Multiset Ordering Constraints*

Alan Frisch and **Ian Miguel**
Dept of Computer Science
University of York
England
{frisch, ianm}@cs.york.ac.uk

Zeynep Kiziltan
Dept of Information Science
Uppsala University
Sweden
Zeynep.Kiziltan@dis.uu.se

Brahim Hnich and **Toby Walsh**
Cork Constraint Computation Centre
University College Cork
Ireland
{brahim,tw}@4c.ucc.ie

Abstract

We identify a new and important global (or non-binary) constraint which ensures that the values taken by two vectors of variables, when viewed as multisets, are ordered. This constraint is useful for a number of different applications including breaking symmetry and fuzzy constraint satisfaction. We propose and implement a linear time algorithm for enforcing generalised arc-consistency on such a multiset ordering constraint. Experimental results show considerable promise.

1 Introduction

Global (or non-binary) constraints are one of the factors central to the success of constraint programming [Régin, 1994; 1996; Beldiceanu, 2000]. Global constraints specify patterns that occur in many problems, and call efficient and effective constraint propagation algorithms. In this paper, we identify a new and important global constraint. This constraint ensures that the values taken by two vectors of variables, when viewed as multisets, are ordered. Such a constraint is useful in a number of domains. For example, in the progressive party problem (prob013 in csplib.org), we wish to assign a host for each guest and period. We can model this with a vector of variables for each period. Each variable is assigned the host for a particular guest. This model has unnecessary symmetry as the periods can be freely permuted. We can break this symmetry by considering the multiset of values associated with each vector and ordering these multisets.The aim of this paper is to study such multiset ordering constraints and to develop efficient and effective techniques for enforcing them.

2 Formal Background

A constraint satisfaction problem (CSP) consists of a set of variables, each with a finite domain of values, and a set of constraints that specify allowed values for subsets of variables. A solution is an assignment of values to the variables satisfying the constraints. To find such solutions, we explore partial assignments enforcing a local consistency like generalized arc-consistency (GAC). A constraint is GAC iff, when a variable in the constraint is assigned a value v, compatible values exist for all the other variables in the constraint. Such compatible values are referred as the support for v. GAC reduces to node-consistency (NC) for unary constraints. Finally, a constraint is bounds consistent (BC) iff, when a variable in the constraint is assigned its maximum or minimum value, there exist compatible values for all the other variables in the constraint. If a constraint c is NC, BC, or GAC then we write NC(c), BC(c), or GAC(c) respectively.

A multiset is an unordered list in which repetition is allowed. A multiset M is ordered smaller than another N, written $M \prec_{\rm m} N$ iff either M is empty and N is not, or the largest value in M is smaller than the largest value in N, or the largest values are the same and, if we eliminate one occurrence of the largest value from both M and N, the resulting two multisets are ordered. Note that $M \preceq_{\rm m} N$ iff $M = N$ or $M \prec_{\rm m} N$. We will often view the values taken by a vector of variables as a multiset. Given two vectors of variables, $\vec{x}$ and $\vec{y}$, we write a multiset ordering constraint as $\vec{x} \leq_{\rm m} \vec{y}$. This constraint is satisfied iff the values for the variables in the vectors, when treated as multisets, satisfy the multiset ordering. Similarly, we write a strict multiset ordering constraint as $\vec{x} <_{\rm m} \vec{y}$. Throughout the paper, we assume that vectors of variables have finite integer domains, and the variables are disjoint and not repeated.

Vectors of variables are indexed from 0. The minimum element in the domain of x_i is $min(x_i)$, and the maximum is $max(x_i)$. The function *floor*($\vec{x}$) assigns all variables in $\vec{x}$ to their minimum values, whilst *ceil*($\vec{x}$) assigns all to their maximums. The vector $\vec{x}_{v=d}$ is identical to $\vec{x}$ except v now has the domain $\{d\}$. An occurrence vector $occ(\vec{x})$ associated with $\vec{x}$ is indexed in decreasing order from the maximum $max(\vec{x})$ to the minimum $min(\vec{x})$ value from the domains in $\vec{x}$. The ith element of $occ(\vec{x})$ is the number of occurrences of $max(\vec{x}) - i$ in $\vec{x}$. When comparing two occurrence vectors, we assume they start and end with the occurrence of the same value, adding leading/ trailing zeroes as necessary. Finally, $\vec{x} \leq_{lex} \vec{y}$ iff $\vec{x}$ is lexicographically less than or equal to $\vec{y}$.

3 Motivating applications

3.1 Matrix symmetry

Many constraints programs contain matrices of decision variables (so called "matrix models"), and the rows and/or

*Support received by EPSRC under GR/R30792 and by the Science Foundation Ireland. We thank Chris Beck and Chris Jefferson.

columns of these matrices are symmetric and can be permuted [Flener *et al.*, 2002]. Such symmetries are very difficult to deal with as there are a super-exponential number of permutations of the rows or columns to consider. There are several ways to break symmetry in a CSP, such as SBDS [Gent and Smith, 2000] or SBDD [Fahle *et al.*, 2001]. One of the most effective, and the one which we will concentrate on as a major application for a multiset ordering constraint, is adding extra symmetry-breaking constraints to an initial model. Existing techniques for dealing with such symmetries typically eliminate only some of the symmetry. Additional techniques, like those proposed here, are therefore of considerable value.

The progressive party problem mentioned earlier has a 2d matrix of decision variables with matrix symmetry. The rows of the matrix are the guests, the columns are the periods. Each variable gives the host assigned to a given guest in a given period. As periods are indistinguishable, the columns of the matrix are symmetric. One way to break such column symmetry is to lex order the columns [Frisch *et al.*, 2002]. Similarly, as guests can be indistinguishable, (some of) the rows may be symmetric and can be lex ordered. Alternatively, we can treat each row and/or column as a multiset and break such symmetry by multiset ordering the rows and/or columns.

Unlike lex ordering, multiset ordering the rows of a matrix may not eliminate all row symmetry. For example, consider the symmetric matrices:

$$\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$$

Both satisfy the constraint that the first row is multiset less than the second. It is therefore a little surprising to discover that multiset ordering (which does not break all row symmetry) is not dominated by lex ordering (which does) but is incomparable. For example, $\langle 0, 2\rangle \leq_{lex} \langle 1, 1\rangle$ but $\langle 1, 1\rangle \leq_m \langle 0, 2\rangle$.

When we have both row and column symmetry, we can multiset order both rows and columns. Like lex ordering both rows and columns, this may not eliminate all row and column symmetry. Consider the symmetric matrices:

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

Both have multiset ordered rows and columns. Unsurprisingly, multiset ordering rows and columns is incomparable to lex ordering rows and columns. Consider the symmetric matrices:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

The first has lex ordered rows and columns, but the columns are not multiset ordered. The second has rows and columns that are multiset ordered but the columns are not lex ordered.

An alternative way to deal with row and column symmetry is to multiset order in one dimension and apply the symmetry breaking method of our choice in the other dimension. This is one of the best features of using multiset ordering to break symmetry. It is compatible with any other method in the other dimension. For instance, we can multiset order the rows and lex order the columns. Preliminary results in [Kiziltan and Smith, 2002] suggest that such a combined method is very promising. This combined method does not eliminate all symmetry (but it is unlikely that any polynomial set of constraints does). Consider the symmetric matrices:

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 2 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix}$$

Both have rows that are multiset ordered, and rows and columns that are lex ordered. Multiset ordering the rows and lex ordering the columns is again incomparable to lex ordering rows and columns. Consider the symmetric matrices:

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 2 & 0 \\ 0 & 2 & 2 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 2 & 2 \\ 1 & 0 & 0 \\ 1 & 0 & 2 \end{pmatrix}$$

The first matrix has rows that are multiset ordered and columns that are lex ordered. However, its rows are not lex ordered. The second matrix has rows and columns that are lex ordered but does not have rows that are multiset ordered. Whilst the two orderings are theoretically incomparable, our experimental results (see later) show that multiset ordering the rows and lex ordering the columns is often the most effective symmetry breaking constraint currently known.

3.2 Fuzzy constraints

Another application for multiset ordering is to fuzzy CSPs. A fuzzy constraint associates a degree of satisfaction to an assignment tuple for the variables it constrains. To combine degrees of satisfaction, we can use a combination operator like the minimum function. Unfortunately, the minimum function may cause a *drowning effect* when one poorly satisfied constraint 'drowns' many highly satisfied constraints. One solution is to collect a vector of degrees of satisfaction, sort these values in ascending order and compare them lexicographically. This *leximin* combination operator identifies the assignment that violates the fewest constraints [Fargier, 1994]. This induces an ordering identical to the multiset ordering except that the lower elements of the satisfaction scale are the more significant. It is simple to modify a multiset ordering constraint to consider the values in a reverse order. To solve such leximin fuzzy CSPs, we can then use branch and bound, adding an ordering constraint when we find a solution to ensure that future solutions are greater in the leximin ordering.

4 GAC algorithm for multiset ordering

The last section motivated why we want multiset ordering constraints. We need, however, to be able to propagate such constraints efficiently. We therefore developed an efficient GAC algorithm for such constraints.

4.1 Background

The algorithm exploits two theoretical results. The first reduces the problem to testing support for upper bounds of $\vec{x}$ and lower bounds of $\vec{y}$ on suitable ground vectors. The second reduces these tests to lex ordering suitable occurrence vectors. Identical results hold for the strict multiset ordering constraint but for reasons of space we omit them here.

Lemma 1 *Given two disjoint and non-repeating vectors of variables, $\vec{x}$ and $\vec{y}$, with non-empty domains, $GAC(\vec{x} \leq_m \vec{y})$ iff $\forall x_i \in \vec{x}, \forall y_j \in \vec{y}$:*

$$\begin{aligned} \mathit{floor}(\vec{x}_{x_i=max(x_i)}) &\leq_m \mathit{ceil}(\vec{y}) \\ \mathit{floor}(\vec{x}) &\leq_m \mathit{ceil}(\vec{y}_{y_j=min(y_j)}) \end{aligned}$$

Proof: ($\Rightarrow$) As the constraint is GAC, all values have support. In particular, $x_i = max(x_i)$ has support. The best support comes if all the other variables in $\vec{x}$ take their minimums, and all the variables in $\vec{y}$ take their maximums. Hence, *floor*$(\vec{x}_{x_i=max(x_i)}) \leq_m$ *ceil*$(\vec{y})$. Similarly, for y_i.

($\Leftarrow$) The first constraint ensures that $max(x_i)$ is supported. The values which support $max(x_i)$ also support all smaller values. Hence, all the values in the domain of x_i are supported. By an analogous argument, all the values in the domain of y_i are supported. Hence the constraint is GAC. QED.

The next lemma reduces these tests for support to lex ordering suitable occurrence vectors.

Lemma 2 *Given two multisets of values, M and N, $M \leq_m N$ iff occ$(M) \leq_{lex}$ occ(N).*

Proof: See [Kiziltan and Walsh, 2002].

4.2 A worked example

Based on these lemmas, we have designed an efficient algorithm for enforcing GAC on a multiset ordering constraint. The algorithm goes through the x_i and y_j checking for support in the appropriate occurrence vectors. Incremental computation of the lex ordering test avoids repetition of work. Consider the multiset ordering constraint $\vec{x} \leq_m \vec{y}$ where:

$$\vec{x} = \langle\{5\},\{4,5\},\{3,4,5\},\{2,4\},\{1\},\{1\}\rangle$$
$$\vec{y} = \langle\{4,5\},\{4\},\{1,2,3,4\},\{2,3\},\{1\},\{0\}\rangle$$

We construct occurrence vectors for *floor*$(\vec{x})$ and *ceil*$(\vec{y})$, indexed from 5 to 0:

$$occ(floor(\vec{x})) = \langle 1,1,1,1,2,0\rangle$$
$$occ(ceil(\vec{y})) = \langle 1,2,1,0,1,1\rangle$$

Recall that index i in *occ*(*ceil*$(\vec{y})$) denotes the number of occurrences of the value i in *ceil*$(\vec{y})$. For example, index 4 is 2 as the value 4 occurs twice.

We first check if *occ*(*floor*$(\vec{x})$) $>_{lex}$ *occ*(*ceil*$(\vec{y})$). If so, we can fail immediately because no value for any variable can have support. Here, *occ*(*floor*$(\vec{x})$) $\leq_{lex}$ *occ*(*ceil*$(\vec{y})$). In fact, we record (in a pointer, α) that the two occurrence vectors are lex ordered by index 4 of *occ*(*floor*$(\vec{x})$), which is strictly smaller than index 4 of *occ*(*ceil*$(\vec{y})$). This means that we will fail to find support in the y_j if any of the x_i is assigned a new value greater than 4. We now go through the x_i checking for support for their maximum values, and then the y_i checking for support for their minimum values.

Consider x_0. As it has a singleton domain, and *occ*(*floor*$(\vec{x})$) $\leq_{lex}$ *occ*(*ceil*$(\vec{y})$), its only value must have support so we skip it. Now consider x_1. Do its values have support? Changing *occ*(*floor*$(\vec{x})$) to $occ(\langle min(x_0), max(x_1), \ldots, min(x_{n-1})\rangle)$ decreases the number of occurrences of $min(x_1) = 4$ by 1, and increases the number of occurrences of $max(x_1) = 5$ by 1. As $min(x_1) \geq \alpha = 4$, this upsets the lex ordering of the two occurrence vectors. We therefore prune all values in the domain of x_1 larger than α. This leaves a single supported value, 4.

Now consider x_2. Changing *occ*(*floor*$(\vec{x})$) to $occ(\langle min(x_0), \ldots, max(x_2), \ldots, min(x_{n-1})\rangle)$ decreases the number of occurrences of $min(x_2) = 3$ by 1, and increases the number of occurrences of $max(x_2) = 5$ by 1. As with x_1, any value of x_2 larger than $\alpha = 4$ upsets the lex ordering. We therefore prune 5 from the domain of x_2. Now consider x_3. Changing *occ*(*floor*$(\vec{x})$) to $occ(\langle min(x_0), \ldots, max(x_3), \ldots, min(x_{n-1})\rangle)$ decreases the number of occurrences of $min(x_3) = 2$ by 1, and increases the number of occurrences of $max(x_3) = 4$ by 1. The occurrence vectors beneath α would now be lex ordered the wrong way. We therefore also prune the value $\alpha = 4$, leaving a single supported value 2 in the domain of x_3. As x_4 and x_5 have singleton domains, their values have support.

Similarly, we check the minimums of the y_j for support. However, rather than prune values above (and in some cases equal to) α, there is now a dual pointer β and we prune values in the domains of y_j up to (and in some cases equal to) β. The pointer β is the largest index such that the occurrence vectors beneath it are lex ordered the wrong way. Any value less than β cannot hope to change the lex ordering as the value at β will still order the vectors the wrong way. Such values can therefore be pruned. Once we have considered each of the y_j, we have the following generalized arc-consistent vectors:

$$\vec{x} = \langle\{5\},\{4\},\{3,4\},\{2\},\{1\},\{1\}\rangle$$
$$\vec{y} = \langle\{5\},\{4\},\{3,4\},\{2,3\},\{1\},\{0\}\rangle$$

4.3 Algorithm details

The algorithm uses two pointers α and β, and two flags γ and δ to avoid traversing the occurrence vectors each time we look for support. The pointer α is set to to the most significant index above which all occurrences are pair-wise equal and at α they are strictly ordered. If the vectors are equal then α is set to $-\infty$. The pointer β is set to the most significant index below α such that the occurrence vectors are lex ordered the wrong way. If no such index exists, we set β to $-\infty$. The flag γ is set to true if all the indices between α and β are pair-wise equal and the flag δ is set to true if the sub-vectors below β are lex ordered the wrong way. For example, given the occurrence vectors in section 4.2, α is set to 4, β to 2, and the flags γ and δ are set to true.

We summarise the major steps the algorithm performs:

A. Build *occ*(*floor*$(\vec{x})$) and *occ*(*ceil*$(\vec{y})$)

B. Set α, β, γ, δ according to their definitions

C. For each x_i: If its maximum disturbs the lex ordering on the occurrence vectors, tighten its upper-bound to α when the occurrence vectors are lex ordered below α, otherwise to $\alpha - 1$.

D. For each y_i: If its minimum disturbs the lex ordering on the occurrence vectors, then tighten its lower-bound to β when the occurrence vectors are lex ordered below α, otherwise to $\beta + 1$.

When we prune a value, we do not need to check recursively for previous support. Pruning changes neither the lower bounds of $\vec{x}$ nor the upper bounds of $\vec{y}$. These values continue to provide support. The exception is when a domain is a singleton, and pruning causes a domain wipe-out.

We now give pseudo-code for an algorithm that maintains GAC on a multiset ordering constraint between vectors $\vec{x}$ and $\vec{y}$ which are of length n and m respectively. As the algorithm reasons about occurrences vectors, the original vectors need not be identical in length (though they often are).

The algorithm is called whenever lower bounds of x_i or upper bounds of y_j change. Lines **A1** to **A3** build the occurrence vectors $\vec{ox}$ and $\vec{oy}$. Line **B1** calls the procedure to set the pointers α and β, and the flags γ and δ. Lines **C1-13** check support for the maximums of the x_i's while lines **D1-14** check support for the minimums of the y_i's.

Procedure GACMSO()

A1. $l := min(\{min(x_i)|i \in [0,n)\} \cup \{min(y_j)|j \in [0,m)\})$
A2. $u := max(\{max(x_i)|i \in [0,n)\} \cup \{max(y_j)|j \in [0,m)\})$
A3. $\vec{ox} = occ(floor(\vec{x})), \vec{oy} = occ(ceil(\vec{y}))$
B1. **SetPointers-and-Flags**(l, u)
C1. **FOR** $i = 0$ **TO** $n-1$ **DO** % check support for x's
C2. **IF** $(min(x_i) \neq max(x_i))$ **THEN**
C3. $a := min(x_i); b := max(x_i);$
C4. **IF** $(a \geq \alpha)$ **THEN** $NC(x_i = a)$;
C5. **IF** $(\alpha > a > \beta \wedge b \geq \alpha)$ **THEN** $NC(x_i \leq \alpha)$;
C6. **IF** $(a = \beta \wedge b \geq \alpha)$ **THEN**
C7. **IF** $(ox_\alpha + 1 = oy_\alpha)$ **THEN**
C8. **IF** $(\gamma \wedge ox_\beta - 1 = oy_\beta \wedge \delta)$ **THEN** $NC(x_i < \alpha)$;
C9. **ELSE** $NC(x_i \leq \alpha)$;
C10. **IF** $(a < \beta \wedge b \geq \alpha)$ **THEN**
C11. **IF** $(ox_\alpha + 1 = oy_\alpha)$**THEN**
C12. **IF** (γ) **THEN** $NC(x_i < \alpha)$;
C13. **ELSE** $NC(x_i \leq \alpha)$;
D1. **FOR** $j = 0$ **TO** $m-1$ **DO** % check support for y's
D2. **IF** $(min(y_j) \neq max(y_j))$ **THEN**
D3. $a := min(y_j); b := max(y_j)$
D4. **IF** $(b > \alpha)$ **THEN** $NC(y_j = b)$;
D5. **IF** $(b = \alpha \wedge ox_\alpha + 1 = oy_\alpha)$ **THEN**
D6. **IF**$(a \leq \beta \wedge \gamma)$ **THEN**
D7. **IF** $(ox_\beta = oy_\beta + 1)$ **THEN**
D8. **IF** (δ) **THEN** $NC(y_j > \beta)$;
D9. **ELSE** $NC(y_j \geq \beta)$;
D10. **ELSE** $NC(y_j > \beta)$;

Procedure SetPointers-and-Flags(l, u)

B1. γ := *false*; δ := *false*; $\alpha := u$;
B2. **WHILE** $(\alpha \geq l \wedge ox_\alpha = oy_\alpha)$ **DO** $\alpha := \alpha - 1$;
B3. **IF** $(\alpha \geq l \wedge ox_\alpha > oy_\alpha)$ **THEN FAIL**;
B4. **IF** $(\alpha = l - 1)$ **THEN** $\alpha := -\infty; \beta := -\infty$;
B5. **ELSE**
B6. $\beta := \alpha - 1; \gamma :=$ *true*;
B7. **WHILE** $(\beta \geq l \wedge ox_\beta \leq oy_\beta)$ **DO**
B8. **IF** $(ox_\beta < oy_\beta)$ **THEN** $\gamma :=$ *false*;
B9. $\beta := \beta - 1$;
B10. **IF** $(\beta = l - 1)$ **THEN** $\beta := -\infty; \gamma :=$ *false*;
B11. **IF** $(\beta \neq -\infty \wedge \beta > l)$ **THEN**
B12. $i := \beta - 1$;
B13. **WHILE** $(i \geq l \wedge ox_i = oy_i)$ **DO** $i := i - 1$;
B14. **IF** $(i \geq l \wedge ox_i > oy_i)$ **THEN** $\delta :=$ *true*;

For each x_i, we only check for support if x_i is not singleton (line **C2**). There are six cases where we prune the domain of x_i: (1) at line **C4**, as $min(x_i) \geq \alpha$, any value in the domain of x_i greater than $min(x_i)$ lacks support because it will disturb the lex ordering; (2) at line **C5**, as $\alpha > min(x_i) > \beta$, and $max(x_i) \geq \alpha$, all the values in the domain of x_i greater than α disturb the lex ordering, and lack support; (3) at line **C6**, **C7**, and **C8**, all values greater than or equal to α lack support. If we assign α to x_i, then the vectors will be equal at index α and β, the values between α and β are also all pair-wise equal (since γ is true), and the vectors below β are ordered the wrong way (since δ is true). Thus, the value α also lacks support and is pruned; (4) at line **C9**, all values greater than α lack support; (5) at line **C10**, **C11**, and **C12**, all values greater than or equal to α lack support. If we assign α to x_i, the vectors will be lex ordered the wrong way as the difference between the number of occurrences at α is exactly one, and γ is true. Thus, the value α also lacks support and is pruned; (6) at line **C13** all values greater than α lack support.

For each y_j, we only check for support if y_j is not singleton (line **D2**). There are four cases where we prune the domain of y_j: (1) at line **D4**, as $max(y_j) > \alpha$, any value smaller than $max(y_j)$ disturbs the lex ordering and lacks support; (2) at lines **D5** to **D8**, the situation is dual to the third case for the x_i, and any value less than or equal to β lacks support; (3) at line **D9**, any value less than β lacks support; (4) at line **D10**, again, any value less than or equal to β lacks support.

4.4 Theoretical properties

In a longer technical report, we prove the following results about the algorithm's complexity and correctness.

Theorem 1 **GACMSO** *runs in time* $O(n + m + d)$ *where* d *is* $u - l$.

If $d \ll n, m$ (and for multisets, we expect this as the number of values is typically less than the cardinalities to permit repetition), then the algorithm is $O(n + m)$.

Theorem 2 *For disjoint and non-repeating vectors,* **GACMSO** *either establishes failure if* $\vec{x} \leq_m \vec{y}$ *is not satisfiable, or prunes values from* $\vec{x}$ *and* $\vec{y}$ *to ensure* $GAC(\vec{x} \leq_m \vec{y})$.

The algorithm can easily be modified to support strict multiset ordering. The only differences are that we fail if $\vec{ox} = \vec{oy}$ and that β is set to $l - 1$ not $-\infty$. The algorithm then enforces a strict inequality on the occurrence sub-vectors above β. Another variant of the algorithm is when $d \gg n, m$. In such a situation, it could be costly to construct the occurrence vectors. We can instead sort the minimums of the x_i and the maximums of the y_j, and compute α, β, γ and δ as if we had the occurrences by scanning these sorted lists. This information is all we need to compute support for each x_i and y_j in turn. The complexity of this modified algorithm is $O(n \log n + m \log m)$ as the cost of sorting dominates.

If we have multiple rows of a matrix that are multiset ordered, we can decompose this into multiset ordering constraint on all pairs of rows, or (further still) onto ordering constraints just on neighbouring pairs of rows. The following result shows that such decompositions hinder constraint propagation. Nevertheless, it will usually be most cost effective to post just the $O(n)$ ordering constraints between neighbouring pairs rather than the $O(n^2)$ constraints between all pairs.

Theorem 3 $GAC(\forall i < j \, . \, \vec{x}_i \leq_m \vec{x}_j)$ *is strictly stronger than* $GAC(\vec{x}_i \leq_m \vec{x}_j)$ *for all* $i < j$*, and this itself is strictly stronger than* $GAC(\vec{x}_i \leq_m \vec{x}_{i+1})$ *for all* i.

5 Alternative approaches

5.1 Arithmetic constraint

Barbara Smith (personal communication) has proposed enforcing $\vec{x} <_m \vec{y}$ on vectors of length n via the arithmetic constraint $n^{x_0} + \ldots n^{x_{n-1}} < n^{y_0} + \ldots n^{y_{n-1}}$. This is similar to the transformation of a leximin fuzzy CSP into an equivalent MAX CSP [Schiex *et al.*, 1995]. BC on such a constraint is equivalent to GAC on the original multiset ordering constraint. However, such an arithmetic constraint is only feasible for small n. Further, most existing solvers will not enforce BC on such an arithmetic constraint, but will delay it until all but one of the variables are instantiated.

5.2 Decomposition

Multiset ordering is equivalent to the lex ordering the associated occurrence vectors. As we have efficient algorithms for constructing occurrence vectors (via the global cardinality constraint [Régin, 1996]) and for lex ordering [Frisch *et al.*, 2002], this might be an alternative approach. However, as the following theorem shows, such a decomposition hinders constraint propagation. Also, the two global cardinality constraints in such a decomposition are more expensive to enforce than the algorithm presented here. We write $gcc(\vec{x}, \vec{ox})$ for the global cardinality constraint that channels from a vector of variables $\vec{x}$ to the associated occurrence vector $\vec{ox}$.

Theorem 4 *$GAC(\vec{x} <_m \vec{y})$ is strictly stronger than simultaneously enforcing $GAC(gcc(\vec{x}, \vec{ox}))$, $GAC(gcc(\vec{y}, \vec{oy}))$, and $GAC(\vec{ox} <_{lex} \vec{oy})$.*

Proof: Clearly it is as strong. To show strictness, consider $\vec{x} = \langle\{1,2\},\{1,2\},\{2\},\{2\}\rangle$ and $\vec{y} = \langle\{1,2\},\{1,2\},\{0,1,2\},\{0,1\}\rangle$. The multiset ordering constraint is not GAC since 0 in y_2 has no support but the decomposition is unable to determine this. QED.

Another approach is to use the sorted constraint in the Eclipse solver. This ensures that the values taken by one vector of variables are identical but in sorted order to the values taken by a second vector of variables. To post a multiset ordering constraint on two vectors, we can channel each into a sorted vector and lex order these. The above example demonstrates that such a decomposition again hinders propagation. The sorting constraint is also more expensive to enforce.

6 Experimental results

We designed some experiments to test three goals. First, is multiset ordering an effective method for dealing with row and/or column symmetry? Second, how does multiset ordering compare to lex ordering? Which one breaks more symmetry? Is a combined method, which multiset orders one dimension and lex orders the other one of the matrix, superior? Third, does our GAC algorithm do more inference in practice than the decomposition? Similarly, is the algorithm more efficient in practice than its decomposition?

The symmetry breaking constraints we used are strict lex ordering on the columns ($<_{lex}$C), on the rows ($<_{lex}$R); multiset ordering on the rows ($\leq_m$R), (strict) multiset ordering on the columns ($\leq_m$C and $<_m$C); and combinations of these constraints. Such constraints are posted between adjacent rows/columns. The results of the experiments are shown in tables where a "-" means no result is obtained in 1 hour (3600 secs). The experiments are done using ILOG Solver 5.2 on a 1000MHz pentium III with 256 Mb RAM using Windows XP.

6.1 Progressive Party Problem

There are a set of host boats, each with a capacity, and a set of guest boats, each with a crew size. We wish to assign a host for each guest and period, such that a guest crew never visits the same host twice, no two guest crews meet more than once, and the spare capacity of each host boat, after accommodating its own crew, is not exceeded (prob013 in csplib.org).

	Fails	Choice points	Time (secs.)
No Symmetry Breaking	180,738	180,860	75.9
$<_{lex}$C	180,738	180,860	81.5
$<_{lex}$R	2,720	2,842	2.7
$<_{lex}$RC	2,720	2,842	2.7
$\leq_m$C	137,185	137,306	71.2
$\leq_m$R	10,853	10,977	8.6
$\leq_m$RC	-	-	-
$<_{lex}$C $\leq_m$R	10,853	10,977	8.6
$\leq_m$C $<_{lex}$R	2,016	2,137	2.6

Table 1: 5-13-29 progressive party problem using row-by-row labelling.

A matrix model of this problem [Smith *et al.*, 1995] is a 2-d matrix of $guests \times periods$ where each variable is assigned a host representing that a host is accommodating a particular guest in a given time period. The rows are the guests, the columns are the periods. This model has column and partial row symmetry: any two periods, and any two guests with the same crew size are indistinguishable. We consider the 13-hosts and 29 guests problem with 5 and 6 time periods, referred as 5-13-29 and 6-13-29. These problems have $p!14!2!4!5!7!$ row and column symmetries where p is the number of time periods. The actual data can be found in csplib.org. Due to the problem constraints, no pair of rows/columns can be equal, hence we can safely pose strict lex ordering. However, any two distinct rows/columns might be equal when viewed as multisets.

As in [Smith *et al.*, 1995], the guest boats are ordered in descending order of their size. We order the host boats in descending order of spare capacity to choose a value in a succeed-first manner. Results obtained by row-by-row, and column-by-column labelling strategies are given in Tables 1 and 2. With row-by-row labelling, we cannot solve 6-13-29 with or without symmetry breaking. For the 5-13-29 problem, $<_{lex}$R breaks a lot more row symmetry than $\leq_m$R. However, the reverse is true for the columns. Here, $<_{lex}$C does not break any symmetry but $\leq_m$C does. Multiset ordering one dimension of a matrix therefore does not necessarily break less symmetry than lex ordering the same dimension. Such phenomena occur through interactions with the search strategy: a search strategy might already lex order, so multiset ordering constraints break additional symmetry. The smallest search tree and also the least solving time is obtained by $\leq_m$C $<_{lex}$R. This supports our conjecture that lex ordering one dimension combined with multiset ordering the other can break more symmetry than lex/multiset ordering both dimensions.

With column-by-column labelling, we are able to solve the 6-13-29 problem. Neither of $<_{lex}$R, $<_{lex}$C, $<_{lex}$RC break any symmetry. The smallest search tree is obtained by $\leq_m$C. This supports our conjecture that multiset ordering one dimension can break more symmetry than lex ordering the same or both dimensions. If the search strategy already orders both dimensions lexicographically, imposing a constraint like multiset ordering in one dimension breaks additional symmetry.

	Fails	Choice points	Time (secs.)
No Symmetry Breaking	20,722	20,871	12.3
$<_{lex}$C	20,722	20,871	12.4
$<_{lex}$R	20,722	20,871	12.5
$<_{lex}$RC	20,722	20,871	12.4
$\leq_m$C	7,053	7,202	4.6
$\leq_m$R	-	-	-
$\leq_m$RC	-	-	-
$<_{lex}$C $\leq_m$R	-	-	-
$\leq_m$C $<_{lex}$R	7,053	7,202	4.6

Table 2: 6-13-29 progressive party problem using column-by-column labelling.

6.2 Sports Scheduling with Odd Teams

This is a modified version of prob026 in csplib.org. We have n teams (n is odd), playing over n weeks. Each week is divided into $(n-1)/2$ periods, and each period is divided into 2 slots, home and away. We wish to find a schedule so that every team plays at most once a week, every team plays twice in the same period over the tournament and every team plays every other team. We slightly modify the model in [Van Hentenryck *et al.*, 1999], where *teams* is a 3-d matrix of $periods \times weeks \times slots$. Each element of *teams* is the team playing in a given period, week and slot. We treat this matrix as 2-d where the rows are the periods and columns are the weeks, and each entry is a list of variables giving the slots.

As the periods and the weeks are indistinguishable, this problem has $n!(n-1/2)!$ row and column symmetries. We pose strict ordering constraints on the rows and columns of $teams$ as the periods and weeks cannot be equal. Due to the constraints on the periods, posing multiset ordering on the rows is not effective.

Results obtained by column-by-column labelling of the $teams$ are given in Table 3. For one column, we first label the first slots; for the other, we first label the second slots. With this strategy, $<_{lex}$R does not break any symmetry, so we omit it in the table. Posing multiset ordering by our algorithm is much more effective and efficient than by gcc and lex ordering constraints. This holds for many other search strategies. In Table 3, we note that $<_m$C gives a smaller search tree than $<_{lex}$C. However, for other search strategies the reverse is true. This supports the theoretical result that lex ordering and multiset ordering are incomparable.

7 Conclusions

We have identified a new and important global (non-binary) constraint. This constraint ensures that the values taken by two vectors of variables, when viewed as multisets, are ordered. We have developed an efficient linear time algorithm for enforcing generalised arc-consistency on such a multiset ordering constraint. We have proposed a number of applications for this new constraint including breaking symmetry in matrix models, and fuzzy constraint satisfaction. We have shown that alternative methods for posting a multiset ordering constraint like an arithmetic constraint or decomposition are inferior. Finally, we have implemented this generalized arc-consistency algorithm in ILOG Solver. Experimental results on a number of problem domains show considerable promise.

n	Model	Failures	Choice points	Time (sec.)
5	No symmetry breaking	3	12	1.1
	$gcc+<_{lex}$C	2	11	1.4
	$<_m$C	1	10	1.4
	$<_{lex}$C	3	12	1.5
7	No symmetry breaking	6,871	6,890	1.9
	$gcc+<_{lex}$C	74	92	1.3
	$<_m$C	69	87	1.1
	$<_{lex}$C	771	788	1.3
9	No symmetry breaking	-	-	-
	$gcc+<_{lex}$C	2,616,149	2,616,177	857.2
	$<_m$C	760,973	761,003	130.5
	$<_{lex}$C	-	-	-

Table 3: Sports scheduling problem.

References

[Beldiceanu, 2000] N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Proc. of CP-2000*, LNCS 1894, pp 52–66, 2000.

[Fahle *et al.*, 2001] T. Fahle, S. Schamberger and M. Sellman. Symmetry Breaking. In *Proc. of CP-2001*, LNCS 2239, pp 93–107, 2001.

[Fargier, 1994] H. Fargier. *Problèmes de satisfaction de contraintes flexibles: application à l'ordonnancement de production*. PhD thesis, Uni. of Paul Sabatier, Toulouse, 1994.

[Flener *et al.*, 2002] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetry in matrix models. In *Proc. of CP-2002*, LNCS 2470, pp 462–476, 2002.

[Frisch *et al.*, 2002] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proc. of CP-2002*, LNCS 2470, pp 93–108, 2002.

[Gent and Smith, 2000] I. Gent and B. Smith. Symmetry Breaking in Constraint Programming. In *Proc. of ECAI '2000*, pp 599–603, 2000.

[Kiziltan and Smith, 2002] Z. Kiziltan and B. Smith. Symmetry breaking constraints for matrix models. In *Proc. of 2nd Int. Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02)*, held alongside CP-2002, 2002.

[Kiziltan and Walsh, 2002] Z. Kiziltan and T. Walsh. Constraint Programming with Multisets. In *Proc. of 2nd Int. Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02)*, held alongside CP-2002, 2002.

[Régin, 1994] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. AAAI'94*, pp 362–367. 1994.

[Régin, 1996] J.C. Régin. Generalized arc consistency for global cardinality constraints. In *Proc. of AAAI'96*, pp 209–215. 1996.

[Schiex *et al.*, 1995] T. Schiex, H. Fargier, and G. Verfaille. Valued constraint satisfaction problems: hard and easy problems. In *Proc. of IJCAI'95*, pp 631–637. 1995.

[Smith *et al.*, 1995] B.M. Smith, S.C. Brailsford, P.M. Hubbard, and H.P. Williams. The progressive party problem. Research Report 95.8, Uni. of Leeds. 1995.

[Van Hentenryck *et al.*, 1999] P. van Hentenryck, L. Michel, L. Perron, and J.C. Régin. Constraint programming in OPL. In *Proc. of PPDP'99*, pp 98–116. Springer, 1999.