Generic SBDD using Computational Group Theory

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

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

Abstract. We introduce a novel approach for symmetry breaking by dominance detection (SBDD). The essence of SBDD is to perform 'dominance checks' at each node in a search tree to ensure that no symmetrically equivalent node has been visited before. While a highly effective technique for dealing with symmetry in constraint programs, SBDD forces a major overhead on the programmer, of writing a dominance checker for each new problem to be solved. Our novelty here is an entirely *generic* dominance checker. This in itself is new, as are the algorithms to implement it. It can be used for any symmetry group arising in a constraint program. A constraint programmer using our system merely has to define a small number (typically 2–6) of generating symmetries, and our system detects and breaks all resulting symmetries. Our dominance checker also performs some propagation, again generically, so that values are removed from variables if setting them would lead to a successful dominance check. We have implemented this generic SBDD and report results on its use. Our implementation easily handles problems involving 10^{36} symmetries, with only four permutations needed to direct the dominance checks during search.

1 Introduction

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

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

The modified search techniques currently broadly fall into two main categories. The first involves adding constraints whenever backtracking occurs, so that symmetric versions of the failed part of the search tree will not be considered in future [1, 11]; these techniques are collectively known as SBDS (Symmetry Breaking During Search). The second category involves performing checks at nodes in the search tree to see whether they are dominated by the symmetric equivalent of some state already considered [5, 7]; we will collectively refer to these techniques as SBDD (Symmetry Breaking by Dominance Detection). A comparison of SBDS and SBDD, together with a dominance check for a highly symmetric problem, is given in [12].

The SBDD approach as implemented to date (with one exception)[3] involves the coding of a dominance checker. This dominance checker is special purpose, as it must be written for each new problem, or at best for a class of problems such as instances of the "golfers' problem". This checker, as part of the constraint system, has to be written by the constraint programmer, who therefore must use the structure of the problem under consideration to detect dominating search nodes. Yet dominance detection is an algebraic operation, and in particular answers a question in group theory: is this node symmetrically equivalent to a previously visited one? Inevitably, if unconsciously, constraint programmers are being required to act as CGT programmers in order to implement SBDD. The contribution we make is to eliminate this necessity by providing all the algebraic equipment to perform dominance checks automatically given the minimum possible information about the symmetries of the particular problem. These dominance checks either succeed (resulting in a backtrack), or fail supplying a set of assignments, any of which, if added to the current partial assignment of values to variables, would result in the dominance check succeeding – this set is used to reduce the size of domains, thus improving search efficiency, as described in [5, 7].

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

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

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

The main contributions of this paper are twofold. First, we show how to combine a constraint programming system with a CGT system to provide an entirely generic implementation of SBDD. Second, we introduce a novel algorithm for performing the dominance check in a generic manner using techniques from CGT.

[3] Although not set in constraint satisfaction terms, *Backtrack Searching in the Presence of Symmetry* [2, 3] contains many SBDD ideas, incidentally predating SBDD itself by a number of years. This work influenced our approach here.

It is instructive to compare this paper with our first application to symmetry breaking using the combination of a CGT system and a constraint programming system, namely our implementation of SBDS reported at CP-02 [9]. The implementation of SBDD using CGT methods raises very different problems from that of SBDS. For SBDS, the task is to compute a set of symmetry breaking constraints. For SBDD, the task is to implement a search algorithm, i.e. the dominance checker. Since the tasks are so different, the implementations themselves are very different. Indeed, a major contribution here is the algorithm for a generic dominance checker, and the resulting CGT program shares no code with the CGT program used for SBDS. Our work here is therefore very novel compared to [9]. The advantage of the current work is that there is no significant space requirement to store the set of constraints, as happens in SBDS. The limiting factor for SBDD is the time taken to search for dominance rather than space. We show that we can deal, in an entirely generic manner, with groups with as many as 10^{36} elements, compared to only a few billion in our implementation of SBDS.

We describe some necessary background on constraint systems and permutation groups in Section 2. We outline SBDD in Section 3, and provide a detailed exposition of the novel algorithm used in our generic implementation in Section 4. Section 5 consists of some experimental results. We discuss our results and highlight future avenues of research in Section 6.

2 Background

We first provide some background material. While we assume familiarity with the basic concepts of constraint satisfaction, we first sketch the problems that arise when a constraint problem contains symmetry. Then, we briefly describe permutation groups and how they can be used in the CGT system GAP, and finally the interface we previously constructed between GAP and ECL^{i}PS^e.

To provide a concrete implementation, we use the constraint logic programming system ECL^{i}PS^e [20] to model the constraint satisfaction problem and to search, while the dominance checks needed for SBDD are performed in a child process, using the world-leading computational group theory system GAP [8]. There is nothing essential about this choice. Barring unforseen technical problems, we could equally well have used other CGT systems such as Magma, or other constraint systems such as Ilog Solver. GAP and ECL^{i}PS^e are large, mature and widely-used systems. Both systems incorporate libraries and packages for computation in specific areas, together with tools and resources for software development. For our purposes, the GAP permutation group libraries and the ECL^{i}PS^e finite domain libraries are of interest.

ECL^{i}PS^e uses standard finite domain constraints, and, during search, applies constraint propagation techniques developed by the AI community [14]. The standard search method is depth-first, and assigns values to variables at choice points. A complete assignment that satisfies the constraints is a solution.

If no symmetry breaking constraints have been posted before search, then ECL^{i}PS^e will search for and return all solutions, irrespective of any symmetries involved. For example, consider the illustrative problem of finding a list $[A, B, C, D, E, F, G]$ of distinct numbers in $1 \ldots 50$ such that $A^3+B^3+C^3+D^3 = E^3+F^3+G^3$. A solution is $[1, 2, 3, 39, 18, 22, 35]$, but this is symmetrically equivalent to those lists with 1, 2, 3, and 39 permuted in any way, and/or 18, 22 and 35 permuted. Our aim is to restrict search to choices which do not lead to one of these other lists. Restricting search to avoid symmetrically equivalent solutions has a larger benefit than avoiding duplicate solutions. Symmetry breaking methods such as SBDD or SBDS also avoid duplicating search from failed nodes. This can have a dramatic effect on time taken, as the same failed search state can reoccur in many symmetric guises, only one of which need be explored.

A permutation is a rearrangement of elements in an ordered list S into a one-to-one correspondence with S itself. The number of permutations on a set of n elements is $n!$. Two permutations can be composed by composing their respective correspondences with S; since all such correspondences are bijective, the composition has an inverse. For example, to construct the resulting composition, one can trace the action of successive permutations. If 1 corresponded to 8 in the first, and 8 corresponded to 3 in the second, then 1 would correspond to 3 in the composition. If we take the identity mapping on S as the required neutral element, composition of permutations forms a group. GAP contains libraries for defining, composing, and manipulating individual permutations, and for computation within permutation groups.

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

Given that there are 7! possible permutations on S, which are the permutations which preserve solutions to the $A^3 + B^3 + C^3 + D^3 = E^3 + F^3 + G^3$ problem? This is straightforward in GAP – we simply supply some example permutations, and let GAP compute the resulting permutation group. Swapping A and D and cycling A, B, C and D are solution preserving symmetries, represented by the permutations $(1, 4)$ and $(1, 2, 3, 4)$ respectively. Similarly $(5, 7)$ and $(5, 6, 7)$ denote the swapping of E and G and a cycling of E, F and G. Supplying these 4 permutations to GAP results in permutation group containing 144 elements ($3!$ permutations of E, F and G for each of the $4!$ permutations of A, B, C and D). In group theoretic terminology this is known as the direct product of the symmetric group on 4 points and the symmetric group on 3 points. In order to use GAP–ECL^{i}PS^e with either the SBDS version given in [9] or the SBDD implementation described in Section 3 of this paper, these 4 permutations are the only information that the constraint logic programmer has to provide to GAP for this problem. GAP can now answer questions such as

– What is the composition of $(1, 4)$ with $(1, 3)(5, 7, 6)$, i.e. the permutation corresponding to performing first $(1, 4)$ and then $(1, 3)(5, 7, 6)$?
  • $(1, 4, 3)(5, 7, 6)$
– Which of our 144 permutations do not move A, C or E?
  • $(), (6, 7), (2, 4), (2, 4)(6, 7)$ – this is the point stabiliser of A, C and E.
– To which points is G mapped to by our group elements?
  • $7, 6, 5$ (i.e. G is mapped to E, F and itself) – this is the orbit of G.

Many of the questions passed to GAP by ECL^{i}PS^e during search are answered by (rather more complicated) calculations similar to those given above. It should be noted that, for computational purposes, a symmetry group consists of a generating set of permutations. This set can usually consist of only two elements, but, for our purposes, contains a few examples of known symmetries. The problem of enabling CSP practitioners to express problem symmetries easily is addressed in [13, 15]. No sensible CGT system computes every element of the group; algorithms construct new elements (or subgroups, or coset representatives, etc.) as required.

In [9] we reported on a simple interface between GAP and ECL^{i}PS^e . In GAP–ECL^{i}PS^e all constraint satisfaction modeling and constraint handling is done in ECL^{i}PS^e, as is the choosing of value to variable assignments during search, and any resulting elimination of values from domains by propagation. GAP runs as a sub-process, and is called as and when symmetry breaking information is needed. In effect, ECL^{i}PS^e is the master, and GAP the slave. The key concept that motivates the interface is that the symmetries of a constraint satisfaction problem are permutations on a suitable initial segment of the natural numbers. Since sets of permutations have a well known algebraic structure, and since GAP uses the algebraic structure to enhance and extend computational capability, we use GAP to provide symmetry information to ECL^{i}PS^e during search. We report further details of the interface in [9].

3 Symmetry Breaking by Dominance Detection

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

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

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

Suppose that we have identified a symmetry group, G, and that we maintain a record in a list S of *fail sets*: sets corresponding to the roots of completed subtrees. Each fail set contains the points from the $M \times N$ array corresponding to the positive decisions made during the search to reach the root of the subtree. Note that we consider decisions, as opposed to domains, as suggested in [12, 18]. As long as we try a positive decision (*Var* $=$ *Val*) before its negative (*Var* $\neq$ *Val*) we are free to ignore the negative decisions [7, 12, 18]. E.g. looking at the $Y = b$ subtree in Figure 1, $Y = b$ has already been fully explored regardless of whether or not $X \neq a$, since the $X = a$ case was covered by the $X = a$ subtree.

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

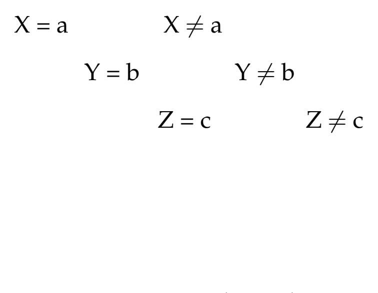

Fig. 1. A partial search tree

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

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

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

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

4 Generic SBDD

```
**generic_sbdd**($Failset$, $depth$) : −
    choose($Var$, $Val$)
    assert($Var = Val$)
    $depth := depth + 1$
    $pt :=$ Point($Var = Val$)
    $NewFailset := [pt, Failset]$
    $Doms := [current\_domains]$
    if consistent($CSP$) and askGAP($Doms$) $= [false, Q]$ then
        reduce_domains($Q$)
        solution_check
        generic_sbdd($NewFailset$, $depth$)
    else
        tellGAP($NewFailset$, $depth$)
        retract($Var = Val$)
        assert( not ($Var = val$))
        $Doms := [current\_domains]$
        if consistent($CSP$) and askGap($Doms$) $= [false, Q]$ then
            reduce_domains($Q$)
            solution_check
            generic_sbdd($Failset$, $depth$)
        else
            backtrack($newdepth$)
            generic_sbdd($Failset$, $newdepth$)
        end if
    end if
```

Fig. 2. Pseudo-code for generic SBDD

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

Provided that the CSP is still consistent, we are now ready to ask GAP for a dominance check, details of which are given in Section 4.1. If this check succeeds (i.e. a dominating state was found), we can backtrack in ECL^{i}PS^e as we have already explored an equivalent state. If this check fails (i.e. if no dominating state is found) then we can still benefit by domain reduction. Our dominance checker supplies a set of points that, if any one of the corresponding assignments is made, would result in a successful dominance check. Clearly we should not allow a search which makes any of these assignments, so we remove them from the domains of the variables involved. This not only reduces the sizes of the value domains, but also allows further propagation based on the removals. This is a significant benefit over obtaining a mere yes/no answer to the dominance check.

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

When the dominance check succeeds, we retract $Var = Val$ and assert $Var \neq Val$. We tell GAP that our current fail set is the most up to date, and remove pt from it. Now that we are at a different node in the search tree, we can obtain *Doms* again perform another dominance check. If this check fails, then, as before, we reduce value domain sizes by removing from variable domains any elements we know would have led to dominance if they had been assigned to the variable, check that any solution is not dominated by a previous one, and carry on searching below the $Var \neq Val$ branch.

If, however, the $Var \neq Val$ dominance check succeeds, then we backtrack to the nearest ancestor node where we have yet to consider the negative branch. This point becomes the root of a completed subtree, we update the fail set accordingly, and carry on searching.

4.1 Dominance Check using Computational Group Theory

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

The dominance check is implemented using a tree-like data structure which encodes all of the fail sets currently applicable, while taking maximum advantage of their overlaps. Every possible $Var = Val$ assignment is identified by a point in a symmetry matrix; the symmetry group for a given problem permutes these points, so that every symmetry is defined by a unique permutation.

We can identify disjoint sets of points $A_1, \ldots, A_k$ and $B_0, \ldots, B_k$ such that the fail sets are $A_1 \cup \cdots \cup A_i \cup B_i$ for each i. The right-pointing edges of the tree are labelled with elements of an A_i, the left-pointing ones with elements of a B_i. Each node of the tree can be associated with the sequence of labels on the path to it from the root.

For example, if the current failsets are [52, 79, 72, 51, 64, 57, 50, 53] and [61, 88, 74, 60, 52, 79, 72, 51, 64, 57, 50, 76] and we are asked to check [98, 48, 90, 42, 35, 77, 27, 96, 82, 14, 70, 13, 69, 40, 26, 19, 61, 46, 88, 32, 74, 18, 60, 7, 6, 5, 4, 87, 94, 31, 24, 17, 10, 52, 37, 44, 79, 72, 16, 9, 51, 22, 29, 36, 43, 64, 57, 50], for dominance (this situation arises in the BIBD(7,7,3,3,1) problem described in Section 5.3), then we have $A_1 = \{52, 79, 72, 51, 64, 57, 50\}$, $B_1 = \{53\}$, $A_2 = \{61, 88, 74, 60\}$, $B_2 = \{76\}$.

We perform the dominance check using a recursive search, which descends this tree, entering each node once for every way of mapping the associated sequence of points into the current point list. If we reach a left-pointing leaf, then we have discovered dominance. The implementation of the search uses the standard group theoretic machinery of stabilizer chains, Schreier vectors and transversals, described, for instance in [19].

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

Full details and the GAP code used will appear in a forthcoming Technical Report.

Since fail sets and point lists are not, in general, the same size, the more powerful machinery of partition backtrack searching also described in [19] does not appear to be helpful.

A useful optimisation is possible in cases where the points fall into more than one *orbit* under the action of the symmetry group. This arises, for instance, if the symmetries permute the variables and not the values. In this case it may happen that all the points appearing in failsets lie in a subset of the orbits. For instance if all the variables are Boolean, then for some labelling strategies, we will only ever see points corresponding to assignments $Var = true$ appearing in the failsets. In this case, points in other orbits are *irrelevant* in the sense that no symmetry can ever map any point of any failset to them. These points can be ignored in the search, and, more importantly, if no new relevant points have been added to the pointlist since a previous dominance check, the entire check can be omitted.

This implementation produces good performance on moderate-sized examples (up to about 10^{36} symmetries), but the internal search can become bogged down when a subset of some F has a large stabilizer, so that we can find elements of G mapping $f_1, \ldots, f_k$ to $p_1, \ldots, p_k$ in any order, but none of these allow us to map f_{k+1} to anything in *Pointset*. Actually computing the set stabilizers of initial segments of F, while possible, seems to be prohibitively expensive in many cases, but such situations usually arise when the group G preserves a system of imprimitivity (for example the rows and columns of a matrix-structured problem) and this condition can be recognized cheaply. Exploiting this information will be an important part of future work. Further implementation details (including source code) are available in [10].

5 Examples

In this section we provide some results of computations using our implemenentation of generic SBDD in GAP–ECL^{i}PS^e. All the examples were run on a 2.6 GHz Pentium IV processor with 512 megabytes of memory, and times are reported in seconds. Where possible we compare the performance of our SBDD implementation with that of our GAP–ECL^{i}PS^e SBDS implementation given in [9], which provided full symmetry breaking in a few seconds for problems having up to 10^9 symmetries. A major cost in dealing with larger symmetry groups in SBDS is the communication of information between GAP and ECL^{i}PS^e – the constraints posted during search are based on large algebraic structures which have to be returned to ECL^{i}PS^e from GAP. In SBDD, however, we expect to be able to deal with much larger groups, since inter-process communication consists of the word *true*, the word *false*, or lists of points of length at most $M \times N$.

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

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

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

We include this example to demonstrate that out implementation breaks all 144 symmetries, with performance comparable to that of SBDS in GAP–ECL^{i}PS^e. The results for all solutions with domains $1 \ldots 20$ are given in Table 1.

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

Table 1. Seven cubes problem – comparative results

We see that SBDD and SBDS both eliminate all the symmetries in roughly the same time, whereas a search which ignores symmetry returns 144×265 solutions.

5.2 Example: Colouring the vertices of a dodecahedron

We consider the problem of colouring the vertices of a dodecahedron, the regular polyhedron having 12 pentagonal faces and 20 vertices. This problem has two useful illustrative features: it involves symmetries in both the variables and values, and obtaining the correct symmetry group is easier than writing a dominance detector for use in SBDD without computational group theory.

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:

- the image of the vertices after one rotation of $72°$ about a face;
- the image of the vertices after one rotation of $120°$ about a vertex;
- the index of the colours with the first two swapped;
- the index of the colours cycled by one place mod m.

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.

GAP–ECL^{i}PS^e (SBDD)							Dominance checks		
Colours	Symms.	Sols.	BT	Gcpu	Ecpu	Σcpu	Success	Fail	Delete
3	360	31	50	0.44	0.07	0.51	71	19	31
4	1440	117902	109502	770.62	109.08	879.70	116396	351720	1176

Table 2. Colouring dodecahedrons – SBDD

From Table 2 we see that, for the 3 colour case, 121 dominance checks were made. Of these, 71 stopped further search in a symmetric sub-branch, 19 failed without providing any near misses, and 31 failed and supplied points which could be deleted from current domains as not leading to any new solutions. All 360 symmetries were broken, with the 31 non-isomorphic colourings returned in less than one second.

5.3 Example: Balanced Incomplete Block Designs

We now present results for examples with much larger symmetry groups. Consider the problem of finding $v \times b$ binary matrices such that each row has exactly r ones, each column has exactly k ones, and the scalar product of each pair of distinct rows is λ. This is a computational version of the (v, b, r, k, λ) BIBD problem [4]. We label the $v \times b$ matrix in column order, since $v \leq b$ for all suitable parameters. We assign zeros before ones whenever $k \geq b/2$, otherwise we assign ones before zeros; the heuristic is to use the minimum domain value whenever there are more ones than zeros in each column.

Solutions do not exist for all parameters, and results are useful in areas such as cryptography and coding theory. A solution has $v! \times b!$ symmetric equivalents: one for each permutation of the rows and/or columns of the matrix. Gent *et al.* [9] reported results with the largest symmetry group having $6! \times 10! \approx 3 \times 10^9$ elements. The results for our generic SBDD implementation are given in Table 3.

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

Table 3. Balanced incomplete block designs – SBDD

The first point to note is that, as expected, we can deal with much larger groups than our GAP–ECL^{i}PS^e implementation of SBDS [9]. SBDS was able to deal only with BIBDs in the first two lines of the table. It was up to about four times slower, while an interesting difference was that most cpu time was in ECL^{i}PS^e with GAP dominating time here.

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

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

The timings obtained are comparable with those presented for the same problems in [6], where lexicographic ordering constraints were use to break the row and column symmetries. The advantage of using SBDD is that all symmetries are broken, whereas a lexicographic solution for the $(6, 20, 10, 3, 4)$ BIBD problem returns 21 solutions. Moreover, while SBDD can work with any variable or value ordering heuristics, a heuristic can interact badly with lexicographic ordering constraints [9].

6 Conclusions

We have presented an implementation of SBDD which

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

We believe that the use of CGT techniques in SBDD solvers is an important contribution. While there is scope for further optimisation of our techniques, we already have significant advantages over related work. Compared to other implementations of SBDD, we have the key advantage of avoiding the need for a separate dominance check to be implemented, either directly [5] or as a separate constraint satisfaction problem [17]. This is an extremely important step forward in the application of SBDD. Compared to the use of GAP for SBDS [9], we avoid the large space overhead, meaning that – as we reported here – we are able to solve completely problems with groups many orders of magnitude larger. Some techniques, such as [6], do not guarantee to eliminate all symmetries, while we do.

Our implementation is robust: both the ECL^{i}PS^e and GAP searches are deterministic, and will break all the supplied symmetries, since a dominance check is performed at each node visited during search. This robustness may have a negative effect on efficiency. There is evidence that performance can be improved by making full dominance checks at a subset of the visited nodes [5, 17], or by using a subset of the full symmetry group of the problem [16]. Both of these approaches depend on the size and structure of the problem being addressed, and we will investigate their applicability to our implementation in the future.

Our work raises a number of questions for further research. First, having implemented both SBDS and SBDD using generic methods, we are in a position to ask whether or not they can be combined in interesting ways to gain the advantages of both. In naïve terms, one can see SBDS as best suited where groups are small, with SBDD effective on larger groups. Yet the symmetries in a constraint problem usually become small as search continues, and it may be possible to implement combined techniques which act like SBDD in some parts of the search tree and like SBDS in other parts. A second question is how far we can integrate SBDS and SBDD with what is perhaps the most commonly used symmetry breaking technique, the use of hand-written symmetry breaking constraints. As these can be very effective, it would be desirable to gain their advantages in terms of simplicity and efficiency, while still having the correctness and uniqueness guarantees of SBDS and SBDD without users having to be expert group theorists.

Acknowledgements

We are very grateful for their helpful comments and other assistance to Iain McDonald, Karen Petrie, Barbara Smith and Toby Walsh. This work is partly supported by EPSRC grant GR/R29666 and by a Royal Society of Edinburgh SEELLD Support Fellowship.

References

1. R. Backofen and S. Will, *Excluding symmetries in constraint-based search*, Proceedings, CP-99, Springer, 1999, LNCS 1713, pp. 73–87.
2. 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.
3. C.A. Brown, L. Finkelstein, and P.W. Purdom Jr., *Backtrack searching in the presence of symmetry*, Nordic Journal of Computing **3** (1996), no. 3, 203–219.
4. C.H. Colbourn and J.H. Dinitz (eds.), *The CRC handbook of combinatorial designs*, CRC Press, Rockville, Maryland, USA, 1996.
5. Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann, *Symmetry breaking*, Proc. CP 2001 (T. Walsh, ed.), 2001, pp. 93–107.
6. Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynap Kızıltan, Ian Miguel, Justin Pearson, and Toby Walsh, *Breaking row and column symmetries in matrix models*, Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming — CP'2002 (Pascal Van Hentenryck, ed.), Lecture Notes in Computer Science, vol. 2470, Springer-Verlag, 2002, pp. 462–476.
7. Filippo Focacci and Michaela Milano, *Global cut framework for removing symmetries*, Proc. CP 2001 (T. Walsh, ed.), 2001, pp. 77–92.
8. The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.2*, 2000, (`http://www.gap-system.org`).
9. Ian P. Gent, Warwick Harvey, and Tom Kelsey, *Groups and constraints: Symmetry breaking during search*, Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming — CP'2002 (Pascal Van Hentenryck, ed.), Lecture Notes in Computer Science, vol. 2470, Springer-Verlag, 2002, pp. 415–430.
10. Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton, *Generic SBDD using computational group theory*, Tech. Report APES-57a-2003, APES Research Group, January 2003, Available from http://www.dcs.st-and.ac.uk/˜apes/apesreports.html.

11. 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.
12. Warwick Harvey, *Symmetry breaking and the social golfer problem*, Proceedings, SymCon-01: Symmetry in Constraints (Pierre Flener and Justin Pearson, eds.), 2001, pp. 9–16.
13. Warwick Harvey, Tom Kelsey, and Karen Petrie, *Symmetry group generation for CSPs*, Tech. Report APES-60-2003, APES Research Group, July 2003, Available from http://www.dcs.st-and.ac.uk/˜apes/apesreports.html.
14. Pascal Van Hentenryck, *Constraint satisfaction in logic programming*, Logic Programming Series, MIT Press, Cambridge, MA, 1989.
15. Iain McDonald, *Symmetry breaking systems*, Tech. Report APES-61-2003, APES Research Group, July 2003, Available from http://www.dcs.st-and.ac.uk/˜apes/apesreports.html.
16. Iain McDonald and Barbara Smith, *Partial symmetry breaking*, Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming — CP'2002 (Pascal Van Hentenryck, ed.), Lecture Notes in Computer Science, vol. 2470, Springer-Verlag, 2002, pp. 431–445.
17. J-F. Puget, *Symmetry breaking revisited*, Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming — CP'2002 (Pascal Van Hentenryck, ed.), Lecture Notes in Computer Science, vol. 2470, Springer-Verlag, 2002, pp. 446–461.
18. Jean-François Puget, *Symmetry breaking revisited*, Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming — CP'2002 (Pascal Van Hentenryck, ed.), Lecture Notes in Computer Science, vol. 2470, Springer-Verlag, 2002, pp. 446–461.
19. Akos Seress, *Permutation group algorithms*, Cambridge tracts in mathematics, no. 152, Cambridge University Press, 2002.
20. M. G. Wallace, S. Novello, and J. Schimpf, *ECLiPSe : A platform for constraint logic programming*, ICL Systems Journal **12** (1997), no. 1, 159–200.