

GAPLex: Combining Static and Dynamic Symmetry Breaking

Chris Jefferson¹, Tom Kelsey², Steve Linton², and Karen Petrie²

¹ Computing Laboratory, University of Oxford, Oxford, UK
Chris.Jefferson@comlab.ox.ac.uk

² School of Computer Science, University of St Andrews,
St Andrews, Fife, KY16 9SX, UK
{kep, tom, sal}@dcs.st-and.ac.uk

Abstract. We describe a novel and effective suite of algorithms that combine the efficiency and ease of use of lex-ordering, with the power of breaking symmetry in CSPs by using computational group theory during search. We show that our new symmetry breaking method, GAPLex, is sound (will neither lose solutions nor return incorrect solutions) and complete (will return exactly one member from each class of symmetrically equivalent solutions). We demonstrate that our implementation of GAPLex is competitive with other methods, being effectively applicable to CSPs with large domains and less than full variable and/or value symmetry. We also describe how a variation of GAPLex can be combined with incomplete symmetry breaking methods – such as double-lex – to provide fast and complete symmetry breaking. We believe this to be the first method that successfully combines the posting of symmetry breaking constraints before search, with symmetry breaking by analysis of search states. We provide empirical evidence that our implementation can be more effective than using either the posting of constraints before search or the analysis of search-state in isolation.

1 Introduction

Constraint systems are a generalization of the Boolean satisfiability problems that play a central role in theoretical computer science. Solving constraint satisfaction problems (CSPs) in general is thus NP-complete; but effective solving of constraint systems arising from real problems, such as airline scheduling, is of enormous industrial importance.

The rationale for symmetry breaking (henceforth SB) in constraints satisfaction is well known: if SB is not taken into account, solutions will be returned which are symmetrically equivalent to others. If a given CSP has n solution symmetries and m symmetrically inequivalent solutions, then up to nm solutions will be found by exhaustive search for all solutions, which is clearly sub-optimal. If only a first solution is wanted (or if $m = 0$) then extra work is, in general, being done by exploring part of a search tree that is symmetrically equivalent to a part already visited.

Four main approaches to SB have been reported in the literature. The first is to add constraints before search that reduce the number of symmetric solutions found. This typically involves constraints that rule out solutions by enforcing some lexicographic-ordering on the variables of the problem [4, 9]. The lex-ordering approach can be thought of as static, in the sense that the constraints are posted before search. The second is to break symmetry using knowledge of the state of search, adapting search as necessary. This typically involves dynamically posting constraints that rule out searching anywhere that is symmetrically equivalent to the current assignment [2, 13], or backtracking away from nodes that are symmetrically equivalent to root nodes of subtrees that have previously been fully explored [5, 7, 19]. One of the major differences between static and dynamic symmetry breaking methods is that some powerful implied constraints can be derived only after statically breaking symmetry [8]. These implied constraints can decrease search size by orders of magnitude.

The third SB approach involves building each node of a search tree by enforcing the twin requirements that (i) no node in the search tree is symmetrically equivalent to any other and (ii) any solution to the CSP is symmetrically equivalent to a satisfying full assignment of the search tree [21]. These search trees are known as GE-trees. The final approach is reformulation of the CSP so that either the number of symmetries is reduced, or one of the other three approaches can be applied more effectively [17, 15]. We are primarily concerned with approaches one and two. In particular, we aim to utilise recent advances [16] in Computational Group Theory (CGT) to obtain answers to symmetry-related questions at any given node in the search tree, in a similar manner to that described in [11, 12]. The motivation for this is that groups (sets of objects with a closed associative binary operation, with a unique neutral element, and with each element having a unique inverse) are the mathematical structures that best encapsulate symmetry. Moreover, many powerful algorithms for investigating group-theoretic questions are known, and have been efficiently implemented in systems such as GAP [10] and MAGMA [3]. We describe the symmetries of a CSP as a permutation group of the literals (variable-value pairs) of the CSP, and obtain information regarding symmetric equivalence of search states from the GAP CGT system.

In this paper we address the open research question: can we combine lex-ordering with a dynamic CGT approach to SB in such a way that we retain the best features of each approach? Our contribution is twofold. We first describe a novel SB method, GAPLex, which involves both ordering constraints and symmetry information regarding the current state of search to break as many solution symmetries of a CSP as are required. We also demonstrate that an adaptation of GAPLex can be combined with fast (but incomplete) static lex-orderings to provide fast and complete SB. We are not aware of any existing SB method that effectively combines static and dynamic approaches, although Harvey has made an interesting initial attempt [14].

In the remainder of this introduction we give a detailed background of SB by lex-ordering, a basic description of permutation groups acting on literals of

CSPs, and describe existing approaches to the use of CGT to break symmetries dynamically. In Section 2 we motivate and describe GAPLex, and provide empirical results for our implementation. Section 3 describes our combination of GAPLex with static lex-orderings, again with empirical results. We conclude with a summary of our results and an outline of future research in this area.

1.1 Lex-ordering to break symmetries

Puget [18] proved that whenever a CSP has symmetry, it is possible to find a ‘reduced form’, with the symmetries eliminated, by adding constraints to the original problem. He also found such a reduction for three simple constraint problems, and showed that this reduced CSP could be solved more efficiently than in its original form. Following this, the key advance was to show a method whereby such a set of constraints could be generated. Crawford, Ginsberg, Luks and Roy outlined a technique, called “lex-leader” for constructing symmetry-breaking ordering constraints for variable symmetries [4]. In later work, Aloul *et al.* also showed how the lex-leader constraints for symmetry breaking can be expressed more efficiently [1]. This method was developed in the context of Propositional Satisfiability (SAT), but the results can also be applied to CP.

The idea of lex-leader is essentially simple. For each equivalence class of solutions under our symmetry group, we predefine one to be the canonical solution. We achieve this by adding constraints before search starts which are satisfied by canonical solutions and not by any others. The technique requires first choosing a static variable ordering. From this, we induce an ordering on full assignments. The ordering on full assignments is straightforward. The tuple is simply the values assigned to variables, listed in the order defined by our static ordering. Since the method is defined for variable symmetries, any permutation g converts this tuple into another tuple, and we prefer the lexicographically least of these. This method is, in principle, simple to implement. Each permutation in the group gives us one \preceq_{lex} constraint. So the set of constraints defined by the lex-leader method is

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

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

An important practical issue with the lex-leader constraints is that they do not “respect” the variable and value ordering heuristics used in search. That is, it may well be that the leftmost solution in the search tree, which would otherwise be found first, is not canonical and so is disallowed, leading to increased search. This is in contrast to dynamic techniques, which do respect the heuristic. Simple examples have been reported where the “wrong” heuristic can lead to dramatic increases in runtime [11]. This problem is inherent in the method,

but in many cases it is easy to work out what is the “right” heuristic. In particular, if the same static variable ordering is used in search as was used to construct the lex-leader ordering, and values are tried from smallest to largest, this conflict should not occur. However, this does limit the power of the constraint programmer to use dynamic variable ordering heuristics.

A less easily solved problem with lex-leader is that many groups contain an exponential number of symmetries. Lex-leader appears to require one constraint for each element of the group. However, for problems in which the variables must take distinct values, Puget has shown that a low-degree polynomial number of binary inequality constraints can break all the variable symmetry, even though an exponential number of symmetries are present [20]. Hence, lex-leader remains of the highest importance, due to its simplicity, ease of application and use in the derivation of new SB methods.

1.2 Group theory for CSPs

Definition 1. A CSP L is a set of constraints \mathcal{C} acting on a finite set of variables $\Delta := \{A_1, A_2, \dots, A_n\}$, each of which has finite domain of possible values $D_i := D(A_i) \subseteq A$. A solution to L is an instantiation of all of the variables in Δ such that all of the constraints in \mathcal{C} are satisfied.

Constraint logic programming systems typically model CSPs using constraints over finite domains. The usual search method is depth-first, with values assigned to variables at choice points. After each assignment a partial consistency test is applied: domain values that are found to be inconsistent are deleted, so that a smaller search tree is produced. Backtrack search is itself a consistency technique, since any inconsistency in a current partial assignment (the current set of choice points) will induce a backtrack. Other techniques include forward-checking, conflict-directed backjumping and look-ahead.

Statements of the form $(Var = val)$ are called *literals*, so a partial assignment is a conjunction of literals. We denote the set of all literals by χ , and generally denote variables in Roman capitals and values by lower case Greek letters.

Definition 2. Given a CSP L , with a set of constraints \mathcal{C} , and a set of literals χ , a symmetry of L is a bijection $f : \chi \rightarrow \chi$ such that a full assignment A of L satisfies all constraints in \mathcal{C} if and only if $f(A)$ does.

We denote the image of a literal $(X = \alpha)$ under a symmetry g by $(X = \alpha)g$. The set of all symmetries of a CSP form a *group*: that is, they are a collection of bijections from the set of all literals to itself that is closed under composition of mappings and under inversion. We denote the symmetry group of a CSP by G .

Definition 3. Let G be a group of symmetries of a CSP. The stabiliser of a literal $(X = \alpha)$ is the set of all symmetries in G that map $(X = \alpha)$ to itself. This set is itself a group. The orbit of a literal $(X = \alpha)$, denoted $(X = \alpha)^G$, is the set of all literals that can be mapped to $(X = \alpha)$ by a symmetry in G . That is

$$(X = \alpha)^G := \{(Y = \beta) : \exists g \in G \text{ s.t. } (Y = \beta)g = (X = \alpha)\}.$$

The orbit of a node is defined similarly.

Given a collection S of literals, the *pointwise* stabiliser of S is the subgroup of G which stabilises each element of S individually. The *setwise* stabiliser of S is the subgroup of G that consists of symmetries mapping the set S to itself.

1.3 Using GAP to break CGT symmetries

There have been three successful implementations of SB methods which use GAP as a black box to provide answers to symmetry related questions during search. All three combined GAP with the constraints system ECLⁱPS^e [22]. GAP-SBDS [11] is an implementation of symmetry breaking during search: at each search node, constraints are posted which ensure that no symmetrically equivalent node will be visited later in search. The overhead comprises maintenance (in GAP) of a stabiliser chain of the symmetry group, the search for group elements which map the current state to a future state (also in GAP), and the posting of the SB constraints. Enough pruning is made, in general, to make GAP-SBDS more efficient than straightforward search. The number of SB constraints grows in line with the size of the group, making GAP-SBDS unattractive for groups of size greater than about 10^9 .

GAP-SBDD [12] also uses GAP to find, where possible, group elements that map the current search state to another state. In this case, we maintain information regarding search so far, and check that our next assignment is not equivalent to a state which is the root of a previously explored sub-tree. Again, the overhead of finding (or failing to find) these group elements is usually more than offset by the reduction in search due to early backtracking. Larger groups – up to about 10^{25} – can be dealt with, simply because the answer from GAP is a straight yes or no to the dominance question; the overhead of passing constraint information is not present. However, GAP also reports literals that can be safely deleted because setting them would have lead to dominance at a lower search depth. Provided that the cost of computing these safe deletions is low enough, the domain reductions – and the resulting propagation – are gain over not making them. We follow the same idea in this paper; we search for literals that, if set, would have lead to a non-lex-least assignment at a lower depth in the current search path.

The third use of GAP is the building of search trees that, by construction, have no symmetrically equivalent nodes and contain an example from each solution equivalence class: GE-trees [15]. In the event that all the symmetries act only on variables (either by suitable formulation or the use of channelling constraints from an unsuitable formulation), these trees can be constructed in low-degree polynomial time per node. This is due to pointwise stabilisers being polynomial time obtainable, whilst setwise stabilisers (the general case) are not known to be obtainable in polynomial time.

In this paper we aim to build upon the strengths of these existing frameworks, by using lex-ordering as the main SB technique, asking GAP to decide whether or not the current partial assignment is lex-smallest of the orbit of the assignment under the symmetry group.

2 GAPLex

2.1 Motivation and rationale

Both lex-ordering and CGT-based SB methods are effective and attractive options for breaking symmetries in CSPs. Can we combine the simplicity and efficiency of lex with the generic, dynamic and complete CGT methods used in GAP-SBDS and GAP-SBDD? The answer to this question is “yes”, provided that we accept a restriction to static variable and value search orderings.

We want to enforce a lex ordering on the literals of a CSP so that only solutions that are minimal in the ordering are returned after complete backtrack search with propagation. Moreover, we want this to work with any symmetry structure induced by the formulation of the CSP.

The key idea is that we can compute the minimum image, under a symmetry group G , of those literals that represent the ground variables in any branch of the search tree. By minimum image, we mean the lex-least ordered list of literals that can be obtained by applying group elements (symmetries) to our set of ground literals. If the minimum image of our current partial assignment is lex-smaller than that assignment, then it is safe to backtrack from the current search node: further search will either fail to find a solution or return a solution that is not the lex-smallest in its equivalence class.

Recent advances in algorithms for finding minimal images have led to dramatic improvements in both worst-case and apparent average-case time complexities [16]. We use these more efficient CGT methods to obtain the minimal images and decide the ordering predicate. As a useful extension to the main technique, we can also use CGT to identify those literals involving non-ground variables that, if taken as assignments, would result in immediate backtrack on the lex-smaller test. The assignments of any such literals can be ruled out immediately by deleting the values from their respective domains. These domain deletions, together with the propagation of the domain deletion decisions, reduce the search required to find solutions (or confirm that no solutions exist) for the CSP. This process is analogous to the deletion of “near misses” in GAP-SBDD, as discussed in [7, 12].

Since the cost of computing minimal images and comparing lists is generally outweighed by the reduction in search due to early backtracking and early domain deletion, our method is a useful and generic extension of lex-ordering as a symmetry breaking technique.

2.2 Motivating example

Suppose that we wish to solve

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

where each letter denotes a member of $\{1, \dots, 9\}$, and, for example, BC denotes $10*B+C$. There are $3!$ permutations of the summands which preserve solutions. We add the constraint that a digit can appear at most twice in a solution.

Suppose now that during search for all solutions we have made the partial assignment $PA : A = 2, B = 3, C = 8, D = 2, E = 1$. This is mapped by the group element $(DAG)(EBH)(FCI)$ as a sequence to $G = 2, H = 3, I = 8, A = 2, B = 1$; or, in sorted order, $A = 2, B = 1, G = 2, H = 3, I = 8$.

This is lexicographically smaller than PA (since $B = 1$ is smaller than $B = 3$) so we would backtrack from this position. Notice that although we map a state which includes $A = 2$ into a smaller state which also includes $A = 2$ we can't do it by mapping the literal $A = 2$ to itself.

Even if we can't immediately backtrack, there are often safe domain deletions that can be made. For example if we have only assigned $A = 7$, it is safe to remove values 1 through 6 from the domains of D and G , since making any of these assignments further down the $A = 7$ branch would lead to an immediate backtrack. This combination of early backtrack and domain reduction provides the search pruning needed to offset the overhead of computing and comparing images of assignments.

2.3 The GAPLex algorithms

We have a CSP and a symmetry group, G , for the CSP. G acts on the set of literals (variable-value pairs) of the problem, written as an initial subset of the natural numbers. The set of literals forms a variables \times values array, so that the literals of the lex-least variable are $1, 2, \dots, |dom(V_1)|$, etc. The GAPLex method, applied at a node N in search, proceeds as follows :

Require: $PA \leftarrow$ current partial assignment
Require: $Var \leftarrow$ next variable w.r.t. any fixed choice heuristic
Require: $val \leftarrow$ next value w.r.t. lex-least value ordering

- 1: set $Var = val$ and propagate
- 2: add $Var = val$ to PA
- 3: $T \leftarrow$ literals involving unassigned variables
- 4: pass PA and T to the GAPLex CGT test
- 5: **if** the test returns **false** **then**
- 6: backtrack
- 7: **else**
- 8: the test returns **true** and a list of literals, D
- 9: **for** $(X = \alpha) \in D$ **do**
- 10: remove α from the domain of X and propagate
- 11: **end for**
- 12: continue search
- 13: **end if**
- 14: **if** $Var = val$ does not lead to a solution **then**
- 15: set $Var \neq val$ and propagate
- 16: **if** a solution is obtained **then**
- 17: check that the solution is not isomorphic to any previous solution
- 18: **else**
- 19: move to next search node

20: **end if**

21: **end if**

There are several points of interest. We assume that some consistency heuristics are in place, which propagate search decisions, backtracking away from no-goods and either stopping at the first solution or continuing search until all solutions are found. Every time we make a search decision (setting $Var = val$, making early domain deletions provided by the CGT test, or retracting $Var = val$), the consistency heuristics can provide additional domain reductions. The list T passed to the CGT test contains only those literals that involve the current domains of non-ground variables, as opposed to the domains before search. In this sense T is the smallest list we can pass, making the CGT test as efficient as possible. The method – if applied at every node in search – is sound, since we only backtrack away from solutions that are not lex-least, and we make no domain deletions involving lex-least solutions. We add an isomorph rejection step whenever retracting $Var = val$ gives a solution, ensuring that the method is complete in the sense that exactly one solution in each symmetric equivalence class will be returned. This isomorph rejection step is identical to that performed in GAP-SBDD. In fact the entire method is very much in the spirit of GAP-SBDD; the aim is to replace dominance detection by the power and simplicity of lex-ordering SB heuristics.

The above presentation is similar to dynamic CGT methods such as GAP-SBDD and GAP-SBDS. Another way of looking at the method is as a propagator for (unposted) lex-ordering SB constraints. The method backtracks and reduces domains in line with the constraints that a static lex-ordering would have posted before search. Indeed, this conceptualisation motivates the notion that we can *combine* GAPLex with static lex-ordering constraints. The combination will be sound, since the same constraints apply, with only the order of their posting or propagation affecting the dynamics of CSP search. We discuss this aspect more fully in Section 3.

As in other CGT-based SB methods, we can only expect a win if the cost of the CGT test is less than the cost of performing the search needed without early backtracking and early domain deletions provided by the test. By using highly efficient implementations of powerful permutation group algorithms, we can achieve this goal. The GAPLex CGT test, written in GAP, is specified as follows:

Require: G – a symmetry group for a CSP

Require: PA and T – as ordered lists of literals

- 1: **if** PA is **not** lex-least in its orbit under G **then**
- 2: result = false
- 3: **else**
- 4: $D \leftarrow t \in T$ if added to PA would make PA **not** lex-least
- 5: result = true and D
- 6: **end if**

The test proceeds by recursive search, similar to that described in [16], terminating when either the elements of PA have been exhausted, or the group at

the bottom of the stabiliser chain consists only of the identity permutation. This stabiliser chain is the sequence stabiliser of the literals involving the decisions made above the current node during search. More precisely, a recursive routine with the following specification is applied:

Require: G – a permutation group

Require: $SOURCE$ and $EXTRA$ – as ordered lists of points

Require: $TARGET$ – an ordered list of points

- 1: **if** $\exists g \in G : g(SOURCE) \prec_{\text{lex}} TARGET$ **then**
- 2: result = false
- 3: **else**
- 4: $D \leftarrow \{t \in EXTRA : \exists g \in G : g(SOURCE \cup \{t\}) \prec_{\text{lex}} TARGET\}$
- 5: appropriate subset of D is added to a global list DL
- 6: result = true
- 7: **end if**

Calling this routine with the same G , and with $SOURCE$ and $TARGET$ both equal to PA and $EXTRA$ equal to T clearly achieves the specification above. The implementation of this routine is:

- 1: **GAPLexSearch**($G, SOURCE, TARGET, EXTRA$)
- 2: **if** $TARGET$ is empty **then**
- 3: return true
- 4: **end if**
- 5: $x \leftarrow TARGET[1]$
- 6: **for** $y \in SOURCE$ **do**
- 7: **if** $\exists g \in G : g(y) < x$ **then**
- 8: return false
- 9: **else**
- 10: **if** $\exists g \in G : g(y) = x$ **then**
- 11: $G' \leftarrow$ the stabiliser of x in G
- 12: $SOURCE' \leftarrow SOURCE \setminus \{y\}$
- 13: $TARGET' \leftarrow TARGET \setminus \{x\}$
- 14: $res \leftarrow$ **GAPLexSearch**($G', SOURCE', TARGET', EXTRA$)
- 15: **if** $res = \text{false}$ **then**
- 16: return false
- 17: **end if**
- 18: **end if**
- 19: **end if**
- 20: **end for**
- 21: **for** $y \in EXTRA$ **do**
- 22: **if** $y \notin DL$ and $\exists g \in G : g(y) < x$ **then**
- 23: add y to DL
- 24: **end if**
- 25: **end for**
- 26: **return true**

The $\exists g \in G$ tests are done using the standard CGT mechanisms of orbits and Schreier vectors. Each test is, in principle, distinct from any other. However,

GAP will maintain stabiliser chain information after each test is made. This means that similar tests make use of persistent data-structures, which aids efficiency. We enforce restrictions that the variable ordering is static – defined before search – and the value ordering is lex-least. In fact, the test will work with any consistent ordering of literals, provided that GAP is aware of the ordering before search. Our restriction to lex-least for values appears to be the only sensible choice.

2.4 Empirical evaluation

Our implementation uses the GAP–ECLⁱPS^e system, with CSP modelling and search performed in ECLⁱPS^e, and with GAP providing black-box answers to symmetry questions. We have tested our implementation on two classes of CSP. The first is Balanced Incomplete Block Designs (BIBDs), problem class 28 in *csplib*. This class was chosen to be a stern test of the effectiveness of GAPLex. The number of symmetries is large (full row and column symmetries where the numbers of rows and columns are the first two parameters in Table 1), and the domains are small. We therefore expect the search for lex-inspired early backtracks to be expensive, with not many useful domain deletions being returned. This expectation is realised in our results given in Table 1. GAPLex appears to increase the number of backtracks (when compared to GAP-SBDD, but at a greater computational effort). Another hurdle with this class of problems is that our group, although containing the same number of symmetries, acts on twice as many points in the GAPLex implementation. This slows the CGT calculations dramatically.

Table 1. GAP-SBDD vs GAPLex. Problem class: BIBDs modelled as binary matrices.

V	B	R	K	λ	GAP-SBDD				GAP-LEX			
					◇	□	△	○	◇	□	△	○
7	7	3	3	1	3	420	50	470	3	1130	20	1150
6	10	5	3	2	4	810	59	869	29	79970	130	80100
7	14	6	3	2	13	502395	230	502625	-	-	-	-
9	12	4	3	1	12	450792	220	451012	-	-	-	-
11	11	5	5	2	11	68700	210	68910	-	-	-	-
8	14	7	4	3	14	219685	260	219945	-	-	-	-

- > 2 hours ◇ Number of Backtracks □ Gap Time in ms
 △ Eclipse time in ms ○ Total runtime in ms

The second problem class is Graceful Graphs. A labelling f of the vertices of a graph with e edges is *graceful* if f assigns each vertex a unique label from $\{0, 1, \dots, e\}$ and when each edge xy is labelled with $|f(x) - f(y)|$, the edge labels are all different. (Hence, the edge labels are a permutation of $1, 2, \dots, e$.) The standard CSP model has a variable for each vertex, x_1, x_2, \dots, x_n each with

domain $\{0, 1, \dots, e\}$ and a variable for each edge, d_1, d_2, \dots, d_e , each with domain $\{1, 2, \dots, e\}$. The symmetries that arise are any symmetries of the graph, combined with symmetries of the labels. In this class the domains are larger, and, in general, there are fewer symmetries. We therefore expect to see more pruning due to domain deletions, and faster CGT testing for safe backtracks and domain deletions.

The results for this class of problems (Table 2) are more encouraging. We see that, in contrast to BIBDs, GAPLex provides fewer backtracks but performs faster than GAP-SBDD. GAPLex appears to perform as well as GAP-SBDS on these problems. We also tested the heuristic observation that no GAPLex tests will fail (resulting in a backtrack) until the first search-related backtrack occurs. This is because our search ordering respects the lex-ordering that is being tested. By not performing these tests we save on computation, but lose any safe domain deletions that may have been found. Our results for this heuristic are inconclusive for this class of problems.

Table 2. Table comparing various symmetry breaking methods. Partial GAP-LEX is where GAP-LEX checks do not commence until after the 1st backtrack.

Instance	GAP-SBDS				GAP-SBDD			
	◇	□	△	○	◇	□	△	○
$K_3 \times P_2$	9	290	110	400	22	310	180	490
$K_4 \times P_2$	165	1140	3590	4730	496	3449	8670	12110
$K_5 \times P_2$	4390	35520	166149	201669	17977	174180	501580	675760
Instance	GAP-LEX				Partial GAP-LEX			
	◇	□	△	○	◇	□	△	○
$K_3 \times P_2$	10	160	100	260	12	150	130	280
$K_4 \times P_2$	184	1550	4020	5570	202	670	4980	5650
$K_5 \times P_2$	4722	47870	176200	224070	5024	18820	224310	243130

◇ Number of Backtracks □ Gap Time in ms
 △ Eclipse time in ms ○ Total runtime in ms

3 Combining GAPLex with Incomplete Static SB methods

In this Section we analyse in more detail the practicalities of combining GAPLex with the posting of SB constraints before search.

3.1 Double-lex

Much research has concentrated on symmetry-breaking constraints for *matrix models* – a constraint program that contains one or more matrices of decision variables – which occur frequently as CSPs. The prime example of this body of

work is that on lexicographically ordering rows and columns in matrix models [6]. Although the work is general in n -dimensions, we restrict ourselves to 2-d examples for simplicity. We deal with matrices where *rows* and *columns* are independently fully permutable. An $n \times m$ matrix with row and column symmetry is acted on by a group of order $n!m!$ (the direct product of the symmetric groups on n and m points respectively). The rows in a 2-d matrix are *lexicographically ordered* if each row is lexicographically smaller (denoted \preceq_{lex}) than the next (if any). Adding lexicographic ordering on the rows breaks all row symmetries. Similarly, we can break all column symmetry by ordering the columns. But the interesting case is where we insist that both rows and columns should be simultaneously lexicographically ordered – double-lex [6]. In general a lexicographic ordering on both the rows and the columns does *not* break all the compositions of the row and column symmetries. In fact, an exponential number of symmetries can remain unbroken. Nevertheless, in practice it often breaks a useful amount of symmetry.

STAB [19] is an algorithm which works on 2d matrices. After each row is assigned, a subset of the complete set of lexicographic symmetry breaking constraints from the complete set of are constructed and imposed on the next row of the matrix. The number of constraints generated is exponential in the length of a row rather than the size of the matrix. While STAB has produced impressive results, the current algorithm is restricted to only working on matrices of boolean matrices with row and column symmetries.

Because of the usefulness of lex-ordering rows and columns, and the simplicity of the constraints, Frisch *et al* introduced an optimal algorithm to establish generalised arc-consistency for the \preceq_{lex} constraint [9] between two vectors. This gives an extremely attractive point on the tradeoff: a linear time to establish a high level of consistency on constraints which often break a lot of the symmetry in matrix models. The algorithm can be used to establish consistency in any use of \preceq_{lex} , so in particular is useful for any use of lex-leader constraints.

3.2 Combining GAPLex and double-lex

Our approach is straightforward. We add static double-lex constraints before search. At each node in the search tree we run GAPLex, *without* supplying the list of candidate domain deletions. The CGT test is now a simple predicate, returning false when it is safe to backtrack and true otherwise. This clearly means that the test is computationally more efficient: the final for-loop in the CGT algorithm isn't performed. We justify this by noting that any safe deletion found would almost certainly be already ruled out by the static double-lex constraints.

In Table 3 we see that GAPLex does not perform as well as simply posting double-lex constraints before search. However, GAPLex returns the correct number of solutions, whilst double-lex returns many symmetrically equivalent solutions. For example, there is exactly one symmetrically distinct BIBD with parameters (9,12,4,3,1), and double-lex returns 92. It seems that combining GAPLex with double-lex is a win over just using GAPLex. These results are not unexpected, as GAPLex was shown to behave poorly on this formulation of

Table 3. Static double-lex *vs* GAPLex with no search for safe deletions *vs* combined GAPLex and double-lex. Problem class: all solutions of BIBDs modelled as binary matrices.

					Double Lex		GAP-Lex no Prop				Combined			
V	B	R	K	λ	\diamond	\circ	\diamond	\square	\triangle	\circ	\diamond	\square	\triangle	\circ
7	7	3	3	1	3	20	21	1330	59	1389	3	1130	20	1150
6	10	5	3	2	5	30	29	79970	130	80100	4	50290	50	50340
7	14	6	3	2	30	110	-	-	-	-	-	-	-	-
9	12	4	3	1	30	120	-	-	-	-	-	-	-	-
11	11	5	5	2	20	140	-	-	-	-	-	-	-	-
8	14	7	4	3	143	720	-	-	-	-	-	-	-	-

- > 2 hours \diamond Number of Backtracks \square Gap Time in ms
 \triangle Eclipse time in ms \circ Total runtime in ms

BIBDs in Section 2.4. We feel that the proof of concept is, however, interesting and useful.

3.3 Combining GAPLex with Puget’s all-different constraints

Puget has recently presented constraints for breaking symmetries in CSPs with all-different constraints [20]. These constraints do not, in general, break all symmetries. They are, therefore, a candidate for being combined with GAPLex. Our approach is the same as for double-lex: we post the static all-different constraints before search, and run the GAPLex test at each search node. Again, we do not search for safe deletions, as these are expected to be already ruled out by the static constraints.

Our results, given in Table 4, show that GAPLex performs slightly better than static constraints, and that combining the two methods is better than using either in isolation. These encouraging results are made better by noting that, for this class of problems, using only static constraints results in twice as many solutions being returned as necessary.

Table 4. Static symmetry breaking all-different constraints *vs* GAPLex with no search for safe deletions *vs* combined GAPLex and static constraints. Problem class: all solutions of Graceful Graphs in the standard model.

Instance	Constraints		GAP-Lex no Prop				Combined			
	\diamond	\circ	\diamond	\square	\triangle	\circ	\diamond	\square	\triangle	\circ
$K_3 \times P_2$	16	800	12	150	130	280	10	140	100	240
$K_4 \times P_2$	369	4530	202	600	5140	5740	188	510	3300	3810
$K_5 \times P_2$	9887	297880	5024	19010	224740	243750	4787	14820	188820	203640

\diamond Number of Backtracks \square Gap Time in ms
 \triangle Eclipse time in ms \circ Total runtime in ms

4 Conclusions and Future Work

We have used recent advances in Computational Group Theory to add lex-ordering to the class of symmetry breaking techniques that can be effectively implemented by using a CGT system to provide black-box answers to symmetry related questions. Our implementation, GAPLex, is competitive with GAP-SBDS and GAP-SBDD. The choice of which method to use for which class of CSPs appears to be an interesting research question. Answers to this question could provide insight into yet more symmetry breaking methods.

We have, moreover, demonstrated the first combination of static and search-based symmetry breaking methods that is (for certain classes of CSP) more efficient than using either the static or search-based method in isolation. This result is important, since the successful combination of symmetry breaking methods is taxing and open area of CSP research, with great potential benefits attached to positive answers.

More work is needed in two areas. Firstly, we must address the questions relating to why a particular symmetry breaking approach works better for some CSPs than others. Secondly, we need to investigate other potentially successful combinations of symmetry breaking techniques.

Acknowledgements

We are very grateful for their helpful comments and other assistance to Ian P. Gent and Barbara Smith, and all the attendees at the 2006 Symnet sandpit on Combining Symmetry Breaking Techniques. This work is supported by EPSRC grants GR/S30580/01 (Symmetry and Inference), EP/CS23229/1 (Critical Mass) and GR/S86037/01 (Symnet Network).

References

1. Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov, *Efficient symmetry breaking for boolean satisfiability*, IJCAI (Georg Gottlob and Toby Walsh, eds.), Morgan Kaufmann, 2003, pp. 271–276.
2. R. Backofen and S. Will, *Excluding symmetries in constraint-based search*, Proceedings, CP-99, Springer, 1999, pp. 73–87.
3. Wieb Bosma and John Cannon, *Handbook of MAGMA functions*, sydneypm, Sydney University, 1993.
4. J. Crawford, M. Ginsberg, E. Luks, and A. Roy, *Symmetry-breaking predicates for search problems*, Proc. KR'96, November 1996, pp. 149–159.
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, Zeynep Kızıltan, Ian Miguel, Justin Pearson, and Toby Walsh, *Breaking row and column symmetries in matrix models*, Proc. CP 2002 (Pascal Van Hentenryck, ed.), 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. Alan M. Frisch, Christopher Jefferson, and Ian Miguel, *Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern.*, ECAI (Ramon López de Mántaras and Lorenza Saitta, eds.), IOS Press, 2004, pp. 171–175.
9. A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh, *Global constraints for lexicographic orderings*, Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (P. van Hentenryck, ed.), Lecture Notes in Computer Science, vol. 2470, Springer, 2002, pp. 93–108.
10. The GAP Group, *GAP – Groups, Algorithms, and Programming, Version 4.2*, 2000, (<http://www.gap-system.org>).
11. 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.
12. Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton, *Generic SBDD using computational group theory*, Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming — CP’2003 (Francesca Rossi, ed.), Lecture Notes in Computer Science, vol. 2833, Springer, 2003, pp. 333–347.
13. I.P. Gent and B.M. Smith, *Symmetry breaking in constraint programming*, Proc. ECAI 2000 (W. Horn, ed.), IOS Press, 2000, pp. 599–603.
14. Warwick Harvey, *A note on the compatibility of static symmetry breaking constraints and dynamic symmetry breaking methods*, Proceedings SymCon-04: Symmetry and Constraint Satisfaction Problems, 2004, pp. 42–47.
15. Tom Kelsey, Steve Linton, and Colva M. Roney-Dougal, *New developments in symmetry breaking in search using computational group theory.*, AISC (Bruno Buchberger and John A. Campbell, eds.), Lecture Notes in Computer Science, vol. 3249, Springer, 2004, pp. 199–210.
16. Steve Linton, *Finding the smallest image of a set*, Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation: ISSAC 2004, ACM, New York, 2004, pp. 229–234.
17. P. Meseguer and C. Torras, *Exploiting Symmetries within Constraint Satisfaction Search*, Artificial Intelligence **129** (2001), 133–163.
18. Jean-Francois Puget, *On the satisfiability of symmetrical constraint satisfaction problems*, Methodologies for Intelligent Systems (Proceedings of ISMIS’93), LNAI 689, Springer, 1993, pp. 350–361.
19. ———, *Symmetry breaking using stabilizers.*, CP (Francesca Rossi, ed.), Lecture Notes in Computer Science, vol. 2833, Springer, 2003, pp. 585–599.
20. ———, *Breaking symmetries in all different problems*, Proceedings of IJCAI05, 2005, pp. 272–277.
21. Colva M. Roney-Dougal, Ian P. Gent, Tom Kelsey, and Steve Linton, *Tractable symmetry breaking using restricted search trees*, Proceedings, 16th European Conference on Artificial Intelligence: ECAI-04 (Ramon López de Mántaras and Lorenza Saitta, eds.), IOS Press, 2004, pp. 211–215.
22. M. G. Wallace, S. Novello, and J. Schimpf, *ECLiPSe : A platform for constraint logic programming*, ICL Systems Journal **12** (1997), no. 1, 159–200.