Constraints for Breaking More Row and Column Symmetries

Alan M. Frisch, Chris Jefferson, and Ian Miguel

Artificial Intelligence Group
Department of Computer Science
University of York
York, United Kingdom
{frisch, caj, ianm}@cs.york.ac.uk

Abstract. Constraint programs containing a matrix of two (or more) dimensions of decision variables often have row and column symmetries: in any assignment to the variables the rows can be swapped and the columns can be swapped without affecting whether or not the assignment is a solution. This introduces an enormous amount of redundancy when searching a space of partial assignments. It has been shown previously that one can remove consistently some of these symmetries by extending such a program with constraints that require the rows and columns to be lexicographically ordered. This paper identifies and studies the properties of a new additional constraint—the first row is less than or equal to all permutations of all other rows—that can be added consistently to break even more symmetries. Two alternative implementations of this stronger symmetry-breaking method are investigated, one of which employs a new algorithm that in time linear in the size of the matrix enforces the constraint that the first row is less than or equal to all permutations of all other rows. It is demonstrated experimentally that our method for breaking more symmetries substantially reduces search effort.

1 Introduction

A common pattern arising in finite domain constraint programs is the matrix of decision variables with two or more dimensions [6]. In two-dimensional matrices it is often the case that some or all of the rows are interchangeable and some or all of the columns are interchangeable. That is, an assignment to the variables in the matrix is a solution if and only if it is still a solution after two of the interchangeable rows are swapped or two of the interchangeable columns are swapped. In a two dimensional matrix this is called *row and column symmetry*. Since this property can also arise in each of many dimensions, more generally it is called *index symmetry*. To simplify our presentation, we assume throughout that we are dealing only with two dimensional matrices in which all rows are interchangeable and all columns are interchangeable.

Symmetry in constraint programs can cause problems for an algorithm that searches a space of partial assignments due to redundancy in the search space.

One of the most popular methods for reducing symmetry is to add to the model extra constraints, so-called *symmetry breaking constraints*.

Flener et al [5] studied index symmetry and showed that one can *consistently* add the symmetry-breaking constraint, called lex^2, that both the rows and the columns are lexicographically ordered.[1] This means that for every assignment to the variables that does not satisfy the symmetry-breaking constraint, there is a symmetric assignment that does. They also showed that for certain problems imposing lex^2 can make the difference between solving and failing to solve the problem. Frisch et al [7] introduced an efficient algorithm for maintaining generalised arc-consistency on the constraint that one vector (a row or column of a matrix) is lexicographically less than or equal to another.

Independently, Flener et al [5] and Shlyakhter [14] also showed that lex^2 does not break all row and column symmetries. That is, lex^2 is *incomplete* in that it can be satisfied by two symmetrical assignments. Consider the following two matrices. Both satisfy lex^2, but the second can be obtained from the first by swapping the rows and rotating the columns to the right.

$$\begin{pmatrix} 2 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 2 \end{pmatrix}$$

It is therefore natural to wonder if more symmetry can be broken *effectively* by imposing a symmetry-breaking constraint stronger than lex^2. That is, can we add extra constraints to lex^2 that both break a significant number of the symmetries left by lex^2 and can be propagated efficiently? This paper addresses this question and answers it in the affirmative. In particular, we introduce a new constraint, called $allperm$, that is satisfied by a matrix if and only if the first row is lexicographically less than or equal to all permutations of all other rows. Only the second matrix above satisfies both lex^2 and $allperm$. Section 3 shows that one can consistently impose both lex^2 and $allperm$ on a matrix that has symmetric rows and symmetric columns. This is shown to break more symmetries than lex^2 on its own but not to break all symmetries.

The $allperm$ constraint can be decomposed into individual lexicographic ordering constraints by adding a constraint for each of the permutations of each row, of which there are a number factorial in the length of the row. This is not an effective way to break symmetry. Instead, we examine other decompositions, leading in Section 6 to an algorithm which implements $allperm$ in time linear in the size of the matrix. This algorithm is then tested in Section 8, where it is demonstrated that some problems can be solved much more effectively when symmetries are broken by lex^2 and $allperm$ together than by lex^2 on its own.

2 Terminology

We are concerned with finite domain constraint satisfaction problems so every variable is associated with a finite domain of values. An *assignment* maps every

[1] Though working in a different context, Shlyakhter [14] independently showed the consistency.

variable to a member of its domain. We write $vars(C)$ to denote the variables constrained by constraint C. We say that a constraint C is *generalised arc consistent* (GAC), written $GAC(C)$, if for every variable $x \in vars(C)$ and every value v in the domain of x, there is an assignment that maps x to v and satisfies C. A finite set of constraints, $\{C_1, \ldots, C_n\}$, is said to be GAC if and only if $C_1 \wedge \cdots \wedge C_n$ is GAC.

A set of constraints S *logically implies* another set of constraints S' if every assignment that satisfies every member of S also satisfies every member of S'. If S and S' logically imply each other, then they are said to be *logically equivalent*.

An $n \times m$ matrix has n columns and m rows. The columns are numbered $1, \ldots, n$ from left to right and the rows are numbered $1, \ldots, m$ from top to bottom. A row of an $n \times m$ matrix can be treated as a vector, $\boldsymbol{x} = \langle x_1, \ldots, x_n \rangle$, by reading it left to right and a column can be treated as a vector, $\boldsymbol{y} = \langle y_1, \ldots, y_m \rangle$, by reading it top to bottom.

This paper focuses on ordering non-empty vectors of equal size, so we shall simplify the presentation by assuming this throughout the paper. One vector, $\boldsymbol{x}$, is defined to be *lexicographically less than or equal to* another, $\boldsymbol{y}$, (written $\boldsymbol{x} \leq_{lex} \boldsymbol{y}$) if $\boldsymbol{x} = \boldsymbol{y}$ or $x_i < y_i$, where i is the smallest index such that $x_i \neq y_i$.

This paper introduces two new orderings on vectors. The first, the anti-multiset ordering, has a definition that mirrors the multiset ordering definition. We write $\sharp \boldsymbol{x}|_v$ to denote the number of occurrences of v in $\boldsymbol{x}$. We define $\boldsymbol{x}$ to be *anti-multiset less than or equal to* $\boldsymbol{y}$, written $\boldsymbol{x} \leq_{\overline{m}} \boldsymbol{y}$, if $\boldsymbol{x} = \boldsymbol{y}$ or $\sharp \boldsymbol{x}|_v > \sharp \boldsymbol{y}|_v$, where v is the smallest value such that $\sharp \boldsymbol{x}|_v \neq \sharp \boldsymbol{y}|_v$. The second ordering is the allperm ordering. We write $\boldsymbol{x} \leq_{ap} \boldsymbol{y}$ if $\boldsymbol{x} \leq_{lex} \boldsymbol{y'}$ for every permutation $\boldsymbol{y'}$ of $\boldsymbol{y}$. It should be noted that this is an abuse of notation as $\leq_{ap}$ is not reflexive (but it is anti-symmetric and transitive).

Using these orderings on vectors, we define some predicates on matrices. Let M be a matrix of values. We write $lex^2(M)$ to mean that the rows of M are lexicographically non-decreasing from top to bottom and the columns are lexicographically non-decreasing from left to right. We write $ams(M)$ to mean that the first row of M is less than or equal to all other rows in the anti-multiset order and, similarly, we write $allperm(M)$ to mean that the first row of M is less than or equal to all other rows in the allperm order.

We also use the symbols $\leq_{lex}$, $\leq_{\overline{m}}$, $\leq_{ap}$, lex^2, ams and $allperm$ to denote constraints that are imposed on vectors or matrices of variables.

3 Basic Properties

In Section 1 we saw an example demonstrating that $allperm$ and lex^2 break more symmetries than lex^2 alone. This section shows that imposing both $allperm$ and lex^2 on a matrix with row and column symmetry is consistent and is incomplete. We show consistency by examining a complete and consistent set of symmetry-breaking constraints known as the row-wise lex-leader constraints, and incompleteness by example.

One way to break all the index symmetries in a matrix is to impose the row-wise lex-leader constraints, which are a specific case of the more general lex-leader constraints introduced by Crawford et al. [4]. The row-wise lex-leader constraints are derived by considering a matrix of distinct variables and all matrices symmetric to it. Each matrix is converted to a string of variables by scanning the matrix row-wise, left-to-right, top-to-bottom. For example, the 3×2 matrix of variables

$$\begin{pmatrix} A & B & C \\ D & E & F \end{pmatrix}$$

yields the string $ABCDEF$. From these strings we produce a set of constraints asserting that the string from the original matrix—called the *lex-leader*—is lexicographically less than or equal to each of the other strings. Thus, continuing our example, there are 11 other matrices that can be produced by permuting the rows and columns of the above matrix. Each of these yields a string of variables that is constrained to be lexicographically less than or equal to $ABCDEF$; thus 11 constraints, called the row-wise lex-leader constraints, are generated:

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

Since the row-wise lex-leader constraints are generated by the method proposed by Crawford et al. [4], we know that they are consistent and complete. However, this does not generally provide an effective way to break row and column symmetries since for an $n \times m$ matrix it yields $n! \cdot m! - 1$ constraints.

It has been shown that the lex^2 constraint is logically implied by the row-wise lex-leader constraints, from which it follows that lex^2 is a consistent symmetry-breaking constraint [14]. We now show a new result—that $allperm$ is logically implied by the row-wise lex-leader constraints—from which it follows that the combination of lex^2 and $allperm$ is consistent.

We first show by example that the row-wise lex-leader constraints logically imply that the first row is lexicographically less than or equal to a permutation of another row. In particular, using the example matrix above, we show that the row-wise lex-leader constraints imply $ABC \leq_{lex} FED$. First observe that there is a matrix that is symmetric to the above and whose first row is FED (obtained by swapping the two rows and swapping the first and third columns):

$$\begin{pmatrix} F & E & D \\ C & B & A \end{pmatrix}$$

This matrix yields the string $FEDCBA$. Hence, $ABCDEF \leq_{lex} FEDCBA$ is one of the row-wise lex-leader constraints, namely constraint (10) above. And this constraint logically implies $ABC \leq_{lex} FED$.

One can easily generalise this example to a proof that the row-wise lex-leader constraints logically imply every constraint of the form that the first row is lexicographically less than or equal to every permutation of every other row. Hence, we have the following theorem:

Theorem 1. *The row-wise lex-leader constraints for a matrix M of variables logically imply both $lex^2(M)$ and $allperm(M)$. Hence the conjunction of $lex^2(M)$ and $allperm(M)$ is a consistent symmetry-breaking constraint.*

The conjunction of lex^2 and $allperm$ is a consistent symmetry-breaking constraint, but it is not complete. To see this observe that the following two matrices can be obtained from each other by permuting rows and columns, yet both satisfy the lex^2 and $allperm$ constraints.

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix} \qquad \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$$

4 An Alternative Characterisation

This section gives an alternative characterisation of the constraint $lex^2(M) \wedge allperm(M)$.

We start by relating the lexicographic and allperm orderings to the anti-multiset ordering. Let $\boldsymbol{x}\uparrow$ denote the vector of values that results from sorting the elements of $\boldsymbol{x}$ from smallest to largest. Notice that $\boldsymbol{x} \leq_{ap} \boldsymbol{y}$ implies $\boldsymbol{x}\uparrow \leq_{ap} \boldsymbol{y}$, which implies $\boldsymbol{x} \leq_{\overline{m}} \boldsymbol{y}$. This chain of implications does not hold in the opposite direction, unless $\boldsymbol{x} = \boldsymbol{x}\uparrow$.

We now extend these two observations to matrices. From the first observation it follows that that $allperm(M)$ implies $ams(M)$. From the second observation it follows that if the first row of M is non-decreasing, then $ams(M)$ implies $allperm(M)$. Now notice that if the columns of matrix M are lexicographically ordered, then the first row of M is nondecreasing. From this our alternative characterisation follows.

Theorem 2 (Characterisation Theorem). *Let M be a matrix of values. Then (1) $allperm(M)$ implies $ams(M)$, (2) $lex^2(M) \wedge ams(M)$ implies $lex^2(M) \wedge allperm(M)$ and, thus, (3) $lex^2(M) \wedge allperm(M)$ if and only if $lex^2(M) \wedge ams(M)$.*

5 Decomposing the Constraints

There is no known algorithm for maintaining GAC on the conjunction of lex^2 and $allperm$ (or, equivalently, the conjunction of lex^2 and ams). Furthermore, the complexity of this conjunction suggests that the development of such an algorithm is probably beyond the reach of current capability. Therefore, the best approach is to decompose the constraint into constituents, for each of which a GAC algorithm is known or could be developed. This section examines and compares some decompositions. But first we present a general theorem about decomposition that is particularly useful for our purposes.

5.1 The Decomposition Theorem

The Decomposition Theorem identifies a simple condition under which a conjunction is GAC if each of its conjuncts is GAC.

Let C be a constraint and x be a variable in $vars(C)$. We say that C is *upwardly monotonic* in x if given any assignment that satisfies C, then replacing the value assigned to x with any larger value also satisfies C. We define downwardly monotonic in a similar fashion. A set S of constraints *agree monotonically* if for any two constraints $C_1, C_2 \in S$ and every variable x in $vars(C_1) \cap vars(C_2)$ either C_1 and C_2 are upwardly monotonic in x or both are downwardly monotonic in x.

Theorem 3 (Decomposition Theorem). *Given a finite set, S, of constraints that agree monotonically, $GAC(S)$ if and only if $GAC(C)$ for every $C \in S$.*

Proof. Clearly the "only-if" part holds, so we turn our attention to the "if" part.

Consider an arbitrary value v in domain of a variable x. We know that we can find support for v assigned to x in all constraints it appears in as $GAC(C)$ holds for all $C \in S$. For any constraint which does not refer to x we can choose any variable in the constraint and any value in that variables domain and find support for it.

Without loss of generality we will assume all the monotonic variables are upwardly monotonic. Given a valid assignment for a constraint C which contains the upwardly monotonic variable v then we can increase the value of v and still have a valid assignment for C. Therefore we can take all of the upwardly monotonic variables (except x) in the assignments of all the constraints and increase the values they take to the maximum value in their domain. Now the assignments we have for all the constraints agree on all the variables that appear in more than one constraint. Therefore we have shown $GAC(C)$ for all $C \in S$ implies $GAC(S)$ □

5.2 First Decomposition

The first, and most obvious, decomposition is to break the conjunctions and maintain GAC on the conjuncts. Doing so reduces pruning: $GAC(lex^2(M) \wedge allperm(M))$ is not implied by $GAC(lex^2(M)) \wedge GAC(allperm(M))$. This is demonstrated by considering the 4×2 matrix M of distinct variables where the domains of the eight variables are:

$$\begin{array}{cccc} \{1,2\} & \{1,2\} & \{4\} & \{4\} \\ \{3\} & \{3\} & \{2\} & \{2\} \end{array}$$

Observe that $GAC(lex^2(M)) \wedge GAC(allperm(M))$ but not $GAC(lex^2(M) \wedge allperm(M))$, as no solution assigns 2 to the first variable of the first row. This same example demonstrates that decomposing $GAC(lex^2(M) \wedge ams(M))$ reduces pruning: we have $GAC(lex^2(M))$ and $GAC(ams(M))$ but not $GAC(lex^2(M) \wedge ams(M))$.

Since both decompositions reduce pruning, we would like to know which suffers the greater reduction. From part (1) of the Characterisation Theorem it follows that $GAC(allperm(M))$ implies $GAC(ams(M))$. So $GAC(lex^2(M)) \wedge GAC(allperm(M))$ is at least as strong as $GAC(lex^2(M)) \wedge GAC(ams(M))$. We now give an example to show that it is strictly stronger. Consider M', a 4×2 matrix of distinct variables where the domains of the eight variables are:

$$\begin{array}{cccc} \{1\} & \{2\} & \{3,4\} & \{3,4\} \\ \{2\} & \{1\} & \{3\} & \{4\} \end{array}$$

Both $lex^2(M')$ and $ams(M')$ are GAC. However $allperm(M')$ is not; no solution assigns 4 to the third variable of the first row.

5.3 Further Decomposition

We now turn our attention to the $allperm(M)$ and $ams(M)$ constraints and show that each can be decomposed without any loss of pruning.

For two vectors of variables $\boldsymbol{x}$ and $\boldsymbol{y}$ that share no variables, the constraints $\boldsymbol{x} \leq_{lex} \boldsymbol{y}$, $\boldsymbol{x} \leq_{ap} \boldsymbol{y}$ and $\boldsymbol{x} \leq_{\overline{m}} \boldsymbol{y}$ are each downwardly monotonic in the variables of $\boldsymbol{x}$ and upwardly monotonic in the variables of $\boldsymbol{y}$.

Let X be a matrix of distinct variables whose rows are $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$. Then we have $GAC(allperm(X))$ if and only if (by the definition of $allperm$) $GAC(\bigwedge_{2\leq j\leq n} \boldsymbol{x_1} \leq_{ap} \boldsymbol{x_j})$ if and only if (by the Decomposition Theorem) $\bigwedge_{2\leq j\leq n} GAC(\boldsymbol{x_1} \leq_{ap} \boldsymbol{x_j})$. Similarly, $GAC(ams(X))$ if and only if (by the definition of ams) $GAC(\bigwedge_{2\leq j\leq n} \boldsymbol{x_1} \leq_{\overline{m}} \boldsymbol{x_j})$ if and only if (by the Decomposition Theorem) $\bigwedge_{2\leq j\leq n} GAC(\boldsymbol{x_1} \leq_{\overline{m}} \boldsymbol{x_j})$.

Thus, without losing any pruning, an algorithm for maintaining GAC on ams can be implemented by $n-1$ instances of an algorithm that maintains GAC on $\leq_{\overline{m}}$. We have obtained an efficient algorithm for maintaining GAC on $\leq_{\overline{m}}$ by implementing a slight modification to an existing algorithm for maintaining GAC on $\leq_m$ [8]. The run time of the algorithm is $O(m + d)$ where m is the length of the vectors and d is the domain size of the variables in the vector. Similarly, without losing any pruning, an algorithm for maintaining GAC on $allperm$ can be implemented by $n-1$ instances of an algorithm that maintains GAC on $\leq_{ap}$. There is no existing algorithm for maintaining GAC on $\leq_{ap}$, except the factorial decomposition described in Section 1, so we turn our attention to that task.

6 A GAC Algorithm for *Allperm*

We now prove correct a method of constructing an algorithm that enforces GAC on *allperm*. We refer throughout to vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of distinct variables (with no variables in common). $\max(x_i)$ and $\max(\boldsymbol{x})$ denote the largest element in the domain of x_i and the vector of values obtained by replacing each variable in $\boldsymbol{x}$ with the maximum value in its domain. Similarly for $\min(x_i)$ and $\min(\boldsymbol{x})$. A element x_i of $\boldsymbol{x}$ is *bound* if there is only one value in its domain. A *permutation* of the variables of $\boldsymbol{y}$ is usually denoted by $\boldsymbol{p}$. We refer to the elements of $\boldsymbol{p}$ either

by their position in $\boldsymbol{p}$, or by their position in $\boldsymbol{y}$. $\boldsymbol{y}\uparrow$ is $\boldsymbol{y}$ sorted using the upper bound of each element. We define $\boldsymbol{y} - y_i$ to be the vector $\boldsymbol{y}$, with the i^{th} element removed. If $\boldsymbol{y}$ is a vector of variables and v is a variable, then $v \cdot \boldsymbol{y}$ denotes the vector that is obtained by placing v at the head of $\boldsymbol{y}$.

Lemma 1. *GAC($\boldsymbol{x} \leq_{ap} \boldsymbol{y}$) implies that for all $y_i \in \boldsymbol{y}$, GAC($x_1 \leq y_i$)*

Proof. Since the first constraint implies the second, GAC of the first constraint implies GAC of the second. □

Theorem 4. *If x_1 is bound to v, some $y_i \in \boldsymbol{y}$ is also bound to v, and for all $y_j \in \boldsymbol{y}$, $min(y_j) \geq v$, then $GAC(\boldsymbol{x} \leq_{ap} \boldsymbol{y})$ if and only if $GAC(\boldsymbol{x} - x_1 \leq_{ap} \boldsymbol{y} - y_i)$.*

Proof. Since for all y_j, $\min(y_j) \geq y_i$ then y_i can be taken as the most significant value of the minimum permutation of any assignment to $\boldsymbol{y}$. Consider some assignment to $\boldsymbol{x}$ and $\boldsymbol{y}$. Now, $\boldsymbol{x} \leq_{ap} \boldsymbol{y}$ if and only if $\boldsymbol{x} \leq_{\text{lex}} \boldsymbol{y}\uparrow$. Since $x_1 = y_i$, this lexicographic order is unaffected by the presence or otherwise of x_1 and y_i □

We can now perform the first stage of a GAC-enforcing algorithm. First prune x_1 and the y_i by Lemma 1. If the preconditions of Theorem 4 are satisfied, remove one element from each of $\boldsymbol{x}$ and $\boldsymbol{y}$ as shown in the theorem and repeat the process. When GAC($x_1 \leq y_i$) for all i and either x_1 is not a singleton or there is no singleton in $\boldsymbol{y}$ equal to x_1, $\boldsymbol{x}$ and $\boldsymbol{y}$ are defined to be "Stage 1 complete".

Theorem 5. *Given "Stage 1 complete" vectors of variables $\boldsymbol{x}$ and $\boldsymbol{y}$, for each permutation $\boldsymbol{p}$ of $\boldsymbol{y}$, $x_1 := max(x_1)$ and $p_1 := min(p_1)$ are the only possible values which can lack support with respect to $\boldsymbol{x} \leq_{lex} \boldsymbol{p}$.*

Proof. From Theorem 4, x_1 and p_1 are not bound and equal and satisfy GAC($x_1 \leq p_1$) and hence x_1 and p_1 can be assigned such that $x_1 < p_1$. □

Lemma 2. *If $\boldsymbol{x} \leq_{ap} \boldsymbol{y}$ is Stage 1 Complete and any element is pruned that lacks support, Stage 1 Completeness is maintained.*

Proof. Enforcing Stage 1 completeness consists of enforcing for all $y_i \in \boldsymbol{y}$, GAC($x_1 \leq y_i$) (Lemma 1) and discarding equal bound pairs (Theorem 4). From Theorem 5 only $\max(x_1)$ and $\min(y_i)$ are candidates for pruning. Pruning in either case cannot produce equal bound pairs nor violate the bounds consistency condition unless an empty domain is produced - in which case we fail. □

¿From Theorems 3 and 5, to establish support for the domains of x_1 and each variable y_i in $\boldsymbol{y}$, it is sufficient to consider only the permutations of $\boldsymbol{y}$ of the form $\langle y_1, \ldots \rangle$. We can consider all these permutations simultaneously, as given $\boldsymbol{y'} = \boldsymbol{y} - y_i$, if $\boldsymbol{x} \leq_{lex} y_i \cdot (\boldsymbol{y'}\uparrow)$ is satisfiable, then so are all similar expressions containing other permutations of $\boldsymbol{y'}$,

Given vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ of variables and $\boldsymbol{p} = \boldsymbol{y}\uparrow$ then the pair $(\boldsymbol{x}, \boldsymbol{y})$ is *tail-inconsistent$_i$* if $\boldsymbol{x'} = \boldsymbol{x} - x_1$ and $\boldsymbol{p'} = \boldsymbol{p} - y_i$ then $\boldsymbol{x'} \leq_{lex} \boldsymbol{p'}$ is unsatisfiable. If $(\boldsymbol{x}, \boldsymbol{y})$ is tail-inconsistent$_i$, the *inconsistency offset* $\eta_{\boldsymbol{x},\boldsymbol{y},i} = j$ is the smallest j such that $min(x'_j) > max(p'_j)$, otherwise $\eta_{\boldsymbol{x},\boldsymbol{y},i} = \infty$. If $\eta_{\boldsymbol{x},\boldsymbol{y},i} < \infty$ there is no valid assignment of $(x) \leq_{ap} \boldsymbol{y}$ with $x_1 = y_i$ so we have to prune the domains of x_1 and y_i to forbid such assignments. After this we will have achieved $GAC(\boldsymbol{x} \leq_{ap} \boldsymbol{y})$ by Theorem 5.

Theorem 6. *Consider two equal length vectors of variables* $\boldsymbol{x}$ *and* $\boldsymbol{y}$*. If* $\eta_{\boldsymbol{x},\boldsymbol{y},i} = j$ $(j \neq \infty)$*, and* $\boldsymbol{p} = (\boldsymbol{y} - y_i)\uparrow$ *then for all* $y_k \in \boldsymbol{y}$ $(max(y_k) \geq max(p_{j+1})$ *or* $max(y_k) > max(p_j))$ *implies* $\eta_{\boldsymbol{x},\boldsymbol{y},k} \leq j$

Proof. Form $(\boldsymbol{y} - \{y_i, y_k\})\uparrow$. Ignoring alternative orderings of variables with equal upper bounds, this permutation is equal to $\boldsymbol{p}$ in at least the first j positions. Now form $(\boldsymbol{y} - y_k)\uparrow$. For the first j positions, $(\boldsymbol{y} - y_k)\uparrow \leq_{\text{lex}} \boldsymbol{p}$, since either $\max(y_i) > \max(p_j)$, or y_i has displaced elements whose upper bound are greater than $\max(y_i)$ which can only lexicographically reduce the first j elements. □

Theorem 7. *Consider vectors of variables* $\boldsymbol{x}$*,* $\boldsymbol{y}$ *and constant* i *s.t. for all* $y_k \in \boldsymbol{y}$*,*$max(y_i) \geq max(y_k)$*. If* $\eta_{\boldsymbol{x},\boldsymbol{y},i} = j$ $(j \neq \infty)$ *and* $\boldsymbol{p} = (\boldsymbol{y} - y_i)\uparrow$ *then for all* $y_k \in \boldsymbol{y}$*,* $max(y_k) < max(p_j)$ *implies* $\eta_{\boldsymbol{x},\boldsymbol{y},k} = \infty$

Proof. (sketch). Form $\boldsymbol{q} = (\boldsymbol{y} - y_k)\uparrow$. Without loss of generality assume y_k is the last element of $\boldsymbol{q}$ with upper bound $= \max(y_k)$. Since $\max(y_k) < \max(p_j)$ we know y_k lies in the first $j-1$ elements of $\boldsymbol{q}$. As $\max(y_i) > \max(y_k)$, $\boldsymbol{q}$ agrees with $\boldsymbol{p}$ until the position p_h where y_k occurs, where it will be replaced by a variable with larger upper bound. However as $\eta_{\boldsymbol{x},\boldsymbol{y},i} = j$, at this position $\max(p_h) = \min(x_h)$. As $\max(q_h) > \min(p_h)$ a consistent instantiation exists. □

Note that $\max(y_i) < \max(y_j)$ implies $\eta_{\boldsymbol{x},\boldsymbol{y},i} < \eta_{\boldsymbol{x},\boldsymbol{y},j}$ so if any k has $\eta_{\boldsymbol{x},\boldsymbol{y},k} < \infty$ the y_k with highest upper bound will have. And once we have $\zeta = \eta_{\boldsymbol{x},\boldsymbol{y},k}$ we can use theorems 6 and 7 tell us the if every other element is tail inconsistent or not except possibly those with the same upper bound as $\boldsymbol{p}_\zeta$ in the minimal permutation which must be tested separately.

6.1 Algorithm Details

The theorems above are embodied in the **GACAP** algorithm below. First, some notes on the structures used. Buckets are objects into which we can place and remove CSP variables. Variables are allocated to buckets according to the largest element in the domain of the variable. When asked to remove an element, a bucket will remove only one, even if it has multiple copies. The function GSNEB returns the smallest non-empty bucket. We *iterate* through a collection of buckets in ascending order.

As described in Section 5.3, an algorithm, **GACAllPerm**, for maintaining GAC on *allperm* on an $n \times m$ matrix can be constructed via $n-1$ instances of **GACAP**.

```
1  Procedure GACAP(vector x, vector y)
2  GLOBAL:  B := newBucketsort(), α := 1, MaxPrune := min(x_1) − 1, n := length(x)
3  FOR (y_i ∈ y) B.add(y_i)
4  Stage1()
5  ηPruning()
6  CleanUp()
7  Procedure Stage1()
8  WHILE (α ≤ n)
9     MaxPrune := max(MaxPrune, min(x_α) − 1)
10    IF (B.GSNEB() ≤ MaxPrune) FAIL
11    x_α.setMax(B.GSNEB())
12    IF ¬(x_α.isBound() AND B.GSNEB() = x_α) RETURN
13    B.remove(B.GSNEB()), α++
14 Procedure GetNextNonEqual(iterator,η)
15 WHILE (max(Iterator.current()) = min(x_η))
16    IF(η = n) RETURN true
17    η++, iterator++
18 RETURN false
19 Procedure ηPruning()
20 η := α + 1, iterator := MakeIterator(B)
21 IF (GetNextNonEqual(iterator, η)) RETURN
22 checker := max(iterator.current())
23 IF(min(x_η) ≤ checker) RETURN
24 FOR (y_i ∈ y s.t. max(y_i) > checker) y_i.setMin(min(x_α) + 1)
25 iterator++
26 IF (GetNextNonEqual(iterator, η)) RETURN
27 IF(min(x_η) ≤ max(iterator.current())) RETURN
28 FOR (y_i ∈ y s.t. max(y_i) = checker) x_α.setMax(max(y_i) − 1), y_i.setMin(min(x_α) + 1)
29 Procedure CleanUp()
30 FOR (y_i ∈ y)
31    IF (max(y_i) ≤ MaxPrune) y_i := max(y_i) ELSE y_i.setMin(MaxPrune + 1)
```

6.2 A Worked Example

Consider applying **GACAP** to the pair: $\boldsymbol{x} = \langle\{2,3\},\{3,4\},\{4,5\},\{6\}\rangle$ and $\boldsymbol{y} = \langle\{4,5\},\{1,2,3\},\{1,2\},\{4,5\}\rangle$ (where the domains of constituent variables are written as sets). A bucket sort of $\boldsymbol{y}$ is performed first according to the upper bound of each variable. This permutation, denoted $\boldsymbol{p}$, is used hereafter for clarity. **Stage1** traverses the vectors, using the pointer α to remove from consideration bound equal pairs. Initially, $\alpha = 1$. *MaxPrune*, incrementally set to the maximum of its current value and $\min(x_\alpha) - 1$, records the maximum extent to which domain elements can be removed from members of $\boldsymbol{y}$. This pruning is done in post-processing by **CleanUp** for efficiency reasons (see below). Line 11 sets $\max(x_1)$ to the smallest non-empty bucket, as justified by Lemma 1:

$$\begin{array}{rcl} & & \alpha \\ \boldsymbol{x} & = & \langle\{2,\not{3}\},\ \{3,4\},\ \{4,5\},\ \{6\}\rangle \\ \boldsymbol{p} & = & \langle\{1,2\},\ \{1,2,3\},\ \{4,5\},\ \{4,5\}\rangle \\ MaxPrune & := & 1 \end{array}$$

x_1 and p_1 form a bound equal pair (ignoring elements of p_1 that are removed by **CleanUp**) which is discarded by incrementing α. The same process is repeated:

$$\begin{array}{rcl} & & \alpha \\ \boldsymbol{x} & = & \langle\{2,\not{3}\},\ \{3,\not{4}\},\ \{4,5\},\ \{6\}\rangle \\ \boldsymbol{p} & = & \langle\{1,2\},\ \{1,2,3\},\ \{4,5\},\ \{4,5\}\rangle \\ MaxPrune & := & 2 \end{array}$$

x_2 and p_2 form another bound equal pair. Since no pruning of x_3 is possible and x_3 is not bound, $\boldsymbol{x}$ and $\boldsymbol{p}$ are Stage 1 Complete. η**Pruning** now prunes $\max(x_1)$

and $\min(p_i)$ for $i \geq \alpha$, as necessary. The pointer η is used to traverse the tail of $\boldsymbol{x}$ and $\boldsymbol{p}$ to test if they can be made consistent.

$$\begin{array}{rcl} & & \alpha \quad \eta \\ \boldsymbol{x} & = & \langle \{2,3\}, \ \{3,4\}, \ \{4,5\}, \ \{6\} \rangle \\ \boldsymbol{p} & = & \langle \{1,2\}, \ \{1,2,3\}, \ \{4,5\}, \ \{4,5\} \rangle \\ MaxPrune & := & 3 \end{array}$$

If the tail is inconsistent (as here because $\max(p_{\eta-1}) < \min(x_\eta)$) we know how to prune all elements except $p_{\eta-1}$. To check this last variable we remove $p_{\eta-1}$ from $\boldsymbol{p}$ and continue checking for consistency. Finally **CleanUp** traverses $\boldsymbol{p}$ and performs pruning with the value of MaxPrune.

$$\begin{array}{rcl} & & \alpha \quad \eta \\ \boldsymbol{x} & = & \langle \{2,3\}, \ \{3,4\}, \ \{4,5\}, \ \{6\} \rangle \\ \boldsymbol{p} & = & \langle \{1,2\}, \ \{1,2,3\}, \ \{4,5\}, \ \{4,5\} \rangle \\ MaxPrune & := & 3 \end{array}$$

6.3 Properties

Theorem 8. *The* **GACAP** *algorithm on two vectors of variables* $\boldsymbol{x}$ *and* $\boldsymbol{y}$ *of length* n *runs in time* $O(n+d)$ *where* d *is the maximum size of the domain of the variables in* $\boldsymbol{x}$ *and* $\boldsymbol{y}$*.*

Proof. (sketch). Sorting $\boldsymbol{y}$ into buckets takes $O(n)$. We traverse $\boldsymbol{x}$ and $\boldsymbol{y}$ examining and/or pruning each element of $\boldsymbol{x}$ and $\boldsymbol{y}$ at most once and also examining (at worst) each of the d buckets, which takes time $O(n+d)$. During **CleanUp** we may prune each element of $\boldsymbol{y}$ once, which is $O(n)$. □

If $d \gg n$ then we use an $O(n \log n)$ algorithm. Instead of sorting $\boldsymbol{y}$ into buckets we sort $\boldsymbol{y}$ from smallest to largest using the upper bound of each domain (in $O(n \log n)$). This alternative algorithm is identical to **GACAP** except instead of iterating through buckets it iterates through this sorted vector.

7 Extensions

The methods used to generate *allperm* from the row-wise lex-leader constraints can also be used to generate useful symmetry-breaking constraints for other symmetry groups. In Molnar's problem (see Section 8.1) we use the result that when a matrix has transpose as well as row and column symmetry then as well as the lex^2 and *allperm* constraints we can also impose consistently $\forall i$: row 1 $\leq_{ap}$ col i. In a similar example, given an $n \times n$ square matrix with rotation as well as row and column symmetry we can impose the extra symmetry breaking constraints row 1 $\leq_{ap}$ col 1 and row 1 $\leq_{ap}$ col n as well as the usual lex^2 and *allperm*.

Index symmetries in matrices of more than two dimensions can be broken by a generalisation of lex^2 and $allperm$, the latter of which can be implemented by using the **GACAllPerm** algorithm. The *allperm* constraint also can be extended and adapted to the case where only some rows/columns are interchangeable. Finally we note that the **GACAllPerm** algorithm can be extended straightforwardly to cope with vectors of different length.

8 Experimental Results

We tested *allperm* on two benchmark problems in number theory and coding theory. Lack of space prevents the consideration of more domains, but further applications, such as Howell designs [1], difference matrices [13], and balanced generalised weighing matrices [12] can readily be found.

We used ILOG Solver 5.3 with the GAC lexicographic algorithm in [7], an anti-multiset algorithm adapted from the GAC multiset algorithm in [8] (see Section 5.3), and the **GACAllPerm** algorithm from Section 6. lex^2 is approximated using lexicographic ordering constraints on adjacent rows and columns. Recently, it has become possible to enforce GAC on a chain of lexicographic ordering constraints [2]. Currently, the effectiveness of this new constraint is unknown. If proven effective it could be used to improve all models tested here.

8.1 Molnar's Problem

Molnar [11] posed the following problem. Given k, construct two $k \times k$ matrices, M_1 and M_2, of integers such that the determinant of M_1 is one, the determinant of M_2 is ± 1, no entry in M_1 or M_2 is ± 1, and each entry in M_2 is the square of the corresponding entry in M_1.

The solutions to this problem are significant in classifying certain types of topological spaces. Guy [9, Section F28] discusses a variant where 0 entries are also disallowed and both determinants must be 1. Guy's variant is studied here as follows. Given k and an upper bound, u, all solutions are sought (since all solutions are interesting mathematically) where each variable in M_1 is assigned a value between 2 and u. Seeking solutions with positive entries only simplifies what is a difficult problem.

Exchanging two rows or columns of a matrix negates the determinant. Hence, this problem does not have row or column symmetry. However, by relaxing the problem such that the absolute value of the determinant is 1, we *can* apply row and column symmetry breaking. If a solution has a determinant of -1, we exchange a pair of rows/columns. The problem also has transpose symmetry which we will exploit.

We modelled this problem with matrices M_1 and M_2 of decision variables, where each entry of M_2 is constrained to be equal to the square of the corresponding entry of M_1. The determinant of each matrix is expressed as a single, large-arity constraint and bound to a variable $detVar$ with domain $\{-1, 1\}$. We considered three models of symmetry breaking:

Model A $lex^2(M_1)$. Since the variables in the bottom row appear most frequently in the definition of the determinant, lex^2 is applied such that this row is least in the lexicographic ordering to agree with the variable ordering described below, ensuring that the column lex constraints are consistent with this choice. Given the transpose symmetry, we also constrain the bottom row to be lexicographically less than or equal to all columns, as discussed in Section 7.

Model B $lex^2(M_1) \wedge ams(M_1)$. In addition, $\leq_{\overline{m}}$ is used to break the transpose symmetry.

Problem	Model A			Model B			Model C		
order, u	Choices	Time	Solutions	Choices	Time	Solutions	Choices	Time	Solutions
3, 3	14	0	-	12	0	-	12	0	-
3, 4	138	0.13	-	114	0.13	-	109	0.13	-
3, 5	742	1.9	-	591	1.6	-	558	1.6	-
3, 6	2,872	18.9	-	2,210	13.8	-	2,067	13.3	-
3, 7	8,695	134	-	6,559	88.4	-	6,172	83.8	-
3, 8	22,948	756	-	17,016	466	-	16,068	440	-
3, 9	53,103	3,470	5	38,979	2,050	3	36,851	1,921	3
3, 10	113,138	13,600	6	82,302	7,800	4	77,793	7,310	4
3, 11	219,383	47,300	7	158,762	26,200	5	150,445	24,800	5
4, 3	139	0.4	-	101	0.4	-	101	0.4	-
4, 4	14,783	155	-	9,267	102	-	8,885	102	-
4, 5	499,836	23,400	19	28,3521	12,500	13	269,373	12,000	13

Table 1. 256Mb PentiumIII 750MHz. Times in seconds given to 3 significant figures.

Model C $lex^2(M_1) \wedge$ **GACAllPerm**(M_1). **GACAP** is used to break the transpose symmetry.

Experiments are on order 3 and 4 matrices. The variable ordering is by row on M_1, starting at the bottom (most constrained) row. Table 1 summarises the results, which show a clear benefit to using *allperm* to break more symmetry even on the relatively small matrices experimented with here. Furthermore, the improvement increases both with the size of the matrix and the size of the domains. Using **GACAllPerm** provides a ten percent reduction in search effort over anti-multiset on these problems. Since both are linear-time algorithms, there is no reason not to use **GACAllPerm**.

8.2 Error-correcting Codes

Given an alphabet F, a fixed-length code of a length n is a set C of strings from F^n. Given a code C and a distance function $d(x, y)$ where x and y are elements of C we define the minimum distance of C as the minimum of the distances between all distinct pairs of elements of C.

If $F = \{0, 1, 2, 3\}$ then a commonly used distance function is the *Lee distance*, which is useful in various areas of coding [10] and also indirectly in other areas, such as packing [3]. For two elements, x, y in a code C of length n the Lee distance is defined as $d(x, y) = \sum_{i=1}^{n} min\{|x_i - y_i|, 4 - |x_i - y_i|\}$. This is what we shall use in our experiments.

Following standard practice we represent the code as an $n \times |C|$ matrix, where each row contains one element of the code. The difference constraint is enforced between pairs of rows. This matrix has row and column symmetry: swapping rows simply changes the order in which the elements of the code are represented in the matrix, and the Lee distance is unaffected by swapping columns.

Codes defined over the Lee distance also have value symmetry. Given a code represented by a matrix as defined above, if we take any column and change each value in it by the mapping $(0 \rightarrow 1, 1 \rightarrow 2, 2 \rightarrow 3, 3 \rightarrow 0)$ then the resulting code has the same Lee distance between each pair of rows. Some of this symmetry can be broken by assuming any code has an element consisting of just zeroes. We do this in the experiments below and not not include this element in the matrix

Length	Size	Min. Dist.	Model A choices	Model A time	Model B choices	Model B time	Model C choices	Model C time
5	6	6	14,931	1.0	13,043	1.0	12,687	1.0
5	7	5	96,942	9.0	40,066	4.1	34,605	3.5
5	8	5	118,712	12.6	57,219	6.6	51,287	6.0
5	9	5	11,311,563	1,230	3,040,390	341	3,027,982	329
6	4	8	1,225	0.1	1,081	0.1	1,080	0.1
6	5	6	1,227,456	80.8	276,525	20.2	237,005	17.7
6	6	6	1,374,943	105	338,689	30.6	286,010	26.8
6	7	6	1,626,743	143	484,923	51.8	418,624	46.5
6	8	6	2,007,190	204	763,918	94.9	687,130	88.4
6	9	6	2,754,911	320	1,409,570	196	1,322,412	189
6	10	6	3,578,967	489	2,130,763	338	2,033,355	329
6	10	6	4,718,395	738	3,166,724	561	3,058,971	531

Table 2. 128Mb PIII 1Ghz. Times given in seconds to 3 significant figures

of variables we use to solve the problem. In conjunction with this basic model three models of symmetry breaking were tried.

Model A $lex^2(M)$ is enforced using lexicographic ordering constraints on adjacent rows and columns.

Model B $lex^2(M) \wedge ams(M)$.

Model C $lex^2(M) \wedge \mathbf{GACAllPerm}(M)$.

The top-most row and left-first column were constrained to be the smallest under the lexicographic order and the *ams* and *allperm* constraints were placed on the rows. The variables were instantiated in row-wise order starting from the top-most row and the variable ordering used was $(0, 1, 2, 3)$.

The experiment performed was to try to find the code with the largest minimum Lee distance given the length and size of the code.

Table 2 summarises the results. Like Molnar's problem these show that applying extra symmetry breaking constraints to lex^2 can result an improvement of over a 50% reduction in both time and number of fails and the magnitude of these improvements increases with the size of the search. There is a small but measurable difference between models B and C.

9 Future Directions

The question now is whether further constraints can be added to break more symmetry effectively than lex^2 plus *allperm*. There are some obvious possibilities, such as attempting to create an $allperm^2$. However, we have explored and rejected as inconsistent all of the possibilities that seemed straightforward to us. It may be possible to look more deeply at the row-wise lex leader constraints to obtain further symmetry breaking constraints or to consider a different lex-leader from which a stronger set of symmetry breaking constraints might be derived.

Another item of future work is to consider the combination of other types of symmetry with row and column symmetry, such as the transpose symmetry seen in our model for Molnar's problem. The interaction of *allperm* with different variable/value ordering heuristics should also be explored. Our preliminary experimentation suggests that choosing a variable/value ordering that does not conflict with *allperm* is not as straightforward as for lex^2 alone.

10 Conclusion

We have identified an extension to the highly successful lex^2 method of breaking row and column symmetries. In some cases, substantial reductions in search effort are achieved. Furthermore, the efficient implementations here incur negligible overhead. This suggests that when a problem is being modelled using lex^2, where applicable it is always worthwhile adding *allperm*.

Acknowledgements. This research is supported by UK-EPSRC grant number GR/N16129. We thank Warwick Harvey, Brahim Hnich, Zeynep Kiziltan, Toby Walsh and our anonymous reviewers.

References

1. P.J. Schellenberg B.A. Anderson and D.R. Stinson. The existance of Howell designs of even side. In *J. Combin. Theory*, pages 23–55, 1984.
2. M. Carlsson and N. Beldiceanu. Arc-consistency for a chain of lexicographic ordering constraints. Technical Report T2002-18, Swedish Institute of Computer Science, 2002.
3. J.H. Conway and N.J.A. Slone. *Sphere-Packings, Lattices and Groups, 3rd ed.* Springer-Verlag, 1992.
4. James Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proceedings of KR'96: Principles of Knowledge Representation and Reasoning*, pages 148–159, 1996.
5. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, pages 462–476, 2002.
6. P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Matrix modelling: Exploiting common patterns in constraint programming. In *Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems*, pages 27–41, 2002.
7. A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, pages 93–108, 2002.
8. A.M. Frisch, I. Miguel, , Z. Kiziltan, B. Hnich, and T. Walsh. Multiset ordering constraints. In *Proceedings of the 18th International Joint Conference on AI*, 2003.
9. R.K. Guy. *Unsolved Problems in Number Theory.* Springer-Verlag, 1994.
10. A.R. Hammons, P.V. Kumar, N.J.A. Slone, and P.Solé. The $\mathbb{Z}_4$-linearity of kerdock, goethals and related codes. *IEEE Trans. Information Theory*, 1993.
11. A.E. Molnar. A matrix problem. *American Mathematical Monthly 81*, pages 383–384, 1974.
12. R.C. Mullin and R.G. Stanton. Group matrices and balanced weighing designs. In *Utilitas Math. 8*, pages 277–301, 1975.
13. G.H.J. van Rees P.J. Schellenberg and S.A. Vanstone. Four pairwise orthogonal latin squares of order 15. In *Ars Combin. 6*, pages 141–150, 1978.
14. Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 2002.