Groups and Constraints: Symmetry Breaking during Search*

Ian P. Gent[1], Warwick Harvey[2], and Tom Kelsey[1]

[1] School of Computer Science, University of St Andrews
St Andrews, Fife, KY16 9SS, UK
{ipg,tom}@dcs.st-and.ac.uk
[2] IC-Parc, Imperial College
Exhibition Road, London SW7 2AZ, UK
wh@icparc.ic.ac.uk

Abstract. We present an interface between the ECL^{i}PS^e constraint logic programming system and the GAP computational abstract algebra system. The interface provides a method for efficiently dealing with large numbers of symmetries of constraint satisfaction problems for minimal programming effort. We also report an implementation of SBDS using the GAP-ECL^{i}PS^e interface which is capable of handling many more symmetries than previous implementations and provides improved search performance for symmetric constraint satisfaction problems.

1 Introduction

Dealing with symmetries in constraint satisfaction problems has become quite a popular topic for research in recent years. One of the main areas of recent study has been the modification of backtracking search procedures so that they only return unique solutions. Such techniques currently broadly fall into two main categories. The first involves adding constraints whenever backtracking occurs, so that symmetric versions of the failed part of the search tree will not be considered in future [1, 12]; we will collectively refer to these techniques as SBDS (Symmetry Breaking During Search). The second category involves performing checks at nodes in the search tree to see whether they are dominated by the symmetric equivalent of some state already considered [7, 9]. Note that these two kinds of approaches are closely related; the main difference is when and how the symmetry-breaking conditions are enforced.

The SBDS approach as implemented to date (with one exception) has one main drawback when it comes to problems with large numbers of symmetry: it requires an explicit list of all the symmetries of the problem. It works well if the number of symmetries is small, and has been used effectively with a list of up to about 8000 symmetries, but clearly if a problem has billions of symmetries, a complete explicit list is not practical. Since the symmetries of a problem form

* This paper is dedicated to Alex Kelsey, 1991–2002

P. Van Hentenryck (Ed.): CP 2002, LNCS 2470, pp. 415–430, 2002.

a group, one obvious candidate for representing and manipulating these symmetries implicitly is to use computational group theory (CGT). Modern CGT systems such as GAP [10] are very efficient: they allow rapid calculations to be done on large groups without the need to iterate over or explicitly represent more than a tiny fraction of the group elements. As well as offering a clear benefit in both time and space, using a CGT approach can make the expression of the symmetries by the programmer much easier: typically only a handful of example symmetries are required to generate the full symmetry group, even for very large groups.

Iain McDonald [13] has performed some early experiments using group theory to represent the symmetries in an SBDS implementation, but still only handled a few thousand symmetries. In this paper we present a much more sophisticated approach capable of handling several orders of magnitude more symmetries than any previous SBDS implementation. We interface the constraint logic programming system ECL^{i}PS^e [14] with GAP, running as a sub-process. We use GAP to specify the group and perform stabiliser calculations, passing the results back to ECL^{i}PS^e, which uses them to reduce the computational effort required to solve the problem. Note that while the examples in this paper have integer domain values, our implementation is general, allowing other classes of domain variables to be used.

In Section 2 of this paper we introduce the group theory concepts utilised by our system. These include the definition of symmetries as bijective mappings from an initial subset of the positive integers to itself, groups formed by composition of these maps, subgroups, generators of groups, elements which leave points unchanged (stable), and points which are the image of maps (orbits).

We describe the GAP-ECL^{i}PS^e interface in Section 3 and our SBDS implementation in Section 4. Examples of the use of GAP-ECL^{i}PS^e to improve SBDS are given in Section 5. We discuss our results and highlight future avenues of research in Section 6.

2 Group Theory for CSPs

Consider a constraint satisfaction problem, $\langle C, D\rangle$, consisting of a constraint, C, over variables $x_1, \ldots, x_n$, with finite domains $D(x_i)$. Suppose that the domain elements are also indexed from $D(x_i)_1$ to $D(x_i)_{m_i}$. It is possible that $\langle C, D\rangle$ contains symmetries in either the variables, the domain values, or both. By symmetry, we mean a permutation of either the variables or the domain values, or both, which preserves solutions.

Example 1: Symmetry in Domains. A graph colouring problem $\langle C, D\rangle$ with domain

$$D(x_i)_1 = red, D(x_i)_2 = green, D(x_i)_3 = blue$$

for each variable, has the same solutions as the problem $\langle C, D'\rangle$, where D' is any permutation of the indexing of D. Hence $D'(x_i)_1 = blue, D'(x_i)_2 = green, D'(x_i)_3 = red$ is a symmetric version of D. Since there are $n!$ permutations of n elements in a set, $\langle C, D\rangle$ has $3! = 6$ symmetric variants.

Example 2: Symmetry in Variables. Suppose that we wish to solve

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

where each letter is a distinct member of $\{1, \ldots, 9\}$, and BC denotes $10 * B + C$. By the associativity and commutativity of arithmetic over the rationals, if $\{A \mapsto 5, B \mapsto 3, C \mapsto 4, \ldots\}$ is a solution, then so is $\{G \mapsto 5, H \mapsto 3, I \mapsto 4, \ldots\}$. Again we have 3! permutations of the variables which preserve solutions.

Symmetries as Group Elements. A group is a tuple $\langle S, \circ \rangle$ where S is a set and $\circ$ is a closed binary operation over S such that:

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

Let $\Omega = \{1, 2, \ldots N\}$ for some N, where each integer might represent (depending on the nature of the symmetries in a given problem) a CSP variable, a value, or a variable/value pair. Our set, S_Ω, is the set of bijective mappings from Ω to itself (i.e. permutations of the elements of Ω), and we take $\circ$ to be the composition of such mappings. $\langle S_\Omega, \circ \rangle$ forms a group since

1. $\circ$ is clearly closed and associative;
2. the identity mapping, denoted by (), is a neutral element: composing () with any other mapping has no effect;
3. by our restriction to bijective mappings, we ensure that each mapping has an inverse.

$\langle S_\Omega, \circ \rangle$ is known as S_N, the symmetric group over N elements. S_N has $N!$ elements (one for each possible bijective permutation), each of which can be represented by the image of the relevant mapping from Ω to Ω. For example, $f(i) = (i + 1) \mod N$ has the image $[2, \ldots, N, 1]$; and g, the swapping of only points 1 and 2, has the image $[2, 1, 3, \ldots, N]$. The inverses of f and g are easy to describe, and their composition $f \circ g$ has the image $[1, 3, \ldots, N, 2]$.

Each member of S_N is a *permutation* of the numbers $\{1, 2, \ldots, N\}$, and each symmetry of a CSP will have an associated permutation. An arbitrary CSP need not have $N!$ symmetries: for example, an N-queens problem has the number of symmetries of a square, which is 8 for any value of N. This motivates a discussion of subgroups of S_N.

Symmetries as Subgroups. The tuple $\langle T, \circ \rangle$ is a subgroup of $\langle S, \circ \rangle$ if T is subset of S which forms a group under the $\circ$ operation. Trivial subgroups are obtained when T consists of only the identity permutation, and when $T = S$. Lagrange's theorem states that the order (number of elements) of a subgroup divides the order of the group. In terms of CSPs, we wish to identify permutations which *generate* the subgroup of S_N which correctly describes the symmetries of a given CSP. The process of subgroup generation involves choosing a small number

of permutations, and repeatedly forming other permutations by composition until a closed subset of S_Ω is obtained. For example, consider a CSP involving the symmetries of a 2×2 square, in which we have labelled the cells $1 \ldots 4$ from top left via top right and lower left to lower right. The S_4 elements $p_1 = [3, 1, 4, 2]$ and $p_2 = [3, 4, 1, 2]$ define a rotation by 90° and a flip through the horizontal axis respectively. We see that p_1 and p_2 generate a subgroup of S_4 order 8:

$$\begin{aligned}
p_1 \circ p_1 &= [4, 3, 2, 1] && \text{rotation by } 180^\circ \\
p_1 \circ p_1 \circ p_1 &= [2, 4, 1, 3] && \text{rotation by } 270^\circ \\
p_1 \circ p_1 \circ p_1 \circ p_1 &= [1, 2, 3, 4] && \text{rotation by } 360^\circ : \text{ identity} \\
p_2 \circ p_1 &= [4, 2, 3, 1] && \text{rotate by } 90^\circ \text{ then flip} \\
p_1 \circ p_2 &= [1, 3, 2, 4] && \text{flip then rotate by } 90^\circ \\
p_1 \circ p_1 \circ p_2 &= [2, 1, 4, 3] && \text{flip through vertical axis}
\end{aligned}$$

It is straightforward to check that any composition involving only p_1 and p_2 gives one of the above elements. The generated group is known as the dihedral group with 8 elements, D_4, and is clearly a subgroup of S_4. Note that 8 divides $4! = 24$ as required by Lagrange's theorem.

In order to identify and deal with symmetries during search, we need to identify the images of points in Ω after permutation by group elements (*orbits* of points), and those elements which leave certain points unchanged after permutation (*stabilisers* of points). The idea is to keep track of the stabilisers of (the identifiers of) forward labelling steps. If a choice is backtracked, we find its orbit in our symmetry group, and add constraints excluding configurations corresponding to each of the points in the orbit. This is justified since points in the orbit are symmetrically equivalent to the choice point, with respect to the current state of our search.

Orbits, Stabilisers and Cosets. Let G be a permutation group acting on N points, so that G is a subgroup of S_N. We define the *orbit*, $O_i(G)$, under G of a point $i \in \{1, \ldots, N\}$ as

$$O_i(G) = \{g(i) \mid g \in G\}.$$

In the above example, $O_3(D_4) = \{1, 2, 3, 4\}$ since p_2 moves 3 to 1, $p_1 \circ p_1$ moves 3 to 2, the identity leaves 3 unchanged, and p_1 moves 3 to 4.

We define the *stabiliser* of point i in G as

$$Stab_G(i) = \{g \in G \mid g(i) = i\}.$$

Using the same example, $Stab_{D_4}(3) = \{(), p_2 \circ p_1\}$, since applying either of these permutations leaves point 3 unchanged (and no others do).

The concepts of orbit and stabiliser can easily be extended to cover more than one point, and it can be shown that any stabiliser of a point acted on by a group is a subgroup of that group. Moreover, the orbit-stabiliser theorem provides the useful result that $|O_i(G)|\ |Stab_G(i)| = |G|$, i.e. the order of a group is the order of the stabiliser of any point times the size of the associated orbit.

In Section 4 we shall consider chains of stabilisers of points, together with representative permutations of associated orbits. To illustrate this, consider the symmetric group consisting of the 24 permutations of $\{1, 2, 3, 4\}$. We compute a chain of stabilisers of each point, starting arbitrarily with point 1:

$$Stab_{S_4}(1) = \{(), [1, 2, 4, 3], [1, 3, 2, 4], [1, 3, 4, 2], [1, 4, 2, 3], [1, 4, 3, 2]\}.$$

This is the group consisting of the 6 permutations of the set $\{2, 3, 4\}$. We can define a binary relation on the elements of any finite G with subgroup H by setting $a \sim b$ iff $b \circ a^{-1} \in H$. This is an equivalence relation (since $a \sim a$, $a \sim b \Rightarrow b \sim a$, and $a \sim b \wedge b \sim c \Rightarrow a \sim c$), and the equivalence classes are called *right cosets* of H in G. A simple calculation shows that the right cosets of $Stab_{S_4}(1)$ in S_4 are:

$$\begin{aligned}
c_1 &= Stab_{S_4}(1) \\
c_2 &= [2, 1, 4, 3], [2, 1, 3, 4], [2, 4, 3, 1], [2, 3, 4, 1], [2, 4, 1, 3], [2, 3, 1, 4] \\
c_3 &= [3, 4, 1, 2], [3, 4, 2, 1], [3, 1, 2, 4], [3, 2, 1, 4], [3, 1, 4, 2], [3, 2, 4, 1] \\
c_4 &= [4, 3, 2, 1], [4, 3, 1, 2], [4, 2, 1, 3], [4, 1, 2, 3], [4, 2, 3, 1], [4, 1, 3, 2]
\end{aligned}$$

Note that the cosets partition S_4, with c_j consisting of the permutations which send 1 to j. A *right-transversal* of H in G is a list of canonical representatives from the cosets. We only need one representative of each coset, since any member of c_j is equivalent to any other member in terms of where the point fixed by the stabiliser, 1, gets mapped to.

To complete the stabiliser chain we have

$$Stab_{Stab_{S_4}(1)}(2) = \{(), [1, 2, 4, 3]\}$$

which is the subgroup of $Stab_{S_4}(1)$ which leaves point 2 unchanged, and

$$Stab_{Stab_{Stab_{S_4}(1)}(2)}(3) = \{()\}$$

since only the identity permutation leaves point 3 unchanged. Stabiliser chains, in general, collapse quickly to the subgroup containing only the identity since the order of each new stabiliser must divide the order of the stabilisers above it. Once an order 1 stabiliser is reached, all stabilisers further down the chain are trivial.

3 GAP-ECL^{i}PS^e

In this Section we briefly describe the interface between the GAP computational group theory system and the ECL^{i}PS^e Constraint Logic Programming system. The idea is that GAP acts as a black box. While an ECL^{i}PS^e implementation is performing tree search to solve a CSP involving symmetries, GAP is asked to provide group theoretic results such as the symmetry group itself, stabilisers of

points, and members of cosets of stabilisers. The ECL^{i}PS^e implementation uses these results to break any symmetries that arise during search.

GAP [10] (Groups, Algorithms and Programming) is a system for computational discrete algebra with particular emphasis on, but not restricted to, computational group theory. GAP includes command line instructions for generating permutation groups, and for computing stabiliser chains and right transversals. Note that GAP does not explicitly create and store each element of a group. This would be impractical for, say, the symmetric group over 30 points, which has 30! elements. Instead group elements are created and used as required by the computation involved, making use of results such as Lagrange's theorem and the orbit-stabiliser theorem to obtain results efficiently. GAP can also be programmed to perform specific calculations in a modular way. GAP is available from `http://www.gap-system.org/`

ECL^{i}PS^e is a Constraint Logic Programming system which includes libraries for finite domain constraint solving. In addition to its powerful modelling and search capabilities, ECL^{i}PS^e has three important features which we utilise to prune search trees using results from computational group theory.

The first, and most important, feature is efficient communication with subprocesses. It is straightforward to write ECL^{i}PS^e programs which start a GAP subprocess and send and receive information which can be used to prune search. We have implemented an ECL^{i}PS^e module which exports predicates for

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

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

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

4 Using GAP-ECL^{i}PS^e to Break Symmetries

We describe a GAP-ECL^{i}PS^e implementation of SBDS in which symmetric equivalents of assignments are determined by computation within GAP. We assume that the ECL^{i}PS^e user has a constraint satisfaction problem of the form

$$C \wedge x_1 \in D(x_1) \wedge \cdots \wedge x_n \in D(x_n),$$

where C represents the constraints of the problem and $D(x_i)$ is the domain of the ith variable. We first obtain some global symmetry information, and then run a binary backtrack search procedure using SBDS [12]. In the context of a partial assignment A, after the assignment $Var = Val$ fails, SBDS posts the constraint $g(A) \Rightarrow g(Var \neq Val)$ for each element g in the symmetry group. This ensures that if we ever visit a symmetric equivalent of A we never try the equivalent of $Var = Val$.

4.1 Mapping between ECL^{i}PS^e Assignments and GAP Points

Before we can begin describing the symmetries of a CSP, we need to decide on a mapping between the points manipulated by GAP ($1 \ldots N$ for some N) and the constraints imposed by ECL^{i}PS^e during search ($x_i = d_j$, $x_i \neq d_j$). To allow full symmetry generality, one needs to assign a distinct point to each possible assignment; for example, one could assign the points $1 \ldots m$ to the assignments $x_1 = 1, \ldots, x_1 = m$, points $m + 1 \ldots 2m$ to the assignments $x_2 = 1, \ldots, x_2 = m$, etc., but any order will do. Note that we do not have to consider disequality constraints separately since they are affected by symmetries in exactly the same way as the corresponding equality constraint, and the type of constraint ($=$ or $\neq$) will be known from context.

If the problem has only simple variable symmetry (or simple value symmetry), then having one point for each potential assignment is overkill since all assignments involving a given variable (resp. value) are affected by the symmetries in the same way. Instead one can associate points with variables (resp. values), and when computing the symmetric equivalents of assignments retain the value (resp. variable) from the original assignment.

In our implementation, the relevant mapping is encapsulated in a pair of ECL^{i}PS^e predicates. The first takes a variable-value pair and returns the corresponding GAP point. The second takes a GAP point and the original variable-value pair (only used for simple variable or value symmetries), and returns the corresponding variable-value pair.

4.2 Using GAP to Obtain a Symmetry Group

There is no need to explicitly code every symmetry of the CSP. Indeed, this would be impractical for a balanced incomplete block design (henceforth, BIBD) with, say, a 7×7 matrix model, which has full symmetry on both the rows and columns. In other words, any solution matrix is symmetrically equivalent to the same matrix after swapping any number of rows and columns. In this case the symmetry group is $S_7 \times S_7$, the direct product of the row and column symmetry groups, with order $7! \times 7! = 25,401,600$.

A more efficient approach is to identify a small number of symmetries, and use GAP to obtain the generated symmetry group. For example, to generate S_N it is sufficient to provide one permutation that switches exactly two elements, and another permutation that cycles each element by one position. The following GAP session illustrates this for the symmetric group on 7 points:

```
gap> p1 := PermList([2,1,3,4,5,6,7]);;
gap> p2 := PermList([2,3,4,5,6,7,1]);;
gap> g := Group(p1,p2);;
gap> Size(g);
                  5040
```

As a further example we generated the symmetry group of a square from one rotational symmetry and one flip symmetry in Section 2. The greatest number of generators needed is $O(log_2(N))$, and this bound is reached only for a small class of permutation groups. In general, it is sufficient to identify one example of each type of symmetry in the problem for GAP to generate the correct symmetry group. A useful heuristic is to use the first 3 or 4 symmetries that can be easily written down. If this process omits a class of symmetries, then symmetry breaking will still happen, but only for a subgroup of the underlying symmetry group. The CSP practitioner then has the option of identifying other symmetries (so as to generate the largest possible group), or accepting the number of symmetric solutions obtained.

4.3 Symmetry Breaking during Search

We assume that our search algorithm has arrived at a value to variable assignment point. The idea is to try assigning *Val* to *Var*, and if that fails, to exclude all symmetrically equivalent assignments. The procedure takes as argument:

- *Stab*, the member of the stabiliser chain computed at the previous assignment node (i.e. the subgroup of G which stabilises each SBDS point assigned so far) ;
- A, the current partial assignment of values to variables;
- *RTchain*, a sequence of the right transversals corresponding to the assignments made so far, $\langle RT_1, \ldots, RT_k \rangle$, where RT_i is a set containing a representative group element for each coset of $Stab_i$ in $Stab_{i-1}$. We define $g \in RTchain$ as any group element g which can be expressed as $g = p_k \circ p_{k-1} \circ \ldots \circ p_1$ where $p_i \in RT_i$. Note that *RTchain* implicitly represents the partial assignments symmetric to A, since any such assignment can be obtained by applying some $g \in RTchain$ to A.

Our implementation is based on the pseudo-code given in Figure 1. We first choose a variable-value pair, and map this pair to a GAP point. The next stage is to use GAP to compute the next member of the stabiliser chain, and the right transversal of this in the previous member of the stabiliser chain. Once this is done, we update *RTchain*. Since *RTchain* is made up of (products of) group elements which appear in the orbits of the points fixed by the current partial assignment, *RTchain* can be thought of as the current set of potentially applicable symmetries. We are now ready to test assignments.

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

```
sbds_search := proc(Stab, A, RTchain)
local Point, NewStab, RT, NewA, BrokenSymms, g;
    choose(Var, Val);
    Point := var_value_to_point(Var, Val);
    NewStab := Stabilizer(Stab, Point);
    RT := RightTransversal(Stab, NewStab);
    NewRTchain = RTchain ^ RT;
    assert(Var = Val);
    if sbds_search(NewStab, A ∧ Var = Val, NewRTchain) = true
    then
        return TRUE
    else
        retract(Var = Val);
        BrokenSymms := lazy_check(NewRTChain, A);
        for g in BrokenSymms do
            assert (g(A) ⇒ g( Var ≠ Val));
        end do;
        return sbds_search(Stab, A, RTchain)
    end if
end proc
```

Fig. 1. SBDS using GAP and ECL^{i}PS^e

backtrack and (effectively) want to impose the constraint $g(A) \Rightarrow g(\mathit{Var} \neq \mathit{Val})$ for each symmetry g.[1] By symmetry we mean a member of our revised *RTchain*, which is, perforce, a member of the original symmetry group. However doing this by iterating over every symmetry would be very inefficient; indeed it would be completely impractical for large symmetry groups. Instead there are several observations we can make which allow us to cut down the number of considered symmetries drastically.

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

The next observation is that we need not post the constraint for any symmetries for which the precondition $g(A)$ is false. For some g this may be true for the entire search subtree under consideration; for some it may be true for some part

[1] Note that we need not explicitly impose $\mathit{Var} \neq \mathit{Val}$, since the identity permutation (or something equivalent) will be one of the symmetries considered.

of the subtree; for others it may never be true. We combine this observation with the previous one to in effect evaluate $g(A)$ lazily, only imposing the constraint $g(\mathit{Var} \neq \mathit{Val})$ when $g(A)$ is known to be true, sharing as much of the evaluation as possible between different g, and deferring that evaluation until it is known to be needed.

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

1. The $i+1$th element of A is mapped to something which is true. In this case, proceed to considering the $i+2$th element.
2. The $i+1$th element of A is mapped to something which is false. In this case, do not consider this p_{i+1} any further. (This excludes all symmetries which map the first $i+1$ elements of A to the same thing from further consideration.)
3. The $i+1$th element of A is mapped to something which is neither true nor false at this point in the search. In this case we delay further computation based on this p_{i+1} until the truth value is known, and then apply the appropriate case above. We delay in order to avoid considering the next right transversal in *RTchain* until we know that we must. This is because each time we consider a new right transversal RT for a permutation under consideration, that permutation is expanded into $|RT|$ candidate permutations for the next iteration, and to remain practical we need to minimise the number of such multiplicative expansions.

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

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

4.4 Comparison with [13]

In a prior implementation [13] of SBDS using group theory, on backtracking the orbit of the current partial assignment under the action of the full symmetry group is computed from scratch, and then a constraint is imposed for each member of the orbit, excluding that particular partial assignment. The idea is to ensure that each constraint posted is different, since different symmetries applied to a partial assignment can yield the same constraint. The main drawbacks of this implementation are that the orbit is computed from scratch each time, and no account is taken of whether the constraints are already entailed. The latter is

important since it may be that many constraints are posted excluding different partial assignments, but that many of these constraints are useless for the same reason; e.g. because they involve excluding configurations where $X = 1$ when we already know that $X = 2$.

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

5 Examples

In this section we present examples of the use of our GAP-ECL^{i}PS^e implementation applied to constraint satisfaction problems having symmetry groups of sizes up to about 10^9. We give CPU times for a 600 MHz Intel PIII processor, version 5.3 of ECL^{i}PS^e and version 4r2 of GAP.

Colouring Dodecahedrons. For our first example we consider the problem of colouring the vertices of a dodecahedron, the regular polyhedron having 12 pentagonal faces and 20 vertices. The variables $x_1, \ldots, x_{20}$ represent the 20 vertices. The values $c_1, \ldots, c_m$ are the m colours in question. It can be shown that the symmetry group of the dodecahedron is isomorphic to the group of even permutations of five objects, known to group theorists as A_5, which has 60 elements. Since any permutation of a colouring is allowed, the symmetry group of the values is S_m. The total number of symmetries is then $60 \times m!$, acting on $20 \times m$ points. We construct this group in GAP from just four generators: one rotation of the vertices about a face, one rotation about a vertex, swapping the first two colour indices, and cycling the colour indices by one place. The constraints of the CSP are of the form $x_i \neq x_j$ whenever vertex i is joined by an edge to vertex j. We seek the number of colourings for a given m, such that no colouring is a symmetric equivalent of another. A standard CSP solver will return all legal assignments. Our approach has the advantage that all symmetries inherent in the problem are dealt with during the initial search.

Table 1. Dodecahedron colouring using GAP-ECL^{i}PS^e

Parameters		GAP-ECL^{i}PS^e			ECL^{i}PS^e		
m	Symms.	Sols.	Time	Backtracks	Sols.	Time	Backtracks
3	360	31	1.0	43	7200	0.2	6840
4	1440	117902	1600	100234	1.7×10^8	5270	1.0×10^8

Table 1 gives a comparison of GAP-ECL^{i}PS^e performance against a standard ECL^{i}PS^e implementation. While the overheads are not repaid for 3-colouring (for

so few symmetries one might as well use one of the existing SBDS approaches), we obtain more than a three-fold speedup for 4-colouring.

Alien Tiles. The alien tiles puzzle (http://www.alientiles.com) is Problem 27 in CSPLib (http://www.csplib.org) and consists of an $n \times n$ array of coloured tiles. A click on any tile in the array changes all tiles in the same row and column by one colour in a cycle of m colours. For $n = 4$, $m = 3$, we look at the following two problems, as described in detail in [11]:

1. What is the largest possible number of clicks required in the shortest sequence of clicks to reach any goal state from the start state?
2. How many distinct goal states are there requiring this number?

By distinct we mean not symmetrically equivalent to another sequence. The symmetries of the problem are given by a flip along any diagonal axis (2 symmetries) followed by any permutation of the rows and columns. The total number of symmetries is $2 \times n! \times n!$, for 1152 when $n = 4$. The group is straightforward to generate using three generators: one for a flip and two for permuting the rows (a permutation of the columns can be obtained by doing a flip, permuting the rows and flipping back).

Table 2. Alien tiles comparison: GAP-ECL^{i}PS^e – SBDS – no symmetry breaking

Problem	GAP-ECL^{i}PS^e				ECL^{i}PS^e SBDS		ECL^{i}PS^e	
	Sol.	GCPU	ECPU	ΣCPU	Sol.	Time	Sol.	Time
min. cost		0.95	8.66	9.61		44.95		600.75
dist. sols.	19	0.98	8.51	9.41	19	43.83	11232	862.63

These questions have been answered previously using SBDS with all 1151 non-trivial symmetries explicitly considered [11]: the minimum number is 10 and there are 19 distinct solutions. The use of SBDS led to a 40-fold runtime speedup using ILOG Solver. We implemented the problem in ECL^{i}PS^e with each symmetry explicitly considered, obtaining identical results with slightly less speedup. We then used our GAP-ECL^{i}PS^e system to solve the problem starting from the three generators. This gave a further run-time improvement of a factor of 5 over SBDS without GAP, again obtaining a minimum cost of 10 with 19 distinct solutions. We see an overall speedup by a factor of 60 to 90. We see that the use of GAP-ECL^{i}PS^e leads to a much faster solution with only a small amount of programming effort required to encode the symmetries of the problem.

So far we have used groups up to size 1,440. Our ability to handle groups of this size efficiently and easily is a significant step forward in the application of symmetry breaking in constraint problems. In fact, up to this size there are as many as 49,500,455,063 distinct groups. Each one could arise as the symmetry

group of a variety of different constraint problems. However, we can also apply SBDS to problems with symmetry groups several orders of magnitude bigger than could be handled previously, as we now show.

Balanced Incomplete Block Designs. To show that large numbers of symmetries can be dealt with, we consider the problem of finding $v \times b$ binary matrices such that each row has exactly r ones, each column has exactly k ones, and the scalar product of each pair of distinct rows is λ. This is a computational version of the (v, b, r, k, λ) BIBD problem [5]. Solutions do not exist for all parameters, and results are useful in areas such as cryptography and coding theory. A solution has $v! \times b!$ symmetric equivalents: one for each permutation of the rows and/or columns of the matrix.

Table 3. Balanced incomplete block designs using GAP-ECL^{i}PS^e

Parameters					GAP-ECL^{i}PS^e				ECL^{i}PS^e	
v	b	r	k	λ	Sols.	GCPU	ECPU	ΣCPU	Sols.	Time
7	7	3	2	1	1	0.71	0.68	1.39	151200	3149.7
6	10	5	3	2	1	0.89	5.57	6.46		$> 4 \times 10^5$

For the $(7, 7, 3, 3, 1)$ problem, GAP-ECL^{i}PS^e finds the unique solution in about one second of combined CPU time. Keen algebraists will note that the number of solutions found by ECL^{i}PS^e with no symmetry breaking is $151,200 = 7!^2/168$. The denominator, 168, is the size of the automorphism group of the projective plane of order 2 defined by the 2-$(7, 3, 1)$ block design. This shows that ECL^{i}PS^e is successfully finding only those solutions which are distinct with respect to the formulation of the problem.

The $(6, 10, 5, 3, 2)$ BIBD has $6! \times 10! = 2,612,736,000$ symmetries. Again, we can find the unique solution in a few seconds. The number of symmetric solutions is so large that an ECL^{i}PS^e program failed to enumerate them after 12 hours elapsed computation time. Taken together, these results demonstrate that many symmetric equivalent solutions are excluded during search in an efficient manner. Both groups were generated from only 4 permutations, and neither group was created as an explicit collection of elements by either ECL^{i}PS^e or GAP, allowing search that is efficient in space as well as time.

For constraint models in the form of a matrix, such as a BIBD, an alternative means of breaking symmetry is to insist that both rows and columns are lexicographically ordered [8]. While this may not break all symmetry, it successfully obtained a unique solution on both BIBDs we studied, in run times at least 10 times faster than our implementation of SBDS.

The lexicographic constraints force only a particular solution to be acceptable. This might conflict with the variable and value ordering heuristics being used, while SBDS will accept the first solution found and remove all symmetric

Table 4. Comparison of symmetry breaking using lexicographic ordering and SBDS in GAP-ECL^{i}PS^e and combinations of heuristics. The variable ordering heuristic + enumerates squares in rows starting from the top left, while - reverses, starting from the bottom right. The value ordering heuristic + tries 0 before 1, while - reverses this

Parameters					Heuristics		Lex-ECL^{i}PS^e		GAP-ECL^{i}PS^e	
v	b	r	k	λ	Var	Val	1^{st} CPU	All CPU	1^{st} ΣCPU	All ΣCPU
7	7	3	3	1	+	+	0.09	0.13	1.06	1.39
					+	-	0.12	0.13	0.70	0.76
					-	+	1.42	1.62	1.12	1.44
					-	-	0.27	1.62	0.80	0.82
6	10	5	3	2	+	+	0.11	0.17	4.51	6.46
					+	-	0.13	0.16	3.06	4.57
					-	+	126.50	243.29	4.57	6.52
					-	-	116.83	242.85	3.08	4.60

equivalents of it. We investigated this experimentally, with results shown in Table 4. Using the wrong variable ordering heuristic can make a difference of three orders of magnitude in the run time using lexicographic constraints, while we see almost no difference in the run time used by SBDS. We suggest this arises because the reversed heuristic starts with the bottom right corner, and the lexicographic constraints are unable to prune until late in the search tree. We do see a change in run time in both methods when reversing the value ordering. For lexicographic constraints, it makes almost no difference to total time, but it can affect dramatically the time to find the first solution. In the (7,7,3,3,1) BIBD, the reversed value ordering is best with the reversed variable ordering heuristic, because the preferred solution has a 1 in the bottom right square. We do see an improved overall run time with the reversed value ordering heuristic for SBDS. While we do not fully understand this, it seems to be more constraining to choose 1 before 0, and SBDS can exploit this to reduce the size of the overall search tree. We conclude that the low overheads of lexicographic ordering can make it very effective, provided that the programmer is aware of how the constraints will interact with the variable and value ordering heuristics. In comparison, SBDS is relatively unaffected by the choice of variable and value orderings.

6 Discussion

We have shown that constraint logic programming and computational group theory systems can be coupled to provide an efficient mechanism for

A: generating symmetry groups for constraint satisfaction problems, and
B: using group theoretic calculations within search to rule out symmetrically equivalent solutions.

Our implementation utilises features of ECL^{i}PS^e and GAP that allow lazy evaluation of properties of representatives of subsets of the symmetries. We obtain

significant increases in computational efficiency for a range of CSP problems with well defined symmetry groups containing as many as 10^9 elements.

There are two areas of future research interest. First, we would like to use the GAP-ECL^{i}PS^e framework to implement another symmetry breaking paradigm. For example, a purely group theoretic approach to symmetry breaking was used to implement efficient backtrack search in [3]. The idea, a precursor of [7], is to perform a backtrack search for broken symmetries within the backtrack search for a solution. It should be possible to implement the subsidiary search in GAP, leaving ECL^{i}PS^e to deal with the resulting (pruned) search tree. It would also be interesting to explore the application of our work to the satisfiability problem, in which Crawford et al [6] successfully applied group theoretic techniques.

The second area is the extension of our approach to more practical CSP problems. In particular the areas of symmetry in model checking [4], and vehicle routing and scheduling [2].

Acknowledgements

The St Andrews' authors are assisted by EPSRC grant GR/R29666. We thank Steve Linton, Ursula Martin, Iain McDonald, Karen Petrie, Joachim Schimpf, Barbara Smith and Mark Wallace for their assistance.

References

[1] R. Backofen and S. Will, *Excluding symmetries in constraint-based search*, Proceedings, CP-99, Springer, 1999, LNCS 1713, pp. 73–87. 415

[2] J. C. Beck, P. Prosser, and E. Selensky, *On the reformulation of vehicle routing problems and scheduling problems*, Tech. Report APES-44-2002, APES Research Group, February 2002. 429

[3] C. A. Brown, L. Finkelstein, and P. W. Purdom, Jr., *Backtrack searching in the presence of symmetry*, Proc. AAECC-6 (T. Mora, ed.), no. 357, Springer-Verlag, 1988, pp. 99–110. 429

[4] M. Calder and A. Miller, *Five ways to use induction and symmetry in the verification of networks of processes by model-checking*, Proc. AVoCS, 2002, pp. 29–42. 429

[5] C. H. Colbourn and J. H. Dinitz (eds.), *The CRC handbook of combinatorial designs*, CRC Press, Rockville, Maryland, USA, 1996. 427

[6] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, *Symmetry breaking predicates for search problems*, Proc. KR 96, Morgan Kaufmann, 1996, pp. 148–159. 429

[7] T. Fahle, S. Schamberger, and M. Sellmann, *Symmetry breaking*, Proc. CP 2001 (T. Walsh, ed.), 2001, pp. 93–107. 415, 429

[8] P. Flener, A. M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, J. Pearson, and T. Walsh, *Symmetry in matrix models*, Tech. Report APES-30-2001, APES Research Group, October 2001. 427

[9] F. Focacci and M. Milano, *Global cut framework for removing symmetries*, Proc. CP 2001 (T. Walsh, ed.), 2001, pp. 77–92. 415

[10] The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.2*, 2000, `http://www.gap-system.org`. 416, 420

[11] 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. 426
[12] 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. 415, 421
[13] I. McDonald, *Unique symmetry breaking in CSPs using group theory*, Proc. SymCon'01 (P. Flener and J. Pearson, eds.), 2001, pp. 75–78. 416, 424
[14] M. G. Wallace, S. Novello, and J. Schimpf, *ECLiPSe : A platform for constraint logic programming*, ICL Systems Journal **12** (1997), no. 1, 159–200. 416