Available online at www.sciencedirect.com

SCIENCE DIRECT

ELSEVIER

Artificial Intelligence 170 (2006) 803–834

Artificial Intelligence

www.elsevier.com/locate/artint

Propagation algorithms for lexicographic ordering constraints

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] *DEIS, University of Bologna, Italy*
[d] *School of Computer Science, University of St Andrews, UK*
[e] *National ICT Australia and Department of CS & E, UNSW, Australia*

Received 12 April 2005; received in revised form 24 March 2006; accepted 27 March 2006

Abstract

Finite-domain constraint programming has been used with great success to tackle a wide variety of combinatorial problems in industry and academia. To apply finite-domain constraint programming to a problem, it is *modelled* by a set of constraints on a set of decision variables. A common modelling pattern is the use of matrices of decision variables. The rows and/or columns of these matrices are often symmetric, leading to redundancy in a systematic search for solutions. An effective method of breaking this symmetry is to constrain the assignments of the affected rows and columns to be ordered lexicographically. This paper develops an incremental propagation algorithm, `GACLexLeq`, that establishes generalised arc consistency on this constraint in $O(n)$ operations, where n is the length of the vectors. Furthermore, this paper shows that decomposing `GACLexLeq` into primitive constraints available in current finite-domain constraint toolkits reduces the strength or increases the cost of constraint propagation. Also presented are extensions and modifications to the algorithm to handle strict lexicographic ordering, detection of entailment, and vectors of unequal length. Experimental results on a number of domains demonstrate the value of `GACLexLeq`.

Keywords: Artificial intelligence; Constraints; Constraint programming; Constraint propagation; Lexicographic ordering; Symmetry; Symmetry breaking; Generalized arc consistency; Matrix models

1. Introduction

Constraints are a natural means of knowledge representation. For instance: the maths class must be timetabled between 9 and 11am on Monday; the helicopter can carry up to four passengers; the sum of the variables must equal 100. This generality underpins the success with which finite-domain constraint programming has been applied to a wide variety of disciplines [27]. To apply finite-domain constraint programming to a given domain, a problem must first be characterised or *modelled* by a set of constraints on a set of decision variables, which its solutions must satisfy. A common pattern arising in the modelling process is the use of matrices of decision variables, so-called *matrix models* [9]. For example, it is simple to represent many types of functions and relations in this way [15].

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

0004-3702/$ – see front matter
doi:10.1016/j.artint.2006.03.002

Concomitant with the selection of a matrix model is the possibility that the rows and/or columns of the matrix are symmetric. Consider, for instance, a matrix model of a constraint problem that requires finding a relation $\mathcal{R}$ on $\mathcal{A} \times \mathcal{B}$ where $\mathcal{A}$ and $\mathcal{B}$ are n-element and m-element sets of interchangeable objects respectively. The matrix, M, has n columns and m rows to represent the elements of $\mathcal{A}$ and $\mathcal{B}$. Each decision variable $M_{a,b}$ can be assigned either 1 or 0 to indicate whether $\langle a, b\rangle \in \mathcal{A} \times \mathcal{B}$ is in $\mathcal{R}$. Symmetry has been introduced because the matrix, whose columns and rows are indexed by $\mathcal{A}$ and $\mathcal{B}$, distinguishes the *position* of the elements of the sets, whereas $\mathcal{A}$ and $\mathcal{B}$ did not. Given a (non-)solution to this problem instance, a (non-)solution can be obtained by permuting columns of assignments and/or permuting rows of assignments. This is known as row and column symmetry [8]. Since similar behaviour can be found in multidimensional matrices of decision variables it is known more generally as *index symmetry*. As is well documented, symmetry can lead to a great deal of redundancy in systematic search [8].

As reviewed in Section 2.5 of this paper, lexicographic ordering constraints have been shown to be an effective method of breaking index symmetry. This paper describes a constraint propagation algorithm, `GACLexLeq`, that enforces this constraint. Given a lexicographic ordering constraint c, the propagation algorithm removes values from the domains of the constrained variables that cannot be part of any solution to c. This paper also shows that `GACLexLeq` establishes a property called generalised arc consistency,—that is it removes *all* infeasible values—while only requiring a number of operations linear in the number of variables constrained. The `GACLexLeq` algorithm is also incremental; if the domain of a variable is reduced the algorithm can re-establish generalised arc consistency without working from scratch.

Although the examples and experiments in the paper employ the lexicographic ordering constraint to break index symmetry, we note that lexicographic ordering can be used to break *any* symmetry that operates on the variables of an instance. The lex-leader method [5] breaks all symmetry by identifying a representative among the elements of the equivalence class of symmetries of an instance and adding a lexicographic ordering constraint for each other element of the equivalence class to ensure that only the representative is allowed.

The paper is organised as follows. Section 2 introduces the necessary background while Section 3 describes a number of applications used to evaluate our approach. Section 4 presents a propagation algorithm for the lexicographic ordering constraint. Then Section 5 discusses the complexity of the algorithm, and proves that the algorithm is sound and complete. Section 6 extends the algorithm to propagate a strict ordering constraint, to detect entailment, and to handle vectors of different lengths. Alternative approaches to propagating the lexicographic ordering constraint are discussed in Section 7. Section 8 demonstrates that decomposing a chain of lexicographic ordering constraints into lexicographic ordering constraints between adjacent or all pairs of vectors hinders constraint propagation. Computational results are presented in Section 9. Finally, we conclude and outline some future directions in Section 10.

2. Background

An instance of the *finite-domain constraint satisfaction problem* (CSP) consists of:

- a finite set of variables $\mathcal{X}$;
- for each variable $X \in \mathcal{X}$, a finite set $\mathcal{D}(X)$ of values (its domain); and
- a finite set $\mathcal{C}$ of constraints on the variables, where each constraint $c(X_1, \ldots, X_n) \in \mathcal{C}$ is defined over the variables $X_1, \ldots, X_n$ by a subset of $\mathcal{D}(X_1) \times \cdots \times \mathcal{D}(X_n)$ giving the set of allowed combinations of values. That is, c is an n-ary relation.

A *variable assignment* maps every variable in a given instance of CSP to a member of its domain. A variable assignment A is said to satisfy a constraint $c(X_1, \ldots, X_n)$ if and only if $\langle A(X_1), \ldots, A(X_n)\rangle$ is in the relation denoted by c. A solution to an instance of CSP is a variable assignment that satisfies all the constraints. An instance 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 instance is an instance of the *constraint optimisation problem*.

To impose total ordering constraints on variables and vectors of variables there must be an underlying total ordering on domains. If the domain of interest is not totally ordered, a total order can be imposed. And now, since domains are always finite, every domain is isomorphic to a finite set of integers. So we shall simplify the presentation by considering all domains to be finite sets of integers.

The minimum element in the domain of variable X is $\min(X)$, and the maximum is $\max(X)$. Throughout, *vars*(c) is used 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, or simply that X is assigned. If two variables X and X' are assigned the same value, then we write $X \doteq X'$, otherwise we write $\neg(X \doteq X')$. If v is assigned to X and v' is assigned to X' and $v < v'$ then we write $X \lessdot X'$.

A constraint c is *entailed* if all assignments of values to *vars*(c) satisfy c. If a constraint can be shown to be entailed then running the (potentially expensive) propagation algorithm can be avoided. Similarly, a constraint c is *disentailed* when all assignments of values to *vars*(c) violate c. Observe that if a constraint in a CSP instance can be shown to be disentailed then the instance has no solution.

2.1. Generalised arc consistency

This paper focuses on solving the CSP by searching for a solution in a space of assignments to subsets of the variables. Solution methods of this type use *propagation* algorithms that make inferences based on the domains of the constrained variables and the assignments that satisfy the constraint. These inferences are typically recorded as reductions in variable domains, where the elements removed cannot form part of any assignment satisfying the constraint, and therefore any solution. At each node in the search, constraint propagation algorithms are used to establish a local *consistency* property. A common example is generalised arc consistency (see [19]).

Definition 1 *(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 [18]).

2.2. Vectors and lexicographic ordering

A one-dimensional matrix, or vector, is an ordered list of elements. We denote a vector of n variables as $\vec{X} = \langle X_0, \dots, X_{n-1}\rangle$, while we denote a vector of n integers as $\vec{x} = \langle x_0, \dots, x_{n-1}\rangle$. In either case, a sub-vector from index a to index b inclusive is denoted by the subscript $a..b$, such as: $\vec{x}_{a..b}$. We define $\min(\langle X_0, \dots, X_{n-1}\rangle)$ to be $\langle \min(X_0), \dots, \min(X_{n-1})\rangle$ and, similarly, $\max(\langle X_0, \dots, X_{n-1}\rangle)$ to be $\langle \max(X_0), \dots, \max(X_{n-1})\rangle$. We define $\langle x_0, \dots, x_{n-1}\rangle \in \langle X_0, \dots, X_{n-1}\rangle$ to be true if and only if $x_i \in \mathcal{D}(X_i)$ for all $i \in [0, n-1]$. Finally, we define $\langle X_0, \dots, X_{n-1}\rangle \doteq \langle Y_0, \dots, Y_{n-1}\rangle$ to be true if and only if $X_i \doteq Y_i$ for all $i \in [1, n)$.

A vector of distinct 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 distinct variables, whose domains are $\{1,3,4\}$, $\{1,2,3,4,5\}$, and $\{1,2\}$, respectively.

Lexicographic ordering is a total ordering on vectors and is used, for instance, to order the words in a dictionary. Lexicographic ordering is defined on equal-sized vectors as follows.

Definition 2. Strict lexicographic ordering $\vec{x} <_{lex} \vec{y}$ between two length n vectors of integers $\vec{x}$ and $\vec{y}$ holds if and only if for some $k \in [0, n)$ it is the case that $\vec{x}_{0..k-1} = \vec{y}_{0..k-1}$ and $x_k < y_k$.

This ordering can be weakened to include equality.

Definition 3. Two equal-length vectors of integers $\vec{x}$ and $\vec{y}$ are lexicographically ordered $\vec{x} \leqslant_{lex} \vec{y}$ if and only if $\vec{x} <_{lex} \vec{y}$ or $\vec{x} = \vec{y}$.

Given two equal-length vectors of variables $\vec{X}$ and $\vec{Y}$, 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 3 and 2, respectively.

2.3. Variable symmetry

Various types of symmetries arise in instances of the CSP. All of the symmetries considered in this paper are *variable symmetries*. A variable symmetry for an instance I of the CSP is a bijection σ on the variables of I such that any total variable assignment A is a solution to I if and only if $A \cdot \sigma$ (the functional composition of A and σ) is. This has the obvious consequence that the identity function is a variable symmetry. It also means that every variable X has the same domain as $\sigma(X)$; otherwise, not every assignment would be mapped to an assignment.

As is the usual practice, we consider a set of symmetries on a problem instance, and the set always forms a group. This means that the inverse of a symmetry is also a symmetry and so is the composition of two symmetries. Such a set of symmetries is called a *symmetry group*. Two assignments, A and A', are said to be symmetric if, for some symmetry σ in the symmetry group, $A \cdot \sigma = A'$. A symmetry group partitions the set of total assignments for a CSP instance into equivalence classes, called *symmetry classes*, where the members of each equivalence class are pairwise symmetric. Notice that either all members of a symmetry class are solutions or none are.

Symmetry in a CSP instance introduces symmetry in its search space of partial assignments. The subtrees rooted at two symmetric partial assignments are symmetric to each other[1] and the solutions, if any, in one subtree are symmetric to those in the other subtree. Since the two symmetric subtrees contain symmetric solutions, there is no need to search both; any solutions found in one can be transformed into the solutions of the other simply by applying the relevant symmetry to it. It is important to note that a search space can contain symmetric subtrees that contain no solutions. Thus, even in cases where an instance has no solutions or where we are searching for only a single solution, subtrees that are symmetric to each other can be encountered.

The search of a space of partial assignments can be sped up by employing some method that avoids searching some or all parts of the space that are symmetric with parts that are searched. Such a method is often referred to as *symmetry breaking*. Symmetry is often broken "statically" by transforming a problem instance into one that has fewer symmetries. This is achieved by adding to the instance a constraint, called a *symmetry-breaking constraint*, that is true of some, but not all, symmetric assignments [22]. For example, consider a CSP instance with the variable symmetry that swaps X and Y. Adding the constraint $X \leqslant Y$ to the instance *breaks* the symmetry—that is, the resulting instance does not have the symmetry. We often talk of a set of symmetry-breaking constraints, which can be considered as the constraint consisting of the conjunction of all the members of the set.

We say that a symmetry-breaking constraint c is *consistent* for a CSP instance with a symmetry group if c is satisfied by at least one assignment in every symmetry class. The constraint c is *complete* if it is satisfied by at most one assignment in every symmetry class.

Crawford et al. [5] showed a method for generating a set of lexicographic ordering constraints that are consistent and complete for breaking any group of variable symmetries. It starts with an enumeration $\vec{X}$ of the variables in the instance. The set of symmetry-breaking constraints contains one constraint of the form $\langle X_0, \ldots, X_{n-1}\rangle \leqslant_{lex} \langle \sigma(X_0), \ldots, \sigma(X_{n-1})\rangle$ for each symmetry σ in the group. Since this set of constraints is often too large to use in practice, what is often used is a subset and/or simplification of these constraints, which gives a consistent, though incomplete, set of symmetry-breaking constraints. For this reason, lexicographic ordering constraints are widely used for breaking variable symmetries.

2.4. Matrix models and index symmetry

A *matrix model* is the formulation of a CSP with one or more matrices of decision variables [9]. Matrix models are a natural way to represent problems that involve finding a function or relation. For example, in the warehouse location problem (prob034 in CSPLib [13]), we need to find a function from stores to warehouses that determines which warehouse supplies each store. As a second example, in the steel mill slab design problem (prob038 in CSPLib), we need to find a function from orders to slabs that determines which slab is used to satisfy each order. Other examples are encountered later in this paper. Matrix models have been long used in integer linear programming [23], and are commonly used in constraint programming. Of the first 38 problems in CSPLib, at least 33 have matrix models, most of them already published and proved successful [9].

[1] For the purposes of this paper precise definitions of symmetric partial assignments and symmetric search trees are not necessary.

Matrices can be of any number of dimensions; the examples used in this paper have two or three dimensions. If a matrix X of variables has n dimensions, we denote each of its elements by $X_{i_1,\dots,i_n}$. In a two-dimensional matrix we refer to the first dimension as the columns of the matrix and the second dimension as the rows. In general, the values that are used to index the matrix can be drawn from any finite set. Without loss of generality, we shall assume that a dimension that has n index values uses $\{0,\dots,n-1\}$ as its index values.

Many matrix models have variable symmetries among the variables of the matrix (matrices). A common pattern of symmetry is that the rows and/or columns of an assignment to a 2D matrix can be swapped without affecting whether or not the assignment is a solution [9]. These are called *row symmetry* or *column* symmetry; the general term is *index symmetry*.

Definition 4. Let I be a CSP instance containing a two-dimensional matrix X of variables. A *column symmetry* for I is a variable symmetry, σ, for I such that for some bijection ρ on the column indices of X,

- $\sigma(X_{i,j}) = X_{\rho(i),j}$, for every variable $X_{i,j}$ in matrix X, and
- $\sigma(Y) = Y$ for every variable not in matrix X.

A *row symmetry* is the same as a column symmetry except that it operates on the second index of the matrix rather than on the first.

Thus, for a particular index i of a matrix, every index symmetry σ on i corresponds to a unique bijection ρ_i on the values of index i. We therefore identify an index symmetry by ρ and the index on which it operates.

Again, we are interested only in groups of index symmetries and, particularly, groups of two kinds. 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 2D matrix has total symmetry, we say that the matrix has *total column symmetry* (resp. *total row symmetry*). We also say that all the columns (resp. rows) of the matrix are interchangeable.

In many matrix models only a subset of the rows or columns are interchangeable. Let I be non-singleton, non-empty subset of the values of index i of a matrix. Let S be the set containing every bijection ρ on the values of index i such that $\rho(v) = v$ for every $v \notin I$. If every member of S is an index symmetry for i then we say that the matrix has *partial index symmetry*. If the first (resp. second) index of a 2D matrix has partial symmetry, we say that the matrix has *partial column symmetry* (resp. *partial row symmetry*). We also say that all the columns (resp. rows) in I of the matrix are interchangeable.

There is one final case to consider: an index may have partial index symmetry on multiple subsets of its values. For example, a CSP instance may have a 2D 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.

Section 3 of this paper gives several examples of CSPs that have index symmetry.

An $n \times m$ matrix with total row and total column symmetry has $n!m!$ symmetries. Consequently, it can be very costly to visit all the symmetric branches in a tree search. The next subsection explains how to break many of these symmetries.

2.5. Lexicographic ordering constraints for breaking index symmetry

The application of lexicographic ordering constraints considered by this paper is to breaking symmetries in CSP instances that have matrices with index symmetry. This section summarises the major results from Flener et al. [8] and Shlyakhter [25] on breaking index symmetries with lexicographic ordering constraints.

If a matrix in a CSP instance has total column (resp. row) symmetry, then the symmetry can be broken completely by a symmetry-breaking constraint that imposes a total ordering on the rows (resp. columns). The total ordering used here is the lexicographic ordering. In particular, we constrain the columns (resp. rows) to be non-decreasing as the value of the index increases. One way to achieve this, which is used in the experiments presented in this paper, is by imposing a constraint between adjacent columns (resp. rows). If X is an n by m matrix of decision variables, then we break column symmetry by imposing the constraints

$$\langle X_{i,0},\dots,X_{i,m}\rangle \leqslant_{lex} \langle X_{i+1,0},\dots,X_{i+1,m}\rangle \quad (i \in [0, n-2])$$

and we break row symmetry by imposing the constraints

$$\langle X_{0,j},\ldots,X_{n,j}\rangle \leqslant_{lex} \langle X_{0,j+1},\ldots,X_{n,j+1}\rangle \quad (j \in [0, m-2]).$$

Though these lexicographic ordering constraints are consistent and complete for total row or total column symmetry individually, they are not complete for a matrix that has both kinds of symmetry. They are, however, consistent and have been shown to be effective at removing many symmetries from the search spaces of many problems. Care must be taken in specifying these constraints; if, the column constraints take $X_{i,0}$ to be the most significant position in each column and the row constraints take $X_{n,j}$ to be the most significant position in each row, then the conjunction of the constraints is an inconsistent symmetry-breaking constraint.

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. 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.

Though it will not arise in this paper, lexicographic ordering constraints can be used in a similar manner to break symmetry in multi-dimensional matrices that have partial or total index symmetry on any number of its dimensions.

3. Applications

This section presents matrix models for three combinatorial problems in which lexicographic ordering constraints can be used to break index symmetry. These models are used in the experiments presented in Section 9.

3.1. Progressive party problem

The *progressive party problem* arises in the context of organising the social programme for a yachting rally (prob013 in www.csplib.org). Given a set of boats, each with a number of crew members and a capacity in terms of the number of guests it can accommodate, the problem is to designate a minimal subset of the boats as hosts and schedule the remaining boats to visit the hosts for a number of half-hour periods. All members of a particular guest crew remain together, and the crew of host boats remain on board their own boat. A guest boat cannot revisit a host and guest crews cannot meet more than once.

A simplified version of this problem, also studied by Smith et al. [24], removes the objective function, pre-designating the host boats and asking for only the schedule to be found. We study this version of the problem here. Let $\mathcal{P}eriods$ be the set of time periods, $\mathcal{G}uests$ the set of guest boats and $\mathcal{H}osts$ the set of host boats. Each host boat k has a capacity c_k (after taking its own crew into consideration), and each guest boat j has a crew size s_j. Smith et al.'s matrix model of this problem is given in Fig. 1. It comprises two matrices, H indexed by $\mathcal{P}eriods \times \mathcal{G}uests$ and B indexed by $\mathcal{P}eriods \times \mathcal{G}uests \times \mathcal{H}osts$. If $H_{i,j} = k$, or equivalently $B_{i,j,k} = 1$, then in period i, guest j visits host k. Although B is redundant given H, it allows the capacity constraints to be specified concisely.

Constraint (1) ensures that every pair of guest crews meet at most once. Note that the constraint sub-expression $H_{i,j_1} = H_{i,j_2}$ is *reified* to a 1 or a 0 value, depending on whether or not it is satisfied. Hence, the summation counts the number of periods in which guest crews j_1 and j_2 are assigned the same host value. Constraint (2) disallows a guest crew from revisiting a host boat over the course of the schedule. Here, for the sake of presentation, the periods are considered to be the integers $0,\ldots,p-1$. Constraint (3) ensures that the capacity of each host boat is never exceeded. Finally, Constraint (4) is a *channelling* constraint [3], which maintains consistency between the H and B matrices.

The time periods are interchangeable, hence there is total symmetry on the first index of H and the first index of B.[2] Guests with equal crew size are interchangeable, which means that there is partial symmetry on the second index of H and the second index of B. Finally, hosts with equal capacity are interchangeable, hence there is partial symmetry on the third index of B.

2 Observe that this is the same symmetry in both matrices and therefore cannot be broken independently in the two matrices. This issue arises elsewhere, but for concision we do not re-address it.

Given:
$\mathcal{P}eriods$, $\mathcal{G}uests$, $\mathcal{H}osts$,
matrix c_k indexed by $k \in \mathcal{H}osts$,
matrix s_j indexed by $j \in \mathcal{G}uests$

Decision Variables:
matrix $H_{i,j}$ with domain $\mathcal{H}osts$ indexed by $i \in \mathcal{P}eriods$, $j \in \mathcal{G}uests$,
matrix $B_{i,j,k}$ with domain $\{0, 1\}$ indexed by $i \in \mathcal{P}eriods$, $j \in \mathcal{G}uests$, $k \in \mathcal{H}osts$

Constraints:
(1) $\sum_{i \in \mathcal{P}eriods}(H_{i,j_1} = H_{i,j_2}) \leqslant 1$ $\quad (j_1, j_2 \in \mathcal{G}uests,\ j_1 < j_2)$
(2) *all-different*$(\langle H_{0,j}, H_{1,j}, \ldots, H_{p-1,j}\rangle)$ $\quad (j \in \mathcal{G}uests)$
(3) $\sum_{j \in \mathcal{G}uests} s_j * B_{i,j,k} \leqslant c_k$ $\quad (i \in \mathcal{P}eriods,\ k \in \mathcal{H}osts)$
(4) $H_{i,j} = k \leftrightarrow B_{i,j,k} = 1$ $\quad (i \in \mathcal{P}eriods,\ j \in \mathcal{G}uests,\ k \in \mathcal{H}osts)$

Fig. 1. Matrix model of the progressive party problem from [24].

Given:
$\mathcal{T}emplates$, $\mathcal{V}ariations$, s,
matrix d_j indexed by $j \in \mathcal{V}ariations$

Decision Variables:
matrix Run_i with domain $\{0, \ldots, \max(d_j)\}$ indexed by $i \in \mathcal{T}emplates$,
matrix $T_{i,j}$ with domain $\{0, \ldots, s\}$ indexed by $i \in \mathcal{T}emplates$, $j \in \mathcal{V}ariations$

Constraints:
(1) $\sum_{j \in \mathcal{V}ariations} T_{i,j} = s$ $\quad (i \in \mathcal{T}emplates)$
(2) $\sum_{i \in \mathcal{T}emplates} Run_i * T_{i,j} \geqslant d_j$ $\quad (j \in \mathcal{V}ariations)$

Objective:
minimize $\sum_{i \in \mathcal{T}emplates} Run_i$

Fig. 2. Matrix model of the template design problem from [21].

3.2. Template design problem

The *template design problem* (prob002 in CSPLib) involves configuring a set of printing templates with design variations that need to be printed to meet specified demand. Given is a set of variations of a design, with a common shape and size and such that the number of required "pressings" of each variation is known. The problem is to design a set of templates, with a common capacity to which each must be filled, by assigning zero or more instances of a variation to each template. A design should be chosen that minimises the total number of "runs" of the templates required to satisfy the number of pressings required for each variation. As an example, the variations might be for cartons for different flavours of cat food, such as fish or chicken, where ten thousand fish cartons and twenty thousand chicken cartons need to be printed. The problem would then be to design a set of templates by assigning a number of fish and/or chicken designs to each template such that a minimal number of runs of the templates is required to print all thirty thousand cartons.

Proll and Smith address this problem by fixing the number of templates and minimising the total number of pressings [21]. We will adopt their model herein. Let $\mathcal{T}emplates$ be the fixed-size set of templates, each with capacity s, to which variations are to be assigned. Let $\mathcal{V}ariations$ be the set of variations. Each variation, j, is described by a demand d_j that specifies the minimum number of pressings required. Proll and Smith's model is given in Fig. 2. It comprises two matrices, *Run* indexed by $\mathcal{T}emplates$, and T indexed by $\mathcal{T}emplates \times \mathcal{V}ariations$. If $Run_i = j$, then template i is printed j times, where j ranges between 0 and the maximum number of pressings required by any single variation. Similarly, if $T_{i,j} = k$ then template i is assigned k instances of variation j, where $0 \leqslant k \leqslant s$.

Given:
v, b, r, k, λ

Let:
$\mathcal{B} = \{0, \ldots, b-1\}$ and $\mathcal{V} = \{0, \ldots, v-1\}$

Decision Variables:
matrix $X_{i,j}$ with domain $\{0, 1\}$ indexed by $i \in \mathcal{B}$, $j \in \mathcal{V}$

Constraints:
(1) $\sum_{i \in \mathcal{B}} X_{i,j} = r \quad (j \in \mathcal{V})$
(2) $\sum_{j \in \mathcal{V}} X_{i,j} = k \quad (i \in \mathcal{B})$
(3) $\sum_{i \in \mathcal{B}} X_{i,j_1} * X_{i,j_2} = \lambda \quad (j_1, j_2 \in \mathcal{V}, j_1 < j_2)$

Fig. 3. Matrix model of the BIBD problem from [20].

Constraint (1) ensures that every template has all its s slots occupied, and constraint (2) specifies that the total production of each variation is at least its demand. The objective is then to minimise the total number of pressings.

In this model all the templates are interchangeable, hence *Run* has total symmetry on its index and T has total symmetry on its first index. Variations of equal demand are interchangeable, hence there is total symmetry on the second index of T.

3.3. Balanced incomplete block design problem

The *balanced incomplete block design* (*BIBD*) *problem* is a standard combinatorial problem from design theory [4] with applications in experimental design and cryptography (prob028 in CSPLib). Given the tuple of natural numbers $\langle v, b, r, k, \lambda \rangle$, the problem is to arrange v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks.

Meseguer and Torras' model [20], which we adopt in this paper, is given in Fig. 3. It comprises one matrix, X, indexed by $\mathcal{B} \times \mathcal{V}$, where $\mathcal{B} = \{0, \ldots, b-1\}$ is the set of blocks and $\mathcal{V} = \{0, \ldots, v-1\}$ is the set of objects. $X_{i,j} = 1$ if and only if block i contains object j. Constraints (1) and (2) ensure, respectively, that each object appears in r blocks and that each block contains k objects. Constraint (3) is a scalar product constraint that requires every pair of objects to meet in exactly λ blocks. Since both the objects and the blocks are interchangeable, the matrix X has total row and total column symmetry.

4. A propagation algorithm

We present a propagation algorithm for the lexicographic ordering constraint that either detects the disentailment of $\vec{X} \leqslant_{lex} \vec{Y}$ or prunes inconsistent values to establish GAC on $\vec{X} \leqslant_{lex} \vec{Y}$.

In order to simplify the presentation, here and throughout the entire paper we consider only the case where $\vec{X}$ and $\vec{Y}$ are variable-distinct in the following sense:

Definition 5. A pair of vectors is *variable-distinct* if each contains only CSP variables, each contains no repeated variables, and there are no variables common to both vectors.

Note that the majority of applications, such as those described in the previous section, involve ordering variable-distinct vectors. Kiziltan [17] gives an algorithm similar to that presented here, but which caters for the cases where variables are repeated. In the presence of repeated variables, the algorithms given herein can be used by the following simple expedient. Consider a constraint c with two occurrences of variable X. We can replace c with $c' \wedge (X = X')$, where X' has the same domain as X and c' results from replacing one occurrence of X in c with X'. This step can be repeated to remove all repeated occurrences of a single variable. This approach preserves soundness, but not completeness.

The key to the algorithm is that there are two significant indices within $\vec{X}$ and $\vec{Y}$. The index α is the least index at which $\vec{X}$ and $\vec{Y}$ are not ground and equal. If there is no such index α is n. The index β is the least index in $[\alpha, n)$ such that $\vec{X}_{\beta..n-1} >_{lex} \vec{Y}_{\beta..n-1}$ is entailed. If there is no such index, β is $n+1$. The algorithm only needs to consider the regions of $\vec{X}$ and $\vec{Y}$ within indices $[\alpha, \beta)$.

4.1. A worked example

We now illustrate the `GACLexLeq` algorithm by considering its operation on the lexicographic ordering constraint $\vec{X} \leqslant_{lex} \vec{Y}$, where $\vec{X}$ and $\vec{Y}$ are variable-distinct and have the domains:

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

The program variables `alpha` and `beta` are used to record the values α and β. When the domains of $\vec{X}$ and $\vec{Y}$ are reduced `alpha` and `beta` may no longer contain the values α and β so the algorithm needs to update these program variables.

We traverse the vectors once in order to initialise `alpha` and `beta`. Starting from index 0, we move first to index 1 and then to index 2 because $X_0 \doteq Y_0$ and $X_1 \doteq Y_1$. We stop at 2 and set `alpha` $= 2 = \alpha$ as Y_2 is not assigned.

$$\begin{array}{lllllll} \vec{X} = \langle\{1\}, \{2\}, & \{2\}, & \{1,3,4\}, & \{1,2,3,4,5\}, & \{1,2\}, & \{3,4,5\}\rangle \\ \vec{Y} = \langle\{1\}, \{2\}, & \{0,1,2\}, & \{1\}, & \{0,1,2,3,4\}, & \{0,1\}, & \{0,1,2\}\rangle \\ & \texttt{alpha}\uparrow & & & & \end{array}$$

We initialise `beta` by traversing the vectors starting from `alpha`. At index 2 $\min(X_2) = \max(Y_2)$, therefore $X_2 \geqslant Y_2$ is entailed. Hence, β may equal 2, but this can only be determined by examining the variables with greater indices. At index 3, $\min(X_3) = \max(Y_3)$. This neither precludes nor confirms $\beta = 2$. At index 4, it is possible to satisfy $X_4 \leqslant Y_4$. Hence, we have determined $\beta \neq 2$. At index 5 $\min(X_5) = \max(Y_5)$, therefore $X_5 \geqslant Y_5$ is entailed. Similarly, β may equal 5, but this can only be determined by examining the variables with greater indices. At index 6, $\min(X_6) > \max(Y_6)$, so $X_6 > Y_6$ is entailed. Hence, $\vec{X}_{5..6} >_{lex} \vec{Y}_{5..6}$ is entailed, and therefore $\beta = 5$, to which `beta` is initialised.

$$\begin{array}{lllllll} \vec{X} = \langle\{1\}, \{2\}, & \{2\}, & \{1,3,4\}, & \{1,2,3,4,5\}, & \{1,2\}, & \{3,4,5\}\rangle \\ \vec{Y} = \langle\{1\}, \{2\}, & \{0,1,2\}, & \{1\}, & \{0,1,2,3,4\}, & \{0,1\}, & \{0,1,2\}\rangle \\ & \texttt{alpha}\uparrow & & & \uparrow\texttt{beta} & \end{array}$$

The algorithm restricts domain pruning to the index `alpha`. As values are removed from the domains of the variables, the value of `alpha` monotonically increases and the value of `beta` monotonically decreases. The constraint is disentailed if the values of `alpha` and `beta` become equal.

Consider the vectors again. As the vectors are assigned and equal at the indices less than `alpha`, there is no support for any value in $\mathcal{D}(Y_{\texttt{alpha}})$ that is less than $\min(X_{\texttt{alpha}})$. We therefore remove 0 and 1 from $\mathcal{D}(Y_{\texttt{alpha}})$ and increment `alpha` to $3 = \alpha$:

$$\begin{array}{lllllll} \vec{X} = \langle\{1\}, \{2\}, \{2\}, & \{1,3,4\}, & \{1,2,3,4,5\}, & \{1,2\}, & \{3,4,5\}\rangle \\ \vec{Y} = \langle\{1\}, \{2\}, \{2\}, & \{1\}, & \{0,1,2,3,4\}, & \{0,1\}, & \{0,1,2\}\rangle \\ & \texttt{alpha}\uparrow & & \uparrow\texttt{beta} & \end{array}$$

Similarly, there is no support for any value in $\mathcal{D}(X_{\texttt{alpha}})$ greater than $\max(Y_{\texttt{alpha}})$. We therefore remove 3 and 4 from $\mathcal{D}(X_{\texttt{alpha}})$ and increment `alpha` to $4 = \alpha$:

$$\begin{array}{llll} \vec{X} = \langle\{1\}, \{2\}, \{2\}, \{1\}, & \{1,2,3,4,5\}, & \{1,2\}, & \{3,4,5\}\rangle \\ \vec{Y} = \langle\{1\}, \{2\}, \{2\}, \{1\}, & \{0,1,2,3,4\}, & \{0,1\}, & \{0,1,2\}\rangle \\ & \texttt{alpha}\uparrow & \uparrow\texttt{beta} & \end{array}$$

Since `alpha` $=$ `beta` $- 1$, there is no support for any value in $\mathcal{D}(X_{\texttt{alpha}})$ greater than or equal to $\max(Y_{\texttt{alpha}})$. Similarly, there is no support for any value in $\mathcal{D}(Y_{\texttt{alpha}})$ less than or equal to $\min(X_{\texttt{alpha}})$. We must therefore establish arc consistency on $X_{\texttt{alpha}} < Y_{\texttt{alpha}}$. Doing so removes 4 and 5 from $\mathcal{D}(X_{\texttt{alpha}})$, and also 0 and 1 from $\mathcal{D}(Y_{\texttt{alpha}})$:

$$\begin{array}{llll} \vec{X} = \langle\{1\}, \{2\}, \{2\}, \{1\}, & \{1,2,3\}, & \{1,2\}, & \{3,4,5\}\rangle \\ \vec{Y} = \langle\{1\}, \{2\}, \{2\}, \{1\}, & \{2,3,4\}, & \{0,1\}, & \{0,1,2\}\rangle \\ & \texttt{alpha}\uparrow & \uparrow\texttt{beta} & \end{array}$$

The constraint $\vec{X} \leqslant_{lex} \vec{Y}$ is now GAC.

4.2. Theoretical background

This section formally defines α and β and presents two theorems that show their significance in propagating the constraint $\vec{X} \leqslant_{lex} \vec{Y}$.

Definition 6. Given two length n variable-distinct vectors, $\vec{X}$ and $\vec{Y}$, α is the least index in $[0, n)$ such that $\neg(X_\alpha \doteq Y_\alpha)$ or, if no such index exists, n.

Definition 7. Given two length n variable-distinct vectors, $\vec{X}$ and $\vec{Y}$, β is the least index in $[\alpha, n)$ such that

$$\exists k \in [\beta, n) \,.\, \big(\min(X_k) > \max(Y_k) \ \wedge\ \min(\vec{X}_{\beta..k-1}) = \max(\vec{Y}_{\beta..k-1})\big)$$

or, if no such index exists, $n + 1$.

The relative values of α and β provide important information about the constraint $\vec{X} \leqslant_{lex} \vec{Y}$. By definition, β cannot be strictly less than α. The following two theorems in turn address the cases when $\alpha = \beta$ and when $\alpha < \beta$. The first theorem states that if $\alpha = \beta$ then the constraint is disentailed.

Theorem 1. *Let $\vec{X}$ and $\vec{Y}$ be a pair of length n variable-distinct vectors. $\alpha = \beta$ if and only if $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed.*

Proof. ($\Rightarrow$) By Definition 6, $\vec{X}_{0..\alpha-1} \doteq \vec{Y}_{0..\alpha-1}$, and by Definition 7 $\vec{X}_{\beta..n-1} >_{lex} \vec{Y}_{\beta..n-1}$ is entailed. Since $\beta = \alpha$ there is no assignment that can satisfy $\vec{X} \leqslant_{lex} \vec{Y}$.

($\Leftarrow$) If $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed then $\vec{X} >_{lex} \vec{Y}$ is entailed. From the definition of strict lexicographic ordering there must be an index i such that $\min(X_i) > \max(Y_i)$. Let j be the least such index. If $j = 0$ then, by Definitions 6 and 7, $\alpha = j = \beta$. Otherwise observe that $\min(X_h) \leqslant \max(X_h)$ for all $h \in [0, j-1]$. If $\vec{X}_{0..j-1} \doteq \vec{Y}_{0..j-1}$ then from Definition 6 $\alpha = j$ and from Definition 7 $\beta = j$. Otherwise, let $g < j$ be the least index such that $\neg(X_g \doteq Y_g)$. Then from Definition 6 $g = \alpha$ and from Definition 7 $g = \beta$. □

The second theorem states that if $\beta > \alpha$ then the constraint is GAC if and only if $\alpha = n$ or all values at index α have support.

Theorem 2. *Let $\vec{X}$ and $\vec{Y}$ be a pair of length n variable-distinct vectors such that $\beta > \alpha$.* GAC($\vec{X} \leqslant_{lex} \vec{Y}$) *if and only if either*

1. $\beta = \alpha + 1$ *and* AC($X_\alpha < Y_\alpha$)*, or*
2. $\beta > \alpha + 1$ *and* AC($X_\alpha \leqslant Y_\alpha$).

Proof. ($\Rightarrow$) Assume $\vec{X} \leqslant_{lex} \vec{Y}$ is GAC but either $X_\alpha < Y_\alpha$ is not AC when $\beta = \alpha + 1$ or $X_\alpha \leqslant Y_\alpha$ is not AC when $\beta > \alpha + 1$. Then either there exists no value in $\mathcal{D}(Y_\alpha)$ greater than (or equal to) a value a in $\mathcal{D}(X_\alpha)$, or there exists no value in $\mathcal{D}(X_\alpha)$ less than (or equal to) a value b in $\mathcal{D}(Y_\alpha)$. Since the variables are all assigned and pairwise equal at indices less than α, a or b lacks support from all the variables in the vectors. This contradicts that $\vec{X} \leqslant_{lex} \vec{Y}$ is GAC.

($\Leftarrow$) All variables with indices less than α are assigned and pairwise equal. Therefore, the assignment $X_\alpha \lessdot Y_\alpha$ provides support for all values at indices greater than α. Hence, given $\beta = \alpha + 1$ and AC($X_\alpha < Y_\alpha$) the constraint is GAC. Similarly, if $\beta > \alpha + 1$ and AC($X_\alpha \leqslant Y_\alpha$) then $X_\alpha \lessdot Y_\alpha$ supports all values at indices greater than α. It remains to consider the assignments that set $X_\alpha \doteq Y_\alpha$. If $\beta = n + 1$, then by definition $\min(X_i) \leqslant \max(Y_i)$ for all $i \in [\alpha, n)$ and the constraint is GAC. Otherwise, from the definition of β, $\min(X_i) \leqslant \max(Y_i)$ for all $i \in [\alpha, \beta)$ and $\neg(X_{\beta-1} \doteq Y_{\beta-1})$. Since $\vec{X}$ and $\vec{Y}$ are variable-distinct, $X_\alpha \doteq Y_\alpha$ is supported by the combination of $X_{\beta-1} \lessdot Y_{\beta-1}$ and $\vec{X}_{\alpha+1..\beta-2} \doteq \vec{Y}_{\alpha+1..\beta-2}$. Hence, the constraint is GAC. □

Algorithm `GACLexLeq`

`EstablishGAC`	
A1	$\texttt{alpha} := 0$
A2	**while** $(\texttt{alpha} < n \wedge X_{\texttt{alpha}} \doteq Y_{\texttt{alpha}})$ **do** $\texttt{alpha} := \texttt{alpha} + 1$
A3	**if** $(\texttt{alpha} = n)$ **then** $\texttt{beta} := n + 1$, **return**
A4	$i := \texttt{alpha}$
A5	$\texttt{beta} := -1$
A6	**while** $(i \neq n \wedge \min(X_i) \leqslant \max(Y_i))$ **do**
A6.1	**if** $(\min(X_i) = \max(Y_i))$ **then** **if** $(\texttt{beta} = -1)$ **then** $\texttt{beta} := i$
A6.2	**else** $\texttt{beta} := -1$
A6.3	$i := i + 1$
A7	**if** $(i = n)$ **then** $\texttt{beta} := n + 1$
A8	**else if** $(\texttt{beta} = -1)$ **then** $\texttt{beta} := i$
A9	**if** $(\texttt{alpha} = \texttt{beta})$ **then** disentailed
A10	`ReEstablishGAC(alpha)`
`ReEstablishGAC`(i in $[0, n)$) Triggered when $\min(X_i)$ or $\max(Y_i)$ changes	
A11	**if** $(i = \texttt{alpha} \wedge i + 1 = \texttt{beta})$ **then** `EstablishAC`$(X_i < Y_i)$
A12	**if** $(i = \texttt{alpha} \wedge i + 1 < \texttt{beta})$ **then**
A12.1	`EstablishAC`$(X_i \leqslant Y_i)$
A12.2	**if** $(X_i \doteq Y_i)$ **then** `UpdateAlpha`
A13	**if** $(\texttt{alpha} < i < \texttt{beta})$ **then**
A13.1	**if** $((i = \texttt{beta} - 1 \wedge \min(X_i) = \max(Y_i)) \vee \min(X_i) > \max(Y_i))$ **then**
A13.2	`UpdateBeta`$(i - 1)$
`UpdateAlpha`	
A14	$\texttt{alpha} := \texttt{alpha} + 1$
A15	**if** $(\texttt{alpha} = n)$ **then** return
A16	**if** $(\texttt{alpha} = \texttt{beta})$ **then** disentailed
A17	**if** $(\neg(X_{\texttt{alpha}} \doteq Y_{\texttt{alpha}}))$ **then** `ReEstablishGAC(alpha)`
A18	**else** `UpdateAlpha`
`UpdateBeta`(i in $[0, n)$)	
A19	$\texttt{beta} := i + 1$
A20	**if** $(\texttt{alpha} = \texttt{beta})$ **then** disentailed
A21	**if** $(\min(X_i) < \max(Y_i))$ **then**
A21.1	**if** $(i = \texttt{alpha})$ **then** `EstablishAC`$(X_i < Y_i)$
A22	**else** `UpdateBeta`$(i - 1)$

Fig. 4. Constituent procedures of the `GACLexLeq` algorithm.

4.3. *Algorithm* `GACLexLeq`

Based on Theorems 1 and 2, we have designed an efficient linear-time propagation algorithm, `GACLexLeq`, which either detects the disentailment of $\vec{X} \leqslant_{lex} \vec{Y}$ or prunes only inconsistent values so as to establish GAC on $\vec{X} \leqslant_{lex} \vec{Y}$. It is presented in Fig. 4.

Throughout the paper, we assume that our propagation algorithms are used in a certain manner, which is common in the practice of constraint programming. If we are searching for a solution to a set of constraints that contains a constraint of the form $\vec{X} \leqslant_{leq} \vec{Y}$, then the constraint will be imposed with a call to `EstablishGAC`. In searching down any path of the search space `ReEstablishGAC` is called whenever the domain of a variable in $\vec{X}$ or $\vec{Y}$ is reduced in a certain manner. As many domain reductions do not destroy the GAC property, our algorithms specify the conditions under which `ReEstablishGAC` is triggered. Finally, we assume that the solver detects when the domain of a variable has been reduced to the empty set and interrupts the execution of the propagation algorithm and signals disentailment. Thus our algorithms do not test for empty domains. When other conditions lead our algorithms detect that the constraint is disentailed, the algorithms signal this and return. We assume that the propagation algorithms are never called with a constraint for which disentailment has been detected.

Let us discuss GACLexLeq, beginning with EstablishGAC. Throughout the paper we refer to lines A1–A9 as the "initialisation step" since these lines initialise the program variables alpha and beta to α and β, as defined in Definitions 6 and 7. Following the initialisation step ReEstablishGAC is called (line A10) to establish generalised arc consistency.

Line A2 traverses $\vec{X}$ and $\vec{Y}$, starting at index 0, until either it reaches the end of the vectors (all pairs of variables are assigned and equal), or it finds an index where the pair of variables are not assigned and equal. In the first case, the algorithm returns (line A3) as $\vec{X} \leqslant_{lex} \vec{Y}$ is entailed. In the second case, alpha is set to the smallest index where the pair of variables are not assigned and equal. The vectors are traversed at line A6, starting at index alpha, until either the end of the vectors is reached (none of the pairs of variables have $\min(X_i) > \max(Y_i)$), or an index i where $\min(X_i) > \max(Y_i)$ is found. In the first case, beta is set to $n + 1$ (line A7). In the second case, beta is guaranteed to be at most i (line A8). If, however, there exists an $h \in [0, i - 1]$ such that $\min(\vec{X}_{h..i-1}) = \max(\vec{Y}_{h..i-1})$, then beta can be revised to the least such h (line A6.1).

If alpha = beta then disentailment is detected and EstablishGAC terminates, signalling that this is the case (line A9). Otherwise, it is sufficient to call ReEstablishGAC with index alpha (line A10) to establish generalised arc consistency, as we will show.

We now consider ReEstablishGAC itself. Apart from the call made by EstablishGAC, this procedure is triggered whenever the lower bound of one of the variables in $\vec{X}$, or the upper bound of one of the variables in $\vec{Y}$, is modified. The justification for this is that lexicographic ordering is a *monotonic* constraint. If a value of X_i has any support then it is supported by the maximum value of Y_i; likewise, if a value of Y_i has any support then it is supported by the minimum value of X_i. Hence new support needs to be sought only if one of these bounds is changed.

ReEstablishGAC consists of three mutually-exclusive branches, which we describe in turn. Line A11 establishes AC on $X_i < Y_i$, in accordance with Theorem 2. Again, because of monotonicity, this is a simple step. First, a check is made to ensure that the upper bound of X_i is supported by the upper bound of Y_i. If not, the upper bound of X_i is revised accordingly. Similarly, the lower bound of Y_i is compared against the lower bound of X_i and revised if necessary.

Lines A12–12.2 cater for the case when alpha and beta are not adjacent. Again exploiting monotonicity, line A12.1 establishes AC on $X_i \leqslant Y_i$, in accordance with Theorem 2. If, following this step, X_i and Y_i are assigned and equal, alpha no longer reflects α and is updated via UpdateAlpha. In lines A14 and A18 of UpdateAlpha the vectors are traversed until α is reached. If $\alpha = n$ the procedure returns (line A15) because GAC has been established. If $\alpha = \beta$, disentailment is signalled (line A16) in accordance with Theorem 1. Otherwise, ReEstablishGAC is called (line A17).

Finally, lines A13–A13.2 deal with a call to ReEstablishGAC with an index between alpha and beta. In this case, it may be the case that beta does not reflect β and must be updated. The condition for updating beta is derived from Definition 7: at i either $\min(X_i) > \max(Y_i)$, or i is beta − 1 and $\min(X_i) = \max(Y_i)$. The program variable beta is updated by calling UpdateBeta($i - 1$). In lines A19 and A22, the vectors are traversed until β is reached. Again, if $\alpha = \beta$, disentailment is signalled (line A20) in accordance with Theorem 1. Otherwise, line A21.1 establishes AC on $X_i \leqslant Y_i$ if α is adjacent to β.

5. Theoretical properties

We begin by analysing the worst-case time complexity of GACLexLeq, before establishing its soundness and completeness.

5.1. *Time complexity*

This section considers the time complexity of using GACLexLeq to establish or re-establish GAC as well as the total time complexity of establishing and then repeatedly re-establishing GAC in moving down a single branch in a search space.

The GACLexLeq algorithm updates the domains of variables by establishing arc-consistency on constraints of the form $X \leqslant Y$ and $X < Y$, both of which modify the domains only by tightening their bounds. Depending on how domains are implemented, the time taken to perform this tightening operation could be constant, a function of the cardinality of the domain, or even a function of the structure of the domain. In order to abstract away from this issue, we shall measure the run time of our algorithms by a pair of functions $\langle f_1, f_2\rangle$, where f_1 gives the total number of operations performed (counting domain bounds modification as a single operation) and f_2 gives the total number of domain bounds modifications. We also assume that $\min(X)$ and $\max(X)$ can be computed in constant time. Thus, the cost of establishing AC on $X \leqslant Y$ or $X < Y$ is $\langle O(1), 0\rangle$ if the constraint is AC, and $\langle O(1), 1\rangle$ if it is not.

Our approach to the complexity analysis is first (in Lemma 2) to characterise the amount of computation performed as a function of $M_{\alpha\beta}$, the total number of times that `alpha` and `beta` are modified, and then obtain the complexity result by establishing that $M_{\alpha\beta}$ is bounded by the vector length plus one.

Lemma 1. *Every execution of* `UpdateAlpha` *increments* `alpha` *and every execution of* `UpdateBeta` *decreases* `beta`. *During the execution of* `ReEstablishGAC` *the value of* `alpha` *never decreases and the value of* `beta` *never increases.*

Proof. Each execution of `UpdateAlpha` increments `alpha` on Line A14 and modifies `alpha` nowhere else. Each execution of `UpdateBeta` assigns a value to `beta` on line A19 and is modified nowhere else. Let us confirm that line A19 decreases `beta`. On the initial call to `UpdateBeta` (line 13.2) the value passed is no greater than `beta`-2 and on subsequent calls (line A22) i is decremented. Hence, immediately before line A19 is executed, $i+1$ is strictly less than `beta`. An execution of `ReEstablishGAC` never modifies `alpha` or `beta` other than through calls to `UpdateAlpha` and `UpdateBeta`; hence it never decreases `alpha` or increases `beta`. □

Lemma 2. *Given a pair of equal-length variable-distinct vectors, the time complexity of* `ReEstablishGAC` *is* $\langle O(M_{\alpha\beta}), O(M_{\alpha\beta})\rangle$, *where* $M_{\alpha\beta}$ *is the total number of times that the* `alpha` *or* `beta` *variables are modified during the execution.*

Proof. With any execution e of `ReEstablishGAC` we can associate a string, s_e, of the symbols "r", "a" and "b" that indicates the sequence of invocations of `ReEstablishGAC`, `UpdateAlpha` and `UpdateBeta`, respectively, that take place during execution e. It is useful to observe, from the structure of the algorithm, that the set of all such strings is $r(a^*|a^+r)b^*$. Consider an arbitrary execution e of `ReEstablishGAC`. We start the proof by first showing that for execution e $M_{\alpha\beta} = \Theta(|s_e|)$. From Lemma 1 we know that each execution of `UpdateAlpha` increments `alpha` and each execution of `UpdateBeta` decreases `beta`. Each execution of `ReEstablishGAC`, except the first, is preceded by an execution of `UpdateAlpha`, whose execution incremented `alpha`. Therefore $M_{\alpha\beta} = \Theta(|s_e|)$. Observing that each of `ReEstablishGAC`, `UpdateAlpha` and `UpdateBeta` perform $O(1)$ total operations, we conclude that the total number of operations performed during execution e is $O(M_{\alpha\beta})$. Observing `ReEstablish-GAC`, `UpdateAlpha` and `UpdateBeta` perform at most one domain bounds modification, we conclude that the total number of such modifications performed during execution e is $O(M_{\alpha\beta})$. □

Theorem 3. *Given a pair of length n variable-distinct vectors the time complexity of both* `ReEstablishGAC` *and* `EstablishGAC` *is* $\langle O(n), O(n)\rangle$.

Proof. First consider `ReEstablishGAC`. Observe that when `ReEstablishGAC` is called on line A10, `alpha` is at least 0 and `beta` is at most $n+1$. From Lemma 1 it follows that this is true for any subsequent call to `ReEstablishGAC`, be it from line A17 or from outside the algorithm. Observe that the value of `alpha` never exceeds that of `beta`. From Lemma 1 and these observations, it follows that, for any execution of `ReEstab-lishGAC`, $M_{\alpha\beta} \leqslant n+1$. Therefore, it follows from Lemma 2 that the complexity of both `ReEstablishGAC` and `EstablishGAC` is $\langle O(n), O(n)\rangle$.

Now consider `EstablishGAC`. Observe that the complexity of the initialisation step (lines A1 through A9) is $\langle O(n), 0\rangle$. Adding this to the execution time of `ReEstablishGAC` (line A10) gives a total complexity of $\langle O(n), O(n)\rangle$. □

As explained at the beginning of Section 4.3, if we are searching for a solution to a set of constraints that contains a constraint of the form $\vec{X} \leqslant_{lex} \vec{Y}$, then the constraint will be imposed with a call to `EstablishGAC` and down any branch of a search space a call will be made to `ReEstablishGAC` every time a lower bound of some X_i is increased, or a upper bound of some Y_i is decreased. Direct application of Theorem 3 would tell us that if such a sequence has

$k+1$ calls then its total execution time is $\langle O(kn), O(kn)\rangle$. However, because the algorithms are incremental, the actual execution time is much less.

Theorem 4. *Given a pair of equal-length variable-distinct vectors, the time complexity of executing* `EstablishGAC` *followed by a sequence of k executions of* `ReEstablishGAC` *is* $\langle O(n+k), O(n+k)\rangle$.

Proof. From Lemma 1 and the arguments put forward in the previous proof, we know that over the entire sequence of operations $M_{\alpha\beta}$ will not exceed $n+1$. Hence, by Lemma 2, the number of operations performed related to moves of `alpha` and `beta` is $O(n)$. But each of the k executions in the sequence will perform at least a constant number of operations, even if `alpha` and `beta` are not moved; this takes $O(k)$ total operations. Thus the total number of operations executed in the sequence is $O(n+k)$. By a similar argument, the number of domain bounds modifications performed in the sequence of executions is also $O(n+k)$. □

5.2. Soundness and completeness

We now turn our attention to the soundness and completeness of `GACLexLeq`. By soundness of a propagation algorithm we mean that the algorithm only removes domain elements that participate in no satisfying assignment to the constrained variables and signals disentailment only if the constraint is disentailed. By completeness we mean that the algorithm signals disentailment if the constraint is disentailed, otherwise it removes all domain elements that participate in no satisfying assignment. We begin with the initialisation part of `EstablishGAC`.

Lemma 3. *Let $\vec{X}$ and $\vec{Y}$ be a pair of length n variable-distinct vectors. The initialisation step of* `EstablishGAC` *on $\vec{X}$ and $\vec{Y}$ sets* `alpha` *and* `beta` *to α and β respectively. It signals that $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed if and only if this is the case.*

Proof. Line A2 of `EstablishGAC` traverses $\vec{X}$ and $\vec{Y}$, starting at index 0, until either it reaches the end of the vectors (all pairs of variables are assigned and equal), or it finds an index where the pair of variables are not assigned and equal. In the first case, $\mathrm{GAC}(\vec{X} \leqslant_{lex} \vec{Y})$ holds and the algorithm returns with `alpha` and `beta` set as per Definitions 6 and 7 (line A3). In the second case, `alpha` is set to the smallest index where the pair of variables are not assigned and equal, as per Definition 6. The vectors are traversed at line A6, starting at index `alpha`, until either the end of the vectors is reached (none of the pairs of variables have $\min(X_i) > \max(Y_i)$), or an index i where $\min(X_i) > \max(Y_i)$ is found. In the first case, `beta` is set to $n+1$ (line A7) as per Definition 7. In the second case, `beta` is guaranteed to be at most i (line A8). If, however, there exists an $h \in [0, i-1]$ such that $\min(\vec{X}_{h..i-1}) = \max(\vec{Y}_{h..i-1})$, then `beta` can be revised to the least such h (line A6.1), as per Definition 7. If `alpha` = `beta` then $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed by Theorem 1 and thus `EstablishGAC` terminates signalling that this is the case (line A9). □

Having established that `alpha` and `beta` are correctly initialised via lines A1 to A9 of `EstablishGAC`, we show that these two variables are correctly updated by `UpdateAlpha` and `UpdateBeta` respectively. Note that these two procedures may trigger pruning. We also show that such pruning is sound and complete.

Lemma 4. *Let $\vec{X}$ and $\vec{Y}$ be a pair of length n variable-distinct vectors,* `alpha` $< \alpha$ *and* `beta` $\geqslant \beta$. `UpdateAlpha` *is sound. If $\alpha < \beta$ or* `beta` $= \beta$, `UpdateAlpha` *is complete. If $\vec{X} \leqslant_{lex} \vec{Y}$ is not disentailed,* `UpdateAlpha` *terminates with α =* `alpha`.

Proof. Induction on $k = n - \alpha$.

Base case: $k = 0$, hence $\alpha = n$. By definition, $\vec{X}_{\texttt{alpha}..\alpha-1} \doteq \vec{Y}_{\texttt{alpha}..\alpha-1}$. At line A14, `alpha` is incremented. If `alpha` is now equal to α, `updateAlpha` terminates at line A15, which is complete. Otherwise, since $\alpha = n$, $\beta = n+1$, so the condition of line A16 cannot be met. Similarly, the condition of line A17 cannot be met. Therefore, `UpdateAlpha` is called recursively at line A18. This process repeats until `alpha` $= n$, and `UpdateAlpha` terminates correctly.

Inductive case: We assume that the theorem is true for $0 \leqslant k < j$. We prove that it is true for $k = j$, equivalently when $\alpha = n - j$. Again, $\vec{X}_{\texttt{alpha}..\alpha-1} \doteq \vec{Y}_{\texttt{alpha}..\alpha-1}$ by definition. As above, `alpha` is updated until it is set correctly to α at line A14. The condition of line A15 cannot be met, by assumption. If alpha $=$ beta, which is possible only if beta $= \beta$, disentailment is signalled at line A16. This is sound and required for completeness. Since alpha $= \alpha$, the condition of line A17 is met, and ReEstablishGAC(alpha) is called. Consider first the case where $\alpha = \beta - 1$. From Theorem 2, establishing AC on $X_{\texttt{alpha}} < Y_{\texttt{alpha}}$ is sound and required for completeness. Since the condition of line A16 was not met, and alpha $= \alpha$, the condition of line A11 is met only if beta $= \beta$, which is the second completeness condition. AC is established at line A11, and the algorithm terminates as required. Consider now the case where $\alpha < \beta - 1$. From Theorem 2, establishing AC on $X_{\texttt{alpha}} \leqslant Y_{\texttt{alpha}}$ is sound and required for completeness. Since alpha $= \alpha$, and $\beta \leqslant$ beta, the condition of line A12 is met. AC is established at line A12.1 as required. If this results in $X_{\texttt{alpha}} \doteq Y_{\texttt{alpha}}$, then UpdateAlpha is called with $k < j$ (since α is now a greater index). This is sound and complete by the induction assumption. Since alpha $= i$ the third branch cannot be entered. □

Lemma 5. *Let $\vec{X}$ and $\vec{Y}$ be a pair of length n variable-distinct vectors, $0 \leqslant$ alpha $\leqslant \alpha$, and for all h in $[\beta, i+1]$, $\min(X_h) \geqslant \max(X_h)$. In addition, assume that $\alpha < \beta$ and $\beta - 1 \leqslant i < n$, or $\alpha = \beta$ and* alpha $-1 \leqslant i < n$. UpdateBeta(i) *is sound and complete. Furthermore, if $\vec{X} \leqslant_{lex} \vec{Y}$ is not disentailed,* UpdateBeta(i) *terminates with $\beta =$* beta.

Proof. We divide the proof into three cases.

Case 1: $\alpha = \beta$. From Theorem 1, it is sound to signal disentailment and completeness requires that disentailment is signalled. We assume that the initial call to UpdateBeta is with $i > \alpha$. An initial call with $i \leqslant \alpha$ is a simple sub-case, as will become clear. By assumption, at every index h in $[\beta, i+1]$, $\min(X_h) \geqslant \max(Y_h)$. Hence, the condition of line A21 cannot be met, and UpdateBeta is called recursively until $\beta =$ beta. By assumption alpha $\leqslant \alpha$. If $\alpha =$ alpha, then the condition of line A20 is met, and disentailment is signalled as required. If alpha $< \alpha$, the recursion continues until alpha $- 1$ since, by Definition 6, $\vec{X}_{\texttt{alpha}..\alpha-1} \doteq \vec{Y}_{\texttt{alpha}..\alpha-1}$. At this point, the condition of line A20 is met, and disentailment is signalled as required.

Case 2: $\alpha = \beta - 1$. From Theorem 2, it is sound to establish $\mathrm{AC}(X_\alpha < Y_\alpha)$ and completeness requires that AC is established. As above, UpdateBeta traverses the vectors until the condition of either line A20 or line A21 is met. However, the condition of line A20 cannot be met: by assumption, alpha $\leqslant \alpha < \beta$ and, by Definition 7, $\min(X_{\beta-1}) < \max(Y_{\beta-1})$. By assumption, at every index h in $[\beta, i+1]$ $\min(X_h) \geqslant \max(Y_h)$. Hence, the recursion will end when i reaches $\alpha = \beta - 1$. At this point the conditions of lines A21 and A21.1 are met and AC is established as required. If this does not cause disentailment, the algorithm returns with beta $= \beta$.

Case 3: $\alpha < \beta - 1$. Similarly to above, beta is updated to β, at which point the condition of line A21 is met. Clearly, however, the condition of line A21.1 cannot be met and the algorithm terminates correctly. □

In the context of a constraint solver, where it is intended that GACLexLeq is used, there is no guarantee that ReEstablishGAC will be invoked immediately following every individual change to the vectors of variables that it constrains. Hence, we must show that ReEstablishGAC is sound and complete following a number of such changes. We also show that both EstablishGAC and ReEstablishGAC ensure that alpha and beta are set correctly upon termination. This, in turn, ensures that successive applications of ReEstablishGAC following a sequence of changes remains sound and complete.

Theorem 5. *Let $\vec{X}$ and $\vec{Y}$ be a pair of length n variable-distinct vectors such that $\mathrm{GAC}(\vec{X} \leqslant_{lex} \vec{Y})$, and let* alpha $= \alpha \neq \beta =$ beta. *Let domain reductions be made to some variables in $\vec{X}$ and $\vec{Y}$, whose indices form the set I. Let P be the process of executing, in any order,* ReEstablishGAC(i) *for each $i \in I$ that meets the trigger conditions. P is sound and complete. If $\vec{X} \leqslant_{lex} \vec{Y}$ is not disentailed, P terminates with $\alpha =$* alpha *and $\beta =$* beta.

Proof. Soundness: The algorithm prunes values directly at lines A11 and A12.1, and as a result of invoking UpdateAlpha and UpdateBeta at lines A12.2 and A13.2. We consider each case in turn. The condition of line A11 is met if 1) alpha $= \alpha$ and beta $= \beta$, in which case the propagation performed is sound from Theorem 2, or 2) $\alpha = \beta$ and alpha $= \alpha$ or beta $= \alpha$. Here, propagation will result in an empty domain and disentailment will be signalled, which is sound from Theorem 1. If alpha $< \alpha$, the propagation at line A12.1 will have no effect and UpdateAlpha is invoked. If alpha $= \alpha$, the propagation at line A12.1 is sound from Theorem 2. If this results in $X_i \doteq Y_i$, `UpdateAlpha` is invoked. In either case, the soundness conditions for `UpdateAlpha` are met by assumption. If the conditions of lines A13 and A13.1 are met, then `UpdateBeta`$(i-1)$ is invoked. The soundness conditions for `UpdateBeta` are met by assumption.

Completeness: We divide completeness into three cases, according to the relative values of α and β.

Case 1: $\alpha < \beta - 1$. From Theorem 2, it suffices to show that AC$(X_\alpha \leqslant Y_\alpha)$ is established. If α remains equal to `alpha`, then there are three possibilities: (a) X_α and Y_α are unchanged and AC$(X_\alpha \leqslant Y_\alpha)$ holds as it did before the sequence of domain reductions. (b) Domain reductions at α have not affected the lower bound of X_α or the upper bound of Y_α. From monotonicity AC$(X_\alpha \leqslant Y_\alpha)$ still holds. (c) The lower bound of X_α and/or the upper bound of Y_α has been revised, in which case $\alpha \in I$ and the propagation is performed at line A12.1 as required. If `alpha` $< \alpha$ then the domain of $X_{\texttt{alpha}}$ and/or $Y_{\texttt{alpha}}$ has been reduced to a single value. Since AC$(X_{\texttt{alpha}} \leqslant Y_{\texttt{alpha}})$ held before the sequence of domain reductions, this means that the lower bound of $X_{\texttt{alpha}}$ and/or the upper bound of $Y_{\texttt{alpha}}$ has been reduced. Hence, `UpdateAlpha` is invoked at line A12.2, which is complete from Lemma 4.

Case 2: $\alpha = \beta - 1$. If $\alpha = n$, no propagation is necessary. Otherwise it suffices to show that AC$(X_\alpha < Y_\alpha)$ is established from Theorem 2. If α remains equal to `alpha`, then there are two possibilities: (a) `beta` $= \beta$. If either X_α and Y_α are unchanged, or domain reductions at α have affected neither the lower bound of X_α nor the upper bound of Y_α, from monotonicity, AC$(X_\alpha < Y_\alpha)$ holds as it did before the sequence of domain reductions. If the lower bound of X_α and/or the upper bound of Y_α has been revised then $\alpha \in I$ and AC$(X_\alpha < Y_\alpha)$ is established at line A11 as required. (b) $\beta <$ `beta`. For some index $j \in [\beta, \texttt{beta})$, the lower bound of X_j or the upper bound of Y_j has been revised such that for all h in $[\beta, j]$, $\min(X_h) \geqslant \max(X_h)$. Hence, $j \in I$ and, `UpdateBeta`$(j-1)$ is called at line A13.2. It is complete from Lemma 5. If `alpha` $< \alpha$ then, as above, `alpha` is updated to α. If, at this point, `beta` $= \beta$, then AC is established at line A11 as required. If `beta` $> \beta$, then `UpdateBeta` is called with its completeness conditions met, as above.

Case 3: $\alpha = \beta$. From Theorem 1, completeness requires that disentailment is signalled. By assumption, it is not the case that both `alpha` $= \alpha$ and `beta` $= \beta$. By similar arguments to the above, either `UpdateAlpha` is invoked and is complete by Lemma 4, or `UpdateBeta` is invoked and is complete by Lemma 5.

Accuracy of α and β on termination (assuming $\vec{X} \leqslant_{lex} \vec{Y}$ is not disentailed): If `alpha` $< \alpha$ then, as argued above, the domain of $X_{\texttt{alpha}}$ and/or $Y_{\texttt{alpha}}$ has been reduced to a single value. Hence, `UpdateAlpha` is invoked at line A12.2, which terminates with $\alpha =$ `alpha` from Lemma 4. If $\beta <$ `beta` then, as also argued above, for some index j, where $\beta \leqslant j <$ `beta`, the lower bound of X_j or the upper bound of Y_j has been revised such that for all h in $[\beta, j]$ $\min(X_h) \geqslant \max(X_h)$. Hence, $j \in I$ and, `UpdateBeta`$(j-1)$ is called at line A13.2. From Lemma 5 it terminates with `beta` $= \beta$. □

Theorem 6. *Let $\vec{X}$ and $\vec{Y}$ be a pair of length n variable-distinct vectors.* `EstablishGAC` *on $\vec{X}$ and $\vec{Y}$ is sound and complete. If $\vec{X} \leqslant_{lex} \vec{Y}$ is not disentailed,* `EstablishGAC` *terminates with $\alpha =$* `alpha` *and $\beta =$* `beta`.

Proof. From Lemma 3, after line A9 of `EstablishGAC` has been executed, `alpha` and `beta` are set to α and β. If disentailment has not already been detected, line A10 invokes `ReEstablishGAC(alpha)`. From Theorem 2, if GAC$(\vec{X} \leqslant_{lex} \vec{Y})$ then `ReEstablishGAC(alpha)` has no effect, which is complete. Otherwise, either AC$(X_{\texttt{alpha}} \leqslant X_{\texttt{beta}})$ or AC$(X_{\texttt{alpha}} < X_{\texttt{beta}})$ does not hold. If `alpha` is adjacent to `beta`, AC$(X_{\texttt{alpha}} < X_{\texttt{beta}})$ is established at line A11, which is sound and complete. If `alpha` and `beta` are not adjacent, AC$(X_{\texttt{alpha}} \leqslant X_{\texttt{beta}})$ is established at line A12.1. If now $X_{\texttt{alpha}} \doteq Y_{\texttt{alpha}}$, `UpdateAlpha` is called at line A12.2. From Lemma 4, `UpdateAlpha` is sound, complete, and if $\vec{X} \leqslant_{lex} \vec{Y}$ is not disentailed, it terminates with $\alpha =$ `alpha` and $\beta =$ `beta`. □

6. Extensions

6.1. Strict lexicographic ordering constraint

With very little effort, `GACLexLeq` can be adapted to obtain a propagation algorithm, `GACLexLess`, which either detects the disentailment of $\vec{X} <_{lex} \vec{Y}$ or prunes only inconsistent values so as to establish GAC on $\vec{X} <_{lex} \vec{Y}$. The reason the two algorithms are so similar is that as soon as β takes a value other than $n+1$, `GACLexLeq` enforces strict lexicographic ordering on the vectors.

Before showing how we modify `GACLexLeq`, we give the necessary theoretical background. We define the index α between two vectors $\vec{X}$ and $\vec{Y}$ as in Definition 6. However, β is now the least index such that $\vec{X}_{\beta..n-1} <_{lex} \vec{Y}_{\beta..n-1}$ is disentailed, or n if no such index exists.

Definition 8. Given two length n variable-distinct vectors, $\vec{X}$ and $\vec{Y}$, β is the least index in $[\alpha, n)$ such that

$$\big(\exists k \in [\beta, n) \,.\, \big(\min(X_k) > \max(Y_k) \;\wedge\; \min(\vec{X}_{\beta..k-1}) = \max(\vec{Y}_{\beta..k-1})\big)\big) \quad \vee \quad \min(\vec{X}_{\beta..n-1}) = \max(\vec{Y}_{\beta..n-1})$$

or, if no such index exists, n.

We again make use of α and β to detect disentailment as well as prune inconsistent values.

Theorem 7. *Let $\vec{X}$ and $\vec{Y}$ be a pair of length n variable-distinct vectors. $\beta = \alpha$ if and only if $\vec{X} <_{lex} \vec{Y}$ is disentailed.*

Proof. ($\Rightarrow$) Identical to the proof of Theorem 1.

($\Leftarrow$) If $\vec{X} <_{lex} \vec{Y}$ is disentailed then $\vec{X} \geqslant_{lex} \vec{Y}$ is entailed. If $\vec{X} >_{lex} \vec{Y}$ is entailed, the proof follows as for Theorem 1, substituting Definition 8 for Definition 7. Otherwise just $\vec{X} \geqslant_{lex} \vec{Y}$ is entailed. Since the vectors are variable-distinct, $\forall i \in [0, n) \min(X_i) \geqslant \max(Y_i)$. Hence, from Definition 6, α is the least index such that $\neg(X_\alpha \doteq Y_\alpha)$. Thus, from Definition 8, $\beta = \alpha$. $\square$

The conditions for GAC to hold are also similar:

Theorem 8. *Given a pair of length n variable-distinct vectors, $\vec{X}$ and $\vec{Y}$, if $\beta > \alpha$ then* GAC($\vec{X} <_{lex} \vec{Y}$) *if and only if either of the following conditions holds*:

1. $\beta = \alpha + 1$ *and* AC($X_\alpha < Y_\alpha$)*, or*
2. $\beta > \alpha + 1$ *and* AC($X_\alpha \leqslant Y_\alpha$).

Proof. ($\Rightarrow$) Similar to the proof of Theorem 2.

($\Leftarrow$) Since $\vec{X}_{0..\alpha-1} \doteq \vec{Y}_{0..\alpha-1}$, the assignment $X_\alpha \lessdot Y_\alpha$ supports all values in the domains of $\vec{X}_{\alpha+1..n-1}$ and $\vec{Y}_{\alpha+1..n-1}$. Hence, given $\beta = \alpha + 1$ and $AC(X_\alpha < Y_\alpha)$, the constraint is GAC. Similarly, if $\beta > \alpha + 1$ and $AC(X_\alpha \leqslant Y_\alpha)$ then $X_\alpha \lessdot Y_\alpha$ supports all values in the domains of $\vec{X}_{\alpha+1..n-1}$ and $\vec{Y}_{\alpha+1..n-1}$. It remains to consider $X_\alpha \doteq Y_\alpha$. Irrespective of whether $\beta = n$, from the definition of β, $\neg(X_{\beta-1} \doteq Y_{\beta-1})$ and $\min(X_i) \leqslant \max(Y_i)$ for all $i \in [\alpha, \beta)$. Since $\vec{X}$ and $\vec{Y}$ are variable-distinct, $X_\alpha \doteq Y_\alpha$ supports and is supported by the combination of $X_{\beta-1} \lessdot Y_{\beta-1}$ and $\vec{X}_{\alpha-1..\beta-2} \doteq \vec{Y}_{\alpha-1..\beta-2}$. Hence, the constraint is GAC. $\square$

We now consider the simple modifications to `GACLexLeq` necessary to obtain the propagation algorithm `GACLexLess`, which is presented in Fig. 5. First, the initialisation step of `EstablishGAC` (i.e. lines B1–B8) must reflect the new definition of β. From Definition 8 and Theorem 7, the constraint is disentailed if $\alpha = n$. Line B3 deals with this case. A further change to the initialisation step is at line B7. Line A7 of `GACLexLeq` has been removed so that `beta` is assigned to n correctly if there is no index from which the tail of the vectors is guaranteed to be ordered lexicographically equal or decreasing. The second modification is to `UpdateAlpha`, as indicated in the figure. Again, if `alpha` reaches n the constraint is disentailed. No other modifications are necessary.

6.2. Entailment

Even though `GACLexLeq` is a sound and complete propagation algorithm, it does not detect entailment. This section extends the `GACLexLeq` algorithm to a new algorithm, `GACLexLeqEntailed`, that can detect entailment. Mimicking the treatment of disentailment, when entailment is detected the algorithm signals this and returns. Once a constraint is entailed it remains GAC even if the domains of the constrained variables are later reduced; hence there is no need ever to re-establish GAC.

It is not hard to show when $\vec{X} \leqslant_{lex} \vec{Y}$ is entailed.

Algorithm `GACLexLess`

`EstablishGAC`

B1 $\texttt{alpha} := 0$
B2 **while** $(\texttt{alpha} < n \wedge X_{\texttt{alpha}} \doteq Y_{\texttt{alpha}})$ **do** $\texttt{alpha} := \texttt{alpha} + 1$
⟩⟩ **B3** **if** $(\texttt{alpha} = n)$ **then** disentailed
B4 $i := \texttt{alpha}$
B5 $\texttt{beta} := -1$
B6 **while** $(i \neq n \wedge \min(X_i) \leqslant \max(Y_i))$ **do**
B6.1   **if** $(\min(X_i) = \max(Y_i))$ **then** **if** $(\texttt{beta} = -1)$ **then** $\texttt{beta} := i$
B6.2   **else** $\texttt{beta} := -1$
B6.3 $i := i + 1$
⟩⟩ **B7** **if** $(\texttt{beta} = -1)$ **then** $\texttt{beta} := i$
B8 **if** $(\texttt{alpha} = \texttt{beta})$ **then** disentailed
B9 `ReEstablishGAC(alpha)`

`ReEstablishGAC`(i in $[0, n)$) Triggered when $\min(X_i)$ or $\max(Y_i)$ changes
Identical to `GACLexLeq`

`UpdateAlpha`

B14 $\texttt{alpha} := \texttt{alpha} + 1$
⟩⟩ **B15** **if** $(\texttt{alpha} = n)$ **then** disentailed
B16 **if** $(\texttt{alpha} = \texttt{beta})$ **then** disentailed
B17 **if** $(\neg(X_{\texttt{alpha}} \doteq Y_{\texttt{alpha}}))$ **then** `ReEstablishGAC(alpha)`
B18 **else** `UpdateAlpha`

`UpdateBeta`(i in $[0, n)$)
Identical to `GACLexLeq`

Fig. 5. Constituent procedures of the `GACLexLess` algorithm. Modifications from `GACLexLeq` are indicated by '⟩⟩'.

Theorem 9. *Given a pair of length n variable-distinct vectors,* $\vec{X}$ *and* $\vec{Y}$, $\vec{X} \leqslant_{lex} \vec{Y}$ *is entailed if and only if* $\max(\vec{X}) \leqslant_{lex} \min(\vec{Y})$.

Proof. (⇒) Since $\vec{X} \leqslant_{lex} \vec{Y}$ is entailed, any combination of assignments satisfies $\vec{X} \leqslant_{lex} \vec{Y}$. Hence, $\min(\vec{X}) \leqslant_{lex} \max(\vec{Y})$.

(⇐) Any $\vec{x} \in \vec{X}$ is lexicographically less than or equal to any $\vec{y} \in \vec{Y}$. Hence, $\vec{X} \leqslant_{lex} \vec{Y}$ is entailed. □

To exploit this fact, we introduce γ, which is the least index in $[\alpha, n)$ such that $\vec{X}_{\gamma..n-1} \leqslant_{lex} \vec{Y}_{\gamma..n-1}$ is entailed, or n if no such index exists.

Definition 9. Given a pair of length n variable-distinct vectors, $\vec{X}$ and $\vec{Y}$, γ is the least index in $[\alpha, n)$ such that

$$\exists k \in [\gamma, n)\,.\,\big(\max(X_k) < \min(Y_k) \wedge \max(\vec{X}_{\gamma..k-1}) = \min(\vec{Y}_{\gamma..k-1})\big) \quad \vee \quad \max(\vec{X}_{\gamma..n-1}) = \min(\vec{Y}_{\gamma..n-1})$$

or, if no such index exists, n.

Theorem 10. *Given a pair of length n variable-distinct vectors,* $\vec{X}$ *and* $\vec{Y}$, $\vec{X} \leqslant_{lex} \vec{Y}$ *is entailed if and only if* $\gamma = \alpha$.

Proof. (⇒) If $\vec{X} \leqslant_{lex} \vec{Y}$ is entailed, then, from the definition of lexicographic ordering and the fact that the vectors are variable distinct, either a) there exists an index i such that $\max(X_i) < \min(Y_i)$ and $\max(\vec{X}_{0..i-1}) = \min(\vec{Y}_{0..i-1})$ or b) $\max(\vec{X}_{0..n-1}) = \min(\vec{Y}_{0..n-1})$. First, consider a). Let k be the least index such that the variables are not assigned and equal. By Definition 6, $\alpha = k$. Clearly, k lies in $[0, i]$. From the first part of Definition 9, $\gamma = k$. Now consider b). If the vectors are assigned and equal, then $\alpha = n = \gamma$ by Definitions 6 and 9. Otherwise, let k be defined as above. From the second part of Definition 9, $\gamma = k$.

(⇐) By Definition 6, $\vec{X}_{0..\alpha-1} = \vec{Y}_{0..\alpha-1}$ is entailed, and by Definition 9 $\vec{X}_{\alpha..n-1} \leqslant_{lex} \vec{Y}_{\alpha..n-1}$ is entailed. Hence, $\vec{X} \leqslant_{lex} \vec{Y}$ is entailed. □

We now consider the modifications necessary to `GACLexLeq` to obtain `GACLexLeqEntailed`, which is presented in Fig. 6. Clearly, if `alpha` reaches n, then the constraint is entailed. This case is dealt with by lines C3 and C21. Otherwise, entailment is detected by considering γ. We introduce the program variable `gamma` to keep track of γ (line C10) and initialise it (lines C11–C12) in much the same way as `beta`.

Two further procedures are required to maintain `gamma` correctly. First, γ may change following an update to $\max(X_i)$ or $\min(Y_i)$, which are the opposite conditions to those that trigger `ReEstablishGAC`. Hence, `CheckEntailment` is triggered by these events (line C19), which calls the `UpdateGamma` procedure if necessary. `UpdateGamma` works similarly to the other two `Update` procedures, signalling entailment if `alpha` and `gamma` meet (line C28).

		Algorithm `GACLexLeqEntailed`
		`EstablishGAC`
	C1	`alpha` $:= 0$
	C2	**while** (`alpha` $< n \wedge X_{\mathtt{alpha}} \doteq Y_{\mathtt{alpha}}$) **do** `alpha` $:=$ `alpha` $+ 1$
⟩⟩	C3	**if** (`alpha` $= n$) **then** entailed
	C4	$i :=$ `alpha`
	C5	`beta` $:= -1$
	C6	**while** ($i \neq n \wedge \min(X_i) \leqslant \max(Y_i)$) **do**
	C6.1	**if** ($\min(X_i) = \max(Y_i)$) **then** **if** (`beta` $= -1$) **then** `beta` $:= i$
	C6.2	**else** `beta` $:= -1$
	C6.3	$i := i + 1$
	C7	**if** ($i = n$) **then** `beta` $:= n + 1$
	C8	**else if** (`beta` $= -1$) **then** `beta` $:= i$
	C9	**if** (`alpha` $=$ `beta`) **then** disentailed
⟩⟩	C10	`gamma` $:= -1$, $i :=$ `alpha`
⟩⟩	C11	**while** ($i \neq n \wedge \max(X_i) \geqslant \min(Y_i)$) **do**
⟩⟩	C11.1	**if** ($\max(X_i) = \min(Y_i)$) **then**
⟩⟩	C11.2	**if** (`gamma` $= -1$) **then** `gamma` $:= i$
⟩⟩	C11.3	**else** `gamma` $:= -1$
⟩⟩	C11.4	$i := i + 1$
⟩⟩	C12	**if** (`gamma` $= -1$) **then** `gamma` $:= i$
⟩⟩	C13	**if** (`alpha` $=$ `gamma`) **then** entailed
	C14	`ReEstablishGAC(alpha)`
		`ReEstablishGAC(`i in $[0, n)$`)` Triggered when $\min(X_i)$ or $\max(Y_i)$ changes
		Identical to `GACLexLeq`
⟩⟩		`CheckEntailment(`i in $[0, n)$`)` Triggered when $\max(X_i)$ or $\min(Y_i)$ changes
⟩⟩	C19	**if** (($i =$ `gamma` $- 1 \wedge \max(X_i) = \min(Y_i)) \vee \max(X_i) < \min(Y_i)$) **then** `UpdateGamma(`$i - 1$`)`
		`UpdateAlpha`
	C20	`alpha` $:=$ `alpha` $+ 1$
⟩⟩	C21	**if** (`alpha` $= n$) **then** entailed
	C22	**if** (`alpha` $=$ `beta`) **then** disentailed
	C23	**if** ($\neg(X_{\mathtt{alpha}} \doteq Y_{\mathtt{alpha}})$) **then** `ReEstablishGAC(alpha)`
	C24	**else** `UpdateAlpha`
		`UpdateBeta(`i in $[0, n)$`)`
		Identical to `GACLexLeq`
⟩⟩		`UpdateGamma(`i in $[0, n)$`)`
⟩⟩	C28	**if** ($i + 1 =$ `alpha`) **then** entailed
⟩⟩	C29	**if** ($\max(X_i) > \min(Y_i)$) **then** `gamma` $:= i + 1$
⟩⟩	C30	**else if** ($\max(X_i) = \min(Y_i)$) **then** `UpdateGamma(`$i - 1$`)`

Fig. 6. Constituent procedures of the `GACLexLeqEntailed` algorithm. Modifications from `GACLexLeq` are indicated by '⟩⟩'.

Section 5.1 analyzed the complexity of GACLexLeq by considering the modifications of alpha and beta. As the same arguments can be made about the modifications of gamma, the results given in Theorems 3 and 4 apply to GACLexLeqEntailed as well.

6.3. Vectors of different length

This section considers vectors of different length. Since two vectors of different length can never be equal, we only consider imposing a strict lexicographic ordering constraint.

Considering vectors of equal length, Definition 2 defined strict lexicographic ordering between two vectors of integers and Theorem 8 stated the necessary conditions for GAC on a strict lexicographic ordering constraint on a pair of vectors of variables. It is, however, straightforward to generalise the definition and the theorem for two vectors of *any*, not necessarily equal, length.

Definition 10. Strict lexicographic ordering $\vec{x} <_{lex} \vec{y}$ between two vectors of integers $\vec{x} = \langle x_0, x_1, \ldots, x_{m-1}\rangle$ and $\vec{y} = \langle y_0, y_1, \ldots, y_{n-1}\rangle$ holds if and only if either of the following conditions hold:

1. $m < n \ \wedge\ \vec{x} \leqslant_{lex} \vec{y}_{0..m-1}$, or
2. $m \geqslant n \ \wedge\ \vec{x}_{0..n-1} <_{lex} \vec{y}$.

In other words, we truncate to the length of the shortest vector and then compare. Either $\vec{x}$ is shorter than $\vec{y}$ and the first m elements of the vectors are lexicographically ordered, or $\vec{x}$ is at least as long as $\vec{y}$ and the first n elements are strict lexicographically ordered. An example is $\langle 0,1,2,1,5\rangle <_{lex} \langle 0,1,2,3,4\rangle <_{lex} \langle 0,1,2,3,4,5,5,5\rangle <_{lex} \langle 0,1,2,4,3\rangle$.

Based on this general definition, GAC on $\vec{X} <_{lex} \vec{Y}$ is either GAC on a lexicographic ordering constraint or GAC on a strict lexicographic ordering constraint.

Proposition 1. *Given a pair of variable-distinct vectors, $\vec{X}$ of length m and $\vec{Y}$ of length n, $\vec{X} <_{lex} \vec{Y}$ is GAC if and only if either of the following conditions hold*:

1. *$m < n$ and* $\mathrm{GAC}(\vec{X} \leqslant_{lex} \vec{Y}_{0..m-1})$, *or*
2. *$m \geqslant n$ and* $\mathrm{GAC}(\vec{X}_{0..n-1} <_{lex} \vec{Y})$.

We can now easily generalise the propagation algorithm GACLexLess based on this theorem. If $m < n$ then we just consider the first m variables of $\vec{Y}$ and use GACLexLeq to enforce GAC on $\vec{X} \leqslant_{lex} \vec{Y}_{0..m-1}$. If $m \geqslant n$ then we just consider the first n variables of $\vec{X}$ and use GACLexLess to enforce GAC on $\vec{X}_{0..n-1} <_{lex} \vec{Y}$).

7. Alternative approaches

Various alternative methods exist for enforcing the lexicographic ordering constraint, ranging from enforcing a collection of simpler constraints to alternative propagation algorithms. This section summarises and discusses these alternative approaches.

7.1. Arithmetic constraints

To ensure that $\vec{X} \leqslant_{lex} \vec{Y}$, we can post the following arithmetic inequality constraint between the vectors $\vec{X}$ and $\vec{Y}$ whose variables range over any subset of $\{0, \ldots, d-1\}$:

$$d^{n-1} * X_0 + d^{n-2} * X_1 + \cdots + d^0 * X_{n-1} \leqslant d^{n-1} * Y_0 + d^{n-2} * Y_1 + \cdots + d^0 * Y_{n-1}$$

This is equivalent to converting two vectors into numbers and posting an ordering on the numbers. In order to enforce strict lexicographic ordering $\vec{X} <_{lex} \vec{Y}$, we post the strict inequality constraint:

$$d^{n-1} * X_0 + d^{n-2} * X_1 + \cdots + d^0 * X_{n-1} < d^{n-1} * Y_0 + d^{n-2} * Y_1 + \cdots + d^0 * Y_{n-1}$$

These two arithmetic constraints are logically equivalent to the corresponding lexicographic ordering constraint. Since these constraints are monotonic—any unsupported domain values are the greatest ones in $\vec{X}$ or the least ones in $\vec{Y}$—many constraint toolkits establish GAC on these constraints. Unfortunately, when n and d become large, d^{n-1} will be much more than the word size of the computer and establishing GAC will be expensive. Hence, this method is only feasible when the vectors and domain sizes are small.

Inspired by [10], Warwick Harvey has suggested an alternative arithmetic constraint [14]. To ensure $\vec{X} \leqslant_{lex} \vec{Y}$, he posts the following logically equivalent constraint:

$$1 = \big(X_0 < Y_0 + \big(X_1 < Y_1 + \big(\ldots + (X_{n-1} < Y_{n-1} + 1)\ldots\big)\big)\big)$$

A constraint of the form $(X_i < Y_i + B)$ is reified into a 0/1 variable and it is interpreted as $X_i < (Y_i + B)$. Strict ordering is achieved by posting:

$$1 = \big(X_0 < Y_0 + \big(X_1 < Y_1 + \big(\ldots + (X_{n-1} < Y_{n-1} + 0)\ldots\big)\big)\big)$$

which disallows $X_{n-1} \doteq Y_{n-1}$ in case the vectors are assigned and equal until the last index. Many toolkits that support reification maintain GAC on these constraints.

Henceforth, we shall refer to Harvey's reification approach by his name and to the purely arithmetic constraint above simply as the "arithmetic constraint".

7.2. Logical decompositions

The first decomposition, which we call the $\wedge$ decomposition, is a conjunction of $n-1$ constraints:

$$\begin{aligned}
X_0 &\leqslant Y_0 \wedge \\
X_0 = Y_0 \rightarrow X_1 &\leqslant Y_1 \wedge \\
X_0 = Y_0 \wedge X_1 = Y_1 \rightarrow X_2 &\leqslant Y_2 \wedge \\
&\vdots \\
X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{n-2} = Y_{n-2} \rightarrow X_{n-1} &\leqslant Y_{n-1}
\end{aligned}$$

That is, we enforce that the most significant bits of the vectors are ordered, and if the most significant bits are equal then the rest of the vectors are lexicographically ordered. In order to decompose the strict lexicographic ordering constraint $\vec{X} <_{lex} \vec{Y}$, we only need to change the last conjunction to $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{n-2} = Y_{n-2} \rightarrow X_{n-1} < Y_{n-1}$.

The second decomposition, which we call $\vee$ decomposition, is a disjunction of $n-1$ constraints:

$$\begin{aligned}
X_0 &< Y_0 \vee \\
X_0 = Y_0 \wedge X_1 &< Y_1 \vee \\
X_0 = Y_0 \wedge X_1 = Y_1 \wedge X_2 &< Y_2 \vee \\
&\vdots \\
X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{n-2} = Y_{n-2} \wedge X_{n-1} &\leqslant Y_{n-1}
\end{aligned}$$

That is, we enforce that either the most significant bits of the vectors are strictly ordered or the most significant bits are equal and the rest of the vectors are lexicographically ordered. For strict lexicographic ordering, it suffices to change the last disjunction to $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{n-2} = Y_{n-2} \wedge X_{n-1} < Y_{n-1}$.

Constraints with logical connectives are imposed in the following manner.

- $C_1 \wedge C_2$: Both C_1 and C_2 are imposed.
- $C_1 \vee C_2$: If one of C_1 or C_2 becomes disentailed, the other constraint is imposed. If C_1 or C_2 becomes entailed then $C_1 \vee C_2$ becomes entailed.
- $C_1 \rightarrow C_2$: If C_1 becomes entailed then C_2 is imposed. If C_2 becomes disentailed then $\neg C_1$ is imposed. If C_1 becomes disentailed or C_2 becomes entailed, then $C_1 \rightarrow C_2$ becomes entailed.

- $\neg C_1$: If C_1 becomes entailed then $\neg C1$ becomes disentailed. If C_1 becomes disentailed then $\neg C1$ becomes entailed.

We consider a propagation algorithm for these decompositions that treat the logical connectives in this manner and establish AC on a binary constraint whenever it is imposed. We now show that these algorithms are guaranteed to establish GAC and that each of the algorithms can prune values that the other cannot.

Theorem 11. *For both $\vec{X} \leqslant_{lex} \vec{Y}$ and $\vec{X} <_{lex} \vec{Y}$ the $\wedge$ decomposition and the $\vee$ decomposition are sound but not complete. Moreover, on each of these constraints, the two decomposition algorithms are incomparable in that one can prune values that the other cannot, and vice-versa.*

Proof. We only consider $(\vec{X} \leqslant_{lex} \vec{Y})$ but the proof also works for $(\vec{X} <_{lex} \vec{Y})$. Suppose that $\vec{X} \leqslant_{lex} \vec{Y}$ is GAC. Thus every value has a support. The vectors $\vec{x}$ and $\vec{y}$ supporting a value are lexicographically ordered $(\vec{x} \leqslant_{lex} \vec{y})$. By Definition 3, either $\vec{x} = \vec{y}$ or there is an index k in $[0, n)$ such that $x_k < y_k$ and $\vec{x}_{0..k-1} = \vec{y}_{0..k-1}$. In either case, all the constraints posted in either of the decompositions are satisfied. That is, every binary constraint imposed in the decompositions is AC. Hence, the $\wedge$ decomposition and the $\vee$ decomposition are sound for $(\vec{X} \leqslant_{lex} \vec{Y})$.

Consider $\vec{X} = \langle\{0, 1\}, \{1\}\rangle$ and $\vec{Y} = \langle\{0, 1\}, \{0\}\rangle$ where $\vec{X} \leqslant_{lex} \vec{Y}$ is not GAC. The $\wedge$ decomposition imposes both of $X_0 \leqslant Y_0$ and $X_0 = Y_0 \rightarrow X_1 \leqslant Y_1$. We have AC$(X_0 \leqslant Y_0)$. Since $X_1 \leqslant Y_1$ is disentailed, $X_0 \neq Y_0$ is imposed. We have AC$(X_0 \neq Y_0)$ so no pruning is possible. The $\vee$ decomposition, however, imposes $X_0 < Y_0$ because $X_0 = Y_0 \wedge X_1 \leqslant Y_1$ is disentailed. This removes 1 from $\mathcal{D}(X_0)$ and 0 from $\mathcal{D}(Y_0)$.

Now consider $\vec{X} = \langle\{0, 1, 2\}, \{0, 1\}\rangle$ and $\vec{Y} = \langle\{0, 1\}, \{0, 1\}\rangle$ where $\vec{X} \leqslant_{lex} \vec{Y}$ is not GAC. The $\wedge$ decomposition removes 2 from $\mathcal{D}(X_0)$ by AC$(X_0 \leqslant Y_0)$. The $\vee$ decomposition, however, leaves the vectors unchanged since neither $X_0 < Y_0$ nor $X_0 = Y_0 \wedge X_1 \leqslant Y_1$ is disentailed.

Since each decomposition prunes a value not pruned by the other, neither is complete. □

Together, the two decompositions behave similarly to the propagation algorithm of the lexicographic ordering constraint: they either prove that $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed or establish GAC$(\vec{X} \leqslant_{lex} \vec{Y})$. However, this requires posting and propagating many constraints, and is likely to be inefficient. Our experimental results in Section 9.1 confirm this expectation.

Theorem 12. *Given a pair of length n variable-distinct vectors, $\vec{X}$ and $\vec{Y}$, the $\wedge$ and $\vee$ decomposition of $\vec{X} \leqslant_{lex} \vec{Y}$ together either prove that $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed, or establish* GAC$(\vec{X} \leqslant_{lex} \vec{Y})$.

Proof. Consider the $\wedge$ decomposition. If $\alpha = \beta$ then either $\min(X_\alpha) > \max(Y_\alpha)$, or there exists an index k in (α, n) such that $\min(X_k) > \max(Y_k)$ and $\min(\vec{X}_{\alpha..k-1}) = \max(\vec{Y}_{\alpha..k-1})$. In the first case, the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{\alpha-1} = Y_{\alpha-1} \rightarrow X_\alpha \leqslant Y_\alpha$ is disentailed. In the second case, the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{k-1} = Y_{k-1} \rightarrow X_k \leqslant Y_k$ is disentailed because $\vec{X}_{\alpha..k-1} \doteq \vec{Y}_{\alpha..k-1}$ due to the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{i-1} = Y_{i-1} \rightarrow X_i \leqslant Y_i$. In any case, $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed. This is correct by Theorem 1. If, however, $\alpha < \beta$ then the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{\alpha-1} = Y_{\alpha-1} \rightarrow X_\alpha \leqslant Y_\alpha$ makes sure that AC$(X_\alpha \leqslant Y_\alpha)$. Now consider the $\vee$ decomposition. If $\beta = \alpha$ then all the disjuncts of the decomposition are disentailed, so $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed. This is correct by Theorem 1. If $\beta = \alpha + 1$ then each disjunct except $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{\alpha-1} = Y_{\alpha-1} \wedge X_\alpha < Y_\alpha$ is disentailed. This makes sure that AC$(X_\alpha < Y_\alpha)$. Given $\beta > \alpha$, we have either

- $\beta = \alpha + 1$ and AC$(X_\alpha < Y_\alpha)$, or
- $\beta > \alpha + 1$ and AC$(X_\alpha \leqslant Y_\alpha)$.

By Theorem 2, we have GAC$(\vec{X} \leqslant_{lex} \vec{Y})$. □

Theorem 13. *Given a pair of length n variable-distinct vectors, $\vec{X}$ and $\vec{Y}$, the $\wedge$ and $\vee$ decomposition of $\vec{X} <_{lex} \vec{Y}$ together either prove that $\vec{X} <_{lex} \vec{Y}$ is disentailed, or establish* GAC$(\vec{X} <_{lex} \vec{Y})$.

Proof. We only need to consider two cases, $\beta = n$ and $\beta < n \wedge \min(\vec{X}_{\beta..n-1}) = \max(\vec{Y}_{\beta..n-1})$; the remaining cases are covered by the proof of Theorem 12. Assume $\beta = n$. Consider the $\wedge$ decomposition. We either have $\alpha + 1 = \beta = n$ or $\alpha + 1 < \beta = n$. In the first case, the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{n-2} = Y_{n-2} \rightarrow X_{n-1} < Y_{n-1}$, which is the last constraint of the conjunction, makes sure that AC($X_\alpha < Y_\alpha$). In the second case, the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{\alpha-1} = Y_{\alpha-1} \rightarrow X_\alpha \leqslant Y_\alpha$ makes sure that AC($X_\alpha \leqslant Y_\alpha$).

Now assume $\beta < n \wedge \min(\vec{X}_{\beta..n-1}) = \max(\vec{Y}_{\beta..n-1})$. Consider the $\wedge$ decomposition. If $\alpha = \beta$ the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{n-2} = Y_{n-2} \rightarrow X_{n-1} < Y_{n-1}$ is disentailed because for all $i \in [\alpha, n-1)$ we get $\vec{X}_i \doteq \vec{Y}_i$ due to the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{i-1} = Y_{i-1} \rightarrow X_i \leqslant Y_i$. Hence, $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed. This is correct by Theorem 1. If, however, $\alpha < \beta$ then the constraint $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{\alpha-1} = Y_{\alpha-1} \rightarrow X_\alpha \leqslant Y_\alpha$ makes sure that AC($X_\alpha \leqslant Y_\alpha$). Now consider the $\vee$ decomposition. If $\beta = \alpha$ then all the disjuncts of the decomposition are disentailed, so $\vec{X} \leqslant_{lex} \vec{Y}$ is disentailed. This is correct by Theorem 1. If $\beta = \alpha + 1$ then each of the disjuncts but $X_0 = Y_0 \wedge X_1 = Y_1 \wedge \cdots \wedge X_{\alpha-1} = Y_{\alpha-1} \wedge X_\alpha < Y_\alpha$ is disentailed. This makes sure that AC($X_\alpha < Y_\alpha$).

Given $\beta > \alpha$, whether $\beta = n$ or $\beta < n$, we have either

1. $\beta = \alpha + 1$ and AC($X_\alpha < Y_\alpha$), or
2. $\beta > \alpha + 1$ and AC($X_\alpha \leqslant Y_\alpha$).

By Theorem 8, we have GAC($\vec{X} <_{lex} \vec{Y}$). □

7.3. Alternative propagation algorithms

The ECLiPSe constraint solver [29] provides a global constraint, called `lexico_le`, for lexicographic ordering of two vectors. The manual [6] does not reveal what propagation is performed by this constraint. Our tests show that it is not complete: if $\vec{X} = \langle\{0,1\},\{0,1\},\{1\}\rangle$ and $\vec{Y} = \langle\{0,1\},\{0\},\{0\}\rangle$ then executing `lexico_le` on $\vec{X} \leqslant_{lex} \vec{Y}$ leaves the vectors unchanged, even though $\vec{X} \leqslant_{lex} \vec{Y}$ is not GAC. Our tests also show that `lexico_le` can prune values that are not pruned by each of the decompositions discussed in Section 7.2. For instance, if $\vec{X} = \langle\{0,1\},\{1\}\rangle$ and $\vec{Y} = \langle\{0,1\},\{0\}\rangle$ then executing `lexico_le` on $\vec{X} \leqslant_{lex} \vec{Y}$ gives $\vec{X} = \langle\{0\},\{1\}\rangle$ and $\vec{Y} = \langle\{1\},\{0\}\rangle$ but the $\wedge$ decomposition leaves the vectors unchanged. Likewise, if $\vec{X} = \langle\{0,1,2\},\{0,1\}\rangle$ and $\vec{Y} = \langle\{0,1\},\{0,1\}\rangle$ then executing `lexico_le` on $\vec{X} \leqslant_{lex} \vec{Y}$ gives $\vec{X} = \langle\{0,1\},\{0,1\}\rangle$ and $\vec{Y} = \langle\{0,1\},\{0.1\}\rangle$ but the $\vee$ decomposition leaves the vectors unchanged. We have found no examples where either of the decompositions prunes values not pruned by `lexico_le`.

Subsequent to the first publication of our `GACLexLeq` algorithm [10], Carlsson and Beldiceanu developed a complete propagation algorithm for the lexicographic ordering constraint using a finite automaton [2]. The algorithm maintains generalised arc consistency or detects (dis)entailment, and runs in linear time for posting plus amortised constant time per propagation event. Their algorithm records the position of α, but has no counterpart of our `beta` variable.

7.4. Encoding `GACLexLeq`

An alternative way of propagating a global constraint is to post a set of constraints that "simulate" the special-purpose propagation algorithm. The success of such an approach was demonstrated in [11] by showing that arc consistency on the CSP representation of the stable marriage problem gives reduced domains that are equivalent to the GS-lists produced by the Extended Gale–Shapley algorithm. Inspired by [10], Gent et al. have developed an encoding of the lexicographic ordering constraint [12].

The encoding introduces a new vector $\vec{\alpha}$ of 0/1 variables indexed from -1 to $n-1$. The intended meaning of $\vec{\alpha}$ is that: if $\alpha_i = 1$ then $\vec{X}_{0..i} = \vec{Y}_{0..i}$, if $\alpha_{i+1} = 0$ but $\alpha_i = 1$ then $X_{i+1} < Y_{i+1}$. They post the following constraints:

$$\alpha_{-1} = 1$$
$$\alpha_i = 0 \rightarrow \alpha_{i+1} = 0 \qquad (i \in [0, n-2])$$
$$\alpha_i = 1 \rightarrow X_i = Y_i \qquad (i \in [0, n-1])$$
$$\alpha_i = 1 \wedge \alpha_{i+1} = 0 \rightarrow X_{i+1} < Y_{i+1} \qquad (i \in [-1, n-2])$$
$$\alpha_i = 1 \rightarrow X_{i+1} \leqslant Y_{i+1} \qquad (i \in [-1, n-2])$$

For strict lexicographic ordering, it suffices to add $\alpha_{n-1} = 0$.

The advantage of using this encoding is that it obviates the need implement a special-purpose propagation algorithm, instead relying on existing and more general propagation algorithms. On the other hand, as our experimental results in Section 9 show, the introduction of extra variables and constraints may be less efficient.

8. Multiple vectors

Chains of lexicographic ordering constraints often arise in practice, such as when posting lexicographic ordering constraints on the rows or columns of a matrix of decision variables. We can treat such a chain as a single global ordering constraint over the whole matrix. Alternatively, we can decompose it into lexicographic ordering constraints between adjacent rows/columns or between all pairs of rows/columns. This section demonstrates that such decompositions hinder constraint propagation.

As this section considers sequences of vectors, it is useful to subscript the individual vectors, such as in the vector sequence $\vec{X}_0, \ldots, \vec{X}_{n-1}$. Notice that the vector accent in $\vec{X}_i$ indicates that we are referring to vector i in a sequence of vectors, as opposed to X_i, which refers to element i in vector X.

Theorem 14. *Let $\vec{X}_0, \ldots, \vec{X}_{n-1}$ be variable-distinct vectors. Then* GAC($\vec{X}_i \leqslant_{lex} \vec{X}_{i+1}$) *for all $i \in [0, n-1)$ does not imply* GAC($\vec{X}_i \leqslant_{lex} \vec{X}_j$) *for all $i, j \in [0, n)$ such that $i < j$.*

Proof. Consider the following three vectors:

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

Observe that $\vec{X}_0 \leqslant_{lex} \vec{X}_1$ and $\vec{X}_1 \leqslant_{lex} \vec{X}_2$ are each GAC. But $\vec{X}_0 \leqslant_{lex} \vec{X}_2$ is not GAC as the constraint has no solution in which the initial element of $\vec{X}_0$ is assigned 1. □

Theorem 15. *Let $\vec{X}_0, \ldots, \vec{X}_{n-1}$ be variable-distinct vectors. Then* GAC($\vec{X}_i <_{lex} \vec{X}_{i+1}$) *for all $i \in [0, n-1)$ does not imply* GAC($\vec{X}_i <_{lex} \vec{X}_j$) *for all $i, j \in [0, n)$ such that $i < j$.*

Proof. This is shown by the example in the proof of Theorem 14. □

Theorem 16. *Let $\vec{X}_0, \ldots, \vec{X}_{n-1}$ be variable-distinct vectors. Then* GAC($\vec{X}_i \leqslant_{lex} \vec{X}_j$) *for all $i, j \in [0, n)$ such that $i < j$ does not imply* GAC($\vec{X}_0 \leqslant_{lex} \vec{X}_1 \leqslant_{lex} \cdots \leqslant_{lex} \vec{X}_{n-1}$).

Proof. Consider the following three vectors:

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

Observe that $\vec{X}_0 \leqslant_{lex} \vec{X}_1$, $\vec{X}_0 \leqslant_{lex} \vec{X}_2$ and $\vec{X}_1 \leqslant_{lex} \vec{X}_2$ are each GAC. But $(X_0 \leqslant_{lex} \vec{X}_1 \leqslant_{lex} X_2)$ is not GAC as the constraint has no solution in which the initial element of $\vec{X}_0$ is assigned 1. □

The proof of Theorem 16 shows that the theorem holds even if attention is restricted to 0/1 variables, demonstrating the incorrectness of a previous claim [10] to the contrary.

Theorem 17. *Let $\vec{X}_0, \ldots, \vec{X}_{n-1}$ be variable-distinct vectors. Then* GAC($\vec{X}_i <_{lex} \vec{X}_j$) *for all $i, j \in [0, n)$ such that $i < j$ does not imply* GAC($\vec{X}_0 <_{lex} \vec{X}_1 <_{lex} \cdots <_{lex} \vec{X}_{n-1}$).

Proof. This is shown by the example in the proof of Theorem 16. □

Subsequent to [10], Carlsson and Beldiceanu [1] introduced a propagation algorithm, called `lex_chain`, that can establish GAC on a constraint of the form $(\vec{X}_0 \leqslant_{lex} \vec{X}_1 \leqslant_{lex} \cdots \leqslant_{lex} \vec{X}_{n-1})$. Every time the constraint is propagated, feasible upper and lower bounds are computed for each vector in the chain and then the vectors are pruned with respect to these bounds. Given m vectors, each of length n, the algorithm maintains generalised arc-consistency or detects (dis)entailment, and performs $O(nm)$ operations.

9. Experimental results

We implemented our algorithms in C++ using ILOG Solver 5.3 [16] and performed experiments to compare them with the alternatives presented in Section 7 and the `lex_chain` algorithm mentioned in Section 8. We experimented on the three matrix models given in Section 3, adding lexicographic ordering constraints to break the partial and total index symmetry.

The results of the experiments are shown in tables where a "−" means no result is obtained in 1 hour. The number of fails gives the number of incorrect decisions at choice points in the search tree. The best result of each entry in a table is typeset in bold. Lexicographic ordering on the rows is enforced via technique *Tech*, then we write *Tech* R. Similarly, we write *Tech* C if *Tech* is used to constrain the columns to be lexicographically ordered, and *Tech* RC if *Tech* is used to constrain both dimensions. In theory, posting lexicographic ordering constraints between every pair of rows (similarly for columns) leads to more pruning than posting between adjacent rows (see Section 8). However, we did not observe any benefit in practice. Therefore, we just posted lexicographic ordering constraints between adjacent rows. The experiments were conducted using ILOG Solver 5.3 on a 1 Ghz Pentium III processor with 256 Mb RAM under Windows XP. We propagate the arithmetic constraint via `IloScalProd`, which maintains GAC on it. We either look for one solution or the optimal solution in optimisation problems.

9.1. Comparison with alternative approaches

We designed some experiments to test two goals. First, does our propagation algorithm(s) do more inference in practice than the $\wedge$ and $\vee$ decompositions? Similarly, is the algorithm more efficient in practice than these decompositions? Second, how does our algorithm compare with the alternatives that also maintain GAC, that is the arithmetic constraint, the combined logical decompositions, and Gent et al.'s encoding?

We do not experiment with `lexico_le`, since it is exclusive to ECLiPSe. Recall, however, that we have shown in Section 7.3 that `lexico_le` does not maintain GAC. Furthermore, we do not experiment with Carlsson and Beldiceanu's pairwise propagation algorithm since it is exclusive to SICStus prolog. As noted in Section 7.3, however, this algorithm has been shown to behave very similarly to our own.

We now consider each of the three problem domains in turn.

Progressive party problem. We use the matrix model introduced in Section 3. The H matrix in this model has partial row and total column symmetry, which we break by posting lexicographic ordering constraints using either our propagation algorithm `GACLexLess` or the various alternative approaches. Due to the problem constraints, no pair of rows/columns can be equal. Given a set of interchangeable guests $\{g_i, g_{i+1}, \ldots, g_j\}$, therefore, we can break the partial row symmetry of H by constraining the corresponding rows, $\vec{R}_i, \vec{R}_{i+1}, \ldots, \vec{R}_j$, to be strictly lexicographically ordered as follows: $\vec{R}_j <_{lex} \vec{R}_{i+1} \cdots <_{lex} \vec{R}_i$. As for the column symmetry of H, we constrain the columns, $\vec{C}_0, \vec{C}_1, \ldots, \vec{C}_{p-1}$, corresponding to the p time periods to be strictly lexicographically ordered as follows: $\vec{C}_{p-1} <_{lex} \vec{C}_{p-2} \cdots <_{lex} \vec{C}_1$.

We consider several instances of the progressive party problem, drawn from the data in CSPLib. We randomly select 13 host boats in such a way that the total spare capacity of the host boats is sufficient to accommodate all the guests. The data is presented in Table 1. The last column gives the percentage of the total capacity used, which is a measure of constrainedness [28].

We branch on the variables of the H matrix. As in [24], 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 first try a value that is most likely to succeed. We therefore order the host boats in descending order of their spare capacity. In terms of variable ordering, we use the *smallest-domain first* heuristic.

Table 1
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

Table 2
Performance of different propagation algorithms on the progressive party problem. All times are given in seconds

Inst. #	GACLexLess RC		$\wedge$ RC		$\vee$ RC		$\wedge\vee$ RC	Gent et al.	Harvey
	Fails	Time	Fails	Time	Fails	Time	Time	Time	Time
1	**446**	**0.86**	–	–	–	–	1.11	0.98	0.92
2	**445**	**0.98**	**445**	1.41	–	–	1.25	1.11	1.04
3	**2,380**	**2.17**	3,651	3.00	–	–	2.95	2.47	2.44
4	**459**	**0.86**	–	–	–	–	1.15	0.98	0.95
5	**443**	**0.98**	**443**	1.12	–	–	1.18	1.16	**0.98**
6	8,481	6.14	**604**	**1.36**	–	–	8.10	6.62	6.82
7	**782**	**1.13**	–	–	–	–	1.50	1.21	1.22
8	33,849	16.79	**773**	**1.31**	–	–	21.62	18.40	17.93
9*	**211,075**	**117.25**	213,568	150.33	–	–	156.84	130.27	131.11

Table 2 summarises the results of the experiments. Note that all the problem instances are solved for 6 time periods. One exception is the last instance, indicated by a "*", as none of the approaches could solve this instance within an hour time limit for 6 time periods. We therefore report results for 5 time periods for this instance of the problem.

In this set of experiments, clearly GACLexLess is superior to the $\vee$ decomposition: none of the instances could be solved within an hour by the $\vee$ decomposition. However, it is difficult to judge which of GACLexLess and the $\wedge$ decomposition is superior. GACLexLess solves instances 1, 4 and 7 very quickly, but the $\wedge$ decomposition fails to return an answer in one hour. Also, instances 3 and 9 are solved with fewer failures by GACLexLess. On the other hand, the $\wedge$ decomposition is superior to GACLexLess for instances 6 and 8. No difference in the size of the search tree is observed for instances 2 and 5. Note that even though the $\wedge$ decomposition of $\vec{X} <_{lex} \vec{Y}$ does not establish GAC, enforcing the $\wedge$ decomposition at every choice point may lead to a smaller search tree than maintaining GAC at every choice point because a dynamic variable ordering is employed.

We now turn our attention to the alternatives that maintain GAC. Note that posting the arithmetic constraint is not feasible for this problem, as the largest coefficient necessary is 13^{28}, which is greater than 2^{31}, the maximum integer size allowed in Solver 5.3. In all cases, the tree explored is identical, hence we focus on run-time. Versus the combined $\wedge\vee$ decomposition, GACLexLess is clearly more efficient, especially for the more difficult instances. Compared with Gent et al.'s encoding and Harvey's arithmetic constraint, GACLexLess holds a small but consistent advantage that scales with the difficulty of the instance. In all cases, this advantage is due to the fact that GACLexLess encapsulates lexicographic ordering in a single compact constraint. The alternatives incur the overhead of several constraints and/or several additional variables (note that the reification performed in Harvey's arithmetic constraint makes use of hidden Boolean variables).

Template design problem. We use the model of [21], which adds symmetry-breaking constraints and implied constraints to the matrix model introduced in Section 3. The T matrix has partial row and partial column symmetry. Again we break the partial row symmetry by posting lexicographic ordering constraints. Given a set of interchangeable variations $\{v_i, v_{i+1}, \ldots, v_j\}$, we can break the partial row symmetry of T by insisting that the rows corresponding to such variations, $\vec{R}_i, \vec{R}_{i+1}, \ldots, \vec{R}_j$, are lexicographically ordered as follows: $\vec{R}_i \leqslant_{lex} \vec{R}_{i+1} \cdots \leqslant_{lex} \vec{R}_j$. The symmetries among interchangeable templates are broken by the constraints

$$Run_i \leqslant Run_{i+1} \quad (i \in [0, t-1))$$

We post the additional constraints proposed in [21]. In presenting these constraints we consider $\mathcal{T}emplates$ be $[0, t)$, $\mathcal{V}ariations$ to be $[0, v)$, and *Surplus* to be $s \cdot \sum_{i \in \mathcal{T}emplates} Run_i - \sum_{j \in \mathcal{V}ariations} d_j$. To ease the presentation of these constraints we assume that the variations are ordered by non-decreasing demand; that is, if $i < j$ then $d_i \leqslant d_j$. The additional constraints are

$$T_{0,j} < T_{0,j+1} \rightarrow T_{1,j} > T_{1,j+1} \qquad (j \in [0, v-1), d_j = d_{j+1}) \tag{1}$$

$$\sum_{i \in \mathcal{T}emplates} Run_i * T_{i,j} \leqslant \sum_{i \in \mathcal{T}emplates} Run_i * T_{i,j+1} \qquad (j \in [0, v-1), d_j < d_{j+1}) \tag{2}$$

$$\sum_{i \in \mathcal{T}emplates} Run_i * T_{i,j} - d_j \leqslant Surplus \qquad (j \in \mathcal{V}ariations) \tag{3}$$

$$\sum_{0 \leqslant j \leqslant k} \left(\sum_{i \in \mathcal{T}emplates} Run_i * T_{i,j} - d_j \right) \leqslant Surplus \qquad (k \in [1, v-1)) \tag{4}$$

The constraints in (1) break symmetries among variations with equal demand when $t = 2$, and (2) are what they call "pseudo-symmetry" breaking constraints. The constraints of (3) and (4) are implied constraints that provide an upper bound on the cost function. Proll and Smith [21] also post some implied constraints on the *Run* variables, but we omit these as they can be shown to be propagation redundant.

We also use the static variable ordering proposed by Proll and Smith [21]: we first label the variables of T, and then the variables of *Run*. The T matrix is labelled row by row starting with variation 0, and each row is labeled in order starting with template 0. The *Run* variables are labelled in order starting with Run_0.

Our experiments are conducted on an instance of the template design problem known as the herbs problem [21], where labels for a variety of herbs are to be printed on templates. The data for this instance are shown in Table 3.

As in [21], we first specify that the over-production of any variation can be at most 10%. With this constraint, there is no solution with 2 templates, and this is trivially proven by all the approaches and `GACLexLeq` in 36 fails, 0.1 seconds. Removing this restriction makes the problem very difficult. A solution with cost 89 is found in 109,683 fails and around 23 seconds by `GACLexLeq`, the $\wedge$ decomposition and the arithmetic constraint, but all of them fail to prove optimality within an hour. Changing the labeling heuristic by assigning the *Run* variables before the T variables helps to find and prove a solution for 2 templates with cost 87, but does not help to find a solution for 3 templates within an hour even with the restricted over-production of 10%.

An alternative way of solving the problem is to allow 10% over- and under-production. We therefore relax the constraint that for all variations the minimum amount produced meets its demand. According to [21], this meets the problem owner's specification. The results of tackling the problem in this way for $t = 2, 3, 4, 5$ templates are shown in Table 4.

We observe in Table 4 that, as the number of templates increase, the search effort and time required to find a solution and prove optimality dramatically increase for the $\wedge$ and $\vee$ decompositions. On the other hand, `GACLexLeq` finds and proves solutions very quickly with much less effort. In particular, the 4 and 5 template problems can only be solved by `GACLexLeq`. On the approaches that maintain GAC we focus on run time since the search trees explored are identical. In these cases, there is very little difference in time taken. This is because the vectors constrained are relatively short, with each variable having a relatively small domain size. Hence, the advantage of having a single compact constraint is not visible.

Table 3
The data for the herbs problem in [21]

Slots per template	Number of variations	Demand (in thousands)
42	30	60, 60, 70, 70, 70, 70 ,70, 70, 70, 80, 80, 80, 80, 90, 90, 90, 90, 90, 90, 90, 100, 100, 100, 100, 150, 230, 230, 230, 230, 280, 280

Table 4
Performance of different propagation algorithms on the herbs problem with 10% over- and under-production. All times are given in seconds

t	Goal	GACLexLeq R		$\wedge$ R		$\vee$ R		$\wedge\vee$ R	Arth R	Gent et al.	Harvey
		Fails	Time	Fails	Time	Fails	Time	Time	Time	Time	Time
2	find	**22**	**0.02**	**22**	**0.02**	**22**	**0.02**	0.08	0.08	**0.02**	**0.02**
	prove	**49**	**0.02**	**49**	**0.02**	**49**	**0.02**	0.08	0.08	**0.02**	**0.02**
3	find	**5**	**0.02**	18,341	7.67	18,842	9.30	0.08	0.08	**0.02**	**0.02**
	prove	**52**	**0.02**	18,341	7.67	18,842	9.30	0.08	0.08	**0.02**	**0.02**
4	find	**6**	**0.02**	–	–	–	–	0.08	0.08	**0.02**	**0.02**
	prove	**70**	**0.02**	–	–	–	–	0.08	0.08	**0.02**	**0.02**
5	find	**4**	**0.02**	–	–	–	–	0.08	0.08	**0.02**	**0.02**
	prove	**77**	**0.02**	–	–	–	–	0.08	0.08	**0.02**	**0.02**

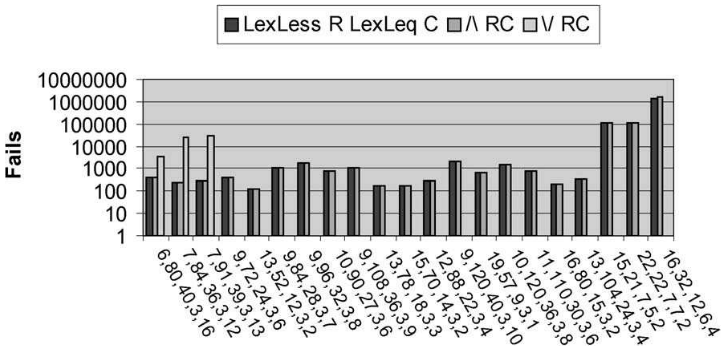

Fig. 7. BIBD: GACLexLeq/GACLexLess vs $\wedge$, $\vee$, $\wedge\vee$ decompositions in terms of fails.

Balanced incomplete block design problem. We use the matrix model, introduced in Section 3. Due to the constraints on the rows, no pair of rows in X can be equal unless $r = \lambda$. To break the row symmetry, we enforce that the rows $\vec{R}_0, \vec{R}_1, \ldots, \vec{R}_{v-1}$ of X corresponding to the v elements are strictly lexicographically ordered as follows: $\vec{R}_{v-1} <_{lex} \vec{R}_{v-2} \cdots <_{lex} \vec{R}_0$. As for the column symmetry, we enforce that the columns $\vec{C}_0, \vec{C}_1, \ldots, \vec{C}_{b-1}$ of X corresponding to the b subsets of $\mathcal{V}$ are lexicographically ordered as follows: $\vec{C}_{b-1} \leqslant_{lex} \vec{C}_{b-2} \cdots \leqslant_{lex} \vec{C}_0$. We post the lexicographic ordering constraints either by using GACLexLess and GACLexLeq, or the corresponding alternative approaches.

Large benchmark instances were selected with which to experiment (as in [4]). For the labelling heuristic, we adopted a static variable ordering, instantiating the matrix X along its rows from top to bottom and exploring the domain of each variable in ascending order. Figs. 7–9 (note the logarithmic scale) summarise our results. Again, we begin by comparing our propagation algorithm with the $\wedge$ and $\vee$ decompositions individually. Given our choice of variable ordering, our algorithm explores the same search tree as the $\wedge$ decomposition on all but two of the instances, where it explores a slightly smaller tree. The $\vee$ decomposition, however, can solve only the first 3 instances within an hour limit, with many more failures. In terms of run time, however, we observe a substantial gain in efficiency by using our algorithms in preference to the other approaches. Even though the $\wedge$ decomposition and our algorithms explore the same search tree, the efficiency of our algorithms dramatically reduces the run times.

We now turn our attention to the alternatives that maintain GAC. Note that these instances require the ordering of relatively long vectors, hence posting the arithmetic constraint is not feasible. The combined $\wedge\vee$ decomposition performs better than the $\wedge$ decomposition alone, but is still significantly less efficient than GACLexLeq. We observe in Fig. 9 that the instances are solved quicker using our algorithm (note the logarithmic scale), though the difference is not as much as the difference between the algorithms and the $\wedge$ decomposition in Fig. 8. In comparison with Gent et al.'s encoding and Harvey's arithmetic constraint, GACLexLeq maintains a small but consistent advantage.

To summarise the three sets of experiments in this section, GACLexLeq provides an efficient and lightweight means of enforcing lexicographic ordering, which provides a consistent improvement over the alternatives when used

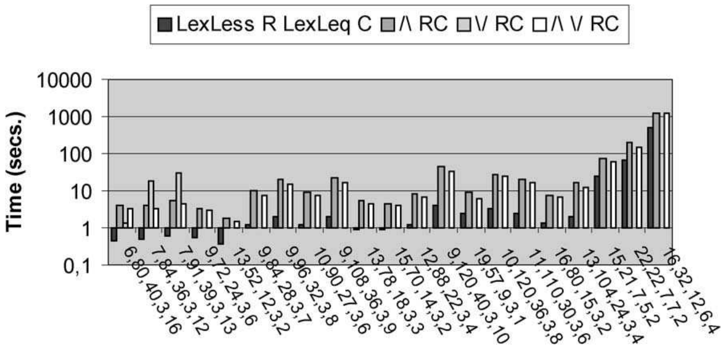

Fig. 8. BIBD: `GACLexLeq`/`GACLexLess` vs $\wedge$, $\vee$, $\wedge\vee$ decompositions in terms of run times.

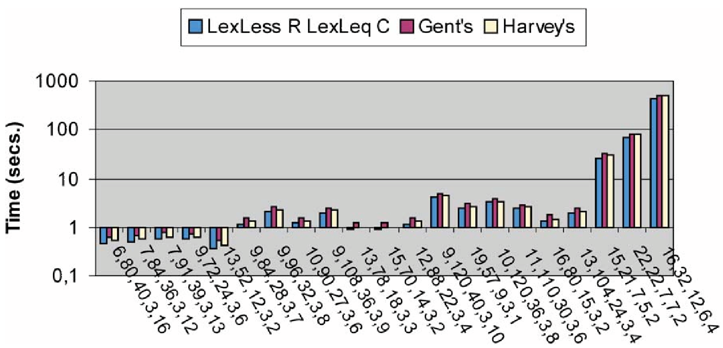

Fig. 9. BIBD: `GACLexLeq`/`GACLexLess` vs Gent et al.'s encoding and Harvey's arithmetic constraint in terms of run times.

with a fixed search strategy. We observed that a weaker propagation algorithm combined with a dynamic variable ordering can sometimes perform better, but this is unsurprising: choosing the 'right' variable to assign next is well known to be crucial in reducing search. On this occasion the dynamic variable was led by chance to a better selection due to weaker propagation.

The alternatives that do maintain GAC have the disadvantage of requiring the introduction of auxiliary variables and constraints. To illustrate this point, consider Figs. 10 and 11, which present the total numbers of variables and memory used for `GACLexLeq`, Gent et al.'s encoding and Harvey's decomposition on the BIBD problem.

9.2. Comparison with `lex_chain`

As Theorem 16 shows, the `lex_chain` algorithm of [1] can do more pruning than lexicographic ordering constraints between adjacent pairs of vectors. We performed a further set of experiments to determine the value of this additional pruning in practice. We focus on the BIBD problem as the most challenging domain considered herein. Table 5 summarises the results of solving BIBDs using SICStus Prolog constraint solver 3.10.1 [26]. Note that we used different instances from those in the previous experiments, since the earlier instances proved to take too long to solve in SICStus. We constrained the columns and rows to be lexicographically ordered non-decreasing or non-increasing, and assigned the variables in the matrix from top to bottom exploring the domains in ascending order. The lexicographic ordering constraints are posted using `lex_chain`. This constraint is either posted once for all the symmetric rows/columns, or between each adjacent symmetric rows/columns.

In all the cases, we observed no benefits of combining a chain of lexicographic ordering constraints. By posting the constraints between the adjacent rows/columns, we obtain the same search trees and very similar run times as posting

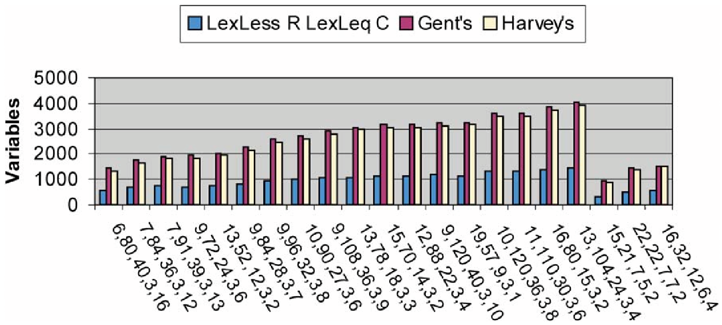

Fig. 10. BIBD: `GACLexLeq`/`GACLexLess` vs Gent et al.'s encoding and Harvey's arithmetic constraint in terms of number of variables.

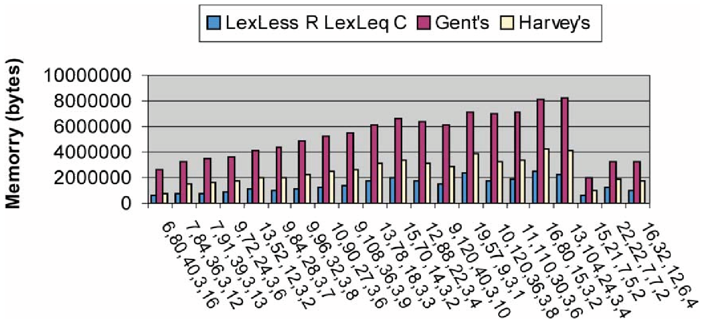

Fig. 11. BIBD: `GACLexLeq`/`GACLexLess` vs Gent et al.'s encoding and Harvey's arithmetic constraint in terms of memory usage.

Table 5
BIBD: **lex_chain**($\langle X_0, \ldots, X_{m-1}\rangle$) vs **lex_chain**($\langle X_i, X_{i+1}\rangle$) for all $i \in [0, m-1)$ with row-wise labeling

v, b, r, k, λ	No symmetry breaking	$<_{lex}$ R $\leqslant_{lex}$ C **lex_chain**		$>_{lex}$ R $\geqslant_{lex}$ C **lex_chain**	
	Backtracks	$\langle X_0, \ldots, X_{m-1}\rangle$ Backtracks	$\langle X_i, X_{i+1}\rangle$ Backtracks	$\langle X_0, \ldots, X_{m-1}\rangle$ Backtracks	$\langle X_i, X_{i+1}\rangle$ Backtracks
6,20,10,3,4	5,201	**84**	**84**	706	706
7,21,9,3,3	1,488	130	130	**72**	**72**
6,30,15,3,6	540,039	**217**	**217**	9216	9216
7,28,12,3,4	23,160	216	216	**183**	**183**
9,24,8,3,2	–	1,472	1,472	**79**	**79**
6,40,20,3,8	–	**449**	**449**	51,576	51,576
7,35,15,3,5	9,429,447	**326**	**326**	395	395
7,42,18,3,6	5,975,823	460	460	**756**	**756**

only one constraint on the rows/columns. This result is in agreement with our previous experiments, comparing posting lexicographic ordering constraints between adjacent rows/columns of a matrix, and between all pairs of rows/columns [10], where the latter is an approximation of `lex_chain`.

10. Conclusion

This paper introduced new algorithms for propagating lexicographic ordering constraints on two vectors of decision variables. Such constraints are useful for breaking row and column symmetries in matrix models. We demonstrated that decomposing such constraints often carries a penalty either in the amount or the cost of constraint propagation.

We have therefore developed efficient propagation algorithms that either ensure that such constraints are GAC or detect disentailment. These algorithms require only $O(n)$ operations where each vector contains n variables and can execute as sequence of k updates in $O(n+k)$ operations. Experimental results on a number of domains demonstrate the value of these new algorithms.

A number of interesting questions remain. First, are there other total orderings of vectors that we can post to break row and column symmetry? For example, can we use the Gray code ordering? Second, how do we design labeling strategies to work in synergy with symmetry breaking constraints like this? Third, are these global constraints useful in multi-criteria optimization problems where the objective function consists of features that are ranked [7]?

Acknowledgements

We thank Chris Jefferson for many useful discussions. Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship. Brahim Hnich received support from Science Foundation Ireland under Grant No. 00/PI.1/C075. The majority of this work was done while Brahim Hnich and Zeynep Kiziltan were at Uppsala University, and Ian Miguel was at the University of York.

References

[1] M. Carlsson, N. Beldiceanu, Arc-consistency for a chain of lexicographic ordering constraints, Technical Report T2002-18, Swedish Institute of Computer Science, 2002.
[2] M. Carlsson, N. Beldiceanu, Revisiting the lexicographic ordering constraint, Technical Report T2002-17, Swedish Institute of Computer Science, 2002.
[3] B.M.W. Cheng, K.M.F. Choi, J.H.M. Lee, J.C.K. Wu, Increasing constraint propagation by redundant modeling: An experience report, Constraints 4 (1999) 167–192.
[4] C.H. Colbourn, J.H. Dinitz, The CRC Handbook of Combinatorial Designs, CRC Press, 1996.
[5] 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.
[6] 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.
[7] M. Ehrgott, X. Gandibleux, A survey and annotated bibliography of multiobjective combinatorial optimization, OR Spektrum 22 (2000) 425–460.
[8] P. Flener, A.M. 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.
[9] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Matrix modelling: Exploiting common patterns in constraint programming, in: Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems, 2002, pp. 27–41.
[10] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Global constraints for lexicographic orderings, 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. 93–108.
[11] I.P. Gent, R.W. Irving, D. Manlove, P. Prosser, B.M. Smith, A constraint programming approach to the stable marriage problem, in: T. Walsh (Ed.), Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming (CP-01), in: Lecture Notes in Computer Science, vol. 2239, Springer, 2001, pp. 462–476.
[12] I.P. Gent, P. Prosser, B.M. Smith, A 0/1 encoding of the gaclex constraint for pairs of vectors, in: Notes of the ECAI-02 Workshop W9 Modelling and Solving Problems with Constraints, 2002.
[13] I.P. Gent, T. Walsh, CSPLib: A benchmark library for constraints, Technical Report APES-09-1999, 1999.
[14] W. Harvey, Personal e-mail communication, 2002.
[15] B. Hnich, Function variables for constraint programming, PhD thesis, Department of Information Science, Uppsala University, 2003.
[16] ILOG S.A., ILOG Solver 5.3 Reference and User Manual, 2002.
[17] Z. Kiziltan, Symmetry breaking ordering constraints, PhD thesis, Uppsala University, 2004.
[18] A.K. Mackworth, Consistency in networks of relations, Artificial Intelligence 8 (1977) 99–118.
[19] A.K. Mackworth. On reading sketch maps, in: Proceedings of the 5th International Joint Conference on Artificial Intelligence, 1977, pp. 598–606.
[20] P. Meseguer, C. Torras, Solving strategies for highly symmetric CSPs, in: T. Dean (Ed.), Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99), Morgan Kaufmann, 1999, pp. 400–405.
[21] L. Proll, B.M. Smith, Integer linear programming and constraint programming approaches to a template design problem, INFORMS Journal on Computing 10 (3) (1998) 265–275.
[22] 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.

[23] A. Schrijver, Theory of Linear Integer Programming, John Wiley and Sons, 1986.
[24] 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.
[25] I. Shlyakhter, Generating effective symmetry-breaking predicates for search problems, Electronic Notes in Discrete Mathematics 9 (2001).
[26] Swedish Institute of Computer Science, SICStus Prolog User's Manual, Release 3.10.1, April 2003.
[27] M. Wallace, Practical applications of constraint programming, Constraints 1 (1/2) (1996) 139–168.
[28] J.P. Walser, Integer Optimization by Local Search—A Domain-Independent Approach, Lecture Notes in Artificial Intelligence, vol. 1637, Springer, 1999.
[29] M.G. Wallace, S. Novello, J. Schimpf, ECLiPSe: A platform for constraint logic programming, ICL Systems Journal 12 (1) (1997) 159–200.