Artificial Intelligence 173 (2009) 299–328

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Filtering algorithms for the multiset ordering constraint

Alan M. Frisch [a], Brahim Hnich [b], Zeynep Kiziltan [c,*], Ian Miguel [d], Toby Walsh [e]

[a] *Department of Computer Science, University of York, UK*
[b] *Faculty of Computer Science, Izmir University of Economics, Turkey*
[c] *Department of Computer Science, University of Bologna, Italy*
[d] *School of Computer Science, University of St Andrews, UK*
[e] *NICTA and School of Computer Science and Engineering, University of New South Wales, Australia*

A R T I C L E   I N F O

Article history:
Received 1 July 2008
Received in revised form 17 October 2008
Accepted 2 November 2008
Available online 6 November 2008

Keywords:
Constraint satisfaction
Constraint programming
Modelling
Global constraints
Constraint propagation
Propagation algorithms
Symmetry breaking
Multiset ordering
Leximin optimal solutions

A B S T R A C T

Constraint programming (CP) has been used with great success to tackle a wide variety of constraint satisfaction problems which are computationally intractable in general. Global constraints are one of the important factors behind the success of CP. In this paper, we study a new global constraint, the multiset ordering constraint, which is shown to be useful in symmetry breaking and searching for leximin optimal solutions in CP. We propose efficient and effective filtering algorithms for propagating this global constraint. We show that the algorithms maintain generalised arc-consistency and we discuss possible extensions. We also consider alternative propagation methods based on existing constraints in CP toolkits. Our experimental results on a number of benchmark problems demonstrate that propagating the multiset ordering constraint via a dedicated algorithm can be very beneficial.

1. Introduction

Constraint satisfaction problems (CSPs) play an important role in various fields of computer science [34] and are ubiquitous in many real-life application areas such as production planning, staff scheduling, resource allocation, circuit design, option trading, and DNA sequencing. In general, solving CSPs is NP-hard and so is computationally intractable [21]. *Constraint programming* (CP) provides a platform for solving CSPs [1,23] and has proven successful in many real-life applications [29,30,37] despite this intractability. One of the jewels of CP is the notion of global (or non-binary) constraints. They encapsulate patterns that occur frequently in constraint models. Moreover, they contain specialised filtering algorithms for powerful constraint inference. Dedicated filtering algorithms for global constraints are vital for efficient and effective constraint solving. A number of such algorithms have been developed (see [4] for examples).

In this paper, we study a new global constraint, the multiset ordering constraint, which ensures that the values taken by two vectors of variables, when viewed as multisets, are ordered. This constraint has applications in breaking row and column symmetry as well as in searching for leximin optimal solutions. We propose two different filtering algorithms for the multiset ordering (global) constraint. Whilst they both maintain generalised arc-consistency, they differ in their complexity. The first algorithm `MsetLeq` runs in time that is in the number of variables (n) and in the number of distinct values (d) and is suitable when n is much bigger than d. Instead, the second algorithm is more suitable when we have large

* Corresponding author.
E-mail addresses: frisch@cs.york.ac.uk (A.M. Frisch), brahim.hnich@ieu.edu.tr (B. Hnich), zeynep@cs.unibo.it (Z. Kiziltan), ianm@dcs.st-and.ac.uk (I. Miguel), toby.walsh@nicta.com.au (T. Walsh).

0004-3702/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.artint.2008.11.001

domains and runs in time $O(n\log(n))$ independent of d. We propose further algorithms by considering some extensions to `MsetLeq`. In particular, we show how we can identify entailment and obtain a filtering algorithm for the strict multiset ordering constraint. These algorithms are proven to maintain generalised arc-consistency.

We consider alternative approaches to propagating the multiset ordering constraint by using existing constraints in CP toolkits. We evaluate our algorithms in contrast to the alternative approaches on a variety of representative problems in the context of symmetry breaking. The results demonstrate that our filtering algorithms are superior to the alternative approaches either in terms of pruning capabilities or in terms of computational times or both. We stress that the contribution of this paper is the study of the filtering algorithms for the multiset ordering constraint. Symmetry breaking is merely used to compare the efficiency of these propagators. A more in depth comparison of symmetry breaking methods awaits a separate study. Such a study would be interesting in its own right as multiset ordering constraints are one of the few methods for breaking symmetry which are not special cases of lexicographical ordering constraints [6]. Nevertheless, we provide experimental evidence to support the need of multiset ordering constraints in the context of symmetry breaking.

The rest of the paper is organised as follows. After we give the necessary formal background in the next section, we present in Section 3 the utility of the multiset ordering constraint. In Section 4, we present our first filtering algorithm, prove that it maintains generalised arc-consistency, and discuss its complexity. Our second algorithm is introduced in Section 5. In Section 6, we extend our first algorithm to obtain an algorithm for the strict multiset ordering constraint and to detect entailment. Alternative propagation methods are discussed in Section 7. We demonstrate in Section 8 that decomposing a chain of multiset ordering constraints into multiset ordering constraints between adjacent or all pairs of vectors hinders constraint propagation. Computational results are presented in Section 9. Finally, we conclude and outline our plans for future work in Section 10.

2. Formal background

2.1. Constraint satisfaction problems and constraint programming

A *finite-domain constraint satisfaction problem* (CSP) consists of: (i) a finite set of variables $\mathcal{X}$; (ii) for each variable $X \in \mathcal{X}$, a finite set $\mathcal{D}(X)$ of values (its domain); (iii) and a finite set $\mathcal{C}$ of constraints on the variables, where each constraint $c(X_i,\ldots,X_j) \in \mathcal{C}$ is defined over the variables $X_i,\ldots,X_j$ by a subset of $\mathcal{D}(X_i) \times \cdots \times \mathcal{D}(X_j)$ giving the set of allowed combinations of values. That is, c is an n-ary relation.

A variable *assignment* or *instantiation* is an assignment to a variable X of one of the values from $\mathcal{D}(X)$. Whilst a *partial assignment* A to $\mathcal{X}$ is an assignment to some but not all $X \in \mathcal{X}$, a *total assignment*[1] A to $\mathcal{X}$ is an assignment to every $X \in \mathcal{X}$. We use the notation $A[\mathcal{S}]$ to denote the projection of A on to the set of variables $\mathcal{S}$. A (partial) assignment A to the set of variables $\mathcal{T} \subseteq \mathcal{X}$ is *consistent* iff for all constraints $c(X_i,\ldots,X_j) \in \mathcal{C}$ such that $\{X_i,\ldots,X_j\} \subseteq \mathcal{T}$, we have $A[\{X_i,\ldots,X_j\}] \in c(X_i,\ldots,X_j)$. A *solution* to the CSP is a consistent assignment to $\mathcal{X}$. A CSP is said to be satisfiable if it has a solution; otherwise it is unsatisfiable. Typically, we are interested in finding one or all solutions, or an optimal solution given some objective function. In the presence of an objective function, a CSP is a *constraint optimisation problem*.

Constraint Programming (CP) has been used with great success to solve CSPs. Recent years have witnessed the development of several CP systems [30]. To solve a problem using CP, we need first to formulate it as a CSP by declaring the variables, their domains, as well as the constraints on the variables. This part of the problem solving is called *modelling*. In the following, we first introduce our notations and then briefly overview modelling and solving in CP. Since we compare our algorithms against the alternative approaches in the context of symmetry breaking, we also briefly review matrix modelling and index symmetry.

2.2. Notation

Throughout, we assume finite integer domains, which are totally ordered. The domain of a variable X is denoted by $\mathcal{D}(X)$, and the minimum and the maximum elements in this domain by $min(X)$ and $max(X)$. We use $vars(c)$ to denote the set of variables constrained by constraint c. If a variable X has a singleton domain $\{v\}$ we say that v is assigned to X and denotes this by $X \leftarrow v$, or simply say that X is assigned. If two variables X and X' are assigned the same value, then we write $X \doteq X'$, otherwise we write $\neg(X \doteq X')$.

A one-dimensional matrix, or vector, is an ordered list of elements. We denote a vector of n variables as $\vec{X} = \langle X_0,\ldots,X_{n-1}\rangle$ and a vector of n integers as $\vec{x} = \langle x_0,\ldots,x_{n-1}\rangle$. In either case, a sub-vector from index a to index b inclusive is denoted by the subscript $a \to b$, such as: $\vec{x}_{a\to b}$. Unless otherwise stated, the indexing of vectors is from left to right, with 0 being the most significant index, and the variables of a vector $\vec{X}$ are assumed to be disjoint and not repeated. The vector $\vec{X}_{X_i \leftarrow d}$ is the vector $\vec{X}$ with some X_i being assigned to d. The functions $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{X})$ assign all the variables of $\vec{X}$ their minimum and maximum values, respectively. A vector $\vec{x}$ in the domain of $\vec{X}$ is designated by $\vec{x} \in \vec{X}$. We write $\{\vec{x} \mid C \wedge \vec{x} \in \vec{X}\}$ to denote the set of vectors in the domain of $\vec{X}$ which satisfy condition C. A vector of variables

[1] Throughout, we will say *assignment* when we mean *total assignment* to the problem variables.

is displayed by a vector of the domains of the corresponding variables. For instance, $\vec{X} = \langle\{1,3,4\},\{1,2,3,4,5\},\{1,2\}\rangle$ denotes the vector of three variables whose domains are $\{1,3,4\}$, $\{1,2,3,4,5\}$, and $\{1,2\}$, respectively.

A set is an unordered list of elements in which repetition is not allowed. We denote a set of n elements as $\mathcal{X} = \{x_0,\ldots,x_{n-1}\}$. A multiset is an unordered list of elements in which repetition is allowed. We denote a multiset of n elements as $\mathbf{x} = \{\!\{x_0,\ldots,x_{n-1}\}\!\}$. We write $max(\mathbf{x})$ or $max\{\!\{x_0,\ldots,x_{n-1}\}\!\}$ for the maximum element of a multiset $\mathbf{x}$. By ignoring the order of elements in a vector, we can view a vector as a multiset. For example, the vector $\langle 0,1,0\rangle$ can be viewed as the multiset $\{\!\{1,0,0\}\!\}$. We will abuse notation and write $\{\!\{\vec{x}\}\!\}$ or $\{\!\{\langle x_0,\ldots,x_{n-1}\rangle\}\!\}$ for the multiset view of the vector $\vec{x} = \langle x_0,\ldots,x_{n-1}\rangle$.

An occurrence vector $occ(\vec{x})$ associated with $\vec{x}$ is indexed in decreasing order of significance from the maximum $max\{\!\{\vec{x}\}\!\}$ to the minimum $min\{\!\{\vec{x}\}\!\}$ value from the values 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, $sort(\vec{x})$ is the vector obtained by sorting the values in $\vec{x}$ in non-increasing order.

2.3. Search, local consistency and propagation

Solutions to CSPs are often found by *searching* systematically the space of partial assignments. A common search strategy is *backtracking search*. We traverse the search space in a depth-first manner and at each step extend a partial assignment by assigning a value to one more variable. If the extended assignment is consistent then one more variable is instantiated and so on. Otherwise, the variable is re-instantiated with another value. If none of the values in the domain of the variable is consistent with the current partial assignment then one of the previous variable assignments is reconsidered.

Backtracking search may be seen as a *search tree* traversal. Each node defines a partial assignment and each branch defines a variable assignment. A partial assignment is extended by branching from the corresponding node to one of its subtrees by assigning a value j to the next variable X_i from the current $\mathcal{D}(X_i)$. Upon backtracking, j is removed from $\mathcal{D}(X_i)$. This process is often called *labelling*. The order of the variables and values chosen for consideration can have a profound effect on the size of the search tree [16]. The order can be determined before search starts, in which case the labelling heuristic is *static*. If the next variable and/or value are determined during search then the labelling heuristic is *dynamic*.

The size of the search tree of a CSP is in the worst case equal to the product of the domain sizes of all variables. It is thus too expensive in general to enumerate all possible assignments using a naive backtracking algorithm. Consequently, many CP solution methods are based on *inference* which reduces the problem to an equivalent (i.e. with the same solution set) but smaller problem. Since complete inference is too computationally expensive to be used in practice, inference methods are often incomplete and enforce *local consistencies*. A local consistency is a property of a CSP defined over "local" parts of the CSP, in other words defined over subsets of the variables and constraints of the CSP. The main idea is to remove from the domains of the variables the values that will not take part of any solution. Such values are said to be *inconsistent*. Inconsistent values can be detected by using a number of consistency properties.

A common consistency property proposed in [22] is generalised arc-consistency. A constraint c is *generalised arc-consistent* (or GAC), written GAC(c), if and only if for every $X \in vars(c)$ and every $v \in \mathcal{D}(X)$, there is at least one assignment to $vars(c)$ that assigns v to X and satisfies c. Values for variables other than X participating in such assignments are known as the *support* for the assignment of v to X. Generalised arc-consistency is established on a constraint c by removing elements from the domains of variables in $vars(c)$ until the GAC property holds. For binary constraints, GAC is equivalent to *arc-consistency* (AC, see [21]). Another useful local consistency is *bound consistency* that treats the domains of the variables as intervals. For integer variables, the values have a natural total order, therefore the domain can be represented by an interval whose lower bound is the minimum value and the upper bound is the maximum value in the domain. A constraint C is *bound consistent* (*BC*) iff for every variable, for its minimum (maximum) there exists a value for every other variable between its minimum and maximum that satisfies C [36].

We will compare local consistency properties applied to (sets of) logically equivalent constraints, c_1 and c_2. As in [7], we say that a local consistency property Φ on c_1 is as strong as Ψ on c_2 iff, given any domains, if Φ holds on c_1 then Ψ holds on c_2; we say that Φ on c_1 is strictly stronger than Ψ on c_2 iff Φ on c_1 is as strong as Ψ on c_2 but not vice versa.

In a constraint program, searching for solutions is interleaved with local consistency as follows. Local consistency is first enforced before search starts to preprocess the problem and prune subsequent search. It is then maintained dynamically at each node of the search tree with respect to the current variable assignment. In this way, the domains of the uninstantiated variables shrink and the search tree gets smaller. Whilst the process of maintaining local consistency over a CSP is known as *propagation*, the process of removing inconsistent values from the domains is known as *pruning* or *filtering*. For effective constraint solving, it is important that propagation removes efficiently as many inconsistent values as possible. Note that GAC is an important consistency property as it is the strongest filtering that be done by reasoning on only a single constraint at a time. Many global constraints in CP toolkits therefore encapsulate their own *filtering algorithm* which typically achieves GAC at a low cost by exploiting the semantics of the constraint. As an example, Régin in [27] gives a filtering algorithm for the *all-different* constraint which maintains GAC in time $O(n^{2.5})$ where n is the number of variables.

The semantics of a constraint can help not only find supports and inconsistent values quickly but also detect entailment and disentailment without having to do filtering. A constraint c is *entailed* if all assignments of values to $vars(c)$ satisfy c.

Similarly, a constraint c is *disentailed* when all assignments of values to $vars(c)$ violate c. If a constraint in a CSP is detected to be entailed, it does not have to propagated in the future, and if it is detected to be disentailed then it is proven that the current CSP has no solution and we can backtrack.

2.4. Modelling

CP toolkits provide constructs for declaring the variables, their domains, as well as the constraints between these variables of a CSP. They often contain a library of predefined constraints with a particular semantics that can be applied to sets of variables with varying arities and domains. For instance, *all-different*$([X_1, \dots, X_3])$ with $D(X_1) = D(X_2) = \{1, 2\}$, $D(X_3) = \{1, 2, 3\}$ is an instance of *all-different*$([X_1, \dots, X_n])$ defined on three variables with the specified domains. It has the semantics that the variables involved take different values [27]. The *all-different*$([X_1, \dots, X_n])$ constraint can be applied to any number of variables with any domains. Such constraints are often referred as *global constraints*. Beldiceanu has catalogued hundreds of global constraints, most of which are defined over finite domain variables [4]. They permit the user to model a problem easily by compactly specifying common patterns that occur in many constraint models. They also provide solving advantages which we shall explain later.

Since constraints provide a rich language, a number of alternative models will often exist, some of which will be more effective than others. However, one of the most common and effective modelling patterns in constraint programming is a *matrix model*. A matrix model is the formulation of a CSP with one or more matrices of decision variables (of one or more dimensions) [12]. Matrix models are a natural way to represent problems that involve finding a function or a relation. We shall illustrate matrix models and the power of global constraints in modelling through the **sport scheduling problem**. This problem involves scheduling games between n teams over $n - 1$ weeks [35]. Each week is divided into $n/2$ periods, and each period is divided into two slots. The team in the first slot plays at home, while the team in the second slot plays away. The goal is to find a schedule such that: (i) every team plays exactly once a week; (ii) every team plays against every other team; (iii) every team plays at most twice in the same period over the tournament. Van Hentenryck et al. propose a model for this problem in [35], where they extend the problem with a "dummy" final week to make the problem more uniform. The model consists of two matrices: a 3-d matrix T of $\mathcal{P}eriods \times \mathcal{E}weeks \times \mathcal{S}lots$ and a 2-d matrix G of $\mathcal{P}eriods \times \mathcal{W}eeks$, where $\mathcal{P}eriods$ is the set of $n/2$ periods, $\mathcal{E}weeks$ is the set of n extended weeks, $\mathcal{W}eeks$ is the set of $n - 1$ weeks, and $\mathcal{S}lots$ is the set of 2 slots. In T, weeks are extended to include the dummy week, and each element takes a value from $\{1, \dots, n\}$ expressing that a team plays in a particular week in a particular period, in the home or away slot. For the sake of simplicity, we will treat this matrix as 2-d where the rows represent the periods and the columns represent the extended weeks, and each entry of the matrix is a pair of variables. The elements of G takes values from $\{1, \dots, n^2\}$, and each element denotes a particular unique combination of home and away teams. More precisely, a game played between a home team h and an away team a is uniquely identified by $(h - 1) * n + a$. (see Fig. 1).

Consider the columns of T which denote the (extended) weeks. The first set of constraints post *all-different* (global) constraints on the columns of T to enforce that each column is a permutation of $1 \dots n$. The second constraint is an *all-different* (global) constraint on G that enforces that all games must be different. Consider the rows of T which represent

Matrices:

	$\mathcal{S}lots$						
	$\nearrow$	3	2	6	6	5	5
$T_{i,j,k}$	1	1	2	3	4	4	
$\downarrow$		4	5	4	5	6	6
$\mathcal{P}eriods$	2	3	3	2	1	1	
$\downarrow$		6	6	5	4	3	3
	5	4	1	1	2	2	
	$\rightarrow \mathcal{E}weeks \rightarrow$						

	$\rightarrow \mathcal{W}eeks \rightarrow$				
$G_{i,j}$	3	2	12	18	23
$\downarrow$	10	17	16	11	6
$\mathcal{P}eriods$	30	24	5	4	9

Constraints:

(1) $\forall i \in \mathcal{E}weeks\,.\ \textit{all-different}(T_i)$

(2) $\textit{all-different}(G)$

(3) $\forall j \in \mathcal{P}eriods\,.\ gcc(\langle T_{0,j,0}, T_{0,j,1}, \dots, T_{n-1,j,0}, T_{n-1,j,1}\rangle, \langle 1, 2, \dots, n\rangle, \langle 2, \dots, 2\rangle)$

(4) $\forall i \in \mathcal{W}eeks\,.\ \forall j \in \mathcal{P}eriods\,.$
$\langle T_{i,j,0}, T_{i,j,1}, G_{i,j}\rangle \in \{\langle h, a, (h-1) * n + a\rangle \mid h, a, h < a \in \{1, \dots, n\}\}$

(5) $\forall i \in \mathcal{E}weeks\,.\ \forall j \in \mathcal{P}eriods\,.\ T_{i,j,0} < T_{i,j,1}$

Fig. 1. The matrix model of the sport scheduling problem in [35].

the periods. The third set of constraints post the global cardinality constraints (*gcc*) on the rows to ensure that each of $1 \ldots n$ occur exactly twice in every row. The fourth set of constraints are called *channelling constraints* and are often used when multiple matrices are used to model the problem and they have to be linked together. In our case, the channelling constraints links a variable representing a game ($G_{i,j}$) with a variable representing the team playing home team ($T_{i,j,0}$) and the corresponding variable representing the away team ($T_{i,j,1}$) such that $G_{i,j} = (T_{i,j,0} - 1) * n + T_{i,j,1}$. The final set of constraints will be discussed after giving an overview of symmetry in CP.

2.5. Symmetry

A *symmetry* is an intrinsic property of an object which is preserved under certain classes of transformations. For instance, rotating a chess board 90° gives us a board which is indistinguishable from the original one. A CSP can have symmetries in the variables or domains or both which preserve satisfiability. In the presence of symmetry, any (partial) assignment can be transformed into a set of symmetrically equivalent assignments without affecting whether or not the original assignment satisfies the constraints.

Symmetry in constraint programs increases the size of the search space. It is therefore important to prune symmetric states so as to improve the search efficiency. This process is referred to as *symmetry breaking*. One of the easiest and most efficient ways of symmetry breaking is adding *symmetry breaking constraints* [6,25]. These constraints impose an ordering on the symmetric objects. Among the set of symmetric assignments, only those that satisfy the ordering constraints are chosen for consideration during the process of search. For instance, in the matrix model of Fig. 1, any solution can be mapped to a symmetric solution by swapping any two teams ($T_{i,j,0}$ and $T_{i,j,1}$). These solutions are essentially the same. We can add the set of constraints (5) in order to break such symmetry between the two teams and speed up search by avoiding visiting symmetric branches.

A common pattern of symmetry in matrix models is that the rows and/or columns of a 2-d matrix represent indistinguishable objects. Consequently the rows and/or columns of an assignment can be swapped without affecting whether or not the assignment is a solution [11]. These are called *row* or *column symmetry*; the general term is *index symmetry*. For instance, in the matrix model of Fig. 1, the (extended) weeks over which the tournament is held, as well the periods of a week are indistinguishable. The rows and the columns of T and G are therefore symmetric. Note that we treat T as a 2-d matrix where the rows represent the periods and columns represent the (extended) weeks, and each entry of the matrix is a pair of variables.

If every bijection on the values of an index is an index symmetry, then we say that the index has *total symmetry*. If the first (resp. second) index of a 2-d matrix has total symmetry, we say that the matrix has *total column symmetry* (resp. *total row symmetry*). In many matrix models only a subset of the rows or columns are interchangeable. If the first (resp. second) index of a 2-d matrix has partial symmetry, we say that the matrix has *partial column symmetry* (resp. *partial row symmetry*).[2] There is one final case to consider: an index may have partial index symmetry on multiple subsets of its values. For example, a CSP may have a 2-d matrix for which rows 1, 2 and 3 are interchangeable and rows 5, 6 and 7 are interchangeable. This can occur on any or all of the indices.

An $n \times m$ matrix with total row and column symmetry has $n!m!$ symmetries, a number which increases super-exponentially. An effective way to deal with this class of symmetry is to use lexicographic ordering constraints.

Definition 1. A strict lexicographic ordering $\vec{x} <_{lex} \vec{y}$ between two vectors of integers $\vec{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ and $\vec{y} = \langle y_0, y_1, \ldots, y_{n-1} \rangle$ holds iff $\exists k\ 0 \leqslant k < n$ such that $x_i = y_i$ for all $0 \leqslant i < k$ and $x_k < y_k$.

The ordering can be weakened to include equality.

Definition 2. Two vectors of integers $\vec{x} = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ and $\vec{y} = \langle y_0, y_1, \ldots, y_{n-1} \rangle$ are lexicographically ordered $\vec{x} \leqslant_{lex} \vec{y}$ iff $\vec{x} <_{lex} \vec{y}$ or $\vec{x} = \vec{y}$.

Given two vectors of variables $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1} \rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1} \rangle$, we write a lexicographic ordering constraint as $\vec{X} \leqslant_{lex} \vec{Y}$ and a strict lexicographic ordering constraint as $\vec{X} <_{lex} \vec{Y}$. These constraints are satisfied by an assignment if the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$ are ordered according to Definitions 2 and 1, respectively.

To deal with column (resp. row) symmetry, we can constrain the columns (resp. rows) to be non-decreasing as the value of the index increases. One way to achieve this is by imposing a lexicographic ordering constraint between adjacent columns (resp. rows). These constraints are *consistent* which means that they leave at least one assignment among the set of symmetric assignments. We can deal with row and column symmetry in a similar way by imposing a lexicographic ordering constraint between adjacent rows and columns simultaneously. Also such constraints are consistent. Even though these constraints may not eliminate all symmetry, they have been shown to be effective at removing many symmetries from the search spaces of many problems. If a matrix has only partial column (resp. partial row) symmetry then the symmetry can be broken by constraining the interchangeable columns (resp. rows) to be in lexicographically non-decreasing order.

[2] Throughout, we will say *row symmetry* (resp. *column symmetry*) when we mean *total row symmetry* (resp. *total column symmetry*) to the problem variables.

This can be achieved in a manner similar to that described above. The method also extends to matrices that have partial or total column symmetry together with partial or total row symmetry. Finally, if the columns and/or rows of a matrix have multiple partial symmetries than each can be broken in the manner just described [11].

3. The multiset ordering constraint and its applications

Multiset ordering is a total ordering on multisets.

Definition 3. Strict multiset ordering $\mathbf{x} <_m \mathbf{y}$ between two multisets of integers $\mathbf{x}$ and $\mathbf{y}$ holds iff:

$$\mathbf{x} = \{\!\{\}\!\} \ \wedge \ \mathbf{y} \neq \{\!\{\}\!\} \vee$$
$$max(\mathbf{x}) < max(\mathbf{y}) \vee$$
$$(max(\mathbf{x}) = max(\mathbf{y}) \ \wedge \ \mathbf{x} - \{\!\{max(\mathbf{x})\}\!\} <_m \mathbf{y} - \{\!\{max(\mathbf{y})\}\!\})$$

That is, either $\mathbf{x}$ is empty and $\mathbf{y}$ is not, or the largest value in $\mathbf{x}$ is less than the largest value in $\mathbf{y}$, or the largest values are the same and, if we eliminate one occurrence of the largest value from both $\mathbf{x}$ and $\mathbf{y}$, the resulting two multisets are ordered. We can weaken the ordering to include multiset equality.

Definition 4. Two multisets of integers $\mathbf{x}$ and $\mathbf{y}$ are multiset ordered $\mathbf{x} \leqslant_m \mathbf{y}$ iff $\mathbf{x} <_m \mathbf{y}$ or $\mathbf{x} = \mathbf{y}$.

Even though this ordering is defined on multisets, it may also be useful to order vectors by ignoring the positions but rather concentrating on the values taken by the variables. We can do this by treating a vector as a multiset. Given two vectors of variables $\vec{X} = \langle X_0, X_1, \ldots, X_{n-1}\rangle$ and $\vec{Y} = \langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$, we write a multiset ordering constraint as $\vec{X} \leqslant_m \vec{Y}$ and a strict multiset ordering constraint as $\vec{X} <_m \vec{Y}$. These constraints ensure that the vectors $\vec{x}$ and $\vec{y}$ assigned to $\vec{X}$ and $\vec{Y}$, when viewed as multisets, are multiset ordered according to Definitions 4 and 3, respectively.

3.1. Breaking index symmetry

One important application of the multiset ordering constraint is in breaking index symmetry [13]. If X is an n by m matrix of decision variables, then we can break its column symmetry by imposing the constraints $\langle X_{i,0}, \ldots, X_{i,m}\rangle \leqslant_m \langle X_{i+1,0}, \ldots, X_{i+1,m}\rangle$ for $i \in [0, n-2]$, or for short $\vec{C}_0 \leqslant_m \vec{C}_1 \leqslant_m \cdots \leqslant_m \vec{C}_{n-1}$ where $\vec{C}_i$ corresponds to the vector of variables $\langle X_{i,0}, \ldots, X_{i,m}\rangle$ which belong to the ith column of the matrix. Similarly we can break its row symmetry by imposing the constraints $\langle X_{0,j}, \ldots, X_{n,j}\rangle \leqslant_m \langle X_{0,j+1}, \ldots, X_{n,j+1}\rangle$ for $j \in [0, m-2]$, or for short $\vec{R}_0 \leqslant_m \vec{R}_1 \leqslant_m \cdots \leqslant_m \vec{R}_{m-1}$ in which $\vec{R}_j$ corresponds to the variables $\langle X_{0,j}, \ldots, X_{n,j}\rangle$ of the jth row. Such constraints are consistent symmetry breaking constraints. Note that when we have partial column (resp. row) symmetry, then the symmetry can be broken by imposing multiset ordering constraints on the symmetric columns (resp. rows) only.

Whilst multiset ordering is a total ordering on multisets, it is not a total ordering on vectors. In fact, it is a preordering as it is not antisymmetric. Hence, each symmetry class may have more than one element where the rows (resp. columns) are multiset ordered. This does not however make lexicographic ordering constraints preferable over multiset ordering constraints in breaking row (resp. column) symmetry. The reason is that they are incomparable as they remove different symmetric assignments in an equivalence class [13].

One of the nice features of using multiset ordering for breaking index symmetry is that by constraining one dimension of the matrix, say the rows, to be multiset ordered, we do not distinguish the columns. We can still freely permute the columns, as multiset ordering the rows ignores positions and is invariant to column permutation. We can therefore consistently post multiset ordering constraints on the rows together with either multiset ordering or lexicographic ordering constraints on the columns when we have both row and column symmetry. Neither approach may eliminate all symmetries, however they are all potentially interesting. Since lexicographic ordering and multiset ordering constraints are incomparable, imposing one ordering in one dimension and the other ordering in the other dimension of a matrix is also incomparable to imposing the same ordering on both dimensions of the matrix [13]. Studying the effectiveness of all these different methods in reducing index symmetry is outside the scope of this paper as we only focus on the design of efficient and effective filtering algorithms for the multiset ordering constraints. Nevertheless, experimental results in Section 9 show that exploiting both multiset ordering and lexicographic ordering constraints can be very effective in breaking index symmetry.

A multiset ordering constraint can also be helpful for implementing other constraints useful to break index symmetry. One such constraint is *allperm* [14]. Experimental results in [14] show that the decomposition of *allperm* using a multiset ordering constraint can be as effective and efficient as the specialised algorithm proposed.

3.2. Searching for leximin optimal solutions

Another interesting application of the multiset ordering constraint arises in the context of searching for leximin optimal solutions. Such solutions can be useful in 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 [10]. 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 a multiset ordering constraint when we find a solution to ensure that future solutions are greater in the leximin ordering.

Leximin optimal solutions can be useful also in other domains. For instance, as shown in [5], they can be exploited as a fairness and Pareto optimality criterion when solving multiobjective problems in CP. Experimental results in [5] show that using a multiset ordering constraint in a branch and bound search can be competitive with the alternative approaches to finding leximin optimal solutions.

4. A filtering algorithm for multiset ordering constraint

In this section, we present our first filtering algorithm which either detects that $\vec{X} \leqslant_m \vec{Y}$ is disentailed or prunes inconsistent values so as to achieve GAC on $\vec{X} \leqslant_m \vec{Y}$. After sketching the main features of the algorithm on a running example in Section 4.1, we first present the theoretical results that the algorithm exploits in Section 4.2 and then give the details of the algorithm in Section 4.3. Throughout, we assume that the variables of the vectors $\vec{X}$ and $\vec{Y}$ are disjoint.

4.1. A worked example

The key idea behind the algorithm is to build a pair of occurrence vectors associated with $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{Y})$. The algorithm goes through every variable of $\vec{X}$ and $\vec{Y}$ checking for support for values in the domains. It suffices to have $occ(\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})) \leqslant_{lex} occ(\texttt{ceiling}(\vec{Y}))$ to ensure that all values of $\mathcal{D}(X_i)$ are consistent. Similarly, we only need $occ(\texttt{floor}(\vec{X})) \leqslant_{lex} occ(\texttt{ceiling}(\vec{Y}_{Y_j \leftarrow min(Y_j)}))$ to hold for the values of $\mathcal{D}(Y_j)$ to be consistent. We can avoid the repeated construction and traversal of these vectors by building, once and for all, the vectors $occ(\texttt{floor}(\vec{X}))$ and $occ(\texttt{ceiling}(\vec{Y}))$, and defining some pointers and flags on them. For instance, assume we have $occ(\texttt{floor}(\vec{X})) \leqslant_{lex} occ(\texttt{ceiling}(\vec{Y}))$. The vector $occ(\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)}))$ can be obtained from $occ(\texttt{floor}(\vec{X}))$ by decreasing the number of occurrences of $min(X_i)$ by 1, and increasing the number of occurrences of $max(X_i)$ by 1. The pointers and flags tell us whether this disturbs the lexicographic ordering, and if so they help us to find quickly the largest $max(X_i)$ which does not.

Consider the multiset ordering constraint $\vec{X} \leqslant_m \vec{Y}$ where:

$$\begin{aligned}\vec{X} &= \langle\{5\},\ \{4,5\},\ \{3,4,5\},\ \{2,4\},\ \{1\},\ \{1\}\rangle\\ \vec{Y} &= \langle\{4,5\},\ \{4\},\ \{1,2,3,4\},\ \{2,3\},\ \{1\},\ \{0\}\rangle\end{aligned}$$

We have $\texttt{floor}(\vec{X}) = \langle 5,4,3,2,1,1\rangle$ and $\texttt{ceiling}(\vec{Y}) = \langle 5,4,4,3,1,0\rangle$. We construct our occurrence vectors $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$, indexed from $max(\{\{\texttt{ceiling}(\vec{X})\}\} \cup \{\{\texttt{ceiling}(\vec{Y})\}\}) = 5$ to $min(\{\{\texttt{floor}(\vec{X})\}\} \cup \{\{\texttt{floor}(\vec{Y})\}\}) = 0$:

$$\begin{array}{rcl} & & \;\;5\;\;\;4\;\;\;3\;\;\;2\;\;\;1\;\;\;0\\ \vec{ox} &=& \langle 1,\ 1,\ 1,\ 1,\ 2,\ 0\rangle\\ \vec{oy} &=& \langle 1,\ 2,\ 1,\ 0,\ 1,\ 1\rangle\end{array}$$

Recall that ox_i and oy_i denote the number of occurrences of the value i in $\{\{\texttt{floor}(\vec{X})\}\}$ and $\{\{\texttt{ceiling}(\vec{Y})\}\}$, respectively. For example, $oy_4 = 2$ as 4 occurs twice in $\{\{\texttt{ceiling}(\vec{Y})\}\}$. Next, we define our pointers and flags on $\vec{ox}$ and $\vec{oy}$. The pointer α points to the most significant index above which the values are pairwise equal and at α we have $ox_\alpha < oy_\alpha$. This means that we will fail to find support if any of the X_i is assigned a new value greater than α, but we will always find support for values less than α. If $\vec{ox} = \vec{oy}$ then we set $\alpha = -\infty$. Otherwise, we fail immediately because no value for any variable can have support. We define β as the most significant index below α such that $ox_\beta > oy_\beta$. This means that we might fail to find support if any of the Y_j is assigned a new value less than or equal to β, but we will always find support for values larger than β. If such an index does not exist then we set $\beta = -\infty$. Finally, the flag γ is *true* iff $\beta = \alpha - 1$ or $\vec{ox}_{\alpha+1\to\beta-1} = \vec{oy}_{\alpha+1\to\beta-1}$, and σ is *true* iff the sub-vectors below β are ordered lexicographically the wrong way. In our example, $\alpha = 4$, $\beta = 2$, $\gamma = true$, and $\sigma = true$:

$$\begin{array}{rcllllll} & & 5 & 4 & 3 & 2 & 1 & 0\\ \vec{ox} &=& \langle 1, & 1, & 1, & 1, & 2, & 0\rangle\\ \vec{oy} &=& \langle 1, & 2, & 1, & 0, & 1, & 1\rangle\\ & & & \alpha\uparrow\ \gamma = true & & \beta\uparrow\ \sigma = true & & \end{array}$$

We now go through each X_i and find the largest value in its domain which is supported. If X_i has a singleton domain then we skip it because we have $\vec{ox} \leqslant_{lex} \vec{oy}$, meaning that its only value has support. Consider X_1. As $min(X_1) = \alpha$, changing $\vec{ox}$ to $occ(\texttt{floor}(\vec{X}_{X_1 \leftarrow max(X_1)}))$ increases the number of occurrences of an index above α by 1. This upsets $\vec{ox} \leqslant_{lex} \vec{oy}$. We therefore prune all values in $\mathcal{D}(X_1)$ larger than α. Now consider X_2. We have $max(X_2) > \alpha$ and $min(X_2) < \alpha$. As with X_1, any value of X_2 larger than α upsets the lexicographic ordering, but any value less than α guarantees the lexicographic ordering. The question is whether α has any support? Changing $\vec{ox}$ to $occ(\mathtt{floor}(\vec{X}_{X_2 \leftarrow \alpha}))$ decreases the number of occurrences of 3 in $\vec{ox}$ by 1, and increases the number of occurrences of α by 1. Now we have $ox_\alpha = oy_\alpha$ but decreasing an entry in $\vec{ox}$ between α and β guarantees lexicographic ordering. We therefore prune from $\mathcal{D}(X_2)$ only the values greater than α. Now consider X_3. We have $max(X_3) = \alpha$ and $min(X_3) < \alpha$. Any value less than α has support but does α have any support? Changing $\vec{ox}$ to $occ(\mathtt{floor}(\vec{X}_{X_3 \leftarrow \alpha}))$ decreases the number of occurrences of *beta* in $\vec{ox}$ by 1, and increases the number of occurrences of α by 1. Now we have $ox_\alpha = oy_\alpha$ and $ox_\beta = oy_\beta$. Since γ and σ are *true*, the occurrence vectors are lexicographically ordered the wrong way. We therefore prune α from $\mathcal{D}(X_3)$. We skip X_4 and X_5.

Similarly, we go through each Y_j and find the smallest value in its domain which is supported. If Y_j has a singleton domain then we skip it because we have $\vec{ox} \leqslant_{lex} \vec{oy}$, meaning that its only value has support. Consider Y_0. As $max(Y_0) > \alpha$, changing $\vec{oy}$ to $occ(\mathtt{ceiling}(\vec{Y}_{Y_0 \leftarrow min(Y_0)}))$ decreases the number of occurrences of an index above α by 1. This upsets $\vec{ox} \leqslant_{lex} \vec{oy}$. We therefore prune all values in $\mathcal{D}(Y_0)$ less than or equal to α. Now consider Y_2. We have $max(Y_2) = \alpha$ and $min(Y_2) \leqslant \beta$. Any value larger than β guarantees lexicographic ordering. The question is whether the values less than or equal to β have any support? Changing $\vec{oy}$ to $occ(\mathtt{ceiling}(\vec{Y}_{Y_2 \leftarrow min(Y_2)}))$ decreases the number of occurrences of α by 1, giving us $ox_\alpha = oy_\alpha$. If $min(Y_2) = \beta$ then we have $ox_\beta = oy_\beta$. This disturbs $\vec{ox} \leqslant_{lex} \vec{oy}$ because γ and σ are both *true*. If $min(Y_2) < \beta$ then again we disturb $\vec{ox} \leqslant_{lex} \vec{oy}$ because γ is *true* and the vectors are not lexicographically ordered as of β. So, we prune from $\mathcal{D}(Y_2)$ the values less than or equal to β. Now consider Y_3. As $max(Y_3) < \alpha$, changing $\vec{oy}$ to $occ(\mathtt{ceiling}(\vec{Y}_{Y_3 \leftarrow min(Y_3)}))$ does not change that $\vec{ox} \leqslant_{lex} \vec{oy}$. Hence, $min(Y_3)$ is supported. We skip Y_4 and Y_5.

We have now the following generalised arc-consistent vectors:

$$\begin{aligned}\vec{X} &= \langle \{5\},\ \{4\},\ \{3,4\},\ \ \{2\},\ \ \{1\},\ \{1\}\rangle\\ \vec{Y} &= \langle \{5\},\ \{4\},\ \{3,4\},\ \{2,3\},\ \{1\},\ \{0\}\rangle\end{aligned}$$

4.2. Theoretical background

The algorithm exploits four theoretical results. The first reduces GAC to consistency on the upper bounds of $\vec{X}$ and on the lower bounds of $\vec{Y}$. The second and the third show in turn when $\vec{X} \leqslant_m \vec{Y}$ is disentailed and what conditions ensure GAC on $\vec{X} \leqslant_m \vec{Y}$. And the fourth establishes that two ground vectors are multiset ordered iff the associated occurrence vectors are lexicographically ordered.

Theorem 1. GAC$(\vec{X} \leqslant_m \vec{Y})$ *iff for all* $0 \leqslant i < n$, $max(X_i)$ *and* $min(Y_i)$ *are consistent.*

Proof. GAC implies that every value is consistent. To show the reverse, suppose for all $0 \leqslant i < n$, $max(X_i)$ and $min(Y_i)$ are supported, but the constraint is not GAC. Then there is an inconsistent value. If this value is in some $\mathcal{D}(X_i)$ then any value greater than this value, in particular $max(X_i)$, is inconsistent. Similarly, if the inconsistent value is in some $\mathcal{D}(Y_i)$ then any value less than this value, in particular $min(Y_i)$, is inconsistent. In any case, the bounds are not consistent. □

A constraint is said to be *disentailed* when the constraint is $false$. The next two theorems show when $\vec{X} \leqslant_m \vec{Y}$ is disentailed and what conditions ensure GAC on $\vec{X} \leqslant_m \vec{Y}$.

Theorem 2. $\vec{X} \leqslant_m \vec{Y}$ *is disentailed iff* $\{\!\{\mathtt{floor}(\vec{X})\}\!\} >_m \{\!\{\mathtt{ceiling}(\vec{Y})\}\!\}$.

Proof. $(\Rightarrow)$ Since $\vec{X} \leqslant_m \vec{Y}$ is disentailed, any combination of assignments, including $\vec{X} \leftarrow \mathtt{floor}(\vec{X})$ and $\vec{Y} \leftarrow \mathtt{ceiling}(\vec{Y})$, does not satisfy $\vec{X} \leqslant_m \vec{Y}$. Hence, $\{\!\{\mathtt{floor}(\vec{X})\}\!\} >_m \{\!\{\mathtt{ceiling}(\vec{Y})\}\!\}$.

$(\Leftarrow)$ Any $\vec{x} \in \vec{X}$ is greater than any $\vec{y} \in \vec{Y}$ under the multiset ordering. Hence, $\vec{X} \leqslant_m \vec{Y}$ is disentailed. □

Theorem 3. GAC$(\vec{X} \leqslant_m \vec{Y})$ *iff for all* i *in* $[0, n)$:

$$\{\!\{\mathtt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})\}\!\} \leqslant_m \{\!\{\mathtt{ceiling}(\vec{Y})\}\!\} \tag{1}$$

$$\{\!\{\mathtt{floor}(\vec{X})\}\!\} \leqslant_m \{\!\{\mathtt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)})\}\!\} \tag{2}$$

Proof. $(\Rightarrow)$ As the constraint is GAC, all values have support. In particular, $X_i \leftarrow max(X_i)$ has a support $\vec{x}_1 \in \{\vec{x} \mid x_i = max(X_i) \ \wedge\ \vec{x} \in \vec{X}\}$ and $\vec{y}_1 \in \vec{Y}$ where $\{\!\{\vec{x}_1\}\!\} \leqslant_m \{\!\{\vec{y}_1\}\!\}$. Any $\vec{x}_2 \in \{\vec{x} \mid x_i = max(X_i) \ \wedge\ \vec{x} \in \vec{X}\}$ less than or equal to $\vec{x}_1$, and any $\vec{y}_2 \in \vec{Y}$ greater than or equal to $\vec{y}_1$, under multiset ordering, support $X_i \leftarrow max(X_i)$. In particular, $min\{\vec{x} \mid x_i = max(X_i) \ \wedge\ \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid \vec{y} \in \vec{Y}\}$ support $X_i \leftarrow max(X_i)$. We get $min\{\vec{x} \mid x_i = max(X_i) \ \wedge\ \vec{x} \in \vec{X}\}$ if all the other variables in $\vec{X}$ take their minimums, and we get $max\{\vec{y} \mid \vec{y} \in \vec{Y}\}$ if all the variables in $\vec{Y}$ take their maximums. Hence, $\{\!\{\mathtt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})\}\!\} \leqslant_m \{\!\{\mathtt{ceiling}(\vec{Y})\}\!\}$.

A dual argument holds for the variables of $\vec{Y}$. As the constraint is GAC, $Y_i \leftarrow min(Y_i)$ has a support $\vec{x}_1 \in \vec{X}$ and $\vec{y}_1 \in \{\vec{y} \mid y_i = min(Y_i) \ \wedge\ \vec{y} \in \vec{Y}\}$ where $\{\!\{\vec{x}_1\}\!\} \leqslant_m \{\!\{\vec{y}_1\}\!\}$. Any $\vec{x}_2 \in \vec{X}$ less than or equal to $\vec{x}_1$, and any $\vec{y}_2 \in \{\vec{y} \mid y_i = min(Y_i) \ \wedge\ \vec{y} \in$

$\vec{Y}\}$ greater than or equal to $\vec{y}_1$, in particular $min\{\vec{x} \mid \vec{x} \in \vec{X}\}$ and $max\{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ support $Y_i \leftarrow min(Y_i)$. We get $min\{\vec{x} \mid \vec{x} \in \vec{X}\}$ if all the variables in $\vec{X}$ take their minimums, and we get $max\{\vec{y} \mid y_i = min(Y_i) \wedge \vec{y} \in \vec{Y}\}$ if all the other variables in $\vec{Y}$ take their maximums. Hence, $\{\{\texttt{floor}(\vec{X})\}\} \leqslant_m \{\{\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)})\}\}$.

($\Leftarrow$) Eq. (1) ensures that for all $0 \leqslant i < n$, $max(X_i)$ is supported, and Eq. (2) ensures that for all $0 \leqslant i < n$, $min(Y_i)$ is supported. By Theorem 1, the constraint is GAC. □

In Theorems 2 and 3, we need to check whether two ground vectors are multiset ordered. The following theorem shows that we can do this by lexicographically comparing the occurrence vectors associated with these vectors.

Theorem 4. $\{\{\vec{x}\}\} \leqslant_m \{\{\vec{y}\}\}$ *iff* $occ(\vec{x}) \leqslant_{lex} occ(\vec{y})$.

Proof. ($\Rightarrow$) Suppose $\{\{\vec{x}\}\} = \{\{\vec{y}\}\}$. Then the occurrence vectors associated with $\vec{x}$ and $\vec{y}$ are the same. Suppose $\{\{\vec{x}\}\} <_m \{\{\vec{y}\}\}$. If $max\{\{\vec{x}\}\} < max\{\{\vec{y}\}\}$ then the leftmost index of $\vec{ox} = occ(\vec{x})$ and $\vec{oy} = occ(\vec{y})$ is $max\{\{\vec{y}\}\}$, and we have $ox_{max\{\{\vec{y}\}\}} = 0$ and $oy_{max\{\{\vec{y}\}\}} > 0$. This gives $\vec{ox} <_{lex} \vec{oy}$. If $max\{\{\vec{x}\}\} = max\{\{\vec{y}\}\} = a$ then we eliminate one occurrence of a from each multiset and compare the resulting multisets.

($\Leftarrow$) Suppose $occ(\vec{x}) = occ(\vec{y})$. Then $\{\{\vec{x}\}\}$ and $\{\{\vec{y}\}\}$ contain the same elements with equal occurrences. Suppose $occ(\vec{x}) <_{lex} occ(\vec{y})$. Then a value a occurs more in $\{\{\vec{y}\}\}$ than in $\{\{\vec{x}\}\}$, and the occurrence of any value $b > a$ is the same in both multisets. By deleting all the occurrences of a from $\{\{\vec{x}\}\}$ and the same number of occurrences of a from $\{\{\vec{y}\}\}$, as well as any $b > a$ from both multisets, we get $max\{\{\vec{x}\}\} < max\{\{\vec{y}\}\}$. □

Theorems 2 and 3 together with Theorem 4 yield to the following propositions:

Proposition 1. $\vec{X} \leqslant_m \vec{Y}$ *is disentailed iff* $occ(\texttt{floor}(\vec{X})) >_{lex} occ(\texttt{ceiling}(\vec{Y}))$.

Proposition 2. GAC($\vec{X} \leqslant_m \vec{Y}$) *iff for all* i *in* $[0, n)$:

$$occ(\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})) \leqslant_{lex} occ(\texttt{ceiling}(\vec{Y})) \tag{3}$$

$$occ(\texttt{floor}(\vec{X})) \leqslant_{lex} occ(\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)})) \tag{4}$$

A naive way to enforce GAC on $\vec{X} \leqslant_m \vec{Y}$ is going through every variable in the vectors, constructing the appropriate occurrence vectors, and checking if their bounds satisfy 3 and 4. If they do, then the bound is consistent. Otherwise, we try the nearest bound until we obtain a consistent bound. We can, however, do better than this by building only the vectors $occ(\texttt{floor}(\vec{X}))$ and $occ(\texttt{ceiling}(\vec{Y}))$, and then defining some pointers and Boolean flags on them. This saves us from the repeated construction and traversal of the appropriate occurrence vectors. Another advantage is that we can find consistent bounds without having to explore the values in the domains.

We start by defining our pointers and flags. We write $\vec{ox}$ for $occ(\texttt{floor}(\vec{X}))$, and $\vec{oy}$ for $occ(\texttt{ceiling}(\vec{Y}))$. We assume $\vec{ox}$ and $\vec{oy}$ are indexed from u to l, and $\vec{ox} \leqslant_{lex} \vec{oy}$.[3]

Definition 5. Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, the pointer α is set either to the index in $[u, l]$ such that:

$$ox_\alpha < oy_\alpha \ \wedge$$
$$\forall i \ u \geqslant i > \alpha \,.\, ox_i = oy_i$$

or (if this is not the case) to $-\infty$.

Informally, α points to the most significant index in $[u, l]$ such that $ox_\alpha < oy_\alpha$ and all the variables above it are pairwise equal. If, however, $\vec{ox} = \vec{oy}$ then α points to $-\infty$.

Definition 6. Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, the pointer β is set either to the index in $(\alpha, l]$ such that:

$$ox_\beta > oy_\beta \ \wedge$$
$$\forall i \ \alpha > i > \beta \,.\, ox_i \leqslant oy_i$$

or (if $\alpha \leqslant l$ or for all $\alpha > i \geqslant l$ we have $ox_i \leqslant oy_i$) to $-\infty$.

[3] In the context of occurrence vector indexing, $u..l$ and $[u, l]$ imply $u \geqslant l$. The exact meaning of the these abused notations will be clear from the context.

Informally, β points to the most significant index in $(\alpha, l]$ such that $\vec{ox}_{\beta\to l} >_{lex} \vec{oy}_{\beta\to l}$. If, such an index does not exist, then β points to $-\infty$. Note that we have $\sum_i ox_i = \sum_i oy_i = n$, as $\vec{ox}$ and $\vec{oy}$ are both associated with vectors of length n. Hence, α cannot be l, and we always have $\vec{ox}_{\alpha-1\to l} >_{lex} \vec{oy}_{\alpha-1\to l}$ when $\alpha \neq -\infty$.

Definition 7. Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, the flag γ is *true* iff:

$$\beta \neq -\infty \ \wedge (\beta = \alpha - 1 \ \vee \ \forall i \ \alpha > i > \beta \,.\, ox_i = oy_i)$$

Informally, γ is *true* if $\beta \neq -\infty$, and either β is jut next to α or the sub-vectors between α and β are equal. Otherwise, γ is *false*.

Definition 8. Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, the flag σ is *true* iff:

$$\beta > l \ \wedge \ \vec{ox}_{\beta-1\to l} >_{lex} \vec{oy}_{\beta-1\to l}$$

Informally, σ is *true* if $\beta > l$ and the sub-vectors below β are lexicographically ordered the wrong way. If, however, $\beta \leqslant l$, or the sub-vectors below β are lexicographically ordered, then σ is *false*.

Using α, β, γ, and σ, we can find the tight upper bound for each $\mathcal{D}(X_i)$, as well as the tight lower bound for each $\mathcal{D}(Y_i)$ without having to traverse the occurrence vectors. In the next three theorems, we are concerned with X_i. When looking for a support for a value $v \in \mathcal{D}(X_i)$, we obtain $occ(\texttt{floor}(\vec{X}_{X_i \leftarrow v}))$ by increasing ox_v by 1, and decreasing $ox_{min(X_i)}$ by 1. Since $\vec{ox} \leqslant_{lex} \vec{oy}$, $min(X_i)$ is consistent. We therefore seek support for values greater than $min(X_i)$.

Theorem 5. *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, if $max(X_i) \geqslant \alpha$ and $min(X_i) < \alpha$ then for all $v \in \mathcal{D}(X_i)$:*

1. *if $v > \alpha$ then v is inconsistent;*
2. *if $v < \alpha$ then v is consistent;*
3. *if $v = \alpha$ then v is inconsistent iff:*

$$(ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma \ \wedge \ ox_\beta > oy_\beta + 1) \ \vee$$
$$(ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma \ \wedge \ ox_\beta = oy_\beta + 1 \ \wedge \ \sigma) \ \vee$$
$$(ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) < \beta \ \wedge \ \gamma)$$

Proof. If $min(X_i) < \alpha$ then $\alpha \neq -\infty$ and $\vec{ox} <_{lex} \vec{oy}$. Let v be a value in $\mathcal{D}(X_i)$ greater than α. Increasing ox_v by 1 gives $\vec{ox} >_{lex} \vec{oy}$. By Proposition 2, v is inconsistent. Now let v be less than α. Increasing ox_v by 1 does not change $\vec{ox} <_{lex} \vec{oy}$. By Proposition 2, v is consistent. Is α a tight upper bound? If any of the conditions in item 3 is *true* then we obtain $\vec{ox} >_{lex} \vec{oy}$ by increasing ox_α by 1 and decreasing $ox_{min(X_i)}$ by 1. By Proposition 2, $v = \alpha$ is inconsistent and therefore the largest value which is less than α is the tight upper bound. We now need to show that the conditions of item 3 are exhaustive. If $v = \alpha$ is inconsistent then, by Proposition 2, we obtain $\vec{ox} >_{lex} \vec{oy}$ after increasing ox_α by 1 and decreasing $ox_{min(X_i)}$ by 1. This can happen only if $ox_\alpha + 1 = oy_\alpha$ because otherwise we still have $ox_\alpha < oy_\alpha$. Now, it is important where we decrease an occurrence. If it is above β (but below α as $min(X_i) < \alpha$) then we still have $\vec{ox} <_{lex} \vec{oy}$ because for all $\alpha > i > max\{l-1, \beta\}$, we have $ox_i \leqslant oy_i$. If it is on or below β (when $\beta \neq -\infty$) and γ is *false*, then we still have $\vec{ox} <_{lex} \vec{oy}$ because γ is *false* when $\beta < \alpha - 1$ and $\vec{ox}_{\alpha-1\to\beta+1} <_{lex} \vec{ox}_{\alpha-1\to\beta+1}$. Therefore, it is necessary to have $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge \ min(X_i) \leqslant \beta \ \wedge \ \gamma$ for α to be inconsistent. Two cases arise here. In the first, we have $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma$. Decreasing ox_β by 1 can give $\vec{ox} >_{lex} \vec{oy}$ in two ways: either we still have $ox_\beta > oy_\beta$, or we now have $ox_\beta = oy_\beta$ but the vectors below β are ordered lexicographically the wrong way. Note that decreasing ox_β by 1 cannot give $ox_\beta < oy_\beta$. Therefore, the first case results in two conditions for α to be inconsistent: $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma \ \wedge \ ox_\beta > oy_\beta + 1$ or $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma \ \wedge \ ox_\beta = oy_\beta + 1 \ \wedge \ \sigma$. Now consider the second case, where we have $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge \ min(X_i) < \beta \ \wedge \ \gamma$. Decreasing $ox_{min(X_i)}$ by 1 gives $\vec{ox} >_{lex} \vec{oy}$. Hence, if $v = \alpha$ is inconsistent then we have either of the three conditions. $\square$

Theorem 6. *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, if $max(X_i) < \alpha$ then $max(X_i)$ is the tight upper bound.*

Proof. If $max(X_i) < \alpha$ then we have $\alpha \neq -\infty$ and $\vec{ox} <_{lex} \vec{oy}$. Increasing $ox_{max(X_i)}$ by 1 does not change this. By Proposition 2, $max(X_i)$ is consistent. $\square$

Theorem 7. *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, if $min(X_i) \geqslant \alpha$ then $min(X_i)$ is the tight upper bound.*

Proof. Any $v > min(X_i)$ in $\mathcal{D}(X_i)$ is greater than α. Increasing ox_v by 1 gives $\vec{ox} >_{lex} \vec{oy}$. By Proposition 2, any $v > min(X_i)$ in $\mathcal{D}(X_i)$ is inconsistent. □

In the next four theorems, we are concerned with Y_i. When looking for a support for a value $v \in \mathcal{D}(Y_i)$, we obtain $occ(\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow v}))$ by increasing oy_v by 1, and decreasing $oy_{max(Y_i)}$ by 1. Since $\vec{ox} \leqslant_{lex} \vec{oy}$, $max(Y_i)$ is consistent. We therefore seek support for values less than $max(Y_i)$.

Theorem 8. *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, if $max(Y_i) = \alpha$ and $min(Y_i) \leqslant \beta$ then for all $v \in \mathcal{D}(Y_i)$*

1. *if $v > \beta$ then v is consistent;*
2. *if $v < \beta$ then v is inconsistent iff $ox_\alpha + 1 = oy_\alpha \wedge \gamma$;*
3. *if $v = \beta$ then v is inconsistent iff:*

$$(ox_\alpha + 1 = oy_\alpha \ \wedge\ \gamma \ \wedge\ ox_\beta > oy_\beta + 1) \ \vee$$
$$(ox_\alpha + 1 = oy_\alpha \ \wedge\ \gamma \ \wedge\ ox_\beta = oy_\beta + 1 \ \wedge\ \sigma)$$

Proof. If $max(Y_i) = \alpha$ and $min(Y_i) \leqslant \beta$ then $\alpha \neq -\infty$, $\beta \neq -\infty$, and $\vec{ox} <_{lex} \vec{oy}$. Let v be a value in $\mathcal{D}(Y_i)$ greater than β. Increasing oy_v by 1 and decreasing oy_α by 1 does not change $\vec{ox} <_{lex} \vec{oy}$. This is because for all $\alpha > i > \beta$, we have $ox_i \leqslant oy_i$. Even if now $\vec{ox}_{\alpha \to v+1} = \vec{oy}_{\alpha \to v+1}$, at v we have $ox_v < oy_v$. By Proposition 2, v is consistent. Now let v be less than β. If the condition in item 2 is *true* then we obtain $\vec{ox} >_{lex} \vec{oy}$ by decreasing oy_α by 1 and increasing oy_v by 1. By Proposition 2, v is inconsistent. We now need to show that this condition is exhaustive. If v is inconsistent then by Proposition 2, we obtain $\vec{ox} >_{lex} \vec{oy}$ after decreasing oy_α by 1 and increasing oy_v by 1. This is in fact the same as obtaining $\vec{ox} >_{lex} \vec{oy}$ after increasing ox_α by 1 and decreasing ox_v by 1. We have already captured this case in the last condition of item 3 in Theorem 5. Hence, it is necessary to have $ox_\alpha + 1 = oy_\alpha \ \wedge\ \gamma$ for v to be inconsistent. What about β then? If any of the conditions in item 3 is *true* then we obtain $\vec{ox} >_{lex} \vec{oy}$ by decreasing oy_α by 1 and increasing oy_β by 1. By Proposition 2, $v = \beta$ is inconsistent. In this case, the values less than β are also inconsistent. Therefore, the smallest value which is greater than β is the tight lower bound. We now need to show that the conditions of item 3 are exhaustive. If $v = \beta$ is inconsistent then by Proposition 2, we obtain $\vec{ox} >_{lex} \vec{oy}$ after decreasing oy_α by 1 and increasing oy_β by 1. This is the same as obtaining $\vec{ox} >_{lex} \vec{oy}$ after increasing ox_α by 1 and decreasing ox_β by 1. We have captured this case in the first two conditions of item 3 in Theorem 5. Hence, if $v = \beta$ is inconsistent then we have either $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge\ \gamma \ \wedge\ ox_\beta > oy_\beta + 1$ or $ox_{\alpha+1} + 1 = oy_\alpha \ \wedge\ \gamma \ \wedge\ ox_\beta = oy_\beta + 1 \ \wedge\ \sigma$. □

Theorem 9. *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, if $max(Y_i) = \alpha$ and $min(Y_i) > \beta$ then $min(Y_i)$ is the tight lower bound.*

Proof. If $max(Y_i) = \alpha$ then $\alpha \neq -\infty$ and $\vec{ox} <_{lex} \vec{oy}$. Increasing $oy_{min(Y_i)}$ by 1 and decreasing oy_α by 1 does not change $\vec{ox} <_{lex} \vec{oy}$. This is because for all $\alpha > i > max\{l-1, \beta\}$, we have $ox_i \leqslant oy_i$. Even if now $\vec{ox}_{\alpha \to min(Y_i)+1} = \vec{oy}_{\alpha \to min(Y_i)+1}$, at $min(Y_i)$ we have $ox_{min(Y_i)} < oy_{min(Y_i)}$. By Proposition 2, $min(Y_i)$ is consistent. □

Theorem 10. *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, if $max(Y_i) < \alpha$ then $min(Y_i)$ is the tight lower bound.*

Proof. If $max(Y_i) < \alpha$ then we have $\alpha \neq -\infty$ and $\vec{ox} <_{lex} \vec{oy}$. Decreasing $oy_{max(Y_i)}$ by 1 does not change this. By Proposition 2, $min(Y_i)$ is consistent. □

Theorem 11. *Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} \leqslant_{lex} \vec{oy}$, if $max(Y_i) > \alpha$ then $max(Y_i)$ is the tight lower bound.*

Proof. Decreasing $oy_{max(Y_i)}$ by 1 gives $\vec{ox} >_{lex} \vec{oy}$. By Proposition 2, any $v < max(Y_i)$ in $\mathcal{D}(Y_i)$ is inconsistent. □

4.3. Algorithm details and theoretical properties

In this subsection, we first explain `MsetLeq` as well as prove that it is correct and complete. We then discuss its time complexity.

The algorithm is based on Theorems 5–11. The pointers and flags are recomputed every time the algorithm is called, as maintaining them incrementally in an easy way is not obvious. Fortunately, incremental maintenance of the occurrence vectors is trivial. When the minimum value in some $\mathcal{D}(X_i)$ changes, we update $\vec{ox}$ by incrementing the entry corresponding to new $min(X_i)$ by 1, and decrementing the entry corresponding to old $min(X_i)$ by 1. Similarly, when the maximum value

Data : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$

Result : $occ(\texttt{floor}(\vec{X}))$ and $occ(\texttt{ceiling}(\vec{Y}))$ are initialised, GAC($\vec{X} \leq_m \vec{Y}$)

1 $l := min(\{\!\{\texttt{floor}(\vec{X})\}\!\} \cup \{\!\{\texttt{floor}(\vec{Y})\}\!\})$;
2 $u := max(\{\!\{\texttt{ceiling}(\vec{X})\}\!\} \cup \{\!\{\texttt{ceiling}(\vec{Y})\}\!\})$;
3 $\vec{ox} := occ(\texttt{floor}(\vec{X}))$;
4 $\vec{oy} := occ(\texttt{ceiling}(\vec{Y}))$;
5 `MsetLeq`;

Algorithm 1. `Initialise`.

in some $\mathcal{D}(Y_i)$ changes, we update $\vec{oy}$ by incrementing the entry corresponding to new $max(Y_i)$ by 1, and decrementing the entry corresponding to old $max(Y_i)$ by 1.

When the constraint is first posted, we need to initialise the occurrence vectors, and call the filtering algorithm `MsetLeq` to establish the generalised arc-consistent state with the initial values of the occurrence vectors. In Algorithm 1, we show the steps of this initialisation.

Theorem 12. `Initialise` *initialises $\vec{ox}$ and $\vec{oy}$ correctly. Then it either establishes failure if $\vec{X} \leqslant_m \vec{Y}$ is disentailed, or prunes all inconsistent values from $\vec{X}$ and $\vec{Y}$ to ensure* GAC($\vec{X} \leqslant_m \vec{Y}$).

Proof. `Initialise` first computes the most and the least significant indices of the occurrence vectors as u and l (lines 1 and 2). An occurrence vector $occ(\vec{x})$ associated with $\vec{x}$ is indexed in decreasing order of significance from $max\{\!\{\vec{x}\}\!\}$ to $min\{\!\{\vec{x}\}\!\}$. Our occurrence vectors are associated with $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{Y})$ but they are also used for checking support for $max(X_i)$ and $min(Y_i)$ for all $0 \leqslant i < n$. We therefore need to make sure that there are corresponding entries. Also, to be able to compare two occurrence vectors, they need to start and end with the occurrence of the same value. Therefore, u is $max(\{\!\{\texttt{ceiling}(\vec{X})\}\!\} \cup \{\!\{\texttt{ceiling}(\vec{Y})\}\!\})$ and l is $min(\{\!\{\texttt{floor}(\vec{X})\}\!\} \cup \{\!\{\texttt{floor}(\vec{Y})\}\!\})$.

Using these indices, a pair of vectors $\vec{ox}$ and $\vec{oy}$ of length $u - l + 1$ are constructed and each entry in these vectors are set to 0. Then, $ox_{min(X_i)}$ and $oy_{max(Y_i)}$ are incremented by 1 for all $0 \leqslant i < n$. Now, for all $u \geqslant v \geqslant l$, ox_v is the number of occurrences of v in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$. Similarly, for all $u \geqslant v \geqslant l$, oy_v is the number of occurrences of v in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. This gives us $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ (lines 3 and 4). Finally, in line 5, `Initialise` calls the filtering algorithm `MsetLeq` which either establishes failure if $\vec{X} \leqslant_m \vec{Y}$ is disentailed, or prunes all inconsistent values from $\vec{X}$ and $\vec{Y}$ to ensure GAC($\vec{X} \leqslant_m \vec{Y}$). □

Note that when $\vec{X} \leqslant_m \vec{Y}$ is GAC, every value in $\mathcal{D}(X_i)$ is supported by $\langle min(X_0), \ldots, min(X_{i-1}), min(X_{i+1}), \ldots, min(X_{n-1})\rangle$, and $\langle max(Y_0), \ldots, max(Y_{n-1})\rangle$. Similarly, every value in $\mathcal{D}(Y_i)$ is supported by $\langle min(X_0), \ldots, min(X_{n-1})\rangle$ and $\langle max(Y_0), \ldots, max(Y_{i-1}), max(Y_{i+1}), \ldots, max(Y_{n-1})\rangle$. So, `MsetLeq` is also called by the event handler whenever $min(X_i)$ or $max(Y_i)$ of some i in $[0, n)$ changes.

In Algorithm 2, we show the steps of `MsetLeq`. Since $\vec{ox}$ and $\vec{oy}$ are maintained incrementally, the algorithm first sets the pointers and flags in line **A1** via `SetPointersAndFlags` using the current state of these vectors.

Theorem 13. `SetPointersAndFlags` *either sets α, β, γ, and σ as per their definitions, or establishes failure as $\vec{X} \leqslant_m \vec{Y}$ is disentailed.*

Proof. Line 2 of `SetPointersAndFlags` traverses $\vec{ox}$ and $\vec{oy}$, starting at index u, until either it reaches the end of the vectors (because $\vec{ox} = \vec{oy}$), or it finds an index i where $ox_i \neq oy_i$. In the first case, α is set to $-\infty$ (line 4) as per Definition 5. In the second case, α is set to i only if $ox_i < oy_i$ (line 5). This is correct by Definition 5 and means that $\vec{ox} <_{lex} \vec{oy}$. If, however, $ox_i > oy_i$ then we have $\vec{ox} >_{lex} \vec{oy}$. By Proposition 1, $\vec{X} \leqslant_m \vec{Y}$ is disentailed and thus `SetPointersAndFlags` terminates with failure (line 3). This also triggers the filtering algorithm to fail.

If $\alpha \leqslant l$ then β is set to $-\infty$ (line 6) as per Definition 6. Otherwise, the vectors are traversed in lines 9–11, starting at index $\alpha - 1$, until either the end of the vectors are reached (because $\vec{ox}_{\alpha-1\to l} \leqslant_{lex} \vec{oy}_{\alpha-1\to l}$), or an index j where $ox_j > oy_j$ is found. In the first case, β is set to $-\infty$ (line 12), and in the second case, β is set j (line 13) as per Definition 6. During this traversal, the Boolean flag *temp* is set to *true* iff $\vec{ox}_{\alpha-1\to max\{l,\beta+1\}} = \vec{oy}_{\alpha-1\to max\{l,\beta+1\}}$. In lines 14 and 15, γ is set to *true* iff $\beta \neq -\infty$, and either $\beta = \alpha - 1$ or *temp* is *true* (because $\vec{ox}_{\alpha-1\to\beta+1} = \vec{oy}_{\alpha-1\to\beta+1}$). This is correct by Definition 7.

In line 14, σ is initialised to $false$. If $\beta \leqslant l$ then σ remains $false$ (line 16) as per Definition 8. Otherwise, the vectors are traversed in line 18, starting at index $\beta - 1$, until either the end of the vectors are reached (because $\vec{ox}_{\beta-1\to l} = \vec{oy}_{\beta-1\to l}$), or an index k where $ox_k \neq oy_k$ is found. In the first case, σ remains $false$ as per Definition 8. In the second case, σ is set to *true* only if $ox_k > oy_k$ (line 19). This is correct by Definition 8 and means that $\vec{ox}_{\beta-1\to l} >_{lex} \vec{oy}_{\beta-1\to l}$. If, however, $ox_k < oy_k$ then σ remains $false$ as per Definition 8. □

We now analyse the rest of `MsetLeq`, where the tight upper bound for X_i and the tight lower bound for Y_i, for all $0 \leqslant i < n$, are sought.

```
     Data    : ⟨X_0, X_1, ..., X_{n-1}⟩, ⟨Y_0, Y_1, ..., Y_{n-1}⟩
     Result  : GAC(X⃗ ≤_m Y⃗)
A1  SetPointersAndFlags;
B1  foreach i ∈ [0, n) do
B2      if min(X_i) ≠ max(X_i) then
B3          if min(X_i) ≥ α then setMax(X_i, min(X_i));
B4          if max(X_i) ≥ α ∧ min(X_i) < α then
B5              setMax(X_i, α);
B6              if ox_α + 1 = oy_α ∧ min(X_i) = β ∧ γ then
B7                  if ox_β = oy_β + 1 then
B8                      if σ then setMax(X_i, α − 1);
                    else
B9                      setMax(X_i, α − 1);
                    end
                end
B10             if ox_α + 1 = oy_α ∧ min(X_i) < β ∧ γ then
B11                 setMax(X_i, α − 1);
                end
            end
        end
    end
C1  foreach i ∈ [0, n) do
C2      if min(Y_i) ≠ max(Y_i) then
C3          if max(Y_i) > α then setMin(Y_i, max(Y_i));
C4          if max(Y_i) = α ∧ min(Y_i) ≤ β then
C5              if ox_α + 1 = oy_α ∧ γ then
C6                  setMin(Y_i, β);
C7                  if ox_β = oy_β + 1 then
C8                      if σ then setMin(Y_i, β + 1);
                    else
C9                      setMin(Y_i, β + 1);
                    end
                end
            end
        end
    end
```

Algorithm 2. `MsetLeq`.

Theorem 14. `MsetLeq` *either establishes failure if $\vec{X} \leqslant_m \vec{Y}$ is disentailed, or prunes all inconsistent values from $\vec{X}$ and $\vec{Y}$ to ensure* GAC($\vec{X} \leqslant_m \vec{Y}$).

Proof. If $\vec{X} \leqslant_m \vec{Y}$ is not disentailed then we have $\vec{ox} \leqslant_{lex} \vec{oy}$ by Proposition 1. This means that $min(X_i)$ and $max(Y_i)$ for all $0 \leqslant i < n$ are consistent by Proposition 2. The algorithm therefore seeks the tight upper bound for X_i only if $max(X_i) > min(X_i)$ (lines **B2–B11**), and similarly the tight lower bound for Y_i only if $min(Y_i) < max(Y_i)$ (lines **C2–C9**).

For each $\mathcal{D}(X_i)$: (1) If $min(X_i) \geqslant \alpha$ then all values greater than $min(X_i)$ are pruned, giving $min(X_i)$ as the tight upper bound (line **B3**). This is correct by Theorem 7. (2) If $max(X_i) \geqslant \alpha \ \wedge \ min(X_i) < \alpha$ then:

- all values greater than α are pruned (line **B5**);
- α is pruned if $ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma \ \wedge \ ox_\beta > oy_\beta + 1$ (line **B9**), or $ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) = \beta \ \wedge \ \gamma \ \wedge \ ox_\beta = oy_\beta + 1 \ \wedge \ \sigma$ (line **B8**), or $ox_\alpha + 1 = oy_\alpha \ \wedge \ min(X_i) < \beta \ \wedge \ \gamma$ (line **B11**).

All the values less than α remain in the domain. By Theorem 5, all the inconsistent values are removed. (3) If, however, $max(X_i) < \alpha$ then $max(X_i)$ is the tight upper bound by Theorem 6, and thus no pruning is necessary.

For each $\mathcal{D}(Y_i)$: (1) If $max(Y_i) > \alpha$ then all values less than $max(Y_i)$ are pruned, giving $max(Y_i)$ as the tight lower bound (line **C3**). This is correct by Theorem 11. (2) If $max(Y_i) = \alpha \ \wedge \ min(Y_i) \leqslant \beta$ then:

- all values less than β are pruned if $ox_\alpha + 1 = oy_\alpha \ \wedge \ \gamma$ (line **C6**);
- β is pruned if $ox_\alpha + 1 = oy_\alpha \ \wedge \ \gamma \ \wedge \ ox_\beta > oy_\beta + 1$ (line **C9**) or $ox_\alpha + 1 = oy_\alpha \ \wedge \ \gamma \ \wedge \ ox_\beta = oy_\beta + 1 \ \wedge \ \sigma$ (line **C8**).

```
i := u;
while i ≥ l ∧ ox_i = oy_i do i := i − 1;
if i ≥ l ∧ ox_i > oy_i then fail;
else if i = l − 1 then α := −∞;
else α := i;
if α ≤ l then β := −∞;
else if α > l then
    j := α − 1, temp := true;
    while j ≥ l ∧ ox_j ≤ oy_j do
        if ox_j < oy_j then temp := false;
        j := j − 1;
    end
    if j = l − 1 then β := −∞;
    else β := j;
end
γ := false, σ := false;
if β ≠ −∞ ∧ temp then γ := true;
if β > l then
    k := β − 1;
    while k ≥ l ∧ ox_k = oy_k do k := k − 1;
    if k ≥ l ∧ ox_k > oy_k then σ := true;
end
```

Procedure. `SetPointersAndFlags`.

All the values greater than β remain in the domain. By Theorem 8, all the inconsistent values are removed. (3) If, however, $max(Y_i) = \alpha \ \wedge \ min(Y_i) > \beta$ or $max(Y_i) < \alpha$ then $min(Y_i)$ is the tight lower bound by Theorems 9 and 10, and thus no pruning is needed.

`MsetLeq` is a correct and complete filtering algorithm, as it either establishes failure if $\vec{X} \leqslant_m \vec{Y}$ is disentailed, or prunes all inconsistent values from $\vec{X}$ and $\vec{Y}$ to ensure GAC($\vec{X} \leqslant_m \vec{Y}$). □

When we prune a value, we do not need to check recursively that previous support remains. The algorithm tightens $max(X_i)$ and $min(Y_i)$ without touching $min(X_i)$ and $max(Y_i)$, for all $0 \leqslant i < n$, which provide support for the values in the vectors. The exception is if a domain wipe out occurs. As the constraint is not disentailed, we have $\vec{ox} \leqslant_{lex} \vec{oy}$. This means $min(X_i)$ and $max(Y_i)$ for all $0 \leqslant i < n$ are supported. Hence, the prunings of the algorithm cannot cause any domain wipe-out.

The algorithm works also when the vectors are of different length as we build and reason about the occurrence vectors as opposed to the original vectors. Also, we do not assume that the original vectors are of the same length when we set the pointer β.

The algorithm corrects a mistake that appears in [13]. We have noticed that in [13] we do not always prune the values greater than α when we have $max(X_i) \geqslant \alpha$ and $min(X_i) < \alpha$. As shown above, this algorithm is correct and complete.

To improve the time complexity, we assume that domains are transformed so that their union is a continuous interval. Suppose, for instance, that we have variables with domains $\{1, 5\}$, $\{1, 100\}$ and $\{5, 100\}$. This transformation normalises the domains to $\{1, 2\}$, $\{1, 3\}$ and $\{2, 3\}$. This technique is widely used (see for instance [19]) and does not change the worst-case complexity of our propagator. It gives us a tighter upper bound on the complexity of our propagator in terms of the number of distinct values as compared to the difference between the largest and smallest values.

Theorem 15. `Initialise` *runs in time $O(n + d)$, where d is the number of distinct values.*

Proof. `Initialise` first constructs $\vec{ox}$ and $\vec{oy}$ of length d where each entry is zero, and then increments $ox_{min(X_i)}$ and $oy_{max(Y_i)}$ by 1 for all $0 \leqslant i < n$. Hence, the complexity of initialisation is $O(n + d)$. □

Theorem 16. `MsetLeq` *runs in time $O(nb + d)$, where b is the cost of adjusting the bounds of a variable, and d is the number of different values.*

Proof. `MsetLeq` does not construct $\vec{ox}$ and $\vec{oy}$, but rather uses their most up-to-date states. `MsetLeq` first sets the pointers and flags which are defined on $\vec{ox}$ and $\vec{oy}$. In the worst case both vectors are traversed once from the beginning until the end, which gives an $O(d)$ complexity. Next, the algorithm goes through every variable in the original vectors $\vec{X}$ and $\vec{Y}$ to check for support. Deciding the tight bound for each variable is a constant time operation, but the cost of adjusting the bound is b. Since we have $O(n)$ variables, the complexity of the algorithm is $O(nb + d)$. □

If $d \ll n$ then the algorithm runs in time $O(nb)$. Since a multiset is a set with possible repetitions, we expect that the number of distinct values in a multiset is often less than the cardinality of the multiset, giving us a linear time filtering algorithm.

5. Multiset ordering with large domains

`MsetLeq` is a linear time algorithm in the n given that $d \ll n$. If instead we have $n \ll d$ then the complexity of the algorithm is $O(d)$, dominated by the cost of the construction of the occurrence vectors and the initialisation of the pointers and flags. This can happen, for instance, when the vectors being multiset ordered are variables in the occurrence representation of a multiset [20]. Is there then an alternative way of propagating the multiset ordering constraint whose complexity is independent of the domains?

5.1. Remedy

In case d is a large number, it could be costly to construct the occurrence vectors. We can instead sort $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{Y})$, and compute α, β, γ, σ, and the number of occurrences of α and β in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ and $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ as if we had the occurrence vectors by scanning these sorted vectors. This information is all we need to find support for the bounds of the variables. Let us illustrate this on an example. To simplify presentation, we assume that the vectors are of the same length. Consider $\vec{X} \leqslant_m \vec{Y}$ where $\vec{sx} = sort(\texttt{floor}(\vec{X}))$ and $\vec{sy} = sort(\texttt{ceiling}(\vec{Y}))$ are as follows:

$$\begin{array}{rcl} \vec{sx} & = & \langle 5,\ 4,\ 3,\ 2,\ 2,\ 2,\ 2,\ 1\rangle \\ \vec{sy} & = & \langle 5,\ 4,\ 4,\ 4,\ 3,\ 1,\ 1,\ 1\rangle \end{array}$$

We traverse $\vec{sx}$ and $\vec{sy}$ until we find an index i such that $sx_i < sy_i$, and for all $0 \leqslant t < i$ we have $sx_t = sy_t$. In our example, i is 2:

$$\begin{array}{rcllllllll} & & & & \downarrow i & & & & & \\ \vec{sx} & = & \langle 5, & 4, & 3, & 2, & 2, & 2, & 2, & 1\rangle \\ \vec{sy} & = & \langle 5, & 4, & 4, & 4, & 3, & 1, & 1, & 1\rangle \end{array}$$

This means that the number occurrences of any value greater than sy_i are equal in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ and in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$, but there are more occurrence of sy_i in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ than in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$. That is, $ox_5 = oy_5$ and $ox_4 < oy_4$. By Definition 5, α is equal to 4. We now move only along $\vec{sy}$ until we find an index j such that $sy_j \neq sy_{j-1}$, so that we reason about the number of occurrences of the smaller values. In our example, j is 4:

$$\begin{array}{rcllllllll} & & & & \downarrow i & & & & & \\ \vec{sx} & = & \langle 5, & 4, & 3, & 2, & 2, & 2, & 2, & 1\rangle \\ \vec{sy} & = & \langle 5, & 4, & 4, & 4, & 3, & 1, & 1, & 1\rangle \\ & & & & & & \uparrow j & & & \end{array}$$

We here initialise γ to *true*, and start traversing $\vec{sx}$ and $\vec{sy}$ simultaneously. We have $sx_i = sy_j = 3$. This adds 1 to ox_3 and oy_3, keeping $\gamma = true$. We move one index ahead in both vectors by incrementing i to 3 and j to 5:

$$\begin{array}{rcllllllll} & & & & & \downarrow i & & & & \\ \vec{sx} & = & \langle 5, & 4, & 3, & 2, & 2, & 2, & 2, & 1\rangle \\ \vec{sy} & = & \langle 5, & 4, & 4, & 4, & 3, & 1, & 1, & 1\rangle \\ & & & & & & & \uparrow j & & \end{array}$$

We now have $sx_i > sy_j$, which suggests that sx_i occurs at least once in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ but does not occur in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. That is, $ox_2 > 0$ and $oy_2 = 0$. By Definition 6, β points to 2. This does not change that γ is *true*. We now move only along $\vec{sx}$ by incrementing i until we find $sx_i \neq sx_{i-1}$, so that we reason about the number of occurrences of the smaller values:

$$\begin{array}{rcllllllll} & & & & & & & & & \downarrow i \\ \vec{sx} & = & \langle 5, & 4, & 3, & 2, & 2, & 2, & 2, & 1\rangle \\ \vec{sy} & = & \langle 5, & 4, & 4, & 4, & 3, & 1, & 1, & 1\rangle \\ & & & & & & & \uparrow j & & \end{array}$$

With the new value of i, we have $sx_i = sy_i = 1$. This increases both ox_1 and oy_1 by one. Reaching the end of only $\vec{sx}$ hints the following: either 1 occurs more than once in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$, or it occurs once but there are values in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ less than 1 and they do not occur in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$. By Definition 8, γ is $false$.

Finally, we need to know the number of occurrences of α and β in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ and $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. Since we already know what α and β are, another scan of $\vec{sx}$ and $\vec{sy}$ gives us the needed information: for all $0 \leqslant i < n$, we increment ox_α (resp. ox_β) by 1 if $sx_i = \alpha$ (resp. $sx_i = \beta$), and also oy_α (resp. oy_β) by 1 if $sy_i = \alpha$ (resp. $sy_i = \beta$).

Data : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$

Result : $sort(\texttt{floor}(\vec{X}))$ and $sort(\texttt{ceiling}(\vec{Y}))$ are initialised, GAC($\vec{X} \leq_m \vec{Y}$)

$\vec{sx} := sort(\texttt{floor}(\vec{X}))$;

$\vec{sy} := sort(\texttt{ceiling}(\vec{Y}))$;

`MsetLeq`;

Algorithm 4. `Initialise`.

5.2. An alternative filtering algorithm

As witnessed in the previous section, it suffices to sort $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{Y})$, and scan the sorted vectors to compute α, β, γ, σ, ox_α, oy_α, ox_β, and oy_β. We can then directly reuse lines **B1–B11** and **C1–C9** of `MsetLeq` to obtain a new filtering algorithm. As a result, we need to change only `Initialise` and `SetPointersAndFlags`.

In Algorithm 4, we show the new `Initialise`. Instead of constructing a pair of occurrence vectors associated with $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{Y})$, we now sort $\texttt{floor}(\vec{X})$ and $\texttt{ceiling}(\vec{Y})$ and then call `MsetLeq`.

Similar to the original algorithm, we recompute the pointers and flags every time we call the filtering algorithm. Maintaining the sorted vectors incrementally is trivial. When the minimum value in some $\mathcal{D}(X_i)$ changes, we update $\vec{sx}$ by inserting the new $min(X_i)$ into, and removing the old $min(X_i)$ from $\vec{sx}$. Similarly, when the maximum value in some $\mathcal{D}(Y_i)$ changes, we update $\vec{sy}$ by inserting the new $max(Y_i)$ into, and removing the old $max(Y_i)$ from $\vec{sy}$. Since these vectors need to remain sorted after the update, such modifications require binary search. The cost of incrementality thus increases from $O(1)$ to $O(log(n))$ compared to the original filtering algorithm.

Given the most up-to-date $\vec{sx}$ and $\vec{sy}$, how do we set our pointers and flags? In line 2 of our new `SetPointersAnd-Flags`, we traverse $\vec{sx}$ and $\vec{sy}$, starting at index 0, until either we reach the end of the vectors (because the vectors are equal), or we find an index i where $sx_i \neq sy_i$. In the first case, we first set α and β to $-\infty$, and γ and σ to $false$, and then return (line 4). In the second case, if $sx_i > sy_i$ then disentailment is detected and `SetPointersAndFlags` terminates with failure (line 3). The reason of the return and failure is due to the following theoretical result.

Theorem 17. *$occ(\vec{x}) \leqslant_{lex} occ(\vec{y})$ iff $sort(\vec{x}) \leqslant_{lex} sort(\vec{y})$.*

Proof. ($\Rightarrow$) If $occ(\vec{x}) <_{lex} occ(\vec{y})$ then a value a occurs more in $\{\!\{\vec{y}\}\!\}$ than in $\{\!\{\vec{x}\}\!\}$, and the occurrence of any value $b > a$ is the same in both multisets. By deleting all the occurrences of a from $\{\!\{\vec{x}\}\!\}$ and the same number of occurrences of a from $\{\!\{\vec{y}\}\!\}$, as well as any $b > a$ from both multisets, we get $max\{\!\{\vec{x}\}\!\} < max\{\!\{\vec{y}\}\!\}$. Since the leftmost values in $sort(\vec{x})$ and $sort(\vec{y})$ are $max\{\!\{\vec{x}\}\!\}$ and $max\{\!\{\vec{y}\}\!\}$ respectively, we have $sort(\vec{x}) <_{lex} sort(\vec{y})$. If $occ(\vec{x}) = occ(\vec{y})$ then we have $\{\!\{\vec{x}\}\!\} = \{\!\{\vec{y}\}\!\}$. By sorting the elements in $\vec{x}$ and $\vec{y}$, we obtain the same vectors. Hence, $sort(\vec{x}) = sort(\vec{y})$.

($\Leftarrow$) Suppose $\vec{ox} = occ(\vec{x})$, $\vec{oy} = occ(\vec{y})$, $\vec{sx} = sort(\vec{x})$, $\vec{sy} = sort(\vec{y})$, and we have $\vec{sx} = \vec{sy}$. Then $\{\!\{\vec{x}\}\!\}$ and $\{\!\{\vec{y}\}\!\}$ contain the same elements with equal occurrences. Hence, $\vec{ox} = \vec{oy}$. Suppose $\vec{sx} <_{lex} \vec{sy}$. If $sx_0 < sy_0$ then the leftmost index of $\vec{ox}$ and $\vec{oy}$ is sy_0, and we have $ox_{sy_0} = 0$ and $oy_{sy_0} > 0$. This gives $\vec{ox} <_{lex} \vec{oy}$. If $sx_0 = sy_0 = a$ then we eliminate one occurrence of a from $\{\!\{\vec{x}\}\!\}$ and $\{\!\{\vec{y}\}\!\}$, and compare the resulting multisets. □

Hence, whenever we have $\vec{sx} \geqslant_{lex} \vec{sy}$, we proceed as if we had $occ(\texttt{floor}(\vec{X})) \geqslant_{lex} occ(\texttt{ceiling}(\vec{Y}))$. But then what do we do if we have $\vec{sx} <_{lex} \vec{sy}$? In line 5, we have $sx_i < sy_i$ and $sx_t = sy_t$ for all $0 \leqslant t < i$. This means that the number occurrences of any value greater than sy_i are equal in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ and in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$, but there are more occurrence of sy_i in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ than in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$. Therefore, we here set α to sy_i.

After initialising γ to $true$ in line 6, we start seeking a value for β. For the sake of simplicity, we here assume our original vectors are of same length. Hence, β cannot be $-\infty$ as α is not $-\infty$. In line 8, we traverse $\vec{sy}$, starting at index $i+1$, until either we reach the end of the vector (because all the remaining values in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ are sy_i), or we find an index j such that $sy_j \neq sy_{j-1}$. In the first case, we set β to sx_i (line 9) because sx_i occurs at least once in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ but does not occur in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. Since no value between α and β occur more in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ than in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$, γ remains $true$. In the second case, sy_j gives us the next largest value in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. In lines 11–14, we traverse $\vec{sx}$ starting from i, and $\vec{sy}$ starting from j. If $sx_i > sy_j$ then we set β to sx_i (line 12) because sx_i occurs more in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ than in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. Having found the value of β, we here exit the while loop using **break**. If $sx_i < sy_j$ then sy_j occurs more in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ than in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$. Since we are still looking for a value for β, we set γ to $false$ (line 13). We then move to the next index in $\vec{sy}$ to find the next largest value in $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$. If $sx_i = sy_j$ then we move to the next index both in $\vec{sx}$ and $\vec{sy}$ to find the next largest values in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ and $\{\!\{\texttt{ceiling}(\vec{Y})\}\!\}$ (line 14). As j is at least one index ahead of i, j can reach to n before i does during this traversal. In such a case, we set β to sx_i (line 15) due to the same reasoning as in line 12.

The process of finding the value of σ (lines 16–25) is very similar to that of β. In line 17, we traverse $\vec{sx}$, starting at index $i+1$, until either we reach the end of the vector (because all the remaining values in $\{\!\{\texttt{floor}(\vec{X})\}\!\}$ are β), or we find an index k such that $sx_k \neq sx_{k-1}$. In the first case, we set σ to $false$ (line 18) because either sy_j occurs at least

```
1  i := 0;
2  while i < n ∧ sx_i = sy_i do i := i + 1;
3  if i < n ∧ sx_i > sy_i then fail;
4  else if i = n then α := −∞, β := −∞, γ := false, σ := false, return;
5  else α := sy_i;
6  γ := true;
7  j := i + 1;
8  while j < n ∧ sy_j = sy_{j−1} do j := j + 1;
9  if j = n then β := sx_i;
10 else if j < n then
11 |   while i < n ∧ j < n do
12 |   |   if sx_i > sy_j then β := sx_i, break;
13 |   |   if sx_i < sy_j then γ := false, j := j + 1;
14 |   |   if sx_i = sy_j then i := i + 1, j := j + 1;
   |   end
15 |   if j = n then β := sx_i;
   end
16 k := i + 1;
17 while k < n ∧ sx_k = sx_{k−1} do k := k + 1;
18 if k = n then σ := false;
19 else if k < n then
20 |   while k < n ∧ j < n do
21 |   |   if sx_k > sy_j then σ := true, break;
22 |   |   if sx_k < sy_j then σ := false, break;
23 |   |   if sx_k = sy_j then k := k + 1, j := j + 1;
   |   end
24 |   if k = n then σ := false ;
25 |   else if j = n then
   |   |   σ := true;
   |   end
   end
26 i := 0, ox_α = 0, oy_α = 0, ox_β = 0, oy_β = 0;
27 foreach i ∈ [0, n) do
28 |   if sx_i = α then ox_α := ox_α + 1;
29 |   if sx_i = β then ox_β := ox_β + 1;
30 |   if sy_i = α then oy_α := oy_α + 1;
31 |   if sy_i = β then oy_β := oy_β + 1;
   end
```

Procedure. `SetPointersAndFlags`.

once in {{`ceiling`($\vec{Y}$)}} but does not occur in {{`floor`($\vec{X}$)}} (due to line 12), or there are no values less than β both in {{`floor`($\vec{X}$)}} and in {{`ceiling`($\vec{Y}$)}} (due to line 15). In the second case, sx_k gives us the next largest value in {{`floor`($\vec{X}$)}}. In lines 20–23, we traverse $\vec{sx}$ starting from k, and $\vec{sy}$ starting from j. The reasoning now is very similar to that of the traversal for β. Instead of setting a value for β, we set σ to *true*, and instead of setting γ to $false$, we set σ to $false$, for the same reasons. If k reaches n before j, then we set σ to $false$ (line 24) due to the same reason as in line 22. If k and j reach n together, then again we set σ to $false$, because we have the same number of occurrences of any value less than β in {{`floor`($\vec{X}$)}} and in {{`ceiling`($\vec{Y}$)}}. If, however, j reaches n before k, then we set σ to *true* (line 25) due to the same reason as in line 21.

Finally, we go through each of sx_i and sy_i in lines 26–31, and find how many times α and β occur in {{`floor`($\vec{X}$)}} and in {{`ceiling`($\vec{Y}$)}}, by counting how many times α and β occur in $\vec{sx}$ and in $\vec{sy}$, respectively.

The complexity of this new algorithm is independent of the domains and is $O(n\log(n))$, as the cost of sorting dominates.

6. Extensions

In this section, we answer two important questions. First, how can we enforce strict multiset ordering? Second, how can we detect entailment?

6.1. Strict multiset ordering constraint

We can easily get a filtering algorithm for strict multiset ordering constraint by slightly modifying `MsetLeq`. This new algorithm, called `MsetLess`, either detects the disentailment of $\vec{X} <_m \vec{Y}$, or prunes inconsistent values to perform GAC on

$\vec{X} <_m \vec{Y}$. Before showing how we modify `MsetLeq`, we first study $\vec{X} <_m \vec{Y}$ from a theoretical point of view. It is not difficult to modify Theorems 2, 3 and 4 so as to exclude the equality and obtain the following propositions:

Proposition 3. *$\vec{X} <_m \vec{Y}$ is disentailed iff $occ(\texttt{floor}(\vec{X})) \geqslant_{lex} occ(\texttt{ceiling}(\vec{Y}))$.*

Proposition 4. GAC($\vec{X} <_m \vec{Y}$) *iff for all i in* $[0, n)$:

$$occ(\texttt{floor}(\vec{X}_{X_i \leftarrow max(X_i)})) <_{lex} occ(\texttt{ceiling}(\vec{Y}))$$
$$occ(\texttt{floor}(\vec{X})) <_{lex} occ(\texttt{ceiling}(\vec{Y}_{Y_i \leftarrow min(Y_i)}))$$

We can exploit the similarity between Propositions 2 and 4, and find the tight consistent bounds by making use of the occurrence vectors $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$, the pointers, and the flags. In Theorems 5 to 11, we have $\vec{ox} \leqslant_{lex} \vec{oy}$. We decide whether a value v in some domain D is consistent or not by first increasing ox_v/oy_v by 1, and then decreasing $min(D)/max(D)$ by 1. The value is consistent for $\vec{X} \leqslant_m \vec{Y}$ iff the change gives $\vec{ox} \leqslant_{lex} \vec{oy}$. In Theorems 7 and 11, changing the occurrences gives $\vec{ox} >_{lex} \vec{oy}$. This means that v is inconsistent not only for $\vec{X} \leqslant_m \vec{Y}$ but also for $\vec{X} <_m \vec{Y}$. In Theorems 6, 9, and 10, however, we initially have $\vec{ox} <_{lex} \vec{oy}$ and changing the occurrences does not disturb the strict lexicographic ordering. This suggests v is consistent also for $\vec{X} <_m \vec{Y}$.

In Theorems 5 and 8, we initially have $\vec{ox} <_{lex} \vec{oy}$, and after the change we obtain either of $\vec{ox} >_{lex} \vec{oy}$, $\vec{ox} = \vec{oy}$, and $\vec{ox} <_{lex} \vec{oy}$. In the first case v is inconsistent, whereas in the third case v is consistent, for both constraints. In the second case, however, v is consistent for $\vec{X} \leqslant_m \vec{Y}$ but not for $\vec{X} <_m \vec{Y}$. This case arises if we get $\vec{ox}_{u \to \beta} = \vec{oy}_{u \to \beta}$ by the change to the occurrence vectors, and we have either $\beta > l$ and $\vec{ox}_{\beta-1 \to l} = \vec{oy}_{\beta-1 \to l}$, or $\beta = l$. We therefore need to record whether there are any sub-vectors below β, and if this is the case we need to know whether they are equal. This can easily be done by extending the definition of σ which already tells us whether we have $\beta > l$ and $\vec{ox}_{\beta-1 \to l} >_{lex} \vec{oy}_{\beta-1 \to l}$.

Definition 9. Given $\vec{ox} = occ(\texttt{floor}(\vec{X}))$ and $\vec{oy} = occ(\texttt{ceiling}(\vec{Y}))$ indexed as $u..l$ where $\vec{ox} <_{lex} \vec{oy}$, the flag σ is *true* iff:

$$(\beta > l \ \wedge \ \vec{ox}_{\beta-1 \to l} \geqslant_{lex} \vec{oy}_{\beta-1 \to l}) \ \vee \ \beta = l$$

Theorems 5 and 8 now declare a value inconsistent if we get $\vec{ox}_{u \to \beta} = \vec{oy}_{u \to \beta}$ when the occurrence vectors change, and we have either $\beta > l$ and $\vec{ox}_{\beta-1 \to l} = \vec{oy}_{\beta-1 \to l}$, or $\beta = l$.

How do we now modify `MsetLeq` to obtain the filtering algorithm `MsetLess`? Theorems 6, 7, 9, 10, and 11 are valid also for $\vec{X} <_m \vec{Y}$. Moreover, Theorems 5 and 8 can easily be adapted for $\vec{X} <_m \vec{Y}$ by changing the definition of σ. Hence, the pruning part of the algorithm need not to be modified, provided that σ is set correctly. Also, by Proposition 3, we need to fail under the new disentailment condition. These suggest we only need to revise `SetPointersAndFlags`, so that we fail whenever we have $\vec{ox} \geqslant_{lex} \vec{oy}$, and set σ to *true* also when we have $\beta = l$, or $\beta > l$ and $\vec{ox}_{\beta-1 \to l} = \vec{ox}_{\beta-1 \to l}$. This corrects a mistake in [13] which claims that failing whenever we have $\vec{ox} \geqslant_{lex} \vec{oy}$ and setting β to $l - 1$ as opposed to $-\infty$ are enough to achieve strict multiset ordering.

6.2. Entailment

`MsetLeq` is a correct and complete filtering algorithm. However, it does not detect entailment. Even though detecting entailment does not change the semantics of the algorithm, it can lead to significant savings from an operational point of view. We thus introduce another Boolean flag, called *entailed*, which indicates whether $\vec{X} \leqslant_m \vec{Y}$ is entailed. More formally:

Definition 10. Given $\vec{X}$ and $\vec{Y}$, the flag *entailed* is set to *true* iff $\vec{X} \leqslant_m \vec{Y}$ is *true*.

The multiset ordering constraint is entailed whenever the largest value that $\vec{X}$ can take is less than or equal to the smallest value that $\vec{Y}$ can take under the ordering in concern.

Data : $\langle X_0, X_1, \ldots, X_{n-1} \rangle$, $\langle Y_0, Y_1, \ldots, Y_{n-1} \rangle$
Result : $occ(\texttt{floor}(\vec{X}))$, $occ(\texttt{ceiling}(\vec{Y}))$, $occ(\texttt{ceiling}(\vec{X}))$, $occ(\texttt{floor}(\vec{Y}))$, and *entailed* are initialised, GAC($\vec{X} \leq_m \vec{Y}$)

```
entailed := false;
  ⋮
ex := occ(ceiling(X));
ey := occ(floor(Y));
MsetLeq;
```

Algorithm 6. `Initialise`.

```
Data    : ⟨X_0, X_1, ..., X_{n-1}⟩, ⟨Y_0, Y_1, ..., Y_{n-1}⟩
Result  : GAC(X ≤_m Y)
A0  if entailed then return;
⇒   if ex ≤_lex ey then entailed := true, return;
A1  SetPointersAndFlags;
B1  foreach i ∈ [0, n) do
B2      if min(X_i) ≠ max(X_i) then
B3          if min(X_i) ≥ α then
⇒               ex_{max(X_i)} := ex_{max(X_i)} − 1, setMax(X_i, min(X_i));
⇒               ex_{max(X_i)} := ex_{max(X_i)} + 1;
            end
B4          if max(X_i) ≥ α ∧ min(X_i) < α then
B5⇒             ex_{max(X_i)} := ex_{max(X_i)} − 1, setMax(X_i, α);
                ⋮
⇒               ex_{max(X_i)} := ex_{max(X_i)} + 1;
            end
        end
    end
⇒   if ex ≤_lex ey then entailed := true, return;
C1  foreach i ∈ [0, n) do
C2      if min(Y_i) ≠ max(Y_i) then
C3          if max(Y_i) > α then
⇒               ey_{min(Y_i)} := ey_{min(Y_i)} − 1, setMin(Y_i, max(Y_i)), ey_{min(Y_i)} := ey_{min(Y_i)} + 1;
            end
C4          if max(Y_i) = α ∧ min(Y_i) ≤ β then
C5              if ox_α + 1 = oy_α ∧ γ then
C6⇒                 ey_{min(Y_i)} := ey_{min(Y_i)} − 1, setMin(Y_i, β);
                    ⋮
⇒                   ey_{min(Y_i)} := ey_{min(Y_i)} + 1
                end
            end
        end
    end
⇒   if ex ≤_lex ey then entailed := true, return;
```

Algorithm 7. MsetLeq.

Theorem 18. *$\vec{X} \leqslant_m \vec{Y}$ is entailed iff* $\{\{\texttt{ceiling}(\vec{X})\}\} \leqslant_m \{\{\texttt{floor}(\vec{Y})\}\}$.

Proof. ($\Rightarrow$) Since $\vec{X} \leqslant_m \vec{Y}$ is entailed, any combination of assignments, including $\vec{X} \leftarrow \texttt{ceiling}(\vec{X})$ and $\vec{Y} \leftarrow \texttt{floor}(\vec{Y})$, satisfies $\vec{X} \leqslant_m \vec{Y}$. Hence, $\{\{\texttt{ceiling}(\vec{X})\}\} \leqslant_m \{\{\texttt{floor}(\vec{Y})\}\}$.

($\Leftarrow$) Any $\vec{x} \in \vec{X}$ is less than or equal to any $\vec{y} \in \vec{Y}$ under multiset ordering. Hence, $\vec{X} \leqslant_m \vec{Y}$ is entailed. $\square$

By Theorems 4 and 18, we can detect entailment by lexicographically comparing the occurrence vectors associated with $\texttt{ceiling}(\vec{X})$ and $\texttt{floor}(\vec{Y})$.

Proposition 5. *$\vec{X} \leqslant_m \vec{Y}$ is entailed iff $occ(\texttt{ceiling}(\vec{X})) \leqslant_{lex} occ(\texttt{floor}(\vec{Y}))$.*

When MsetLeq is executed, we have three possible scenarios in terms of entailment: (1) $\vec{X} \leqslant_m \vec{Y}$ has already been entailed in the past due to the previous modifications to the variables; (2) $\vec{X} \leqslant_m \vec{Y}$ was not entailed before, but after the recent modifications which invoked the algorithm, $\vec{X} \leqslant_m \vec{Y}$ is now entailed; (3) $\vec{X} \leqslant_m \vec{Y}$ has not been entailed, but after the prunings of the algorithm, $\vec{X} \leqslant_m \vec{Y}$ is now entailed. In all cases, we can safely return from the algorithm. We need to, however, record entailment in our flag *entailed* in the second and the third cases, before returning.

To deal with entailment, we need to modify both Initialise and MsetLeq. In Algorithm 6, we show how we revise Algorithm 1. We add line 0 to initialise the flag *entailed* to $false$. We replace line 5 of Algorithm 1 with lines 5–7. Before calling MsetLeq , we now initialise our new occurrence vectors $occ(\texttt{ceiling}(\vec{X}))$ and $occ(\texttt{floor}(\vec{Y}))$ in a similar way to that of $occ(\texttt{floor}(\vec{X}))$ and $occ(\texttt{ceiling}(\vec{Y}))$: we create a pair of vectors $\vec{ex}$ and $\vec{ey}$ of length $u - l + 1$ where each ex_i and ey_i are first set to 0. Then, for each value v in $\{\{\texttt{ceiling}(\vec{X})\}\}$, we increment ex_v by 1. Similarly, for each v in $\{\{\texttt{floor}(\vec{Y})\}\}$, we increment ey_v by 1. These vectors are then used in MsetLeq to detect entailment. It is possible to maintain $\vec{ex}$ and $\vec{ey}$

incrementally. When the maximum value in some $\mathcal{D}(X_i)$ changes, we update $\vec{ex}$ by incrementing the entry corresponding to new $max(X_i)$ by 1, and decrementing the entry corresponding to old $max(X_i)$ by 1. Likewise, when the minimum value in some $\mathcal{D}(Y_i)$ changes, we update $\vec{ey}$ by incrementing the entry corresponding to new $min(Y_i)$ by 1, and decrementing the entry corresponding to old $min(Y_i)$ by 1.

In Algorithm 7, we show how we modify the filtering algorithm given in Algorithm 2 to deal with the three possible scenarios described above. We add line **A0** where we return if the constraint has already been entailed in the past. Moreover, just before setting our pointers and flags, we check whether the recent modifications that triggered the algorithm resulted in entailment. If this is the case, we first set *entailed* to *true* and then return from the algorithm. Furthermore, we check entailment after the algorithm goes through its variables. Lines **B1–B11** visit the variables of $\vec{X}$ and prune inconsistent values from the upper bounds, affecting $\vec{ex}$. Even if we have $\vec{ex} >_{lex} \vec{ey}$ when the algorithm is called, we might get $\vec{ex} \leqslant_{lex} \vec{ey}$ just before the algorithm proceeds to the variables of $\vec{Y}$. In such case, we return from the algorithm after setting *entailed* to *true*. As an example, assume we have $\vec{X} \leqslant_m \vec{Y}$, and `MsetLeq` is called with $\vec{X} = \langle\{1,2\},\{1,2,4\}\rangle$ and $\vec{Y} = \langle\{2,3\},\{2,3\}\rangle$. As 4 in $\mathcal{D}(X_1)$ lacks support, it is pruned. Now we have $\vec{ex} = \vec{ey}$. Alternatively, the constraint might be entailed after the algorithm visits the variables of $\vec{Y}$ and prunes inconsistent values from the lower bounds, affecting $\vec{ey}$. In this case, we return from the algorithm by setting *entailed* to *true*. As an example, assume we also have 0 in $\mathcal{D}(Y_1)$ in the previous example. The constraint is entailed only after the variables of $\vec{Y}$ are visited and 0 is removed.

Finally, before/after the algorithm modifies $max(X_i)$ or $min(Y_i)$ of some i in $[0,n)$, we keep our occurrence vectors $\vec{ex}$ and $\vec{ey}$ up-to-date by decrementing/incrementing the necessary entries.

7. Alternative approaches

There are several alternative ways known for posting and propagating multiset ordering constraints. We can, for instance, post arithmetic inequality constraints, or decompose multiset ordering constraints into other constraints. In this section, we explore these approaches and argue why it is preferable to propagate multiset ordering constraints using our filtering algorithms.

7.1. Arithmetic constraint

We can achieve multiset ordering between two vectors by assigning a weight to each value, summing the weights along each vector, and then insisting the sums to be non-decreasing. Since the ordering is determined according to the maximum value in the vectors, the weight should increase with the value. A suitable weighting scheme was proposed in [18], where each value v gets assigned the weight n^v, where n is the length of the vectors. $\vec{X} \leqslant_m \vec{Y}$ on vectors of length n can then be enforced via the following arithmetic inequality constraint:

$$n^{X_0} + \cdots + n^{X_{n-1}} \leqslant n^{Y_0} + \cdots + n^{Y_{n-1}}$$

Therefore, a vector containing one element with value v and $n-1$ 0s is greater than a vector whose n elements are only $v-1$. This is in fact similar to the transformation of a leximin fuzzy CSP into an equivalent MAX CSP [32]. Strict multiset ordering constraint $\vec{X} <_m \vec{Y}$ is enforced by disallowing equality:

$$n^{X_0} + \cdots + n^{X_{n-1}} < n^{Y_0} + \cdots + n^{Y_{n-1}}$$

BC on such arithmetic constraints does the same pruning as GAC on the original multiset ordering constraints. However, such arithmetic constraints are feasible only for small n and u, where u is the maximum value in the domains of the variables. As n and u get large, n^{X_i} or n^{Y_i} will be a very large number and therefore it might be impossible to implement the multiset ordering constraint. Consequently, it can be preferable to post and propagate the multiset ordering constraints using our global constraints.

Theorem 19. GAC($\vec{X} \leqslant_m \vec{Y}$) *and* GAC($\vec{X} <_m \vec{Y}$) *are equivalent to BC on the corresponding arithmetic constraints.*

Proof. We just consider GAC($\vec{X} \leqslant_m \vec{Y}$) as the proof for GAC($\vec{X} <_m \vec{Y}$) is entirely analogous. As $\vec{X} \leqslant_m \vec{Y}$ and the corresponding arithmetic constraint are logically equivalent, BC($\vec{X} \leqslant_m \vec{Y}$) and BC on the arithmetic constraint are equivalent. By Theorem 1, BC($\vec{X} \leqslant_m \vec{Y}$) is equivalent to GAC($\vec{X} \leqslant_m \vec{Y}$). □

7.2. Decomposition

Global ordering constraints can often be built out of the logical connectives ($\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\neg$) and existing (global) constraints. We can thus compose other constraints between $\vec{X}$ and $\vec{Y}$ so as to obtain the multiset ordering constraint between $\vec{X}$ and $\vec{Y}$. We refer to such a logical constraint as a decomposition of the multiset ordering constraint.

The multiset view of two vectors of integers $\vec{x}$ and $\vec{y}$ are multiset ordered $\{\{\vec{x}\}\} \leqslant_m \{\{\vec{y}\}\}$ iff $occ(\vec{x}) \leqslant_{lex} occ(\vec{y})$ by Theorem 4. One way of decomposing the multiset ordering constraint $\vec{X} \leqslant_m \vec{Y}$ is thus insisting that the occurrence vectors associated with the vectors assigned to $\vec{X}$ and $\vec{Y}$ are lexicographically ordered. Such occurrence vectors can be constructed via an extended global cardinality constraint (*gcc*). Given a vector of variables $\vec{X}$ and a vector of values $\vec{d}$, the constraint $gcc(\vec{X}, \vec{d}, \vec{OX})$ ensures that OX_i is the number of variables in $\vec{X}$ assigned to d_i. To ensure multiset ordering, we can enforce lexicographic ordering constraint on a pair of occurrence vectors constructed via *gcc* where $\vec{d}$ is the vector of values that the variables can be assigned to, arranged in descending order, without any repetition:

$$gcc(\vec{X}, \vec{d}, \vec{OX}) \ \wedge \ gcc(\vec{Y}, \vec{d}, \vec{OY}) \ \wedge \ \vec{OX} \leqslant_{lex} \vec{OY}$$

In order to decompose the strict multiset ordering constraint $\vec{X} <_m \vec{Y}$, we need to enforce strict lexicographic ordering constraint on the occurrence vectors:

$$gcc(\vec{X}, \vec{d}, \vec{OX}) \ \wedge \ gcc(\vec{Y}, \vec{d}, \vec{OY}) \ \wedge \ \vec{OX} <_{lex} \vec{OY}$$

We call this way of decomposing a multiset ordering constraint as *gcc* decomposition.

The *gcc* constraint is available in, for instance, ILOG Solver 5.3 [17], SICStus Prolog 3.10.1 [33], and the FaCiLe constraint solver 1.0 [9]. These solvers propagate the *gcc* constraint using the algorithm proposed in [28]. Among the various filtering algorithms of *gcc*, which maintain either GAC [26,28] or BC [19,26], only the algorithms in [19] prune values from $\vec{OX}$ and $\vec{OY}$. Even though the algorithm integrated in ILOG Solver 5.3 may also prune the occurrence vectors, this may not always be the case. For instance, when we have $gcc(\langle\{1\}, \{1,2\}, \{1,2\}, \{2\}, \{3,4\}, \{3,4\}\rangle, \langle 4,3,2,1\rangle, \langle\{1\}, \{1\}, \{1,2\}, \{1,2,3\}\rangle)$, ILOG Solver 5.3 leaves $\vec{OX}$ unchanged even though 1 in $\mathcal{D}(OX_3)$ is not consistent. This shows that there is currently very limited support in the constraint toolkits to propagate the multiset ordering constraint using the *gcc* decomposition. Also, as the following theorems demonstrate, the *gcc* decomposition of a multiset ordering constraint hinders constraint propagation.

Theorem 20. GAC($\vec{X} \leqslant_m \vec{Y}$) *is strictly stronger than* GAC($gcc(\vec{X}, \vec{d}, \vec{OX})$), GAC($gcc(\vec{Y}, \vec{d}, \vec{OY})$), *and* GAC($\vec{OX} \leqslant_{lex} \vec{OY}$), *where $\vec{d}$ is the vector of values that the variables can take, arranged in descending order, without any repetition.*

Proof. Since $\vec{X} \leqslant_m \vec{Y}$ is GAC, every value has a support $\vec{x}$ and $\vec{y}$ where $occ(\vec{x}) \leqslant_{lex} occ(\vec{y})$, in which case all the three constraints posted in the decomposition are satisfied. Hence, every constraint imposed is GAC, and GAC($\vec{X} \leqslant_m \vec{Y}$) is as strong as its decomposition. To show strictness, consider $\vec{X} = \langle\{0,3\}, \{2\}\rangle$ and $\vec{Y} = \langle\{2,3\}, \{1\}\rangle$. The multiset ordering constraint $\vec{X} \leqslant_m \vec{Y}$ is not GAC as 3 in $\mathcal{D}(X_0)$ has no support. By enforcing GAC($gcc(\vec{X}, \langle 3,2,1,0\rangle, \vec{OX})$) and GAC($gcc(\vec{Y}, \langle 3,2,1,0\rangle, \vec{OY})$) we obtain the following occurrence vectors:

$$\begin{aligned} \vec{OX} &= \langle\{0,1\}, \ \ \{1\}, \ \ \{0\}, \ \{0,1\}\rangle \\ \vec{OY} &= \langle\{0,1\}, \ \{0,1\}, \ \{1\}, \ \ \{0\}\rangle \end{aligned}$$

Since we have GAC($\vec{OX} \leqslant_{lex} \vec{OY}$), $\vec{X}$ and $\vec{Y}$ remain unchanged. □

Theorem 21. GAC($\vec{X} <_m \vec{Y}$) *is strictly stronger than* GAC($gcc(\vec{X}, \vec{d}, \vec{OX})$), GAC($gcc(\vec{Y}, \vec{d}, \vec{OY})$), *and* GAC($\vec{OX} <_{lex} \vec{OY}$), *where $\vec{d}$ is the vector of values that the variables can take, arranged in descending order, without any repetition.*

Proof. The example in Theorem 20 shows the strictness. □

In Theorem 17, we have established that $occ(\vec{x}) \leqslant_{lex} occ(\vec{y})$ iff $sort(\vec{x}) \leqslant_{lex} sort(\vec{y})$. Putting Theorems 4 and 17 together, the multiset view of two vectors of integers $\vec{x}$ and $\vec{y}$ are multiset ordered $\{\!\{\vec{x}\}\!\} \leqslant_m \{\!\{\vec{y}\}\!\}$ iff $sort(\vec{x}) \leqslant_{lex} sort(\vec{y})$. This suggests another way of decomposing a multiset ordering constraint $\vec{X} \leqslant_m \vec{Y}$: we insist that the sorted versions of the vectors assigned to $\vec{X}$ and $\vec{Y}$ are lexicographically ordered. For this purpose, we can use the constraint *sorted* which is available in, for instance, ECLiPSe constraint solver 5.6 [8], SICStus Prolog 3.10.1 [33], and the FaCiLe constraint solver 1.0 [9]. Given a vector of variables $\vec{X}$, $sorted(\vec{X}, \vec{SX})$ ensures that $\vec{SX}$ is of length n and is a sorted permutation of $\vec{X}$. To ensure multiset ordering, we can enforce lexicographic ordering constraint on a pair of vectors which are constrained to be the sorted versions of the original vectors in descending order:

$$sorted(\vec{X}, \vec{SX}) \ \wedge \ sorted(\vec{Y}, \vec{SY}) \ \wedge \ \vec{SX} \leqslant_{lex} \vec{SY}$$

A strict multiset ordering constraint $\vec{X} <_m \vec{Y}$ is then achieved by enforcing strict lexicographic ordering constraint on the sorted vectors:

$$sorted(\vec{X}, \vec{SX}) \ \wedge \ sorted(\vec{Y}, \vec{SY}) \ \wedge \ \vec{SX} <_{lex} \vec{SY}$$

We call this way of decomposing a multiset ordering constraint as the *sort* decomposition.

The *sorted* constraint has previously been studied and some BC filtering algorithms have been proposed [2,3,24]. Unfortunately, we lose in the amount of constraint propagation also by the *sort* decomposition of a multiset ordering constraint.

Theorem 22. GAC($\vec{X} \leqslant_m \vec{Y}$) *is strictly stronger than* GAC($sorted(\vec{X}, \vec{SX})$), GAC($sorted(\vec{Y}, \vec{SY})$), *and* GAC($\vec{SX} \leqslant_{lex} \vec{SY}$).

Proof. Since $\vec{X} \leqslant_m \vec{Y}$ is GAC, every value has a support $\vec{x}$ and $\vec{y}$ where $sort(\vec{x}) \leqslant_{lex} sort(\vec{y})$, in which case all the three constraints posted in the decomposition are satisfied. Hence, every constraint imposed is GAC, and GAC($\vec{X} \leqslant_m \vec{Y}$) is as strong as its decomposition. To show strictness, consider $\vec{X} = \langle\{0, 3\}, \{2\}\rangle$ and $\vec{Y} = \langle\{2, 3\}, \{1\}\rangle$. The multiset ordering constraint $\vec{X} \leqslant_m \vec{Y}$ is not GAC as 3 in $\mathcal{D}(X_0)$ has no support. By enforcing GAC($sorted(\vec{X}, \vec{SX})$) and GAC($sorted(\vec{Y}, \vec{SY})$) we obtain the following vectors:

$$\vec{SX} = \langle\{2, 3\},\ \{0, 2\}\rangle$$
$$\vec{SY} = \langle\{2, 3\},\ \ \{1\}\rangle$$

Since we have GAC($\vec{SX} \leqslant_{lex} \vec{SY}$), $\vec{X}$ and $\vec{Y}$ remain unchanged. □

Theorem 23. GAC($\vec{X} <_m \vec{Y}$) *is strictly stronger than* GAC($sorted(\vec{X}, \vec{SX})$), GAC($sorted(\vec{Y}, \vec{SY})$), *and* GAC($\vec{SX} <_{lex} \vec{SY}$).

Proof. The example in Theorem 22 shows strictness. □

How do the two decompositions compare? Assuming that GAC is enforced on every n-ary constraint of a decomposition, the *sort* decomposition is superior to the *gcc* decomposition.

Theorem 24. *The sort decomposition of $\vec{X} \leqslant_m \vec{Y}$ is strictly stronger than the gcc decomposition of $\vec{X} \leqslant_m \vec{Y}$.*

Proof. Assume that a value is pruned from $\vec{X}$ due to the *gcc* decomposition. Then, there is an index α such that $\neg(OX_\alpha \doteq OY_\alpha)$ and for all $i > \alpha$ we have $OX_i \doteq OY_i$. Moreover, we have $min(OX_i) = max(OY_i)$ and $max(OX_i) > max(OY_i)$. The reason is that, only in this case, GAC($\vec{OX} \leqslant_{lex} \vec{OY}$) will not only prune values from OX_α but also from $\vec{X}$. In any other case, we will either get no pruning at OX_α, or the pruning at OX_α will reduce the number of occurrences of α in $\vec{X}$ without deleting any of α from $\vec{X}$. Now consider the vectors $\vec{SX}$ and $\vec{SY}$. We name the index of $\vec{SX}$ and $\vec{SY}$, where α first appears in the domains of $\vec{SX}$ and $\vec{SY}$, as i. Since the number of occurrences of any value greater than α is already determined and is the same in both $\vec{X}$ and $\vec{Y}$, the sub-vectors of $\vec{SX}$ and $\vec{SY}$ above i are ground and equal. For all $i \leqslant j < i + min(OX_i)$, we have $SX_j \doteq SY_j \leftarrow \alpha$. Since $max(OX_i) > max(OY_i)$, at position $k = i + min(OX_i)$ we will have α in $\mathcal{D}(SX_k)$ but not in $\mathcal{D}(SY_k)$ whose values are less than α. To have $\vec{SX} \leqslant_{lex} \vec{SY}$, α in $\mathcal{D}(SX_k)$ is eliminated. This propagates to the pruning of α from the remaining variables of $\vec{SX}$, as well as from domains of the uninstantiated variables of $\vec{X}$. Hence, any value removed from $\vec{X}$ due to the *gcc* decomposition is removed from $\vec{X}$ also by the *sort* decomposition. The proof can easily be reverted for values being removed from $\vec{Y}$.

To show that the *sort* decomposition dominates the *gcc* decomposition, consider $\vec{X} = \langle\{1, 2\}\rangle$ and $\vec{Y} = \langle\{0, 1, 2\}\rangle$ where 0 in $\mathcal{D}(Y_0)$ is inconsistent and therefore $\vec{X} \leqslant_m \vec{Y}$ is not GAC. We have $\vec{SX} = \langle\{1, 2\}\rangle$ and $\vec{SY} = \langle\{0, 1, 2\}\rangle$ by GAC($sorted(\vec{X}, \vec{SX})$) and GAC($sorted(\vec{Y}, \vec{SY})$), and $\vec{OX} = \langle\{0, 1\}, \{0, 1\}, \{0\}\rangle$ and $\vec{OY} = \langle\{0, 1\}, \{0, 1\}, \{0, 1\}\rangle$ by GAC($gcc(\vec{X}, \langle 2, 1, 0\rangle, \vec{OX})$) and GAC($gcc(\vec{Y}, \langle 2, 1, 0\rangle, \vec{OY})$). To achieve GAC($\vec{SX} \leqslant_{lex} \vec{SY}$), 0 in $\mathcal{D}(SY_0)$ is pruned. This leads to the pruning of 0 also from $\mathcal{D}(Y_0)$ so as to establish GAC($sorted(\vec{Y}, \vec{SY})$). On the other hand, we have GAC($\vec{OX} \leqslant_{lex} \vec{OY}$), in which case no value is pruned from any variable. □

Theorem 25. *The sort decomposition of $\vec{X} <_m \vec{Y}$ is strictly stronger than the gcc decomposition of $\vec{X} <_m \vec{Y}$.*

Proof. The example in Theorem 24 shows strictness. □

Even though the *sort* decomposition of $\vec{X} \leqslant_m \vec{Y}$ is stronger than the *gcc* decomposition of $\vec{X} \leqslant_m \vec{Y}$, GAC on $\vec{X} \leqslant_m \vec{Y}$ can lead to more pruning than any of the two decompositions. A similar argument holds also for $\vec{X} <_m \vec{Y}$. Hence, it can be preferable to post and propagate multiset ordering constraints via our global constraints.

8. Multiple vectors

We often have multiple multiset ordering constraints. For example, we post multiset ordering constraints on the rows or columns of a matrix of decision variables because we want to break row or column symmetry. We can treat such a problem as a single global ordering constraint over the whole matrix. Alternatively, we can decompose it into multiset ordering constraints between adjacent or all pairs of vectors. In this section, we demonstrate that such decompositions hinder constraint propagation.

The following theorems hold for n vectors of m constrained variables.

Theorem 26. GAC($\vec{X}_i \leqslant_m \vec{X}_j$) *for all* $0 \leqslant i < j \leqslant n - 1$ *is strictly stronger than* GAC($\vec{X}_i \leqslant_m \vec{X}_{i+1}$) *for all* $0 \leqslant i < n - 1$.

Proof. GAC($\vec{X}_i \leqslant_m \vec{X}_j$) for all $0 \leqslant i < j \leqslant n-1$ is as strong as GAC($\vec{X}_i \leqslant_m \vec{X}_{i+1}$) for all $0 \leqslant i < n-1$, because the former implies the latter. To show strictness, consider the following 3 vectors:

$$\begin{aligned}\vec{X}_0 &= \langle\{0,3\}, \quad \{2\}\rangle\\ \vec{X}_1 &= \langle\{0,1,2,3\}, \ \{0,1,2,3\}\rangle\\ \vec{X}_2 &= \langle\{2,3\}, \quad \{1\}\rangle\end{aligned}$$

We have GAC($\vec{X}_i \leqslant_m \vec{X}_{i+1}$) for all $0 \leqslant i < 2$. The assignment $X_{0,0} \leftarrow 3$ forces $\vec{X}_0$ to be $\langle 3,2\rangle$, and we have $\texttt{ceiling}(\vec{X}_2) = \langle 3,1\rangle$. Since $\{\{3,2\}\} >_m \{\{3,1\}\}$, GAC($\vec{X}_0 \leqslant_m \vec{X}_2$) does not hold. □

Theorem 27. GAC($\vec{X}_i <_m \vec{X}_j$) *for all* $0 \leqslant i < j \leqslant n-1$ *is strictly stronger than* GAC($\vec{X}_i <_m \vec{X}_{i+1}$) *for all* $0 \leqslant i < n-1$.

Proof. The example in Theorem 26 shows strictness. □

Theorem 28. GAC($\forall ij\ 0 \leqslant i < j \leqslant n-1.\ \vec{X}_i \leqslant_m \vec{X}_j$) *is strictly stronger than* GAC($\vec{X}_i \leqslant_m \vec{X}_j$) *for all* $0 \leqslant i < j \leqslant n-1$.

Proof. GAC($\forall ij\ 0 \leqslant i < j \leqslant n-1.\ \vec{X}_i \leqslant_m \vec{X}_j$) is as strong as GAC($\vec{X}_i \leqslant_m \vec{X}_j$) for all $0 \leqslant i < j \leqslant n-1$, because the former implies the latter. To show strictness, consider the following 3 vectors:

$$\begin{aligned}\vec{X}_0 &= \langle\{0,3\}, \quad \{1\}\rangle\\ \vec{X}_1 &= \langle\{0,2\}, \ \{0,1,2,3\}\rangle\\ \vec{X}_2 &= \langle\{0,1\}, \ \{0,1,2,3\}\rangle\end{aligned}$$

We have GAC($\vec{X}_i \leqslant_m \vec{X}_j$) for all $0 \leqslant i < j \leqslant 2$. The assignment $X_{0,0} \leftarrow 3$ is supported by $X_0 \leftarrow \langle 3,1\rangle$, $X_1 \leftarrow \langle 2,3\rangle$, and $X_2 \leftarrow \langle 1,3\rangle$. In this case, $\vec{X}_1 \leqslant_m \vec{X}_2$ is $false$. Therefore, GAC($\forall ij\ 0 \leqslant i < j \leqslant 2.\ \vec{X}_i \leqslant_m \vec{X}_j$) does not hold. □

Theorem 29. GAC($\forall ij\ 0 \leqslant i < j \leqslant n-1.\ \vec{X}_i <_m \vec{X}_j$) *is strictly stronger than* GAC($\vec{X}_i <_m \vec{X}_j$) *for all* $0 \leqslant i < j \leqslant n-1$.

Proof. GAC($\forall ij\ 0 \leqslant i < j \leqslant n-1.\ \vec{X}_i <_m \vec{X}_j$) is as strong as GAC($\vec{X}_i <_m \vec{X}_j$) for all $0 \leqslant i < j \leqslant n-1$, because the former implies the latter. To show strictness, consider the following 3 vectors:

$$\begin{aligned}\vec{X}_0 &= \langle\{0,3\}, \quad \{1\}\rangle\\ \vec{X}_1 &= \langle\{1,3\}, \ \{0,1,3\}\rangle\\ \vec{X}_2 &= \langle\{0,2\}, \ \{0,1,2,3\}\rangle\end{aligned}$$

We have GAC($\vec{X}_i <_m \vec{X}_j$) for all $0 \leqslant i < j \leqslant 2$. The assignment $X_{0,0} \leftarrow 3$ is supported by $X_0 \leftarrow \langle 3,1\rangle$, $X_1 \leftarrow \langle 3,3\rangle$, and $X_2 \leftarrow \langle 2,3\rangle$. In this case, $\vec{X}_1 <_m \vec{X}_2$ is $false$. Therefore, GAC($\forall ij\ 0 \leqslant i < j \leqslant 2.\ \vec{X}_i <_m \vec{X}_j$) does not hold. □

9. Experiments

We implemented our global constraints $\leqslant_m$ and $<_m$ in C++ using ILOG Solver 5.3 [17]. Due the absence of the *sorted* constraint in Solver 5.3, the multiset ordering constraint is decomposed via the *gcc* decomposition using the **IloDistribute** constraint. This constraint is the *gcc* constraint but it does not always prune completely the occurrence vectors as described before.

In the experiments, we have a matrix of decision variables where the rows and/or columns are (partially) symmetric. To break the symmetry, we post multiset ordering constraints on the adjacent symmetric rows or columns, and address several questions in the context of looking for one solution or the optimal solution. First, does our filtering algorithm(s) do more inference in practice than its decomposition? Similarly, is the algorithm more efficient in practice than its decomposition? Second, is it feasible to post the arithmetic constraint? How does our algorithm compare to BC on the arithmetic constraint? Even though studying the effectiveness of the multiset ordering constraints in breaking symmetry is out of the scope of this paper, we provide experimental evidence of their value in symmetry breaking.

We report experiments on three problem domains: the progressive party problem, the rack configuration problem, and the sport scheduling problem. The decisions made when modelling and solving a problem are tuned by our initial experimentation. The results are shown in tables where a "-" means no result is obtained in 1 hour (3600 secs). The best result of each entry in a table is typeset in bold. If posing an ordering constraint on the rows (resp. columns) is done via a technique called *Tech* then we write *Tech* R (resp. *Tech* C). The ordering constraints are enforced just between the adjacent rows and/or columns as we have found it not worthwhile to post them between all pairs.

Finally, the hardware used for the experiments is a 1 Ghz pentium III processor with 256 Mb RAM running Windows XP.

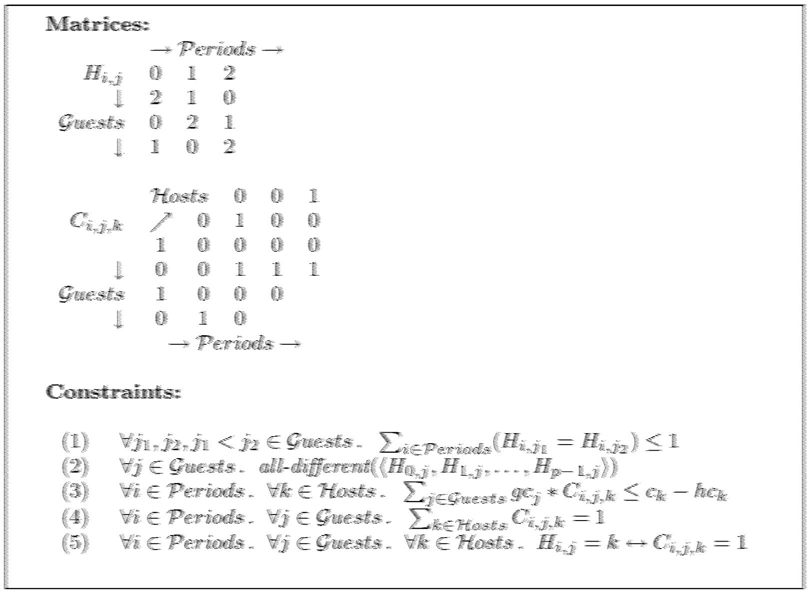

Fig. 2. The matrix model of the progressive party problem in [31].

9.1. Progressive party problem

The **progressive party problem** arises in the context of organising the social programme for a yachting rally (prob013 in CSPLib). We consider a variant of the problem proposed in [31]. There is a set $\mathcal{H}osts$ of host boats and a set $\mathcal{G}uests$ of guest boats. Each host boat i is characterised by a tuple $\langle hc_i, c_i\rangle$, where and hc_i is its crew size and c_i is its capacity; and each guest boat is described by gc_i giving its crew size. The problem is to assign hosts to guests over p time periods such that:

- a guest crew never visits the same host twice;
- no two guest crews meet more than once;
- the spare capacity of each host boat, after accommodating its own crew, is not exceeded.

A matrix model of this problem is given in [31]. It has a 2-d matrix H to represent the assignment of hosts to guests in time periods (see Fig. 2). The matrix H is indexed by the set $\mathcal{P}eriods$ of time periods and $\mathcal{G}uests$, taking values from $\mathcal{H}osts$. The first constraint enforces that two guests can meet at most once by introducing a new set of 0/1 variables:

$$\forall i \in \mathcal{P}eriods.\quad \forall j_1, j_2, j_1 < j_2 \in \mathcal{G}uests.\quad M_{i,j1,j2} = 1 \leftrightarrow H_{i,j1} = H_{i,j2}$$

The sum of these new variables are then constrained to be at most 1. The *all-different* constraints on the rows of this matrix ensure that no guest revisits a host. Additionally, a 3-d 0/1 matrix C of $\mathcal{P}eriods \times \mathcal{G}uests \times \mathcal{H}osts$ is used. A variable $C_{i,j,k}$ in this new matrix is 1 iff the host boat k is visited by guest j in period i. Even though C replicates the information held in the 2-d matrix, it allows capacity constraints to be stated concisely. The sum constraints on C ensure that a guest is assigned to exactly one host on a time period. Finally, channelling constraints are used to link the variables of H and C.

The time periods as well as the guests with equal crew size are indistinguishable. Hence, this model of the problem has partial row symmetry between the indistinguishable guests of H, and column symmetry. In the following we first show that multiset ordering constraints are useful in breaking index symmetry.

To break the row and column symmetries, we can utilise both lexicographic ordering and multiset ordering constraints, as well as combine lexicographic ordering constraints in one dimension of the matrix with multiset ordering constraints in the other. Due to the problem constraints, no pair of rows/columns can have equal assignments, but they can be equal when viewed as multisets. This gives us the models $<_{lex}$ RC, $\leqslant_m$ RC, $\leqslant_m$ R $\geqslant_m$ C, $\leqslant_m$ R $<_{lex}$ C, $\leqslant_m$ R $>_{lex}$ C, $<_{lex}$ R $\leqslant_m$ C, and $<_{lex}$ R $\geqslant_m$ C. As the matrix H has partial row symmetry, the ordering constraints on the rows are posted on only the symmetric rows. The ordering constraints on the columns are, however, posted on all the columns.

In our experiments, we compare the models described above in contrast to the initial model of the problem in which no symmetry breaking ordering constraints are imposed. We consider the original instance of the progressive party problem described in [31], with 5 and 6 time periods. As in [31], we give priority to the largest crews, so the guest boats are ordered in descending order of their size. Also, when assigning a host to a guest, we try a value first which is most likely to succeed. We therefore order the host boats in descending order of their spare capacity. We adopt two static variable orderings, and instantiate H either along its rows from top to bottom, or along its columns from left to right.

Table 1
Progressive party problem with row-wise labelling of H.

Model	Problem					
	5 time periods			6 time periods		
	Fails	Choice points	Time (secs.)	Fails points	Choice points	Time (secs.)
No symmetry breaking	180,738	180,860	75.9	–	–	–
$<_{lex}$ RC	2720	2842	2.7	–	–	–
$\leqslant_m$ RC	–	–	–	–	–	–
$\leqslant_m$ R $\geqslant_m$C	9, 207	9329	8.0	–	–	–
$\leqslant_m$ R $<_{lex}$C	10,853	10,977	8.6	–	–	–
$\leqslant_m$ R $>_{lex}$C	2289	2405	2.6	–	–	–
$<_{lex}$ R $\leqslant_m$C	**2016**	**2137**	**2.0**	–	–	–
$<_{lex}$ R $\geqslant_m$C	–	–	–	–	–	–

Table 2
Progressive party problem with column-wise labelling of H.

Model	Problem					
	5 time periods			6 time periods		
	Fails	Choice points	Time (secs.)	Fails points	Choice points	Time (secs.)
No symmetry breaking	20,546	20,676	9.0	20,722	20,871	12.3
$<_{lex}$ RC	20,546	20,676	9.0	20,722	20,871	12.4
$\leqslant_m$ RC	–	–	–	–	–	–
$\leqslant_m$ R $\geqslant_m$C	–	–	–	–	–	–
$\leqslant_m$ R $<_{lex}$C	–	–	–	–	–	–
$\leqslant_m$ R $>_{lex}$C	–	–	–	–	–	–
$<_{lex}$ R $\leqslant_m$C	**7038**	**7168**	**3.4**	**7053**	**7202**	**4.6**
$<_{lex}$ R $\geqslant_m$C	–	–	–	–	–	–

Table 3
Instance specification for the progressive party problem.

Instance #	Host boats	Total host spare capacity	Total guest size	%Capacity
1	2-12, 14, 16	102	92	.90
2	3-14, 16	100	90	.90
3	3-12, 14, 15, 16	101	91	.90
4	3-12, 14, 16, 25	101	92	.91
5	3-12, 14, 16, 23	99	90	.91
6	3-12, 15, 16, 25	100	91	.91
7	1, 3-12, 14, 16	100	92	.92
8	3-12, 16, 25, 26	100	92	.92
9	3-12, 14, 16, 30	98	90	.92

The results of the experiments are shown in Tables 1 and 2. With row-wise labelling of H, we cannot solve the problem with 6 time periods with or without the symmetry breaking ordering constraints. As for the other instance, whilst many of the models we have considered give significantly smaller search trees and shorter run-times, $\leqslant_m$RC and $<_{lex}$R $\geqslant_m$C cannot return an answer within an hour time limit. The smallest search tree and also the shortest solving time is obtained by $<_{lex}$R $\leqslant_m$C, in which case the reduction in the search effort is noteworthy compared to the model in which no ordering constrains are imposed. This supports our conjecture that lexicographic ordering constraints in one dimension of a matrix combined with multiset ordering constraints in the other can break more symmetry than lexicographic ordering or multiset ordering constraints on both dimensions.

Next, we show that our filtering algorithm is the best way to propagate multiset ordering constraints. To simplify the presentation, we address only the row symmetry. Given a set of indistinguishable guests $\{g_i, g_{i+1}, \ldots, g_j\}$, we insist that the rows corresponding to such guests are multiset ordered: $\vec{R}_i \leqslant_m \vec{R}_{i+1} \leqslant_m \cdots \leqslant_m \vec{R}_j$. We impose such constraints by either using our filtering algorithm `MsetLeq`, or the *gcc* decomposition, or the arithmetic constraint.

We now consider several instances of the problem using the problem data given in CSPLib. We randomly select the host boats in such a way that the total spare capacity of the host boats is sufficient to accommodate all the guests. Table 3 shows the data. The last column of Table 3 gives the percentage of the total capacity used, which is a measure of constrainedness [38]. We instantiate H row-wise following the same protocol described previously.

The results of the experiments are shown in Table 4. Note that all the problem instances are solved for 5 time periods. The results show that `MsetLeq` maintains a significant advantage over the *gcc* decomposition and the arithmetic constraint. The solutions to the instances, which can be solved within an hour limit, are found quicker and compared to the *gcc*

Table 4
Progressive party problem: `MsetLeq` vs *gcc* decomposition and the arithmetic constraint with row-wise labelling.

Instance #	`MsetLeq` R			Arithmetic constraint R	*gcc* R		
	Fails	Choice points	Time (secs.)	Time (secs.)	Fails	Choice points	Time (secs.)
1	**10,839**	**10,963**	**8.3**	16	20,367	20,491	11.6
2	**56,209**	**56,327**	**46.8**	123.7	57,949	58,067	48.6
3	**27,461**	**27,575**	**17.1**	39.1	42,741	42,855	20.5
4	**420,774**	**420,888**	**280.5**	621.7	586,902	587,016	298.1
5	–	–	–	–	–	–	–
6	**5052**	**5170**	**3.8**	7.3	8002	8123	4.3
7	**86,432**	**86,547**	**65.5**	135.2	128,080	128,195	75.7
8	–	–	–	–	–	–	–
9	–	–	–	–	–	–	–

decomposition with much less failures. Note that `MsetLeq` and the arithmetic constraint methods create the same search tree.

9.2. Rack configuration problem

The **rack configuration problem** consists of plugging a set of electronic cards into racks with electronic connectors (prob031 in CSPLib). Each card is a certain card type. A card type i in the set $\mathcal{C}types$ is characterised by a tuple $\langle cp_i, d_i\rangle$, where cp_i is the power it requires, and d_i is the demand, which designates how many cards of that type have to be plugged. In order to plug a card into a rack, the rack needs to be assigned a rack model.

Each rack model i in the set $\mathcal{R}ackModels$ is characterised by a tuple $\langle rp_i, c_i, s_i\rangle$, where rp_i is the maximal power it can supply, c_i is its number of connectors, and s_i is its price. Each card plugged into a rack uses a connector. The problem is to decide how many among the set $\mathcal{R}acks$ of available racks are needed, and which model the racks are in order to plug all the cards such that:

- the number of cards plugged into a rack does not exceed its number of connectors;
- the total power of the cards plugged into a rack does not exceed its power;
- all the cards are plugged into some rack;
- the total price of the racks is minimised.

A matrix model of this problem is given in [17] and shown in Fig. 3. The idea is to assign a rack model to every available rack. Since some of the racks might not be needed in an optimal solution, a "dummy" rack model is introduced (i.e., a rack is assigned the dummy rack model when the rack is not needed). Furthermore, for every available rack, the number of cards of a particular card type plugged into the rack has to be determined. The assignment of rack models to racks is represented by a 1-d matrix R, indexed by $\mathcal{R}acks$, taking values from $\mathcal{R}ackModels$ which includes the dummy rack model. In order to represent the number of cards of a particular card type plugged into a particular rack, a 2-d matrix C of $\mathcal{C}types \times \mathcal{R}acks$ is introduced. A variable in this matrix takes values from $\{0, \ldots, maxConn\}$ where $maxConn$ is the maximum number of cards that can be plugged into any rack.

The dummy rack model is defined as a rack model where the maximal power it can supply, its number of connectors, and its price are all set to 0. The constraints enforce that the connector and the power capacity of each rack is not exceeded and every card type meets its demand. The objective is then to minimise the total cost of the racks.

The 2-d matrix C has partial row symmetry, because racks of the same rack model are indistinguishable and therefore their card assignments can be interchanged. To break this symmetry, we post multiset ordering constraints on the rows conditionally. Given two racks i and j, we enforce that the rows corresponding to such racks are multiset ordered if the racks are assigned the same rack model. That is:

$$R_i = R_j \rightarrow \langle C_{0,i}, \ldots, C_{n-1,i}\rangle \leqslant_m \langle C_{0,j}, \ldots, C_{n-1,j}\rangle$$

where n is the number of card types. We impose such constraints by either using our filtering algorithm `MsetLeq` or the arithmetic constraint. Unfortunately, we are unable to compare `MsetLeq` against the *gcc* decomposition in this problem, as Solver 5.3 does not allow us to post **IloDistribute** constraint conditionally.

We consider several instances of the rack configuration problem, which are described in Tables 5 and 6. In the experiments, we use the rack model and card type specifications given in [17], but we vary the demand of the card types randomly. As in [17], we search for the optimal solution by exploring the racks in turn. For each rack, we first instantiate its model and then determine how many cards from each card type are plugged into the rack.

The results of the experiments are shown in Table 7. `MsetLeq` is clearly much more efficient than the arithmetic constraint on every instance considered. Note that the two methods create the same search tree.

Matrices:

$$\begin{array}{cccccc} & \multicolumn{5}{c}{\longrightarrow \mathcal{R}acks \longrightarrow} \\ R_i & 0 & 0 & 1 & 2 & 2 \end{array}$$

$$\begin{array}{ccccc} & \multicolumn{4}{c}{\rightarrow \mathcal{C}types \rightarrow} \\ C_{i,j} & 0 & 0 & 0 & 0 \\ \downarrow & 0 & 0 & 0 & 0 \\ \mathcal{R}acks & 0 & 1 & 2 & 0 \\ \downarrow & 0 & 3 & 0 & 1 \\ & 10 & 0 & 0 & 0 \end{array}$$

Constraints:

$$\begin{array}{lll} (1) & \forall j \in \mathcal{R}acks\,. & \sum_{i \in \mathcal{C}types} C_{i,j} \leq c_{R_j} \\ (2) & \forall j \in \mathcal{R}acks\,. & \sum_{i \in \mathcal{C}types} C_{i,j} * cp_i \leq rp_{R_j} \\ (3) & \forall i \in \mathcal{C}types\,. & \sum_{j \in \mathcal{R}acks} C_{i,j} = d_i \end{array}$$

Objective:

$$minimize \sum_{i \in \mathcal{R}acks} s_{R_i}$$

Fig. 3. The matrix model of the rack configuration problem in [17].

Table 5
Rack model and card type specifications in the rack configuration problem [17].

Rack model	Power	Connectors	Price
1	150	8	150
2	200	16	200
Card type			Power
1			20
2			40
3			50
4			75

Table 6
Demand specification for the cards in the rack configuration problem.

Instance #	Demand			
	Type 1	Type 2	Type 3	Type 4
1	10	4	2	2
2	10	4	2	4
3	10	6	2	2
4	10	4	4	2
5	10	6	4	2
6	10	4	2	4

Table 7
Rack configuration problem: `MsetLeq` vs the arithmetic constraint.

Inst. #	`MsetLeq` R			Arithmetic constraint R
	Fails	Choice points	Time (secs.)	Time (secs.)
1	3052	3063	**0.2**	2.8
2	15,650	15,657	**0.6**	15.6
3	3990	3999	**0.2**	2.6
4	8864	8872	**0.4**	7.1
5	40,851	40,858	**1.5**	41.3
6	42,013	42,026	**1.6**	35.2

Table 8
Sport scheduling problem: `MsetLess` vs *gcc* decomposition and the arithmetic constraint with column-wise labelling. For one column, we first label the first slots; for the other, we first label the second slots.

n	Model	Fails	Choice points	Time (sec.)
5	`MsetLess` C	**1**	**10**	**0.8**
	Arithmetic constraint C	1	10	0.9
	gcc C	2	11	1.2
7	`MsetLess` C	**69**	**87**	**0.8**
	Arithmetic constraint C	69	87	1.3
	gcc C	74	92	1.3
9	`MsetLess` C	760,973	761,003	**121.3**
	Arithmetic constraint C	760,973	761,003	2500
	gcc C	2,616,148	2,616,176	656.4

9.3. Sport scheduling problem

This problem was introduced in Section 2. Fig. 1 shows a matrix model. The (extended) weeks over which the tournament is held, as well the periods of a week are indistinguishable. The rows and the columns of T and G are therefore symmetric. Note that we treat T as a 2-d matrix where the rows represent the periods and columns represent the (extended) weeks, and each entry of the matrix is a pair of values. The global cardinality constraints posted on the rows of T ensure that each of $1 \ldots n$ occur exactly twice in every row. In any solution to the problem, the rows when viewed as multisets are therefore equal. The *all-different* constraints posted on the columns state that each column is a permutation of $1 \ldots n$. Thus, the columns are also equal when viewed as multisets. Therefore, we cannot utilise multiset ordering constraints to break row and/or column symmetry of this model of the problem.

Scheduling a tournament between n teams means arranging $n(n-1)/2$ games. The model described in Fig. 1 assumes n is an even number. If n is an odd number instead, then we can still schedule $n(n-1)/2$ games provided that the games are played over n weeks and each week is divided into $(n-1)/2$ periods. The problem now requires that each team plays at most once a week, and every team plays exactly twice in the same period over the tournament. This version of the problem can be modelled using the original model in Fig. 1, as the *all-different* constraints on the rows and the cardinality constraints on the columns enforce the new problem constraints.

We can now post multiset ordering constraints on the columns of T to break column symmetry. Since the games are all different, no pair of columns can be equal, when viewed as multisets. Hence, we insist that the columns corresponding to the n weeks are strict multiset ordered: $\vec{C}_0 <_m \vec{C}_1 <_m \cdots <_m \vec{C}_{n-1}$. We enforce such constraints by either using our filtering algorithm `MsetLess`, or the *gcc* decomposition, or the arithmetic constraint. Since the multiset ordering constraints are posted on the columns, we instantiate T column-by-column. For one column, we first label the first slots; for the other, we first label the second slots. The results are shown in Table 8.

We observe that `MsetLess` is superior to the *gcc* decomposition. As the problem gets more difficult, `MsetLess` does more pruning and solves the problem quicker. The results moreover indicate a substantial gain in efficiency by using `MsetLess` in preference to the arithmetic constraint. Even though the same search tree is created by the two, constructing and propagating the arithmetic constraints is much more costly than running `MsetLess` to solve the multiset ordering constraints.

10. Conclusions

We have developed filtering algorithms for the multiset ordering (global) constraint $\vec{X} \leqslant_m \vec{Y}$ defined on a pair of vectors of variables. It ensures that the values taken by the vectors $\vec{X}$ and $\vec{Y}$, when viewed as multisets, are ordered. This global constraint is useful for breaking row and column symmetries of a matrix model and when searching for leximin solutions in fuzzy constraints. The filtering algorithms either prove that $\vec{X} \leqslant_m \vec{Y}$ is disentailed, or ensure GAC on $\vec{X} \leqslant_m \vec{Y}$.

The first algorithm `MsetLeq` is useful when $d \ll n$ and runs in $O(n)$ where n is the length of the vectors and d is the number of distinct values. This is often the case as the number of distinct values in a multiset is typically less than its cardinality to permit repetition. We further proposed another variant of the algorithm suitable when $d \gg n$. This identifies support by lexicographically ordering suitable sorted vectors. The complexity is then independent of the number of distinct values and is $O(n \log(n))$, as the cost of sorting dominates. We also have shown that `MsetLeq` can easily be modified for $\vec{X} <_m \vec{Y}$ by changing the definition of one of the flags. Moreover, the ease of maintaining the occurrence vectors incrementally helps detect entailment in a simple and dual manner to detecting disentailment.

Our experiments on the progressive party problem, the rack configuration problem, and the sport scheduling problem support the usefulness of multiset ordering constraints in the context of symmetry breaking and support our theoretical studies: even if it is feasible to post the arithmetic constraint, it is much more efficient to propagate the multiset ordering constraint using our filtering algorithm; furthermore, decomposing the multiset constraint carries a penalty either in the amount or the cost of constraint propagation.

In our future work, we plan to investigate whether the incremental cost for propagation can be made less than linear time. Moreover, we plan to understand whether it is worthwhile to a propate a chain of multiset ordering constraints and if that is the case devise an efficient filtering algorithm.

Acknowledgements

The authors would like to thank the anonymous reviewers for their useful comments on the presentation and Chris Jefferson for fruitful discussions on the work described in the article. B. Hnich is supported by Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No: SOBAG-108K027. I. Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship.

References

[1] K.R. Apt, Principles of Constraint Programming, Cambridge University Press, 2003.
[2] N. Bleuzen-Guernalec, A. Colmerauer, Narrowing a block of sortings in quadratic time, in: G. Smolka (Ed.), Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP-97), in: Lecture Notes in Computer Science, vol. 1330, Springer, 1997, pp. 2–16.
[3] N. Bleuzen-Guernalec, A. Colmerauer, Optimal narrowing of a block of sortings in optimal time, Constraints 5 (1–2) (2000) 85–118.
[4] N. Beldiceanu, M. Carlsson, J.-X. Rampon, Global constraints catalog, Technical Report T2005/08, Swedish Institute of Computer Science (SICS), 2005, Available at http://www.emn.fr/x-info/sdemasse/gccat/.
[5] S. Bouveret, M. Lemaître, Computing leximin-optimal solutions in constraint networks, Artificial Intelligence 173 (2) (2009) 343–364, this issue.
[6] J. Crawford, G. Luks, M. Ginsberg, A. Roy, Symmetry breaking predicates for search problems, in: Proceedings of the 5th International Conference on Knowledge Representation and Reasoning (KR '96), 1996, pp. 148–159.
[7] R. Debruyne, C. Bessière, Some practicable filtering techniques for the constraint satisfaction problem, in: M.E. Pollack (Ed.), Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97), Morgan Kaufmann, 1997, pp. 412–417.
[8] P. Brisset, H. El Sakkout, T. Frühwirth, C. Gervet, W. Harvey, M. Meier, S. Novello, T. Le Provost, J. Schimpf, K. Shen, M.G. Wallace, ECLiPSe Constraint Library Manual Release 5.6, 2003. Available at http://www.icparc.ic.ac.uk/eclipse/doc/doc/libman/libman.html.
[9] N. Barnier, P. Brisset, FaCiLe: A Functional Constraint Library Release 1.0, 2001. Available at http://www.recherche.enac.fr/opti/facile/doc/.
[10] H. Fargier, Problèmes de satisfaction de constraintes flexibles: application à l'ordonnancement de production, PhD thesis, University of Paul Sabatier, Tolouse, 1994.
[11] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh, Breaking row and column symmetry in matrix models, in: P. van Hentenryck (Ed.), Proceedings of the 8th International Conference on Principles and Practice of Constraint programming (CP-02), in: Lecture Notes in Computer Science, vol. 2470, Springer, 2002, pp. 462–476.
[12] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Matrix modelling: Exploiting common patterns in constraint programming, in: A.M. Frisch (Ed.), Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems, 2002, pp. 27–41.
[13] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Multiset ordering constraints, in: G. Gottlob, T. Walsh (Eds.), Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03), Morgan Kaufmann, 2003, pp. 221–226.
[14] A.M. Frisch, C. Jefferson, I. Miguel, Constraints for breaking more row and column symmetries, in: F. Rossi (Ed.), Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03), in: Lecture Notes in Computer Science, vol. 2833, Springer, 2003, pp. 318–332.
[15] P. Flener, J. Pearson (Eds.), Notes of the 2nd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02), CP-02 Post-conference Workshop, 2002. Available at http://www.it.uu.se/research/group/astra/SymCon02/.
[16] R.M. Haralick, G.L. Elliot, Increasing tree search efficiency for constraint satisfaction problems, Artificial Intelligence 14 (1980) 263–313.
[17] ILOG S.A., ILOG Solver 5.3 Reference and User Manual, 2002.
[18] Z. Kiziltan, B.M. Smith, Symmetry breaking constraints for matrix models, in: [15], 2002.
[19] I. Katriel, S. Thiel, Complete bound consistency for the global cardinality constraint, Constraints 10 (3) (2005) 191–217.
[20] Z. Kiziltan, T. Walsh, Constraint programming with multisets, in: [15], 2002.
[21] A.K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1) (1977) 99–118.
[22] A.K. Mackworth, On reading sketch maps, in: Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77), William Kaufmann, 1977, pp. 598–606.
[23] K. Marriott, P.J. Stuckey, Programming with Constraints, The MIT Press, 1998.
[24] K. Mehlhorn, S. Thiel, Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint, in: R. Dechter (Ed.), Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP-00), in: Lecture Notes in Computer Science, vol. 1894, Springer, 2000, pp. 306–319.
[25] J.F. Puget, On the satisfiability of symmetrical constrained satisfaction problems, in: H.J. Komorowski, Z.W. Ras (Eds.), Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems (ISMIS-93), in: Lecture Notes in Computer Science, vol. 689, Springer, 1993, pp. 350–361.
[26] C.-G. Quimper, P. van Beek, A. Lopez-Ortiz, A. Golynski, S.B. Sadjad, An efficient bounds consistency algorithm for the global cardinality constraint, in: F. Rossi (Ed.), Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP-03), in: Lecture Notes in Computer Science, vol. 2833, Springer, 2003, pp. 600–614.
[27] J.C. Régin, A filtering algorithm for constraints of difference in CSPs, in: Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94), AAAI Press, 1994, pp. 362–367.
[28] J.C. Régin, Generalized arc consistency for global cardinality constraints, in: Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI-96), AAAI Press/The MIT Press, 1996, pp. 209–215.
[29] F. Rossi, Constraint (logic) programming: a survey on research and applications, in: K.R. Apt, A.C. Kakas, E. Monfroy, F. Rossi (Eds.), New Trends in Constraints, in: Lecture Notes in Computer Science, vol. 1865, Springer, 2000, pp. 40–74.
[30] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, Elsevier, 2006.
[31] B.M. Smith, S.C. Brailsford, P.M. Hubbard, H.P. Williams, The progressive party problem: integer linear programming and constraint programming compared, Constraints 1 (1996) 119–138.
[32] T. Schiex, H. Fargier, G. Verfaille, Valued constraint satisfaction problems: hard and easy problems, in: C.S. Mellish (Ed.), Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95), Morgan Kaufmann, 1995, pp. 631–637.
[33] Swedish Institute of Computer Science, SICStus Prolog User's Manual, Release 3.12.0, November 2004. Available at http://www.sics.se/sicstus/docs/latest/pdf/sicstus.pdf.
[34] E.P.K. Tsang, Foundations of Constraint Satisfaction, Academic Press, 1993.

[35] P. van Hentenryck, L. Michel, L. Perron, J.C. Régin, Constraint programming in OPL, in: G. Nadathur (Ed.), Proceedings of the International Conference on the Principles and Practice of Declarative Programming (PPDP-99), in: Lecture Notes in Computer Science, vol. 1702, Springer, 1999, pp. 98–116.
[36] P. van Hentenryck, V.A. Saraswat, Y. Deville, Design, implementation and evaluation of the constraint language cc(FD), Journal of Logic Programming 37 (1–3) (1998) 139–164.
[37] M.G. Wallace, Practical applications of constraint programming, Constraints 1 (1–2) (1996) 139–168.
[38] J.P. Walser, Integer Optimization by Local Search—A Domain-Independent Approach, Lecture Notes in Artificial Intelligence, vol. 1637, Springer, 1999.