Combining Symmetry Breaking with Other Constraints: lexicographic ordering with sums *

Brahim Hnich[1], Zeynep Kiziltan[2], and Toby Walsh[1]

[1] Cork Constraint Computation Center, University College Cork, Ireland.
{brahim, tw}@4c.ucc.ie
[2] Department of Information Science, Uppsala University, Sweden.
Zeynep.Kiziltan@dis.uu.se

Abstract. We introduce a new global constraint which combines together the lexicographic ordering constraint with some sum constraints. Lexicographic ordering constraints are frequently used to break symmetry, whilst sum constraints occur in many problems involving capacity or partitioning. Our results show that this global constraint is useful when there is a very large space to explore, such as when the problem is unsatisfiable, or when the search strategy is poor or conflicts with the symmetry breaking constraints. By studying in detail when combining lexicographical ordering with other constraints is useful, we propose a new heuristic for deciding when to combine constraints together.

1 Introduction

Global constraints specify patterns that reoccur in many problems. For example, we often have row and column symmetry on a 2-d matrix of decision variables and can post lexicographic ordering constraints on the rows and columns to break much of this symmetry [5, 3]. There are, however, only a limited number of common constraints like the lexicographic ordering constraint which repeatedly occur in problems. New global constraints are therefore likely to be increasingly more specialized. An alternative strategy for developing global constraints that might be useful in a wide range of problems is to identify constraints that often occur together, and develop efficient constraint propagation algorithms for their combination. In this paper, we explore this strategy.

We introduce a new global constraint on 0/1 variables that combines together the lexicographic ordering constraint with two sum constraints. Sum and lexicographic ordering constraints frequently occur together in problems involving capacity or partitioning that are modelled with symmetric matrices of decision variables. Examples are the ternary Steiner problem, the balanced incomplete block design problem, the rack configuration problem, social golfers, etc. Our results show that this new constraint is most useful when there is a very large space to explore, such as when the problem is unsatisfiable, or when the branching heuristics are poor or conflict with the symmetry breaking constraints. The

* This research is supported by Science Foundation Ireland. We thank the other members of the 4C lab, the APES research group, Alan Frisch and Chris Jefferson.

combined constraint gives additional pruning and this can, for example, help compensate in part for the branching heuristic trying to push the search in a different direction to the symmetry breaking constraints. Combining constraints is a significant step towards tackling one of the most common criticisms of using symmetry breaking constraints. By increasing the amount of propagation, we can, partly, tackle conflict between the branching heuristic and the symmetry breaking constraints. Finally, by studying in detail when combining lexicographical ordering with other constraints is useful, we propose a new heuristic for deciding when to combine heuristics together. The heuristic suggests that the combination should be likely to prune a significant number of shared variables.

2 Preliminaries

A constraint satisfaction problem (CSP) is a set of variables, each with a finite domain of values, and a set of constraints that specify allowed values for subsets of variables. A solution to a CSP is an assignment of values to the variables satisfying the constraints. To find such solutions, constraint solvers often explore the space of partial assignments enforcing a local consistency like generalized arc-consistency (GAC). A constraint is GAC iff, when a variable in the constraint is assigned any of its values, compatible values exist for all the other variables in the constraint. For totally ordered domains, like integers, another level of consistent is bounds-consistency (BC). A constraint is bounds consistent (BC) iff, when a variable in the constraint is assigned its maximum or minimum value, there exist compatible values for all the other variables in the constraint. If a constraint c is BC or GAC then we write BC(c) or GAC(c) respectively.

In this paper, we are interested in lexicographic ordering of vectors of variables in the presence of sum constraints on the vectors. We denote a vector x of n finite integer variables as $\boldsymbol{X} = \langle X_0, \ldots, X_{n-1} \rangle$, while we denote a vector x of n ground values as $\boldsymbol{x} = \langle x_0, \ldots, x_{n-1} \rangle$. The sub-vector of $\boldsymbol{x}$ with start index a and last index b inclusive is denoted by $\boldsymbol{x}_{a \to b}$. The domain of a finite integer variable V is denoted by $\mathcal{D}(V)$, and the minimum and the maximum elements in this domain by $min(\mathcal{D}(V))$ and $max(\mathcal{D}(V))$.

Given two vectors, $\boldsymbol{X}$ and $\boldsymbol{Y}$ of variables, we write a lexicographical ordering constraint as $\boldsymbol{X} \leq_{\text{lex}} \boldsymbol{Y}$ and a strict lexicographic ordering constraint as $\boldsymbol{X} <_{\text{lex}} \boldsymbol{Y}$. $\boldsymbol{X} \leq_{\text{lex}} \boldsymbol{Y}$ ensures that: $X_0 \leq Y_0$; $X_1 \leq Y_1$ when $X_0 = Y_0$; $X_2 \leq Y_2$ when $X_0 = Y_0$ and $X_1 = Y_1$; $\ldots$; $X_{n-1} \leq Y_{n-1}$ when $X_0 = Y_0$, $X_1 = Y_1$, $\ldots$, and $X_{n-2} = Y_{n-2}$. $\boldsymbol{X} <_{\text{lex}} \boldsymbol{Y}$ ensures that: $\boldsymbol{X} \leq_{\text{lex}} \boldsymbol{Y}$; and $X_{n-1} < Y_{n-1}$ when $X_0 = Y_0$, $X_1 = Y_1$, $\ldots$, and $X_{n-2} = Y_{n-2}$. We write $\texttt{LexLeqAndSum}(\boldsymbol{X}, \boldsymbol{Y}, Sx, Sy)$ for the constraint which ensure that $\boldsymbol{X} \leq_{lex} \boldsymbol{Y}$, and that $\sum_i X_i = Sx$ and that $\sum_i Y_i = Sy$. Similarly, we write $\texttt{LexLessAndSum}(\boldsymbol{X}, \boldsymbol{Y}, Sx, Sy)$ for $\boldsymbol{X} <_{lex} \boldsymbol{Y}$, $\sum_i X_i = Sx$, and $\sum_i Y_i = Sy$. We denote the dual cases as $\texttt{LexGeqAndSum}(\boldsymbol{X}, \boldsymbol{Y}, Sx, Sy)$ and as $\texttt{LexGreaterAndSum}\ (\boldsymbol{X}, \boldsymbol{Y}, Sx, Sy)$. We assume that the variables being ordered are disjoint and not repeated. We also assume that Sx and Sy are ground and discuss the case when they are bounded variables in Section 5.

3 A worked example

We consider the special (but nevertheless useful) case of vectors of 0/1 variables. Generalizing the algorithm to non Boolean variables remains a significant challenge as it will involve solving subset sum problems. Fortunately, many of the applications of our algorithm merely require 0/1 variables. To maintain GAC on `LexLeqAndSum(`$\boldsymbol{X}, \boldsymbol{Y}, Sx, Sy$`)`, we minimize $\boldsymbol{X}$ lexicographically with respect to the sum and identify which positions in $\boldsymbol{Y}$ support 0 or 1. We then maximize $\boldsymbol{Y}$ lexicographically with respect to the sum and identify which positions in $\boldsymbol{X}$ support 0 or 1. Since there are two values and two vectors to consider, the algorithm has 4 steps.

In each step, we maintain a pair of lexicographically minimal and maximal ground vectors $\boldsymbol{sx} = \langle sx_0, \ldots, sx_{n-1}\rangle$ and $\boldsymbol{sy} = \langle sy_0, \ldots, sy_{n-1}\rangle$, and a flag α where for all $i < \alpha$ we have $sx_i = sy_i$ and $sx_\alpha \neq sy_\alpha$. That is, α is the most significant index where $\boldsymbol{sx}$ and $\boldsymbol{sy}$ differ. Additionally, we may need to know whether $\boldsymbol{sx}_{\alpha+1\to n-1}$ and $\boldsymbol{sy}_{\alpha+1\to n-1}$ are lexicographically ordered. Therefore, we introduce a boolean flag γ whose value is *true* iff $\boldsymbol{sx}_{\alpha+1\to n-1} \leq_{lex} \boldsymbol{sy}_{\alpha+1\to n-1}$.

Consider the vectors

$$\begin{array}{rl}\boldsymbol{X} = & \langle\{0,1\},\ \{0,1\},\ \ \{0\},\ \ \{0\},\ \{0,1\},\ \{0,1\},\ \{0\},\ \ \{0\}\rangle \\ \boldsymbol{Y} = & \langle\{0,1\},\ \{0,1\},\ \{0,1\},\ \{1\},\ \{0,1\},\ \{0,1\},\ \{0\},\ \{0,1\}\rangle\end{array}$$

and the constraints $\boldsymbol{X} \leq_{lex} \boldsymbol{Y}$, $\sum_i X_i = 3$, and $\sum_i Y_i = 2$. Each of these constraints are GAC and thus no pruning is possible. Our algorithm that maintains GAC on `LexLeqAndSum`$(\boldsymbol{X}, \boldsymbol{Y}, 3, 2)$ starts with step 1 in which we have

$$\begin{array}{rl}\boldsymbol{sx} = & \langle 0,\ 0,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0\rangle \\ \boldsymbol{sy} = & \langle 1,\ 0,\ 0,\ 1,\ 0,\ 0,\ 0,\ 0\rangle \\ & \ \uparrow \alpha\end{array}$$

where $\boldsymbol{sx} = min\{\boldsymbol{x} |\ \sum_i x_i = 2 \wedge \boldsymbol{x} \in \boldsymbol{X}\}$, and $\boldsymbol{sy} = max\{\boldsymbol{Y} |\ \sum_i y_i = 2 \wedge \boldsymbol{y} \in \boldsymbol{Y}\}$. We check where we can place one more 1 in $\boldsymbol{sx}$ to make the sum 3 as required without disturbing $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$. We have $\alpha = 0$ and $\gamma = true$. We can safely place 1 to the right of α as this does not affect $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$. Since γ is *true*, placing 1 at α also does not affect the order of the vectors. Therefore, all the 1s in $\boldsymbol{X}$ have support.

In step 2 we have

$$\begin{array}{rl}\boldsymbol{sx} = & \langle 1,\ \ 1,\ \ 0,\ 0,\ 1,\ 1,\ 0,\ 0\rangle \\ \boldsymbol{sy} = & \langle 1,\ \ 0,\ \ 0,\ 1,\ 0,\ 0,\ 0,\ 0\rangle \\ & \qquad \uparrow \alpha\end{array}$$

where $\boldsymbol{sx} = min\{\boldsymbol{x} |\ \sum_i x_i = 4 \wedge \boldsymbol{x} \in \boldsymbol{X}\}$, and $\boldsymbol{sy}$ is as before. We check where we can place one more 0 in $\boldsymbol{sx}$ to make the sum 3 as required to obtain $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$. We have $\alpha = 1$ and $\gamma = true$. Placing 0 to the left of α makes $\boldsymbol{sx}$ smaller than $\boldsymbol{sy}$. Since γ is *true*, placing 0 at α also makes $\boldsymbol{sx}$ smaller than $\boldsymbol{sy}$. However, placing 0 to the right of α orders the vectors lexicographically the wrong way around. Hence, we remove 0 from the domains of the variables of $\boldsymbol{X}$ on the right hand side of α. The vector $\boldsymbol{X}$ is now $\langle\{0, 1\}, \{0, 1\}, \{0\}, \{0\}, \{1\}, \{1\}, \{0\}, \{0\}\rangle$.

In step 3 we have

$$\begin{array}{rl}\boldsymbol{sx} = & \langle 0,\ 1,\ 0,\ 0,\ 1,\ 1,\ 0,\ 0\rangle \\ \boldsymbol{sy} = & \langle 1,\ 1,\ 0,\ 1,\ 0,\ 0,\ 0,\ 0\rangle \\ & \ \uparrow \alpha\end{array}$$

where $\boldsymbol{sx} = min\{\boldsymbol{x} | \sum_i x_i = 3 \wedge \boldsymbol{x} \in \boldsymbol{X}\}$, and $\boldsymbol{sy} = max\{\boldsymbol{Y} | \sum_i y_i = 3 \wedge \boldsymbol{y} \in \boldsymbol{Y}\}$. We check where we can place one more 0 in $\boldsymbol{sy}$ to make the sum 2 as required without disturbing $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$. We have $\alpha = 0$ and $\gamma = true$. We can safely place 0 to the right of α as this does not affect $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$. Since γ is $true$, placing 0 at α also does not affect the order of the vectors. Therefore, all the 0s in $\boldsymbol{Y}$ have support.

Finally, in step 4 we have

$$\begin{array}{rl} \boldsymbol{sx} = & \langle 0,\ \ 1,\ \ 0, 0, 1, 1, 0, 0\rangle \\ \boldsymbol{sy} = & \langle 0,\ \ 0,\ \ 0, 1, 0, 0, 0, 0\rangle \\ & \quad\ \ \uparrow \alpha \end{array}$$

where $\boldsymbol{sx}$ is as before, and $\boldsymbol{sy} = max\{\boldsymbol{Y} | \sum_i y_i = 1 \wedge \boldsymbol{y} \in \boldsymbol{Y}\}$. We check where we can place one more 1 in $\boldsymbol{sy}$ to make the sum 2 as required to obtain $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$. We have $\alpha = 1$ and $\gamma = true$. Placing 1 to the left of α makes $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$ and so is safe. Since γ is $true$, we can also safely place 1 at α. However, placing 1 to the right of α makes $\boldsymbol{sx} >_{lex} \boldsymbol{sy}$. Hence, we remove 1 from the domains of the variables of $\boldsymbol{Y}$ on the right hand side of α. The algorithm now terminates with:

$$\begin{array}{rl} \boldsymbol{X} = & \langle \{0,1\}, \{0,1\}, \{0\}, \{0\}, \{1\}, \{1\}, \{0\}, \{0\}\rangle \\ \boldsymbol{Y} = & \langle \{0,1\}, \{0,1\}, \{0\}, \{1\}, \{0\}, \{0\}, \{0\}, \{0\}\rangle \end{array}$$

4 Algorithm

The algorithm first establishes BC on the sum constraints via the call `BC`, which is based on proposition 1 in [10]. Note that for 0/1 variables, BC is equivalent to GAC. If no failure is encountered we continue with 4 pruning steps. In step 1, we are concerned with support for 1s in $\boldsymbol{X}$. We first construct $\boldsymbol{sx}$ as the minimum $\boldsymbol{X}$ with $\sum_i X_i = Sx - 1$, and $\boldsymbol{sy}$ as the maximum $\boldsymbol{Y}$ with $\sum_i Y_i = Sy$. We then determine where we can place one more 1 on $\boldsymbol{sx}$ and have $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$.

If $\boldsymbol{sx} \geq_{lex} \boldsymbol{sy}$ then placing another 1 on any location of $\boldsymbol{sx}$ makes $\boldsymbol{sx} >_{lex} \boldsymbol{sy}$. Since the constraint is not disentailed, we already have $min\{\boldsymbol{x} | \sum_i x_i = Sx \wedge \boldsymbol{x} \in \boldsymbol{X}\} \leq_{lex} \boldsymbol{sy}$ which implies that $\boldsymbol{sx} <_{lex} \boldsymbol{sy}$. Now, placing 1 on $\boldsymbol{sx}$ to the left of α makes $\boldsymbol{sx} >_{lex} \boldsymbol{sy}$. This is true also for α provided that the subvectors after α are not lexicographically ordered. Anywhere after α has support since $\boldsymbol{sx} <_{lex} \boldsymbol{sy}$ remains valid. We therefore prune all 1s in $\boldsymbol{X}$ to the left of α and at α if $\boldsymbol{sx}_{\alpha+1 \rightarrow n-1} >_{lex} \boldsymbol{sy}_{\alpha+1 \rightarrow n-1}$, maintain $BC(\sum_i X_i = Sx)$ (via the call `BC`), and continue with step 2.

In step 2, we are concerned with support for 0s in $\boldsymbol{X}$. We first construct $\boldsymbol{sx}$ as the minimum $\boldsymbol{X}$ with $\sum_i X_i = Sx + 1$, and $\boldsymbol{sy}$ as the maximum $\boldsymbol{Y}$ with $\sum_i Y_i = Sy$. We then determine where we can place one more 0 on $\boldsymbol{sx}$ and have $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$. If $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$ then placing another 0 on any location of $\boldsymbol{sx}$ makes $\boldsymbol{sx} < \boldsymbol{sy}$, so we do no pruning. If however $\boldsymbol{sx} >_{lex} \boldsymbol{sy}$, placing 0 on $\boldsymbol{sx}$ to the right of α does not change that $\boldsymbol{sx} > \boldsymbol{sy}$. This is true also for α provided that the subvectors after α are not lexicographically ordered. Anywhere before α has support as in this case we have $\boldsymbol{sx} <_{lex} \boldsymbol{sy}$. We therefore prune all 0s

Algorithm 1: `LexLeqAndSum`

Data : $\langle X_0, X_1, \ldots, X_{n-1}\rangle$ with $\mathcal{D}(X_i) \subseteq \{0,1\}$, Integer Sx,
$\langle Y_0, Y_1, \ldots, Y_{n-1}\rangle$ with $\mathcal{D}(Y_i) \subseteq \{0,1\}$, Integer Sy

Result : GAC($\boldsymbol{X} \leq_{lex} \boldsymbol{Y} \wedge \sum_i X_i = Sx \wedge \sum_i Y_i = Sy$)

```
BC(X, Sx);
BC(Y, Sy);
sx := min{x | Σ_i x_i = Sx ∧ x ∈ X};
sy := min{y | Σ_i y_i = Sy ∧ y ∈ Y};
if sx >_lex sy then fail;
if ∃i ∈ {0,...,n−1} . |D(X_i)| > 1 then
    sx := min{x | Σ_i x_i = Sx − 1 ∧ x ∈ X};
    PruneLeft(sx, sy, X);
    BC(X, Sx);
    if ∃i ∈ {0,...,n−1} . |D(X_i)| > 1 then
        sx := min{x | Σ_i x_i = Sx + 1 ∧ x ∈ X};
        PruneRight(sx, sy, X);
        BC(X, Sx);
    end
end
if ∃i ∈ {0,...,n−1} . |D(Y_i)| > 1 then
    sx := min{x | Σ_i x_i = Sx ∧ x ∈ X};
    sy := max{y | Σ_i y_i = Sy + 1 ∧ y ∈ Y};
    PruneLeft(sx, sy, Y);
    BC(Y, Sy);
    if ∃i ∈ {0,...,n−1} . |D(Y_i)| > 1 then
        sy := max{y | Σ_i y_i = Sy − 1 ∧ y ∈ Y};
        PruneRight(sx, sy, Y);
        BC(Y, Sy);
    end
end
```

in $\boldsymbol{X}$ to the right of α and at α if $\boldsymbol{sx}_{\alpha+1\to n-1} >_{lex} \boldsymbol{sy}_{\alpha+1\to n-1}$, and maintain $BC(\sum_i x_i = Sx)$ (via the call `BC`).

Step 3 is very similar to step 1, except we identify support for the 0s in $\boldsymbol{Y}$. We first construct $\boldsymbol{sx}$ as the minimum $\boldsymbol{X}$ with $\sum_i X_i = Sx$, and $\boldsymbol{sy}$ as the maximum $\boldsymbol{Y}$ with $\sum_i Y_i = Sy + 1$. We then determine where we can place one more 0 on $\boldsymbol{sy}$ and have $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$. Instead of pruning 1s, we now prune from $\boldsymbol{Y}$ those 0s which lack support. Due to this similarity, we can perform the first and the third steps of the algorithm with the same procedure, `PruneLeft`. The input to the procedure is the vectors $\boldsymbol{sx}$ and $\boldsymbol{sy}$ and either of $\boldsymbol{X}$ and $\boldsymbol{Y}$. `PruneLeft` prunes either 1s from $\boldsymbol{X}$ (via the call `setMax`) or 0s from $\boldsymbol{Y}$ (via the call `setMin`) between the beginning of the vector and index α.

Step 4 is very similar to step 2, except we identify support for the 1s in $\boldsymbol{Y}$. We first construct $\boldsymbol{sx}$ as the minimum $\boldsymbol{X}$ with $\sum_i X_i = Sx$, and $\boldsymbol{sy}$ as the maximum $\boldsymbol{Y}$ with $\sum_i Y_i = Sy - 1$. We then determine where we can place one more 1 on $\boldsymbol{sy}$ and have $\boldsymbol{sx} \leq_{lex} \boldsymbol{sy}$. Instead of pruning 0s, we now prune from $\boldsymbol{Y}$

Procedure `BC(`$\boldsymbol{X}, Sx$`)`

$min = \sum_{i \in [0,n)} min(\mathcal{D}(X_i))$;
$max = \sum_{i \in [0,n)} max(\mathcal{D}(X_i))$;
`setMin`(Sx, min);
`setMax`(Sx, max);
foreach $i \in [0, n)$ **do**
    `setMin`($X_i, min(\mathcal{D}(Sx)) - max + max(\mathcal{D}(X_i))$);
    `setMax`($X_i, max(\mathcal{D}(Sx)) - min + min(\mathcal{D}(X_i))$);
end

Procedure `PruneLeft(`$\boldsymbol{sx}, \boldsymbol{sy}, \boldsymbol{V}$`)`

$\alpha := 0$;
while $sx_\alpha = sy_\alpha$ **do** $\alpha := \alpha + 1$;
$\gamma := false$;
if $\boldsymbol{sx}_{\alpha \to n-1} \leq_{lex} \boldsymbol{sy}_{\alpha \to n-1}$ **then** $\gamma := true$;
$i := 0$;
while $i < \alpha$ **do**
    if $|\mathcal{D}(V_i)| > 1$ **then**
        if $\boldsymbol{V} = \boldsymbol{X}$ **then** `setMax`($V_i, 0$);
        else if $\boldsymbol{V} = \boldsymbol{Y}$ **then** `setMin`($V_i, 1$);
    end
    $i := i + 1$;
end
if $\neg\gamma \ \wedge \ |\mathcal{D}(V_\alpha)| > 1$ **then**
    if $\boldsymbol{V} = \boldsymbol{X}$ **then** `setMax`($V_\alpha, 0$);
    else if $\boldsymbol{V} = \boldsymbol{Y}$ **then** `setMin`($V_\alpha, 1$);
end

those 1s which lack support. Due to this similarity, we can perform the second and the fourth steps of the algorithm with the same procedure, `PruneRight`. The input to the procedure is the vectors $\boldsymbol{sx}$ and $\boldsymbol{sy}$ and either of $\boldsymbol{X}$ and $\boldsymbol{Y}$. `PruneRight` prunes either 0s from $\boldsymbol{X}$ (via the call `setMin`) or 1s from $\boldsymbol{Y}$ (via the call `setMax`) between the end of the vector and index α.

In `PruneLeft` and `PruneRight`, we consider only the variables the domains of which are not singleton. Also, we skip a step of the algorithm if the variables of the corresponding vector have all singleton domains. The reason is as follows. At the beginning of the algorithm, we check whether $min\{\boldsymbol{x} | \ \sum_i x_i = Sx \wedge \boldsymbol{x} \in \boldsymbol{X}\} >_{lex} max\{\boldsymbol{y} | \ \sum_i y_i = Sy \ \wedge \ \boldsymbol{y} \in \boldsymbol{Y}\}$. If yes then we fail, otherwise we have $min\{\boldsymbol{x} | \ \sum_i x_i = Sx \wedge \boldsymbol{x} \in \boldsymbol{X}\} \leq_{lex} max\{\boldsymbol{y} | \ \sum_i y_i = Sy \wedge \boldsymbol{y} \in \boldsymbol{Y}\}$. This means that there is at least one value in the domain of each variable which is consistent. The algorithm therefore seeks support for a variable only if its domain is not a singleton.

Note that the normal execution flow of the algorithm is steps 1 and 2, and then steps 3 and 4. There are a number of reasons for this choice. First, in step 2, we can re-use the vector $\boldsymbol{sy}$ constructed in step 1. Similarly, in step 4 we can

Procedure `PruneRight`($\boldsymbol{sx}, \boldsymbol{sy}, \boldsymbol{V}$)

> **while** $\alpha < n \;\wedge\; sx_\alpha = sy_\alpha$ **do** $\alpha := \alpha + 1$;
> **if** $\alpha = n \;\vee\; sx_\alpha < sy_\alpha$ **then** return;
> $\gamma := false$;
> **if** $\boldsymbol{sx}_{\alpha\to n-1} \leq_{lex} \boldsymbol{sy}_{\alpha\to n-1}$ **then** $\gamma := true$;
> $i := n - 1$;
> **while** $i > \alpha$ **do**
>   **if** $|\mathcal{D}(V_i)| > 1$ **then**
>   **if** $\boldsymbol{V} = \boldsymbol{X}$ **then** `setMin`($V_i, 1$);
>   **else if** $\boldsymbol{V} = \boldsymbol{Y}$ **then** `setMax`($V_i, 0$);
>   **end**
> $i := i - 1$;
> **end**
> **if** $\neg\gamma \;\wedge\; |\mathcal{D}(V_\alpha)| > 1$ **then**
>   **if** $\boldsymbol{V} = \boldsymbol{X}$ **then** `setMin`($V_\alpha, 1$);
>   **else if** $\boldsymbol{V} = \boldsymbol{Y}$ **then** `setMax`($V_\alpha, 0$);
> **end**

work on $\boldsymbol{sx}$ constructed in step 3. Second, from step 1 to step 2, we increase the number of 1s in $\boldsymbol{sx}$ by two and do not change $\boldsymbol{sy}$. Note that in step 1, all 1s in $\boldsymbol{X}$ before X_α and may be 1 at X_α are pruned. In step 2, α is either unchanged (i.e., 1 in X_α is pruned in step 1) or it moves only to the right. Therefore, in step 2 we can reuse the previous value of α as the lower bound while looking for its new value. Similar argument holds between steps 3 and 4. As step 3 prunes all 0s before Y_α, the new value of α in step 4 is at least its previous value. Third, from step 2 to step 3, we decrease the number of 1s in $\boldsymbol{sx}$ and increase in $\boldsymbol{sy}$ by one. So α may move to the left and thus we must recompute α in step 3. Note that similar argument holds from step 4 to step 1, therefore it does not matter which of steps 1-2 and steps 3-4 we perform first.

None of the prunings require any recursive calls back to the algorithm. We tighten only $min\{\boldsymbol{y}|\ \sum_i y_i = Sy \wedge \boldsymbol{y} \in \boldsymbol{Y}\}$ and $max\{\boldsymbol{x}|\ \sum_i x_i = Sx \wedge \boldsymbol{x} \in \boldsymbol{X}\}$ without touching $max\{\boldsymbol{y}|\ \sum_i y_i = Sy \wedge \boldsymbol{y} \in \boldsymbol{Y}\}$ and $min\{\boldsymbol{x}|\ \sum_i x_i = Sx \wedge \boldsymbol{x} \in \boldsymbol{X}\}$ which provide support for the values in the vectors. The exception is when a domain wipe-out occurs. As we have $min\{\boldsymbol{x}|\ \sum_i x_i = Sx \;\wedge\; \boldsymbol{x} \in \boldsymbol{X}\} \leq_{lex} max\{\boldsymbol{y}|\ \sum_i y_i = Sy \;\wedge\; \boldsymbol{y} \in \boldsymbol{Y}\}$, $min(\mathcal{D}(X_i))$ and $max(\mathcal{D}(Y_i))$ for all $0 \leq i < n$ are consistent. This means that the prunings of the algorithm cannot cause any domain wipe-out.

5 Theoretical Properties and Extensions

The algorithm `LexLeqAndSum` runs in linear time in the length of the vectors and is correct.

Theorem 5.1. `LexLeqAndSum` *runs in time O(n).*

Proof. BC runs in O(n) [10]. Each of PruneLeft and PruneRight runs in 0(n) as in the worst the case the whole vectors need to be traversed. Checking whether a vector is ground can be done in constant time by comparing $\sum_i min(\mathcal{D}(X_i))$ and $\sum_i max(\mathcal{D}(X_i))$ that are recomputed at every call to BC. Hence the algorithm runs in 0(n). □

Theorem 5.2. LexLeqAndSum *either establishes failure if* LexLeqAndSum*($\boldsymbol{X}, \boldsymbol{Y}, Sx, Sy$) is not satisfiable, or prunes all inconsistent values from $\boldsymbol{X}$ and $\boldsymbol{Y}$ to ensure GAC(*LexLeqAndSum*($\boldsymbol{X}, \boldsymbol{Y}, S_x, S_y$)).*

Proof. For reasons of space, the proof is given in [6]. □

Strict Ordering The algorithm can easily be modified for the strict ordering constraint LexLessAndSum ($\boldsymbol{X}, \boldsymbol{Y}, Sx, Sy$). To do so, we need to disallow equality between the vectors. This requires just two modifications to the algorithm. First, we change the definition of γ. The flag γ is *true* iff $\boldsymbol{sx}_{\alpha+1\to n-1} <_{lex} \boldsymbol{sy}_{\alpha+1\to n-1}$. Second, we fail if we have $min\{\boldsymbol{x} |\ \sum_i x_i = Sx \ \wedge \ \boldsymbol{x} \in \boldsymbol{X}\} \geq_{lex} max\{\boldsymbol{y} |\ \sum_i y_i = Sy \ \wedge \ \boldsymbol{y} \in \boldsymbol{Y}\}$.

Bounded Sums We can also deal with sums that are not ground but bounded. Assume we have $lx \leq Sx \leq ux$ and $ly \leq Sy \leq uy$. We now need to find support first for the values in the domains of the vectors and second for the values in the range of $lx..ux$ and $ly..uy$. In the first part, we can run our algorithm LexLeqAndSum with $\sum_i X_i = lx$ and $\sum_i Y_i = uy$. In the second part, we tighten the upper bound of Sx with respect to the upper bound of Sy so that $max\{\boldsymbol{x} | \sum_i x_i = ux \ \wedge \ \boldsymbol{x} \in \boldsymbol{X}\} \leq_{lex} max\{\boldsymbol{y} | \sum_i y_i = uy \ \wedge \ \boldsymbol{y} \in \boldsymbol{Y}\}$. The support for the upper bound of Sx is also the support for all the other values in the domain of Sx. Similarly, we tighten the lower bound of Sy with respect to the lower bound of Sx so that $min\{\boldsymbol{x} | \sum_i x_i = lx \ \wedge \ \boldsymbol{x} \in \boldsymbol{X}\} \leq_{lex} min\{\boldsymbol{y} | \sum_i y_i = ly \ \wedge \ \boldsymbol{y} \in \boldsymbol{Y}\}$. The support for the lower bound of Sy is also the support for all the other values in the domain of Sy. The values in the vectors are supported by lx and uy. The prunings of the second part tighten only ux and ly. Hence the prunings performed in the second part do not require any calls back to the first part. It is easy to show that the modified algorithm is correct and runs in linear time.

Entailment A constraint is entailed when any assignment of values to its variables satisfy the constraint. Detecting entailment does not change the worst-case complexity but is very useful for avoiding unnecessary work. For this purpose, we can maintain a flag *entailed*, which is set to true whenever the constraint LexLeqAndSum is entailed, and the algorithm directly returns on future calls if *entailed* is set to *True*. The constraint is entailed when we have $max\{\boldsymbol{x} |\ \sum_i x_i = Sx \ \wedge \ \boldsymbol{x} \in \boldsymbol{X}\} \leq_{lex} min\{\boldsymbol{y} |\ \sum_i y_i = Sy \ \wedge \ \boldsymbol{y} \in \boldsymbol{Y}\}$.

6 Multiple Vectors

If we have multiple rows of a matrix that are lexicographically ordered and are constrained by sum constraints, we can decompose this into lexicographic ordering with sum constraints on all pairs of rows, or (further still) onto ordering constraints just on neighbouring pairs of rows. Such decompositions hinder constraint propagation. However, we identify a special case which occurs in a number of applications where it does not hinder propagation. This case is when the sums are just 1, which occurs in many assignment problems. Nevertheless, it is usually more cost effective to post just the $O(n)$ ordering constraints between neighbouring pairs rather than the $O(n^2)$ constraints between all pairs.

Theorem 6.3. *$GAC(\texttt{LexLeqAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, S_i, S_j))$ for all $i < j$ is strictly stronger than $GAC(\texttt{LexLeqAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_{i+1}, S_i, S_{i+1}))$ for all i.*

Proof. Consider the following 3 vectors:

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

with $S_{X_0} = S_{X_1} = S_{X_2} = 2$. We have $GAC(\texttt{LexLeqAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_{i+1}, S_i, S_{i+1}))$ for all i. The assignment $X_{0,0} = \{1\}$ forces $\boldsymbol{X}_0$ to be $\langle 1,1,0,0\rangle$, which is not supported by $\boldsymbol{X}_2$ whose largest value is $\langle 1,0,1,0\rangle$. Therefore, $GAC(\texttt{LexLeqAndSum}$ $(\boldsymbol{X}_0, \boldsymbol{X}_2, S_0, S_2))$ does not hold. □

Theorem 6.4. *$GAC(\texttt{LexLessAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, S_i, S_j))$ for all $i < j$ is strictly stronger than $GAC(\texttt{LexLessAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_{i+1}, S_i, S_{i+1}))$ for all i.*

Proof. Consider the following 3 vectors:

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

with $S_{X_0} = S_{X_1} = S_{X_2} = 2$. We have $GAC(\texttt{LexLessAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_{i+1}, S_i, S_{i+1}))$ for all i. The assignment $X_{0,0} = \{1\}$ forces $\boldsymbol{X}_0$ to be $\langle 1,0,1,0,0\rangle$, which is not supported by $\boldsymbol{X}_2$ whose largest value is $\langle 1,0,0,1,0\rangle$. Therefore, $GAC(\texttt{LexLessAndSum}$ $(\boldsymbol{X}_0, \boldsymbol{X}_2, S_0, S_2))$ does not hold. □

Theorem 6.5. *$GAC(\forall i < j\ .\texttt{LexLeqAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, S_i, S_j))$ is strictly stronger than $GAC(\texttt{LexLeqAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, S_i, S_j))$ for all $i < j$.*

Proof. Consider the following 3 vectors:

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

with $S_{X_0} = 2$, $S_{X_1} = 3$, and $S_{X_2} = 3$. We have $GAC(\texttt{LexLeqAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, S_i, S_j))$ for all $i < j$. The assignment $X_{0,0} = \{1\}$ forces $\boldsymbol{X}_0$ to be $\langle 1,0,1,0,0,0\rangle$, which is supported by $\boldsymbol{X}_1$ only with the assignment $\langle 1,1,0,1,0,0\rangle$. The latter assignment is however not supported by $\boldsymbol{X}_2$ whose largest value is $\langle 1,1,0,0,1,0\rangle$. Therefore, $GAC(\forall i < j\ .\texttt{LexLeqAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, S_i, S_j))$ does not hold. □

Theorem 6.6. *$GAC(\forall i < j$.*$\texttt{LexLessAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, S_i, S_j))$ *is strictly stronger than* $GAC(\texttt{LexLessAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, S_i, S_j))$ *for all* $i < j$.

Proof. Consider 7 vectors $\boldsymbol{X}_0$ to $\boldsymbol{X}_6$ with $\boldsymbol{X}_i = \langle\{0,1\},\{0,1\},\{0,1\},\{0,1\}\rangle$ and $S_{X_i} = 2$ for all $i \in [0, 6]$. Although $GAC(\texttt{LexLessAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, S_i, S_j))$ for all $i < j$ there is no globally consistent solution as there are only $\binom{4}{2} = 6$ possible distinct vectors. □

Theorem 6.7. *$GAC(\forall i < j$.*$\texttt{LexLessAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, 1, 1))$ *is equivalent to* $GAC(\texttt{LexLessAndSum}(\boldsymbol{X}_i, \boldsymbol{X}_j, 1, 1))$ *for all* $i < j$.

Proof. If $m > n$ then no solution is allowed. If $m \leq n$ then 1s in each row are only supported in a diagonal band of width $n - m + 1$ that moves from top-right to bottom-left. The decomposition is able to prune all other values. □

7 Experimental Results

We performed a wide range of experiments to test when this combination of constraints is useful in practice. In the following tables, the results for finding the first solution or that none exists are shown, where "-" means no result is obtained in 1 hour (3600 secs) using ILOG Solver 5.3 on a 1GHz pentium III processor with 256 Mb RAM under Windows XP.

Ternary Steiner Problem (tSp). tSp originates from the computation of hypergraphs in combinatorial mathematics. The tSp of order n is to find $n(n-1)/6$ subsets of $\{1, \ldots, n\}$ such that each subset is of cardinality 3 and any two subsets have at most one element in common [8]. One way of modelling this problem is to represent each subset using its characteristic function. We thus have a 0/1 matrix of n columns and $n(n-1)/6$ rows, with constraints enforcing exactly 3 ones per row, and a scalar product of at most 1 between any pair of distinct rows. Since the subsets are indistinguishable, we can freely permute the subsets to obtain symmetric partial assignments. Moreover, permuting the elements of the set $\{1, \ldots, n\}$ does not affect the cardinality of the subsets, nor the number of elements in common between any two subsets. Hence, the matrix has row and column symmetry. Such row and column symmetries can effectively be eliminated by imposing lexicographic ordering constraints on rows and columns [4]. Since the rows are also constrained by sum constraints, instead of posing the lexicographic ordering constraints between the rows and the sum constraints on the rows separately, we can impose our new global constraint between the rows.

There are two ways to pose lexicographic ordering constraints on a matrix with row and column symmetry: double-lex ($\leq_{lex}$RC) where both the rows and columns are ordered in increasing order, or double-antilex ($\geq_{lex}$RC) where both the rows and columns are ordered in decreasing order [4]. Our initial experimentation with tSps using lexicographic ordering constraints for symmetry breaking showed that the best way to solve the problem is to pose double-antilex constraints on the matrix. Also, due to the intersection constraint between any two

Problem n	No symmetry breaking Failures	Time (sec.)	$>_{lex}$R$\geq_{lex}$C Failures	Time (sec.)	`LexGreaterAndSum` R $\geq_{lex}$C Failures	Time (sec.)
6	6,195	0.3	14	0	11	0
7	6	0	2	0	1	0
8	-	>1hr	741	0.1	390	0.1
9	4,521	0.4	336	0.1	250	0.1
10	-	>1hr	723,210	128.8	433,388	105.3

Table 1. Ternary Steiner problem: row-wise labelling.

Problem n	No symmetry breaking Failures	Time (sec.)	$>_{lex}$R$\geq_{lex}$C Failures	Time (sec.)	`LexGreaterAndSum` R $\geq_{lex}$C Failures	Time (sec.)
6	12,248	0.5	22	0	11	0
7	115	0.3	21	0	14	0
8	-	>1hr	1,259	0.2	410	0.1
9	4,289,520	697.6	2,106	0.4	619	0.2
10	-	>1hr	4,153,162	940.5	643,152	217.2

Table 2. Ternary Steiner problem: row-and-column-wise labelling.

Problem n	No symmetry breaking Failures	Time (sec.)	$>_{lex}$R$\geq_{lex}$C Failures	Time (sec.)	`LexGreaterAndSum` R $\geq_{lex}$C Failures	Time (sec.)
6	26,352	1.2	47	0	27	0
7	585,469	40.4	146	0	52	0
8	-	>1hr	6,826	0.7	1,962	0.4
9	-	>1hr	89,760	14.1	8,971	2
10	-	>1hr	-	>1hr	3,701,480	1323.7

Table 3. Ternary Steiner problem: column-wise labelling.

subsets, no two rows can be equal. We can therefore strengthen the model by using strict lexicographical ordering constraints.

We tried many different search strategies, and obtained the best results by instantiating the matrix along its rows from top to bottom, and exploring the domain of each variable in *decreasing* order (i.e. 1 first and then 0). This search strategy together with the double-antilex constraints works very well for solving tSps. Table 1 shows the results on some tSp instances. Note that tSp has been proven to have solutions iff n modulo 6 is equal to 1 or 3 [8]. Therefore, only $n = 7, 9$ have solutions in the table.

We observe in Table 1 that the symmetry breaking double-antilex constraints significantly reduce the size of the search tree giving significantly shorter run-times compared to no symmetry breaking. The difference between the lexicographic ordering constraints and lexicographic ordering with sum constraints are not striking for the soluble instances ($n = 7, 9$). The difference becomes apparent with the unsatisfiable instances when we have a larger search space to explore.

If we change the search strategy slightly, the problem becomes much more difficult to solve. We then observe a dramatic increase in the size of the search tree if symmetries are not eliminated. Tables 2 and 3 show the search tree and the run-times obtained when labelling along the rows and columns, and along the columns of the matrix, respectively. Reasoning with lexicographic ordering constraints in the presence of sum constraints now becomes very useful, and the instances are solved much more quickly than the lexicographic ordering constraints alone, with notable differences in the size of the search tree. Note for instance that $n = 10$ in Table 3 could be proved to have no solution only with our algorithm, given the time limit 1 hour.

Balanced Incomplete Block Designs (BIBD). BIBD generation is a standard combinatorial problem from design theory with applications in cryptography and experimental design. A BIBD is a set V of $v \geq 2$ elements and a collection of $b > 0$ subsets of V, such that each subset consists of exactly k elements ($v > k > 0$), each element appears in exactly r subsets ($r > 0$), and each pair of elements appear simultaneously in exactly λ subsets ($\lambda > 0$).

A BIBD can be specified as a constraint program by a 0/1 matrix of b columns and v rows, with constraints enforcing exactly r ones per row, k ones per column, and a scalar product of λ between any pair of distinct rows. Since the objects and the subsets are indistinguishable, given a partial assignment of the matrix, we can exchange any pair of rows, and any pair of columns to obtain another symmetric partial assignment. This model is very similar to that of the tSp except that the columns are now also constrained by sum constraints. Hence, we can impose our new global constraint on both the rows and on the columns.

Our initial experiments with solving BIBDs using lexicographic ordering constraints for breaking symmetry showed that the best way to solve BIBDs is again to post double-antilex constraints on the matrix[3]. Also, due to the constraint between every pair of elements, no two rows can be equal. We therefore again strengthen the model by enforcing strict lexicographical ordering constraints.

Instantiating the matrix along its rows from top to bottom and exploring the domain of each variable in *increasing* order works extremely well with the double-antilex constraints. All the instances of [9] are solved within a few seconds. Bigger instances such as $\langle 15, 21, 7, 5, 2\rangle$ and $\langle 22, 22, 7, 7, 2\rangle$ are solved in less than a minute. With this search strategy, we observe no difference between the inference of our algorithm and its decomposition. Examples can be found in Table 4.

If there are no lexicographic ordering constraints posted on the matrix, which row to instantiate next is irrelevant. The same search tree is generated whether the rows are instantiated from top to bottom, or bottom to up, or in any order of choice. However, if the matrix is constrained by the lexicographic ordering constraints, then the order of the rows being instantiated affects the size of the search tree: many solutions are now considered as dead-ends as they do not match the ordering imposed by the lexicographic ordering constraints. Instead of exploring the rows from top to bottom, if we explore them from bottom to top then the problem becomes very difficult to solve in the presence of double-antilex constraints, i.e. even small instances become hard to solve within an hour. We can make the problem more difficult to solve by choosing one row from the top

[3] This was also pointed out by Jean-François Puget at his CP'02 talk on symmetry breaking.

Problem $\langle v, b, r, k, \lambda \rangle$	No symmetry breaking Failures	$>_{lex}$R $\geq_{lex}$C Failures	`LexGreaterAndSum` R `LexGeqAndSum` C Failures
6,20,10,3,4	8,944	76	76
7,21,9,3,3	7,438	42	42
6,30,15,3,6	1,893,458	68	68
7,28,12,3,4	229,241	64	64
9,24,8,3,2	6,841	48	48
6,40,20,3,8	-	108	108
7,35,15,3,5	7,814,878	88	88
7,42,18,3,6	-	115	115

Table 4. BIBDs: row-wise labelling (1).

Problem		No symmetry breaking		$>_{lex}$R $\geq_{lex}$C		`LexGreaterAndSum` R `LexGeqAndSum` C	
#	$\langle v, b, r, k, \lambda \rangle$	Failures	Time (sec.)	Failures	Time (sec.)	Failures	Time (sec.)
1	6,20,10,3,4	8,944	0.7	916	0.2	327	0.1
2	7,21,9,3,3	7,438	0.7	20,182	5.3	5,289	2.1
3	6,30,15,3,6	1,893,458	192.3	10,618	3.7	1,493	1
4	7,28,12,3,4	229,241	26.1	801,290	330.7	52,927	27
5	9,24,8,3,2	6,841	1.1	2,338,067	1115.9	617,707	524.3
6	6,40,20,3,8	-	>1hr	117,126	67.5	4,734	4.4
7	7,35,15,3,5	7,814,878	1444.4	-	> 1hr	382,173	311.2
8	7,42,18,3,6	-	>1hr	-	>1hr	2,176,006	2,603.7

Table 5. BIBDs: row-wise labelling (2).

and then one row from the bottom, and so on. Comparing Table 4 and Table 5 shows how the search tree is affected with double-antilex constraints, though there is no difference if there are no ordering constraints imposed.

We make a number of observations about these result. First, imposing double-antilex constraints significantly reduces the size of the search tree and time to solve the problem compared to no symmetry breaking. Moreover, the additional inference performed by our algorithm gives much smaller search trees in much shorter run-times. See entries 1, 3, and 6. Second, lexicographic ordering constraints and the search strategy clash, resulting in bigger search trees. However, the extra inference of our algorithm is able to compensate for this. This suggests that even if the ordering imposed by symmetry breaking constraints conflicts with the search strategy, more inference incorporated into the symmetry breaking constraints can significantly reduce the size of the search tree. See entries 2, 4, and 7. Third, increased inference scales up better, and recovers from mistakes much quicker. See entry 5. Finally, the problem can sometimes only be solved, when using a poor search strategy, by imposing our new global constraint. See entry 8.

8 Lexicographic Ordering with Other Constraints

We obtained similar results with other problems like rack configuration, steel mill slab design and the social golfers problem. In each case, the combined constraint was only useful when the symmetry breaking conflicted with the branching heuristic, the branching heuristic was poor, or there was a very large search space to explore. Why is this so?

Katsirelos and Bacchus have proposed a simple heuristic for combining constraints together [7]. The heuristic suggests grouping constraints together if they share many variables in common. This heuristic would suggest that combining lexicographical ordering and sum constraints would be very useful as they intersect on many variables. However, this ignores how the constraints are propagated. The lexicographical ordering constraint only prunes at one position, α[4]. If the vectors are already ordered at this position then any future assignments are irrelevant. Of course, α can move to the right but on average it moves only one position for each assignment. Hence, the lexicographic ordering constraint interacts on average with one variable from each of the sum constraints. Such interaction is of limited value because the constraints are already communicating with each other via the domain of that variable. This explains why combining lexicographical ordering and sum constraints is only of value on problems where there is a lot of search and even a small amount of extra inference may save exploring large failed subtrees.

A similar argument will hold for combining lexicographic ordering constraints with other constraints. For example, Carlsson and Beldiceanu have introduced a new global constraint, called **lex_chain**, which combines together a chain of lexicographic ordering constraints [2]. When we have a matrix say with row symmetry, we can now post a single lexicographic ordering constraint on all the m vectors corresponding to the rows as opposed to posting $m-1$ of them. In theory, such a constraint can give more propagation. However, our experiments on BIBDs indicate no gain over posting lexicographic ordering constraints between the adjacent vectors. In Table 6, we report the results of solving BIBDs using SICStus Prolog 3.10.1. We either post lexicographic ordering or anti-lexicographic ordering constraints on the rows and columns, and instantiate the matrix from top to bottom exploring the domains in ascending order. The lexicographic ordering constraints are posted using **lex_chain** of Carlsson and Beldiceanu, which is available in SICStus Prolog 3.10.1. This constraint is either posted once for all the symmetric rows/columns, or between adjacent rows/columns. In all the cases, we observe no benefits in combining a chain of lexicographic ordering constraints. The interaction between the constraints is again very restricted. Each of them is concerned only with a pair of variables and it interacts with its neighbour either at this position or at a position above its α where the variable is already ground. This argument suggests a new heuristic for combining constraints: the combination should be likely to prune a significant number of shared variables.

9 Conclusion

We have introduced a new global constraint on 0/1 variables which combines a lexicographical ordering constraint with sum constraints. Lexicographic ordering

[4] α points to the most significant index of $\boldsymbol{X}$ and $\boldsymbol{Y}$ where X_i and Y_i are not ground and equal.

v,b,r,k,λ	No symmetry breaking	$<_{lex}$ R $\leq_{lex}$ C **lex_chain**		$>_{lex}$ R $\geq_{lex}$ C **lex_chain**	
		$\langle X_0,\ldots,X_{m-1}\rangle$	$\langle X_i, X_{i+1}\rangle$	$\langle X_0,\ldots,X_{m-1}\rangle$	$\langle X_i, X_{i+1}\rangle$
	Backtracks	Backtracks	Backtracks	Backtracks	Bactracks
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	-	1,473	1,473	**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**

Table 6. BIBD: **lex_chain**($\langle X_0,\ldots,X_{m-1}\rangle$) vs **lex_chain**($\langle X_i, X_{i+1}\rangle$) for all $0 \leq i < m-1$ with row-wise labelling.

constraints are frequently used to break symmetry, whilst sum constraints occur in many problems involving capacity or partitioning. Our results showed that this global constraint is useful when there is a very large space to explore, such as when the problem is unsatisfiable, or when the branching heuristic is poor or conflicts with the symmetry breaking constraints. However, our combined constraint did not compensate in full for a poor branching heuristic. Overly, it was better to use a good branching heuristic. Finally, by studying in detail when combining lexicographical ordering with other constraints is useful, we proposed a new heuristic for deciding when to combine constraints together.

References

1. C. Bessière and J-C. Régin. Local consistency on conjunctions of constraints. Presented at ECAI-98 Workshop on Non-binary Constraints, 1998.
2. M. Carlsson and N. Beldiceanu. Arc-consistency for a chain of lexicographic ordering constraints. Technical Report T2002-18, SICS, 2002.
3. M. Carlsson and N. Beldiceanu. Revisiting the lexicographic ordering constraint. Technical Report T2002-17, SICS, 2002.
4. P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetry in matrix models. In P. van Hentenryck, editor, *Proceedings of 8th CP (CP-2002)*, pages 462–476. Springer, 2002.
5. A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In P. van Hentenryck, editor, *Proceedings of 8th CP (CP-2002)*, pages 93–108. Springer, 2002.
6. B. Hnich, Z. Kiziltan, and T. Walsh. Global constraints by composition: lexicographic ordering with sums. Technical Report APES-59-2003, 2003.
7. G. Katsirelos and F. Bacchus. GAC on conjunctions of constraints. In T. Walsh, editor, *Proceedings of 7th CP (CP-2001)*, pages 610–614. Springer, 2001.
8. C.C. Lindner and A. Rosa. Topics on steiner systems. *Annals of Discrete Mathematics*, 7, 1980.
9. P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129(1-2):133–163, 2001.
10. J.C. Régin and M. Rueher. A global constraint combining a sum constraint and difference constraints. In R. Dechter, editor, *Proceedings of the 6th CP (CP-2000)*, pages 384–395. Springer, 2000.