important for the symmetry but the order assumed by the symmetry does not match that of the problem.

The final part of a symmetry specification indicates what should be done with the other dimensions. Often, all the indices of the other dimensions should be treated in exactly the same way (e.g. when permuting the columns of a matrix, all rows should be affected in the same way, as should all values). Other times, some indices should be affected while others are not (e.g. permuting the groups in one round of a tournament schedule while leaving the other rounds unchanged, or permuting the columns of the first three rows of a matrix). By default, all indices of the other dimensions are affected synchronously; if some indices of some dimensions are to be affected while others not, this can be specified in the (optional) third part of a symmetry specification.

Typically one such expressed symmetry does not suffice for describing all the symmetries of a problem. Thus the user may provide a list of such symmetry specifications. This will be interpreted as meaning all the symmetries from each of the specifications, plus all possible combinations of these symmetries. (In group-theoretic terms, we take the generators corresponding to each of the specifications and put them together to generate a new group, which will be isomorphic to the direct product of the groups corresponding to each individual specification.)

4.2 Examples

Example 1 (Balanced Incomplete Block Design (BIBD)). Suppose we have the standard model for a BIBD, with a two-dimensional array of booleans. The symmetries of this problem with this model are full row and full column permutation. If the row dimension is called `rows` and the column dimension called `cols`, then the expression of the symmetries is just:

```
   symmetry(full_perm, rows)
   symmetry(full_perm, cols)
```

Example 2 (Social Golfer Problem). Suppose we have a boolean model for the Social Golfer Problem, with one boolean variable for each round/group/player combination. The symmetries of this problem with this model are full permutation of the rounds (`rounds`), full permutation of the groups (`groups`) within any round, and full permutation of the players (`players`):

```
   symmetry(full_perm, rounds)
   symmetry(full_perm, groups, rounds:1)
   symmetry(full_perm, players)
```

Note that the second line indicates that the full permutation of the groups specified operates on just the first round, leaving the other rounds unchanged. This permutation of the groups within a round does not need to be specified for every round; the fact that the rounds are interchangeable means that specifying it for one round is sufficient.

Example 3 (N Queens Problem). Suppose we have the standard model for the N Queens Problem, with one integer variable for each row, specifying which column the queen appears in. The symmetries of this problem with this model are those of the square, including reflection, applied to the row/column combination:

```
   symmetry(square, [rows, cols])
```

Note that the fact that the columns of the board correspond to the value dimension rather than a variable dimension is irrelevant for the specification of the symmetries. Indeed, exactly the same specification can be given if the model is a boolean model, with one boolean variable per square on the board. However, the group given to GAP will operate on a different number of points, and the mapping between CSP assignments and GAP points will be different; these differences derive from differences in the specification of the dimensions.

Example 4 (Miscellaneous examples).
Permuting a subset of the columns:

```
   symmetry(full_perm, cols:[1,3,5..7])
```

Applying the symmetries of a square to the top left 4×4 submatrix:

```
    symmetry(square, [rows:1..4, cols:1..4])
```
Permuting the columns of the first three rows of a matrix:
```
    symmetry(full_perm, cols, rows:1..3)
```
Reversing the order of the values (i.e. swapping 1 and n, 2 and $n-1$, etc. — can also be thought of as a reflection in the given dimension):
```
    symmetry(reverse, values)
```
Swapping value pairs $1 \leftrightarrow 2$, $3 \leftrightarrow 4$, $5 \leftrightarrow 6$ and $7 \leftrightarrow 8$ (works by rearranging the values and using the reversal symmetry):
```
    symmetry(reverse, values:[1,3,5,7,8,6,4,2])
```

4.3 User-provided symmetries

Of course, we only provide a limited number of pre-defined symmetries (permutations, rotations, reflections, etc.) and sometimes the user will wish to express a symmetry which is not provided. Currently any such symmetry must be added to the library before it can be used. However, we envisage extending the library to allow user-provided symmetries in a number of ways. The first is by providing a function which, when given an index for each of the dimensions the symmetry operates on (and, say, some information about the size of those dimensions), returns the indices that result from applying that symmetry. The second is by providing a function which, when given information about the number and size of the dimensions to be acted upon, returns an array of that size, where each entry in the array indicates where that entry would be mapped to by the symmetry (effectively, a tabulated version of the user-provided function). The final way is by providing a function which, when given information about the number and size of the dimensions to be acted upon, returns a GAP group representing the symmetry. Of course, with this last method, some mapping must be agreed upon between the points acted upon by the GAP group and the indices of the dimension(s) acted upon by the symmetry.

In each case, the user need only provide an implementation for the base symmetry; support for operating on only a subset of indices and handling the presence of other dimensions is all handled automatically.

4.4 Extensions and Generalisations

There are a number of ways this symmetry expression interface could be extended or generalised. One such extension, which has been implemented, is to support set variables rather than finite domain variables. In this case the decisions are whether a value should be in a set or excluded from it (rather than whether a value should be assigned to a variable or not). This simply requires a different mapping between the decisions and the GAP points.

It ought to be possible to make a similar extension, again by varying the mapping between decisions and points, for when the branching decisions are $\leq$ vs. $>$ rather than $=$ vs. $\neq$; at least in the absence of value symmetry. More challenging is to come up with a generalisation that would cope with any suitable combination of domain variable and branching decision.

Also, it would be nice to support more than just pure matrix models. At the very least it ought to be possible for the user to have multiple arrays of variables, if that suits their model better.

Finally, this way of expressing symmetries is in no way limited to methods based on the GAP–ECL^{i}PS^e interface. For example it ought to be possible to implement something similar for other combinations of constraint solver and computational group theory tool, assuming they have suitable interfacing facilities. But more than this, it can be used for expressing symmetries in any constraint programming context. For instance, with a different back end it could generate the symmetry functions required for "classic" SBDS [3,9][4], a set of symmetry breaking constraints (complete or otherwise), or perhaps even a problem-specific dominance checker for SBDD. Not only is using our interface for expressing symmetries easier than doing any of these things directly, but having a common interface would make comparison of different symmetry-breaking techniques much easier, by avoiding the need to do problem-specific implementation for each technique.

4.5 Implementation

Implementation of the interface is relatively straightforward. Based on the information about the model provided by the user (number and size of dimensions), one determines a suitable mapping between GAP points and the variable-value pairs of the constraint model. Each basic symmetry (full permutation, rotation, etc.) generally corresponds to one or two generators which are easy to construct for the size(s) of the dimension(s) they apply to. The only somewhat involved part of the process is then taking these base generators and mapping/extending them to the appropriate points, based on the dimension specifiers for the symmetry — but this is simply a matter of programming. The generating set for the problem as a whole is then just the set of generators constructed in this way for each of the symmetry specifiers.

5 Conclusions

We have presented a number of ways of generating the groups required for expressing the symmetries of a problem, as required for symmetry breaking techniques based on the GAP–ECL^{i}PS^e interface (among others). In particular, we have presented a user interface which provides a relatively simple yet powerful way of expressing symmetries in terms of the CSP model, without requiring any specialist knowledge of a system such as GAP. We have used this interface to express the symmetries of quite a few problems (e.g. it was used in [10]), and it does greatly reduce the complexity of specifying symmetries for use with the implementations described in [6,7]. In particular, it has allowed us to express

[4] GAP has already been used to generate SBDS symmetry functions from a group for the Alien Tiles Puzzle [8], for example.

the symmetries of a problem with such complicated (instance-specific) wreath product combinations that it was just too complex to contemplate doing any other way.

Acknowledgements

This work is supported by EPSRC grants GR/R29666, GR/S30658/01 and GR/R29673. We are thankful to Neil Yorke-Smith for his useful comments on this work.

References

1. Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah, *Solving difficult SAT instances in the presence of symmetry*, Proc. SAT 2002, 2002, pp. 338–345.
2. Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah, *Symmetry-breaking for pseudo-boolean formulas*, Proc. SymCon'03, 2003, to appear.
3. R. Backofen and S. Will, *Excluding symmetries in constraint-based search*, Proc. CP-99 (J. Jaffar, ed.), LNCS 1713, Springer, 1999, pp. 73–87.
4. Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann, *Symmetry breaking*, Proc. CP 2001 (T. Walsh, ed.), LNCS 2239, 2001, pp. 93–107.
5. Filippo Focacci and Michaela Milano, *Global cut framework for removing symmetries*, Proc. CP 2001 (T. Walsh, ed.), LNCS 2239, 2001, pp. 77–92.
6. Ian P. Gent, Warwick Harvey, and Tom Kelsey, *Groups and constraints: Symmetry breaking during search*, Proc. CP 2002 (P. Van Hentenryck, ed.), LNCS 2470, Springer-Verlag, 2002, pp. 415–430.
7. Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton, *Generic SBDD using computational group theory*, Proc. CP 2003, 2003, to appear.
8. I.P. Gent, S.A. Linton, and B.M. Smith, *Symmetry breaking in the Alien Tiles Puzzle*, Tech. Report APES-22-2000, APES Research Group, October 2000.
9. I.P. Gent and B.M. Smith, *Symmetry breaking in constraint programming*, Proceedings of ECAI-2000 (W. Horn, ed.), IOS Press, 2000, pp. 599–603.
10. Warwick Harvey, *The fully social golfer problem*, Proc. SymCon'03, 2003, to appear.
11. Iain McDonald, *NuSBDS: Symmetry breaking made easy*, Proc. SymCon'03, 2003, to appear.