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 two 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 [4]. 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 for the branching heuristic trying to push the search in a different direction to the symmetry breaking constraints. Combining constraints is a 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 `LexLeqAndSum`$(\boldsymbol{X}, \boldsymbol{Y}, Sx, Sy)$ for the constraint which ensure that $\boldsymbol{X} \leq_{lex} \boldsymbol{Y}$, $\sum_i X_i = Sx$ $\sum_i Y_i = Sy$. Similarly, we write `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 `LexGeqAndSum`$(\boldsymbol{X}, \boldsymbol{Y},$
$Sx, Sy)$ and as `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 $\boldsymbol{sy} = \langle sy_0, \ldots, sy_{n-1}\rangle$. To serve repeatedly traversing the vectors we have 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

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

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

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

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 domains that are GAC:

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

4 Algorithm

The algorithm first establishes BC on the sum constraints. 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 $\boldsymbol{X}$ as in the worked example. In step 2, we are concerned with support for 0s in $\boldsymbol{X}$. Step 3 is very similar to step 1, except we identify support for the 0s in $\boldsymbol{Y}$. Step 4 is very similar to step 2, except we identify support for the 1s in $\boldsymbol{Y}$. None of the prunings require any recursive calls back to the algorithm. The algorithm runs in linear time in the length of the vectors and is correct. But, for reasons of space, the details of the algorithm and the proofs are given in [5].

The algorithm can easily be modified for `LexLessAndSum`$(\boldsymbol{X}, \boldsymbol{Y}, Sx, Sy)$ (the strict ordering constraint). 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 $min\{\boldsymbol{x} | \sum_i x_i = Sx \ \wedge \ \boldsymbol{x} \in \boldsymbol{X}\} \geq_{lex} max\{\boldsymbol{y} | \sum_i y_i = Sy \ \wedge \ \boldsymbol{y} \in \boldsymbol{Y}\}$.

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.

Finally, we can extend the algorithm to detect constraint 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 $max\{\boldsymbol{x} | \; \sum_i x_i = Sx \;\wedge\; \boldsymbol{x} \in \boldsymbol{X}\} \leq_{lex} min\{\boldsymbol{y} | \; \sum_i y_i = Sy \;\wedge\; \boldsymbol{y} \in \boldsymbol{Y}\}$.

5 Experimental Results

We performed a wide range of experiments to test when this combination of constraints is useful in practice. But, for reasons of space, we only show the results for BIBDs. In the following table, 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.

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. This matrix model has row and column symmetry[3], and both the rows and the columns are now also constrained by sum constraints. Hence, we can impose our new global constraint (both) on the rows and the columns.

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 symmetry breaking constraints. All the instances of [7] 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

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 1. BIBDs: row-wise labelling.

less than a minute. With this search strategy, we observe no difference between the inference of our algorithm and its decomposition into seperate lexicographic ordering and sum constraints.

To see a difference, we need either a poor branching heuristic or a large search space (e.g., an unsatisfiable problem). 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 the symmetry breaking 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 and then one row from the bottom, and so on. Table 1 shows how the search tree is affected. We make a number of observations about these result. First, imposing the symmetry breaking 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.

6 Lexicographic Ordering with Other Constraints

We obtained similar results with other problems like ternary Steiner, 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 [6]. 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 (α points to the most significant index of $\boldsymbol{X}$ and $\boldsymbol{Y}$ where X_i and Y_i are not ground and equal). 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 [1]. 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 2, 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.

7 Conclusion

We have introduced a new global constraint on 0/1 variables which combines a lexicographical ordering constraint with sum constraints. Lexicographic ordering constraints are frequently used to break symmetry, whilst sum constraints occur in many problems involving capacity or partitioning. Our results showed that

v, b, r, k, λ	No symmetry breaking	$<_{lex}$ R $\leq_{lex}$ C		$>_{lex}$ R $\geq_{lex}$ C	
		lex_chain		**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 2. 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.

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. Overall, 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. M. Carlsson and N. Beldiceanu. Arc-consistency for a chain of lexicographic ordering constraints. Technical Report T2002-18, SICS, 2002.
2. M. Carlsson and N. Beldiceanu. Revisiting the lexicographic ordering constraint. Technical Report T2002-17, SICS, 2002.
3. 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.
4. 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.
5. Z. Kiziltan. Symmetry Breaking Ordering Constraints. PhD thesis, Uppsala University. Due to be submitted late 2003.
6. 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.
7. P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129(1-2):133–163, 2001.